Programmer's Reference Guide

Zend Framework

Programmer's Reference Guide: Zend Framework

Published 2008-08-30

Table of Contents

1. Introduction to Zend Framework 1
    - Overview 1
    - Installation 1
2. Zend_Acl 3
    - Introduction 3
        - About Resources 3
        - About Roles 3
        - Creating the Access Control List (ACL) 5
        - Registering Roles 5
        - Defining Access Controls 6
        - Querying the ACL 7
    - Refining Access Controls 7
        - Precise Access Controls 7
        - Removing Access Controls 9
    - Advanced Usage 10
        - Storing ACL Data for Persistence 10
        - Writing Conditional ACL Rules with Assertions 11
3. Zend_Auth 13
    - Introduction 13
        - Adapters 13
        - Results 14
        - Identity Persistence 15
        - Using Zend_Auth 18
    - Database Table Authentication 20
        - Introduction 20
        - Advanced Use: Persisting a DbTable Result Object 22
        - Advanced Usage By Example 23
    - Digest Authentication 25
        - Introduction 25
        - Specifics 25
        - Identity 25
    - HTTP Authentication Adapter 26
        - Introduction 26
        - Design Overview 26
        - Configuration Options 27
        - Resolvers 27
        - Basic Usage 28
    - LDAP Authentication 29
        - Introduction 29
        - Usage 29
        - The API 31
        - Server Options 33
        - Collecting Debugging Messages 35
        - Common Options for Specific Servers 36
    - Open ID Authentication 37
        - Introduction 37
        - Specifics 38
4. Zend_Cache 40
    - Introduction 40
    - The theory of caching 43
        - The `Zend_Cache` factory method 43

Tagging records ... 44
Cleaning the cache ... 44
Zend_Cache frontends ... 45
Zend_Cache_Core ... 45
Zend_Cache_Frontend_Output ... 48
Zend_Cache_Frontend_Function ... 48
Zend_Cache_Frontend_Class ... 49
Zend_Cache_Frontend_File ... 50
Zend_Cache_Frontend_Page ... 51
Zend_Cache backends ... 56
Zend_Cache_Backend_File ... 56
Zend_Cache_Backend_Sqlite ... 56
Zend_Cache_Backend_Memcached ... 57
Zend_Cache_Backend_Apc ... 57
Zend_Cache_Backend_Xcache ... 58
Zend_Cache_Backend_ZendPlatform ... 58
5. Zend_Captcha ... 59
Introduction ... 59
Captcha Operation ... 59
Captcha Adapters ... 60
Zend_Captcha_Word ... 60
Zend_Captcha_Dumb ... 61
Zend_Captcha_Figlet ... 61
Zend_Captcha_Image ... 61
Zend_Captcha_ReCaptcha ... 62
6. Zend_Config ... 63
Introduction ... 63
Theory of Operation ... 64
Zend_Config_Ini ... 65
Zend_Config_Xml ... 67
7. Zend_Console_Getopt ... 70
Introduction to Getopt ... 70
Declaring Getopt Rules ... 71
Declaring Options with the Short Syntax ... 71
Declaring Options with the Long Syntax ... 71
Fetching Options and Arguments ... 72
Handling Getopt Exceptions ... 72
Fetching Options by Name ... 73
Reporting Options ... 74
Fetching Non-option Arguments ... 74
Configuring Zend_Console_Getopt ... 74
Adding Option Rules ... 74
Adding Help Messages ... 75
Adding Option Aliases ... 75
Adding Argument Lists ... 76
Adding Configuration ... 76
8. Zend_Controller ... 78
Zend_Controller Quick Start ... 78
Introduction ... 78
Quick Start ... 78
Zend_Controller Basics ... 81
The Front Controller ... 84
Overview ... 84
Primary Methods ... 84

Environmental Accessor Methods ... 86
Front Controller Parameters ... 88
Subclassing the Front Controller ... 88
The Request Object ... 89
Introduction ... 89
HTTP Requests ... 89
Subclassing the Request Object ... 92
The Standard Router ... 94
Introduction ... 94
Using a router ... 95
Basic Rewrite Router operation ... 95
Default routes ... 96
Base URL and subdirectories ... 98
Route Types ... 98
Using Zend_Config with the RewriteRouter ... 106
Subclassing the Router ... 107
The Dispatcher ... 107
Overview ... 107
Subclassing the Dispatcher ... 108
Action Controllers ... 112
Introduction ... 112
Object initialization ... 113
Pre- and Post-Dispatch Hooks ... 114
Accessors ... 114
View Integration ... 115
Utility Methods ... 117
Subclassing the Action Controller ... 117
Action Helpers ... 119
Introduction ... 119
Helper Initialization ... 120
The Helper Broker ... 120
Built-in Action Helpers ... 122
Writing Your Own Helpers ... 156
The Response Object ... 157
Usage ... 157
Manipulating Headers ... 159
Named Segments ... 159
Testing for Exceptions in the Response Object ... 161
Subclassing the Response Object ... 162
Plugins ... 162
Introduction ... 162
Writing Plugins ... 163
Using Plugins ... 163
Retrieving and Manipulating Plugins ... 164
Plugins Included in the Standard Distribution ... 164
Using a Conventional Modular Directory Structure ... 169
Introduction ... 169
Specifying Module Controller Directories ... 170
Routing to modules ... 172
Module or Global Default Controller ... 172
MVC Exceptions ... 172
Introduction ... 172
How can you handle exceptions? ... 173
MVC Exceptions You May Encounter ... 174

Migrating from Previous Versions 176
Migrating from 1.5.x to 1.6.0 or newer 177
Migrating from 1.0.x to 1.5.0 or newer 177
Migrating from 0.9.3 to 1.0.0RC1 or newer 178
Migrating from 0.9.2 to 0.9.3 or newer 180
Migrating from 0.6.0 to 0.8.0 or newer 180
Migrating from 0.2.0 or before to 0.6.0 181
9. Zend_Currency 185
Introduction to Zend_Currency 185
Why should `Zend_Currency` be used ? 185
How to work with currencies 185
Create output from an currency 187
Changing the format of a currency 188
Informational methods for Zend_Currency 190
Settings new default values 191
Speed up Zend_Currency 191
Migrating from Previous Versions 192
Migrating from 1.0.2 to 1.0.3 or newer 192
10. Zend_Date 194
Introduction 194
Always Set a Default Timezone 194
Why Use Zend_Date? 194
Theory of Operation 195
Internals 195
Basic Methods 196
The current date 196
Zend_Date by Example 196
Zend_Date API Overview 198
Zend_Date Options 198
Working with Date Values 200
Basic `Zend_Date` Operations Common to Many Date Parts 201
Comparing Dates 205
Getting Dates and Date Parts 206
Working with Fractions of Seconds 208
Sunrise / Sunset 208
Creation of dates 208
Create the actual date 208
Create a date from database 209
Create dates from an array 210
Constants for General Date Functions 210
Using Constants 210
List of All Constants 211
Self-Defined OUTPUT Formats with ISO 216
Self-defined OUTPUT formats using PHP's date() format specifiers 220
Working examples 223
Checking dates 223
Sunrise and Sunset 224
Timezones 226
11. Zend_Db 229
Zend_Db_Adapter 229
Connecting to a Database using an Adapter 229
The example database 234
Reading Query Results 236

Writing Changes to the Database 239
Quoting Values and Identifiers 243
Controlling Database Transactions 246
Listing and Describing Tables 247
Closing a Connection 248
Running Other Database Statements 249
Notes on Specific Adapters 249
Zend_Db_Statement 252
Creating a Statement 252
Executing a Statement 253
Fetching Results from a SELECT Statement 253
Zend_Db_Profiler 256
Introduction 256
Using the Profiler 258
Advanced Profiler Usage 259
Specialized Profilers 261
Zend_Db_Select 262
Overview of the Select Object 262
Creating a Select Object 262
Building Select queries 263
Executing Select Queries 278
Other methods 279
Zend_Db_Table 281
Introduction to Table Class 281
Defining a Table Class 281
Creating an Instance of a Table 285
Inserting Rows to a Table 286
Updating Rows in a Table 289
Deleting Rows from a Table 290
Finding Rows by Primary Key 290
Querying for a Set of Rows 292
Querying for a Single Row 296
Retrieving Table Metadata Information 296
Caching Table Metadata 297
Customizing and Extending a Table Class 299
Zend_Db_Table_Row 303
Introduction 303
Fetching a Row 303
Writing rows to the database 305
Serializing and unserializing rows 307
Extending the Row class 309
Zend_Db_Table_Rowset 313
Introduction 313
Fetching a Rowset 313
Retrieving Rows from a Rowset 314
Retrieving a Rowset as an Array 316
Serializing and Unserializing a Rowset 317
Extending the Rowset class 318
Zend_Db_Table Relationships 319
Introduction 319
Defining Relationships 319
Fetching a Dependent Rowset 322
Fetching a Parent Row 324
Fetching a Rowset via a Many-to-many Relationship 325

Cascading Write Operations ... 328
12. Zend_Debug ... 332
Dumping Variables ... 332
13. Zend_Dojo ... 333
Introduction ... 333
Zend_Dojo_Data: dojo.data Envelopes ... 333
Zend_Dojo_Data Usage ... 333
Advanced Use Cases ... 335
Dojo View Helpers ... 337
dojo() View Helper ... 337
Dijit-Specific View Helpers ... 343
Dojo Form Elements and Decorators ... 355
Dijit-Specific Form Decorators ... 356
Dijit-Specific Form Elements ... 358
Dojo Form Examples ... 374
14. Zend_Dom ... 382
Introduction ... 382
Zend_Dom_Query ... 382
Theory of Operation ... 382
Methods Available ... 383
15. Zend_Exception ... 385
Using Exceptions ... 385
16. Zend_Feed ... 386
Introduction ... 386
Importing Feeds ... 387
Custom feeds ... 388
Retrieving Feeds from Web Pages ... 391
Consuming an RSS Feed ... 392
Consuming an Atom Feed ... 393
Consuming a Single Atom Entry ... 394
Modifying Feed and Entry structures ... 395
Custom Feed and Entry Classes ... 396
17. Zend_File ... 399
Zend_File_Transfer ... 399
Validators for Zend_File_Transfer ... 400
Using validators with `Zend_File_Transfer` ... 401
Count validator ... 403
Exists validator ... 403
Extension validator ... 404
FilesSize validator ... 405
ImageSize validator ... 406
MimeType validator ... 407
NotExists validator ... 408
Size validator ... 408
18. Zend_Filter ... 410
Introduction ... 410
What is a filter? ... 410
Basic usage of filters ... 410
Using the static `get()` method ... 410
Standard Filter Classes ... 411
Alnum ... 411
Alpha ... 411
BaseName ... 411

Digits 411
Dir 411
HtmlEntities 411
Int 411
StripNewlines 412
RealPath 412
StringToLower 412
StringToUpper 412
StringTrim 412
StripTags 412
Filter Chains 412
Writing Filters 413
Zend_Filter_Input 413
Declaring Filter and Validator Rules 414
Creating the Filter and Validator Processor 416
Retrieving Validated Fields and other Reports 416
Using Metacommands to Control Filter or Validator Rules 419
Adding Filter Class Namespaces 424
Zend_Filter_Inflector 425
Operation 426
Setting Paths To Alternate Filters 426
Setting the Inflector Target 427
Inflection Rules 428
Utility Methods 430
Using Zend_Config with Zend_Filter_Inflector 431
19. Zend_Form 432
Zend_Form 432
Zend_Form Quick Start 432
Create a form object 432
Add elements to the form 433
Render a form 435
Check if a form is valid 437
Get error status 437
Putting it together 438
Using a Zend_Config object 440
Conclusion 440
Creating Form Elements Using Zend_Form_Element 441
Plugin Loaders 441
Filters 443
Validators 444
Decorators 449
Metadata and Attributes 452
Standard Elements 453
Zend_Form_Element Methods 453
Configuration 455
Custom Elements 456
Creating Forms Using Zend_Form 458
Plugin Loaders 458
Elements 459
Display Groups 465
Sub Forms 469
Metadata and Attributes 470
Decorators 472
Validation 474

Methods ... 476
Configuration ... 479
Custom forms ... 480
Creating Custom Form Markup Using Zend_Form_Decorator ... 482
Operation ... 482
Standard Decorators ... 483
Custom Decorators ... 483
Standard Form Elements Shipped With Zend Framework ... 486
Zend_Form_Element_Button ... 486
Zend_Form_Element_Captcha ... 487
Zend_Form_Element_Checkbox ... 488
Zend_Form_Element_File ... 488
Zend_Form_Element_Hidden ... 489
Zend_Form_Element_Hash ... 489
Zend_Form_Element_Image ... 490
Zend_Form_Element_MultiCheckbox ... 490
Zend_Form_Element_Multiselect ... 491
Zend_Form_Element_Password ... 491
Zend_Form_Element_Radio ... 492
Zend_Form_Element_Reset ... 492
Zend_Form_Element_Select ... 492
Zend_Form_Element_Submit ... 493
Zend_Form_Element_Text ... 493
Zend_Form_Element_Textarea ... 493
Standard Form Decorators Shipped With Zend Framework ... 493
Zend_Form_Decorator_Callback ... 493
Zend_Form_Decorator_Captcha ... 494
Zend_Form_Decorator_Description ... 494
Zend_Form_Decorator_DtDdWrapper ... 494
Zend_Form_Decorator_Errors ... 495
Zend_Form_Decorator_Fieldset ... 495
Zend_Form_Decorator_Form ... 495
Zend_Form_Decorator_FormElements ... 495
Zend_Form_Decorator_HtmlTag ... 495
Zend_Form_Decorator_Image ... 495
Zend_Form_Decorator_Label ... 496
Zend_Form_Decorator_ViewHelper ... 496
Zend_Form_Decorator_ViewScript ... 496
Internationalization of Zend_Form ... 498
Initializing I18n in Forms ... 498
Standard I18n Targets ... 499
Advanced Zend_Form Usage ... 500
Array Notation ... 500
Multi-Page Forms ... 503
20. Zend_Gdata ... 514
Introduction to Gdata ... 514
Structure of Zend_Gdata ... 514
Interacting with Google Services ... 515
Obtaining instances of Zend_Gdata classes ... 515
Google Data Client Authentication ... 516
Dependencies ... 516
Creating a new Gdata client ... 516
Common query parameters ... 517
Fetching a feed ... 518

Working with multi-page feeds ... 518
Working with data in feeds and entries ... 519
Updating entries ... 519
Posting entries to Google servers ... 520
Deleting entries on Google servers ... 520
Authenticating with AuthSub ... 521
Creating an AuthSub authenticated Http Client ... 521
Revoking AuthSub authentication ... 522
Authenticating with ClientLogin ... 523
Creating a ClientLogin authenticated Http Client ... 523
Terminating a ClientLogin authenticated Http Client ... 524
Using Google Calendar ... 524
Connecting To The Calendar Service ... 524
Retrieving A Calendar List ... 527
Retrieving Events ... 528
Creating Events ... 531
Modifying Events ... 534
Deleting Events ... 535
Accessing Event Comments ... 535
Using Google Documents List Data API ... 536
Get a List of Documents ... 536
Upload a Document ... 536
Searching the documents feed ... 537
Using Google Spreadsheets ... 538
Create a Spreadsheet ... 538
Get a List of Spreadsheets ... 538
Get a List of Worksheets ... 539
Interacting With List-based Feeds ... 539
Interacting With Cell-based Feeds ... 542
Using Google Apps Provisioning ... 543
Setting the current domain ... 543
Interacting with users ... 544
Interacting with nicknames ... 548
Interacting with email lists ... 550
Interacting with email list recipients ... 552
Handling errors ... 553
Using Google Base ... 553
Connect To The Base Service ... 554
Retrieve Items ... 557
Insert, Update, and Delete Customer Items ... 558
Using the YouTube Data API ... 561
Authentication ... 561
Developer Keys and Client ID ... 561
Retrieving public video feeds ... 561
Retrieving video comments ... 564
Retrieving playlist feeds ... 564
Retrieving a list of a user's subscriptions ... 565
Retrieving a user's profile ... 565
Uploading Videos to YouTube ... 565
Browser-based upload ... 567
Checking upload status ... 568
Other Functions ... 569
Using Picasa Web Albums ... 569
Connecting To The Service ... 569

Understanding and Constructing Queries 572
Retrieving Feeds And Entries 573
Creating Entries 577
Deleting Entries 579
Catching Gdata Exceptions 581
21. Zend_Http 583
Zend_Http_Client - Introduction 583
Introduction 583
Configuration Parameters 583
Performing Basic HTTP Requests 584
Adding GET and POST parameters 585
Accessing Last Request and Response 586
Zend_Http_Client - Advanced Usage 586
HTTP Redirections 586
Adding Cookies and Using Cookie Persistence 587
Setting Custom Request Headers 588
File Uploads 589
Sending Raw POST Data 590
HTTP Authentication 590
Sending Multiple Requests With the Same Client 591
Zend_Http_Client - Connection Adapters 592
Overview 592
The Socket Adapter 593
The Proxy Adapter 594
The Test Adapter 595
Creating your own connection adapters 597
Zend_Http_Cookie and Zend_Http_CookieJar 600
Introduction 600
Instantiating Zend_Http_Cookie Objects 600
Zend_Http_Cookie getter methods 602
Zend_Http_Cookie: Matching against a scenario 603
The Zend_Http_CookieJar Class: Instantiation 605
Adding Cookies to a Zend_Http_CookieJar object 605
Retrieving Cookies From a Zend_Http_CookieJar object 605
Zend_Http_Response 606
Introduction 606
Boolean Tester Methods 607
Accessor Methods 608
Static HTTP Response Parsers 609
22. Zend_InfoCard 611
Introduction 611
Basic Theory of Usage 611
Using as part of Zend_Auth 612
Using the Zend_InfoCard component standalone 613
Working with a Claims object 614
Attaching Information Cards to existing accounts 614
Creating Zend_InfoCard adapters 615
23. Zend_Json 617
Introduction 617
Basic Usage 617
JSON Objects 617
XML to JSON conversion 618
Zend_Json_Server - JSON-RPC server 620
Advanced Details 623

24. Zend_Layout 629
Introduction 629
Zend_Layout Quick Start 629
Layout scripts 629
Using Zend_Layout with the Zend Framework MVC 630
Using Zend_Layout as a Standalone Component 632
Sample Layout 632
Zend_Layout Configuration Options 633
Examples 633
Zend_Layout Advanced Usage 635
Custom View Objects 635
Custom Front Controller Plugins 636
Custom Action Helpers 636
Custom Layout Script Path Resolution: Using the Inflector 637
25. Zend_Ldap 639
Introduction 639
Theory of Operation 639
26. Zend_Loader 644
Loading Files and Classes Dynamically 644
Loading Files 644
Loading Classes 644
Testing if a File is Readable 645
Using the Autoloader 645
Loading Plugins 646
Basic Use Case 647
Manipulating Plugin Paths 648
Testing for Plugins and Retrieving Class Names 649
27. Zend_Locale 650
Introduction 650
What is Localization 650
What is a Locale? 651
How are Locales Represented? 651
Selecting the Right Locale 652
Usage of automatic Locales 652
Using a default Locale 653
ZF Locale-Aware Classes 654
Application wide locale 655
Zend_Locale_Format::setOptions(array $options) 655
Speed up Zend_Locale and its subclasses 656
Using Zend_Locale 656
Copying, Cloning, and Serializing Locale Objects 657
Equality 657
Default locales 657
Set a new locale 658
Getting the language and region 658
Obtaining localized strings 659
Obtaining translations for "yes" and "no" 669
Get a list of all known locales 670
Normalization and Localization 671
Number normalization: getNumber($input, Array $options) 671
Number localization 672
Number testing 674
Float value normalization 674
Floating point value localization 674

Floating point value testing 675
Integer value normalization 675
Integer point value localization 675
Integer value testing 676
Numeral System Conversion 676
Working with Dates and Times 678
Normalizing Dates and Times 678
Testing Dates 682
Normalizing a Time 683
Testing Times 683
Supported locales 684
28. Zend_Log 696
Overview 696
Creating a Log 696
Logging Messages 696
Destroying a Log 697
Using Built-in Priorities 697
Adding User-defined Priorities 698
Understanding Log Events 698
Writers 699
Writing to Streams 699
Writing to Databases 700
Writing to Firebug 700
Stubbing Out the Writer 703
Testing with the Mock 703
Compositing Writers 703
Formatters 704
Simple Formatting 704
Formatting to XML 704
Filters 705
Filtering for All Writers 706
Filtering for a Writer Instance 706
29. Zend_Mail 707
Introduction 707
Getting started 707
Configuring the default sendmail transport 708
Sending via SMTP 708
Sending Multiple Mails per SMTP Connection 709
Using Different Transports 710
HTML E-Mail 711
Attachments 711
Adding Recipients 712
Controlling the MIME Boundary 712
Additional Headers 713
Character Sets 713
Encoding 713
SMTP Authentication 713
Securing SMTP Transport 714
Reading Mail Messages 715
Simple example using Pop3 715
Opening a local storage 715
Opening a remote storage 716
Fetching messages and simple methods 717
Working with messages 717

Checking for flags 720
Using folders 721
Advanced Use 723
30. Zend_Measure 727
Introduction 727
Creation of Measurements 727
Creating measurements from integers and floats 728
Creating measurements from strings 728
Measurements from localized strings 729
Outputting measurements 730
Automatic output 730
Outputting values 730
Output with unit of measurement 731
Output as localized string 731
Manipulating Measurements 731
Convert 731
Add and subtract 732
Compare 733
Compare 734
Manually change values 734
Manually change types 734
Types of measurements 735
Hints for Zend_Measure_Binary 737
Hints for Zend_Measure_Number 737
Roman numbers 738
31. Zend_Memory 739
Overview 739
Introduction 739
Theory of Operation 739
Memory Manager 741
Creating a Memory Manager 741
Managing Memory Objects 741
Memory Manager Settings 742
Memory Objects 743
Movable 743
Locked 743
Memory container 'value' property. 744
Memory container interface 744
32. Zend_Mime 747
Zend_Mime 747
Introduction 747
Static Methods and Constants 747
Instantiating Zend_Mime 747
Zend_Mime_Message 748
Introduction 748
Instantiation 748
Adding MIME Parts 748
Boundary handling 748
parsing a string to create a Zend_Mime_Message object (experimental) 748
Zend_Mime_Part 749
Introduction 749
Instantiation 749
Methods for rendering the message part to a string 749
33. Zend_OpenId 750

Introduction ... 750
What is OpenID? ... 750
How Does it Work? ... 750
Zend_OpenId Structure ... 751
Supported Standards ... 751
Zend_OpenId_Consumer Basics ... 751
OpenID Authentication ... 751
Combine all Steps in One Page ... 753
Realm ... 754
Immediate Check ... 755
Zend_OpenId_Consumer_Storage ... 755
Simple Registration Extension ... 759
Integration with Zend_Auth ... 761
Integration with Zend_Controller ... 763
Zend_OpenId_Provider ... 763
Quick Start ... 763
Combine all together ... 766
Simple Registration Extension ... 767
What Else? ... 769
34. Zend_Paginator ... 770
Introduction ... 770
Usage ... 770
Paginating data collections ... 770
Rendering pages with view scripts ... 771
Configuration ... 775
Advanced usage ... 776
Custom data source adapters ... 776
Custom scrolling styles ... 776
35. Zend_Pdf ... 778
Introduction. ... 778
Creating and loading PDF documents. ... 778
Save changes to the PDF document. ... 779
Document pages. ... 780
Page creation. ... 780
Page cloning. ... 781
Drawing ... 782
Geometry ... 782
Colors ... 782
Shape Drawing ... 782
Text Drawing ... 785
Using fonts ... 786
Starting in 1.5, Extracting fonts. ... 789
Image Drawing ... 791
Line drawing style ... 792
Fill style ... 792
Rotations ... 793
Save/restore graphics state ... 793
Clipping draw area ... 794
Styles ... 795
Transparency ... 798
Document Info and Metadata. ... 799
Zend_Pdf module usage example ... 801
36. Zend_Registry ... 805
Using the Registry ... 805

Setting Values in the Registry 805
Getting Values from the Registry 805
Constructing a Registry Object 806
Accessing the Registry as an Array 806
Accessing the Registry as an Object 807
Querying if an index exists 807
Extending the Registry 808
Unsetting the Static Registry 808
37. Zend_Rest 810
Introduction 810
Zend_Rest_Client 810
Introduction 810
Responses 811
Request Arguments 812
Zend_Rest_Server 813
Introduction 813
REST Server Usage 813
Calling a Zend_Rest_Server Service 814
Sending A Custom Status 814
Returning Custom XML Responses 815
38. Zend_Search_Lucene 817
Overview 817
Introduction 817
Document and Field Objects 817
Understanding Field Types 819
HTML documents 819
Building Indexes 821
Creating a New Index 821
Updating Index 821
Updating Documents 822
Retrieving Index Size 822
Index optimization 822
Permissions 824
Limitations 824
Searching an Index 825
Building Queries 825
Search Results 826
Limiting the Result Set 827
Results Scoring 828
Search Result Sorting 828
Search Results Highlighting 829
Query Language 829
Terms 830
Fields 830
Wildcards 831
Term Modifiers 831
Range Searches 831
Fuzzy Searches 832
Proximity Searches 832
Boosting a Term 833
Boolean Operators 833
Grouping 835
Field Grouping 835
Escaping Special Characters 835

Query Construction API 836
Query Parser Exceptions 836
Term Query 836
Multi-Term Query 837
Boolean Query 838
Wildcard Query 840
Fuzzy Query 841
Phrase Query 842
Range Query 845
Character Set 846
UTF-8 and single-byte character set support 846
Default text analyzer 846
UTF-8 compatible text analyzers 846
Extensibility 848
Text Analysis 848
Tokens Filtering 850
Scoring Algorithms 851
Storage Containers 853
Interoperating with Java Lucene 855
File Formats 855
Index Directory 856
Java Source Code 856
Advanced 856
Starting from 1.6, handling index format transformations. 856
Using the index as static property 857
Best Practices 858
Field names 858
Indexing performance 859
Index during Shut Down 861
Retrieving documents by unique id 862
Memory Usage 863
Encoding 863
Index maintenance 864
39. Zend_Server 866
Introduction 866
Zend_Server_Reflection 866
Introduction 866
Usage 866
40. Zend_Service 868
Introduction 868
Zend_Service_Akismet 868
Introduction 868
Verify an API key 869
Check for spam 869
Submitting known spam 870
Submitting false positives (ham) 871
Zend-specific Accessor Methods 871
Zend_Service_Amazon 872
Introduction 872
Country Codes 873
Looking up a Specific Amazon Item by ASIN 873
Performing Amazon Item Searches 874
Using the Alternative Query API 875
Zend_Service_Amazon Classes 875

Zend_Service_Audioscrobbler 881
Introduction to Searching Audioscrobbler 881
Users 881
Artists 883
Tracks 884
Tags 884
Groups 884
Forums 885
Zend_Service_Delicious 885
Introduction 885
Retrieving posts 886
Zend_Service_Delicious_PostList 886
Editing posts 888
Deleting posts 888
Adding new posts 889
Tags 890
Bundles 890
Public data 890
HTTP client 891
Zend_Service_Flickr 892
Introduction to Searching Flickr 892
Finding Flickr Users' Photos and Information 892
Finding photos From a Group Pool 893
Retrieving Flickr Image Details 893
Zend_Service_Flickr Result Classes 894
Zend_Service_Nirvanix 896
Introduction 896
Registering with Nirvanix 896
API Documentation 896
Features 896
Getting Started 897
Understanding the Proxy 897
Examining Results 898
Handling Errors 899
Zend_Service_ReCaptcha 900
Introduction 900
Simplest use 900
Zend_Service_Simpy 901
Introduction 901
Links 901
Tags 903
Notes 904
Watchlists 905
Introduction 906
Getting Started with `Zend_Service_SlideShare` 906
The SlideShow object 907
Retrieving a single slide show 910
Retrieving Groups of Slide Shows 910
`Zend_Service_SlideShare` Caching policies 911
Changing the behavior of the HTTP Client 912
Zend_Service_StrikeIron 912
Overview 912
Registering with StrikeIron 913

Getting Started ... 913
Making Your First Query ... 914
Examining Results ... 914
Handling Errors ... 915
Checking Your Subscription ... 916
Zend_Service_StrikeIron: Bundled Services ... 917
ZIP Code Information ... 917
U.S. Address Verification ... 918
Sales & Use Tax Basic ... 918
Zend_Service_StrikeIron: Advanced Uses ... 919
Using Services by WSDL ... 919
Viewing SOAP Transactions ... 920
Zend_Service_Technorati ... 921
Introduction ... 921
Getting Started ... 921
Making Your First Query ... 921
Consuming Results ... 922
Handling Errors ... 924
Checking Your API Key Daily Usage ... 924
Available Technorati Queries ... 925
Zend_Service_Technorati Classes ... 929
Zend_Service_Yahoo ... 933
Introduction ... 933
Searching the Web with Yahoo! ... 933
Finding Images with Yahoo! ... 934
Finding videos with Yahoo! ... 934
Finding Local Businesses and Services with Yahoo! ... 934
Searching Yahoo! News ... 935
Searching Yahoo! Site Explorer Inbound Links ... 935
Searching Yahoo! Site Explorer's PageData ... 935
Zend_Service_Yahoo Classes ... 936
41. Zend_Session ... 942
Introduction ... 942
Basic Usage ... 942
Tutorial Examples ... 943
Iterating Over Session Namespaces ... 944
Accessors for Session Namespaces ... 944
Advanced Usage ... 945
Starting a Session ... 945
Locking Session Namespaces ... 946
Namespace Expiration ... 947
Session Encapsulation and Controllers ... 947
Preventing Multiple Instances per Namespace ... 948
Working with Arrays ... 949
Using Sessions with Objects ... 951
Using Sessions with Unit Tests ... 951
Global Session Management ... 953
Configuration Options ... 953
Error: Headers Already Sent ... 956
Session Identifiers ... 956
`rememberMe(integer $seconds)` ... 958
`forgetMe()` ... 958
`sessionExists()` ... 958

`destroy(bool $remove_cookie = true, bool $readonly = true)` 958
`stop()` 959
`writeClose($readonly = true)` 959
`expireSessionCookie()` 959
`setSaveHandler(Zend_Session_SaveHandler_Interface $interface)` 959
`namespaceIsset($namespace)` 959
`namespaceUnset($namespace)` 960
`namespaceGet($namespace)` 960
`getIterator()` 960
Zend_Session_SaveHandler_DbTable 960
42. Zend_Soap 963
Zend_Soap_Server 963
`Zend_Soap_Server` constructor. 963
Methods to define Web Service API. 964
Request and response objects handling. 965
Zend_Soap_Client 967
`Zend_Soap_Client` Constructor 967
Performing SOAP Requests 968
WSDL Accessor 970
`Zend_Soap_Wsdl` constructor. 970
`addMessage()` method. 970
`addPortType()` method. 971
`addPortOperation()` method. 971
`addBinding()` method. 971
`addBindingOperation()` method. 971
`addSoapBinding()` method. 972
`addSoapOperation()` method. 972
`addService()` method. 972
Type mapping. 973
`addDocumentation()` method. 974
Get finilised WSDL document. 974
AutoDiscovery. 974
AutoDiscovery. Introduction 974
Class autodiscovering. 975
Functions autodiscovering. 976
Autodiscovering. Datatypes. 977
43. Zend_Test 978
Introduction 978
Zend_Test_PHPUnit 978
Bootstrapping your TestCase 980
Testing your Controllers and MVC Applications 981
Assertions 983
Examples 985
44. Zend_Text 991
Zend_Text_Figlet 991
45. Zend_TimeSync 993
Introduction 993
Why `Zend_TimeSync` ? 993
What is NTP ? 994
What is SNTP? 994
Problematic usage 994

Decide which server to use ... 994
Working with Zend_TimeSync ... 995
Generic timeserver request ... 995
Multiple timeservers ... 995
Protocols of timeservers ... 996
Using ports for timeservers ... 996
Options for timeservers ... 997
Using different timeservers ... 997
Informations from timeservers ... 998
Taking care of exceptions ... 998
46. Zend_Translate ... 999
Introduction ... 999
Starting multi-lingual ... 999
Adapters for Zend_Translate ... 1000
How to decide which translation adapter to use ... 1000
Integrate self written Adapters ... 1002
Speedup all Adapters ... 1003
Using Translation Adapters ... 1003
Translation Source Structures ... 1005
Creating array source files ... 1007
Creating Gettext Source Files ... 1008
Creating TMX Source Files ... 1009
Creating CSV Source Files ... 1009
Creating INI Source Files ... 1010
Options for adapters ... 1011
Handling languages ... 1012
Automatic source detection ... 1015
Checking for translations ... 1018
Access to the source data ... 1019
47. Zend_Uri ... 1021
Zend_Uri ... 1021
Overview ... 1021
Creating a New URI ... 1021
Manipulating an Existing URI ... 1021
URI Validation ... 1022
Common Instance Methods ... 1022
48. Zend_Validate ... 1024
Introduction ... 1024
What is a validator? ... 1024
Basic usage of validators ... 1024
Customizing messages ... 1025
Using the static `is()` method ... 1026
Standard Validation Classes ... 1027
Alnum ... 1027
Alpha ... 1027
Barcode ... 1027
Between ... 1027
Ccnum ... 1027
Date ... 1027
Digits ... 1027
EmailAddress ... 1027
Float ... 1029
GreaterThan ... 1030
Hex ... 1030

Hostname 1030
InArray 1032
Int 1032
Ip 1032
LessThan 1032
NotEmpty 1032
Regex 1032
StringLength 1033
Validator Chains 1033
Writing Validators 1034
49. Zend_Version 1040
Reading the Zend Framework Version 1040
50. Zend_View 1041
Introduction 1041
Controller Script 1041
View Script 1041
Options 1042
Short Tags with View Scripts 1043
Utility Accessors 1044
Controller Scripts 1044
Assigning Variables 1044
Rendering a View Script 1045
View Script Paths 1045
View Scripts 1046
Escaping Output 1047
Using Alternate Template Systems 1048
View Helpers 1053
Initial Helpers 1054
Helper Paths 1081
Writing Custom Helpers 1082
Zend_View_Abstract 1084
51. Zend_Wildfire 1085
Zend_Wildfire 1085
52. Zend_XmlRpc 1086
Introduction 1086
Zend_XmlRpc_Client 1086
Introduction 1086
Method Calls 1086
Types and Conversions 1087
Server Proxy Object 1089
Error Handling 1090
Server Introspection 1091
From Request to Response 1091
HTTP Client and Testing 1092
Zend_XmlRpc_Server 1092
Introduction 1092
Basic Usage 1092
Server Structure 1093
Conventions 1093
Utilizing Namespaces 1094
Custom Request Objects 1094
Custom Responses 1095
Handling Exceptions via Faults 1095
Caching Server Definitions Between Requests 1095

Usage Examples ... 1096
A. Zend Framework Requirements ... 1100
PHP Version ... 1100
PHP Extensions ... 1100
Zend Framework Components ... 1106
Zend Framework Dependencies ... 1111
B. Zend Framework Coding Standard for PHP ... 1119
Overview ... 1119
Scope ... 1119
Goals ... 1119
PHP File Formatting ... 1119
General ... 1119
Indentation ... 1119
Maximum Line Length ... 1120
Line Termination ... 1120
Naming Conventions ... 1120
Classes ... 1120
Filenames ... 1120
Functions and Methods ... 1121
Variables ... 1121
Constants ... 1122
Coding Style ... 1122
PHP Code Demarcation ... 1122
Strings ... 1122
Arrays ... 1123
Classes ... 1124
Functions and Methods ... 1125
Control Statements ... 1127
Inline Documentation ... 1128
C. Copyright Information ... 1131
Index ... 1132

List of Tables

2.1. Access Controls for an Example CMS ... 5
3.1. Configuration Options ... 27
3.2. Server Options ... 34
3.3. Debugging Messages ... 35
3.4. Options for Active Directory ... 36
3.5. Options for OpenLDAP ... 37
4.1. Core frontend options ... 45
4.2. Function frontend options ... 48
4.3. Class frontend options ... 49
4.4. File frontend options ... 51
4.5. Page frontend options ... 53
4.6. File backend options ... 56
4.7. Sqlite backend options ... 57
4.8. Memcached backend options ... 57
4.9. Xcache backend options ... 58
6.1. Zend_Config_Ini Constructor parameters ... 67
9.1. Constants for the selecting the currency description ... 189
9.2. Constants for the selecting the currency position ... 189
10.1. Date Parts ... 203
10.2. Basic Operations ... 205
10.3. Date Comparison Methods ... 206
10.4. Date Output Methods ... 207
10.5. Date Output Methods ... 208
10.6. Miscellaneous Methods ... 208
10.7. Operations involving Zend_Date::HOUR ... 211
10.8. Day Constants ... 212
10.9. Week Constants ... 212
10.10. Month Constants ... 213
10.11. Year Constants ... 213
10.12. Time Constants ... 214
10.13. Timezone Constants ... 214
10.14. Date Format Constants (formats include timezone) ... 215
10.15. Date and Time Formats (format varies by locale) ... 216
10.16. Constants for **ISO 8601 date output** ... 218
10.17. Constants for **PHP date output** ... 221
10.18. Types of supported horizons for sunset and sunrise ... 225
11.1. Metadata fields returned by describeTable() ... 248
11.2. Constants used by getPart() and reset() ... 280
20.1. Metadata used in the code-sample below ... 566
21.1. Zend_Http_Client configuration parameters ... 584
21.2. Zend_Http_Client_Adapter_Socket configuration parameters ... 593
21.3. Zend_Http_Client configuration parameters ... 594
25.1. Zend_Ldap Options ... 641
25.2. `accountCanonicalForm` ... 642
27.1. Details for getTranslationList($type = null, $locale = null, $value = null) ... 660
27.2. Details for getTranslation($value = null, $type = null, $locale = null) ... 664
27.3. Differences between ZF 1.0 and ZF 1.5 ... 667
27.4. Format tokens for self generated number formats ... 673
27.5. List of supported numeral systems ... 678
27.6. Key values for getDate() with option 'fix_date' ... 679
27.7. Return values ... 679

27.8. Format definition 682
27.9. Example formats 682
27.10. List of all supported languages 685
28.1. Firebug Logging Styles 702
29.1. Mail Read Feature Overview 715
29.2. Mail Folder Names 722
30.1. List of measurement types 736
34.1. Adapters for `Zend_Paginator` 770
34.2. Scrolling styles for `Zend_Paginator` 772
34.3. Properties available to view partials 775
34.4. Configuration methods for `Zend_Paginator` 775
38.1. Zend_Search_Lucene_Field Types 819
40.1. Zend_Service_Amazon_Item Properties 877
40.2. Zend_Service_Amazon_Image Properties 877
40.3. Zend_Service_Amazon_OfferSet Properties 878
40.4. Properties 879
40.5. Zend_Service_Amazon_SimilarProduct Properties 879
40.6. Zend_Service_Amazon_Accessories Properties 879
40.7. Zend_Service_Amazon_CustomerReview Properties 880
40.8. Zend_Service_Amazon_EditorialReview Properties 880
40.9. Zend_Service_Amazon_Listmania Properties 880
40.10. Methods for retrieving public data 890
40.11. Methods of the `Zend_Service_Delicious_SimplePost` class 891
40.12. Zend_Service_Flickr_ResultSet Properties 894
40.13. Zend_Service_Flickr_Result Properties 895
40.14. Zend_Service_Flickr_Image Properties 895
40.15. Zend_Service_Yahoo_ResultSet 937
40.16. Zend_Service_Yahoo_LocalResultSet Properties 938
40.17. Zend_Service_Yahoo_Result Properties 939
40.18. Zend_Service_Yahoo_WebResult Properties 939
40.19. Zend_Service_Yahoo_ImageResult Properties 939
40.20. Zend_Service_Yahoo_VideoResult Properties 940
40.21. Zend_Service_Yahoo_LocalResult Properties 940
40.22. Zend_Service_Yahoo_NewsResult Properties 941
40.23. Zend_Service_Yahoo_Image Properties 941
46.1. Adapters for Zend_Translate 1000
46.2. Options for Translation Adapters 1012
52.1. PHP and XML-RPC Type Conversions 1087
52.2. `Zend_XmlRpc_Value` Objects for XML-RPC Types 1089
A.1. PHP Extensions Used in Zend Framework by Component 1101
A.2. Zend Framework Components and the PHP Extensions they use 1107
A.3. Zend Framework Components and their dependency to other Zend Framework Components 1112

List of Examples

2.1. Multiple Inheritance amoung Roles 4
3.1. Modifying the Session Namespace 16
3.2. Using a Custom Storage Class 17
3.3. Basic Usage 21
4.1. Getting a frontend with `Zend_Cache::factory()` 40
4.2. Caching a database query result 41
4.3. Caching output with `Zend_Cache` output frontend 42
6.1. Using Zend_Config Per Se 63
6.2. Using Zend_Config with a PHP Configuration File 64
6.3. Using Zend_Config_Ini 66
6.4. Using Zend_Config_Xml 68
6.5. Using tag attributes in Zend_Config_Xml 69
7.1. Using the Short Syntax 71
7.2. Using the Long Syntax 72
7.3. Catching Getopt Exceptions 73
7.4. Using getOption() 73
7.5. Using __get() and __isset() magic methods 73
7.6. Using getRemainingArgs() 74
7.7. Using addRules() 75
7.8. Using setHelp() 75
7.9. Using setAliases() 76
7.10. Using addArguments() and setArguments() 76
7.11. Using setOption() 77
7.12. Using setOptions() 77
8.1. How to Handle Non-Existent Actions 118
8.2. Adding a task using action, controller and module names 123
8.3. Adding a task using a request object 124
8.4. AutoCompletion with Dojo Using Zend MVC 127
8.5. Allowing Actions to Respond To Ajax Requests 137
8.6. Setting Options 141
8.7. Using Defaults 142
8.8. Using goto()'s _forward() API 143
8.9. Using route assembly with gotoRoute() 144
8.10. Basic Usage 151
8.11. Disabling autorender 152
8.12. Choosing a different view script 153
8.13. Modifying the registered view 154
8.14. Changing the path specifications 155
8.15. Rendering multiple view scripts from a single action 156
8.16. Standard usage 168
8.17. Setting a different error handler 168
8.18. Using accessors 168
9.1. Creating an instance of Zend_Currency from the actual locale 185
9.2. Other examples for creating an instance of Zend_Currency 187
9.3. Creating output for an currency 188
9.4. Changing the displayed format of a currency 190
9.5. Getting informations from currencies 191
9.6. Setting a new locale 191
9.7. Caching currencies 192
10.1. Setting a default timezone 194
10.2. Creating the current date 196

10.3. get() - output a date 197
10.4. set() - set a date 197
10.5. add() - adding dates 197
10.6. compare() - compare dates 198
10.7. equals() - identify a date or date part 198
10.8. User-specified input date format 200
10.9. Operating on Parts of Dates 201
10.10. Date creation by instance 208
10.11. Static date creation 209
10.12. Quick creation of dates from database date values 209
10.13. Convenient creation of dates from database date values 209
10.14. Date creation by array 210
10.15. Example usage for self-defined ISO formats 217
10.16. Example usage for self-defined formats with PHP specifier 220
10.17. Checking dates 224
10.18. Getting all available cities 224
10.19. Getting the location for a city 225
10.20. Calculating sun informations 226
10.21. Working with timezones 227
10.22. Multiple timezones 228
11.1. Using an Adapter constructor 230
11.2. Using the Adapter factory method 230
11.3. Using the Adapter factory method for a custom adapter class 231
11.4. Using the Adapter factory method with a Zend_Config object 231
11.5. Passing the case-folding option to the factory 232
11.6. Passing the auto-quoting option to the factory 233
11.7. Passing PDO driver options to the factory 233
11.8. Handling connection exceptions 234
11.9. Using fetchAll() 236
11.10. Using setFetchMode() 237
11.11. Using fetchAssoc() 237
11.12. Using fetchCol() 238
11.13. Using fetchPairs() 238
11.14. Using fetchRow() 238
11.15. Using fetchOne() 239
11.16. Inserting to a table 239
11.17. Inserting expressions to a table 240
11.18. Using lastInsertId() for an auto-increment key 240
11.19. Using lastInsertId() for a sequence 240
11.20. Using lastSequenceId() 241
11.21. Updating rows 242
11.22. Updating rows using an array of expressions 242
11.23. Deleting rows 243
11.24. Using quote() 244
11.25. Using quote() with a SQL type 244
11.26. Using quoteInto() 245
11.27. Using quoteInto() with a SQL type 245
11.28. Using quoteIdentifier() 246
11.29. Managing a transaction to ensure consistency 247
11.30. Closing a database connection 249
11.31. Running a non-prepared statement in a PDO adapter 249
11.32. Creating a SQL statement object with query() 252
11.33. Using a SQL statement constructor 253
11.34. Executing a statement with positional parameters 253

11.35. Executing a statement with named parameters ... 253
11.36. Using fetch() in a loop ... 254
11.37. Using fetchAll() ... 254
11.38. Setting the fetch mode ... 255
11.39. Using fetchColumn() ... 255
11.40. Using fetchObject() ... 256
11.41. DB Profiling with `Zend_Controller_Front` ... 261
11.42. DB Profiling without `Zend_Controller_Front` ... 262
11.43. Example of the database adapter's select() method ... 263
11.44. Example of creating a new Select object ... 263
11.45. Example of the using methods to add clauses ... 263
11.46. Example of the using the fluent interface ... 264
11.47. Example of the from() method ... 264
11.48. Example of specifying a table correlation name ... 264
11.49. Example of specifying a schema name ... 265
11.50. Examples of specifying columns ... 266
11.51. Examples of specifying columns containing expressions ... 267
11.52. Examples of quoting columns in an expression ... 268
11.53. Examples of adding columns with the `columns()` method ... 268
11.54. Example of the join() method ... 269
11.55. Example of specifying no columns ... 269
11.56. Example of the joinUsing() method ... 271
11.57. Example of the where() method ... 271
11.58. Example of a parameter in the where() method ... 272
11.59. Example of multiple where() methods ... 272
11.60. Example of the orWhere() method ... 273
11.61. Example of parenthesizing Boolean expressions ... 273
11.62. Example of the group() method ... 274
11.63. Example of the having() method ... 275
11.64. Example of the order() method ... 276
11.65. Example of the limit() method ... 276
11.66. Example of the limitPage() method ... 277
11.67. Example of the distinct() method ... 277
11.68. Example of forUpdate() method ... 278
11.69. Example using the Db adapter's query() method ... 278
11.70. Example using the Select object's query method ... 278
11.71. Example of the __toString() method ... 279
11.72. Example of the getPart() method ... 280
11.73. Example of the reset() method ... 281
11.74. Declaring a table class with explicit table name ... 282
11.75. Declaring a table class with implicit table name ... 282
11.76. Declaring a table class with schema ... 282
11.77. Declaring table and schema names upon instantiation ... 283
11.78. Example of specifying the primary key ... 283
11.79. Example of overriding the _setupTableName() method ... 284
11.80. Example usage of init() method ... 285
11.81. Example of constructing a Table using an Adapter object ... 285
11.82. Example of constructing a Table using a the Default Adapter ... 286
11.83. Example of constructing a Table using a Registry key ... 286
11.84. Example of inserting to a Table ... 287
11.85. Example of inserting expressions to a Table ... 287
11.86. Example of declaring a Table with auto-incrementing primary key ... 288
11.87. Example of declaring a Table with a sequence ... 288

11.88. Example of declaring a Table with a natural key 289
11.89. Example of updating rows in a Table 289
11.90. Example of deleting rows from a Table 290
11.91. Example of finding rows by primary key values 291
11.92. Example of finding rows by compound primary key values 292
11.93. Simple usage 293
11.94. Example of fluent interface 293
11.95. Example of finding rows by an expression 294
11.96. Example of finding rows by an expression 294
11.97. Retrieving specific columns 295
11.98. Retrieving expressions as columns 295
11.99. Using a lookup table to refine the results of fetchAll() 295
11.100. Removing the integrity check on Zend_Db_Table_Select to allow JOINed rows 296
11.101. Example of finding a single row by an expression 296
11.102. Example of getting the table name 297
11.103. Using a Default Metadata Cache for all Table Objects 298
11.104. Using a Metadata Cache for a Specific Table Object 299
11.105. Example of specifying the Row and Rowset classes 300
11.106. Example of changing the Row and Rowset classes 300
11.107. Custom logic to manage timestamps 301
11.108. Custom method to find bugs by status 302
11.109. Example of an abstract table class that implements inflection 303
11.110. Example of fetching a row 303
11.111. Example of reading a row in a rowset 304
11.112. Example of reading a column in a row 304
11.113. Example of using the toArray() method 305
11.114. Example of changing a column in a row 305
11.115. Example of creating a new row for a table 306
11.116. Example of populating a new row for a table 306
11.117. Example of using setFromArray() to set values in a new Row 307
11.118. Example of deleting a row 307
11.119. Example of serializing a row 308
11.120. Example of unserializing a serialized row 308
11.121. Example of reactivating a row 309
11.122. Specifying a custom Row class 309
11.123. Example usage of init() method 310
11.124. Example of custom logic in a Row class 311
11.125. Example of a Row class that logs insert data for multiple tables 312
11.126. Example of defining an inflection transformation 313
11.127. Example of fetching a rowset 314
11.128. Counting the Rows in a Rowset 314
11.129. Reading a Single Row from a Rowset 314
11.130. Iterating through a Rowset 315
11.131. Seeking to a known position into a Rowset 315
11.132. Using toArray() 316
11.133. Serializing a Rowset 317
11.134. Unserializing a Serialized Rowset 317
11.135. Reactivating a Rowset as Live Data 318
11.136. Specifying a custom Rowset class 318
11.137. Example of Rowset class with a new method 319
11.138. Fetching a Dependent Rowset 322
11.139. Fetching a Dependent Rowset By a Specific Rule 322
11.140. Fetching a Dependent Rowset using a Zend_Db_Table_Select 323
11.141. Fetching Dependent Rowsets using the Magic Method 324

11.142. Fetching the Parent Row 324
11.143. Fetching a Parent Row By a Specific Rule 325
11.144. Fetching the Parent Row using the Magic Method 325
11.145. Fetching a Rowset with the Many-to-many Method 326
11.146. Fetching a Rowset with the Many-to-many Method By a Specific Rule 327
11.147. Fetching Rowsets using the Magic Many-to-many Method 328
11.148. Example of a Cascading Delete 329
11.149. Example Declaration of Cascading Operations 330
12.1. Example of dump() method 332
13.1. Zend_Dojo_Data initialization via constructor 334
13.2. Zend_Dojo_Data initialization via mutators 334
13.3. Appending data to Zend_Dojo_Data 334
13.4. Specifying a label field in Zend_Dojo_Data 335
13.5. Populating Zend_Dojo_Data from JSON 335
13.6. Using Dojo View Helpers 337
13.7. dojo() View Helper Usage Example 338
13.8. Specifying Declarative and Programmatic Dojo Usage 340
13.9. Registering the Dojo View Helper Prefix Path 344
13.10. BorderContainer layout dijit example 347
13.11. Enabling Dojo in your existing forms 356
13.12. DijitElement Decorator Usage 357
13.13. DijitContainer Decorator Usage 358
13.14. Example Button dijit element usage 359
13.15. Example CheckBox dijit element usage 360
13.16. ComboBox dijit element usage as select input 361
13.17. ComboBox dijit element usage with datastore 361
13.18. Example CurrencyTextBox dijit element usage 362
13.19. Example DateTextBox dijit element usage 363
13.20. Example HorizontalSlider dijit element usage 365
13.21. Example NumberSpinner dijit element usage 366
13.22. Example NumberTextBox dijit element usage 367
13.23. Example PasswordTextBox dijit element usage 368
13.24. Example RadioButton dijit element usage 368
13.25. Example SubmitButton dijit element usage 369
13.26. Example TextBox dijit element usage 370
13.27. Example Textarea dijit element usage 370
13.28. Example TimeTextBox dijit element usage 371
13.29. Example ValidationTextBox dijit element usage 372
13.30. Example VerticalSlider dijit element usage 373
13.31. Using Zend_Dojo_Form 375
13.32. Modifying an existing form to utilize Dojo 381
15.1. Example of catching an exception 385
16.1. Putting Zend_Feed to Work on RSS Feed Data 387
16.2. Basic Use of an Atom Feed 394
16.3. Reading a Single-Entry Atom Feed 395
16.4. Using the Entry Object Directly for a Single-Entry Atom Feed 395
16.5. Modifying an Existing Feed Entry 395
16.6. Creating an Atom Entry with Elements of Custom Namespaces 396
16.7. Extending the Atom Entry Class with Custom Namespaces 397
17.1. Simple File-Upload Form 400
17.2. Add validators to a file transfer 402
17.3. Limit validators to single files 402
17.4. Add multiple validators 402
17.5. Using the Count validator 403

17.6. Using the Exists validator 404
17.7. Using the Extension validator 405
17.8. Using the FilesSize validator 406
17.9. Using the ImageSize validator 407
17.10. Using the MimeType validator 407
17.11. Using the NotExists validator 408
17.12. Using the Size validator 409
18.1. Transforming MixedCase and camelCaseText to another format 426
18.2. Setting Multiple Rules at Once 430
18.3. Using Zend_Config with Zend_Filter_Inflector 431
19.1. Custom Label 442
19.2. Setting Prefix Paths for All Elements 461
19.3. Setting Decorators for All Elements 462
19.4. Setting Decorators for Some Elements 463
19.5. Setting Filters for All Elements 464
19.6. Setting Decorator Prefix Path for All Display Groups 466
19.7. Setting Decorators for All Display Groups 466
19.8. File form element usage 489
19.9. Registration Form Example 504
21.1. Instantiating a Zend_Http_Client object 583
21.2. Performing a Simple GET Request 584
21.3. Using Request Methods Other Than GET 585
21.4. Setting GET Parameters 585
21.5. Setting POST Parameters 586
21.6. Forcing RFC 2616 Strict Redirections on 301 and 302 Responses 586
21.7. Setting Cookies Using setCookie() 587
21.8. Enabling Cookie Stickiness 588
21.9. Setting A Single Custom Request Header 588
21.10. Setting Multiple Custom Request Headers 589
21.11. Using setFileUpload to Upload Files 589
21.12. Sending Raw POST Data 590
21.13. Setting HTTP Authentication User and Password 591
21.14. Performing consecutive requests with one client 592
21.15. Changing the HTTPS transport layer 594
21.16. Using Zend_Http_Client behind a proxy server 595
21.17. Testing Against a Single HTTP Response Stub 596
21.18. Testing Against Multiple HTTP Response Stubs 597
21.19. Creating your own connection adapter 599
21.20. Instantiating a Zend_Http_Cookie object 601
21.21. Stringifying a Zend_Http_Cookie object 602
21.22. Using getter methods with Zend_Http_Cookie 603
21.23. Matching cookies 604
21.24. Instantiating a Zend_Http_Response object using the factory method 607
21.25. Using the isError() method to validate a response 608
21.26. Using Zend_Http_Response Accessor Methods 608
21.27. Accessing Response Headers 609
23.1. Zend_Json_Server Usage 621
24.1. Passing options to the constructor or startMvc() 634
24.2. Using setOption() and setConfig() 635
24.3. Using Accessors 635
24.4. Using Zend_Layout accessors to modify the inflector 637
24.5. Direct modification of Zend_Layout inflector 637
24.6. Custom inflectors 638
26.1. Example of loadFile() method 644

26.2. Example of loadClass() method 645
26.3. Example of isReadable() method 645
26.4. Example of registering the autoloader callback method 646
26.5. Example of registering the autoload callback method from an extended class 646
27.1. Choosing a specific locale 651
27.2. Automatically selecting a locale 652
27.3. Using automatic locales 653
27.4. Handling locale exceptions 654
27.5. Setting a default locale 654
27.6. Dates default to correct locale of web users 654
27.7. Overriding default locale selection 655
27.8. Performance optimization when using a default locale 655
27.9. Usage of an application wide locale 655
27.10. Dates default to correct locale of web users 656
27.11. Using STANDARD definitions for setOptions() 656
27.12. clone 657
27.13. Check for equal locales 657
27.14. Get default locales 658
27.15. setLocale 658
27.16. getLanguage and getRegion 659
27.17. getTranslationList 659
27.18. getTranslationList 668
27.19. Converting country name in one language to another 668
27.20. All available translations 668
27.21. All Languages written in their native language 669
27.22. getQuestion() 670
27.23. getLocaleList() 670
27.24. Number normalization 671
27.25. Number normalization with precision 672
27.26. Number localization 672
27.27. Number localization with precision 673
27.28. Using a self defined number format 673
27.29. Number testing 674
27.30. Floating point value normalization 674
27.31. Floating point value localization 675
27.32. Floating point value testing 675
27.33. Integer value normalization 675
27.34. Integer value localization 676
27.35. Integer value testing 676
27.36. Converting numerals from Eastern Arabic scripts to European/Latin scripts 677
27.37. Converting numerals from Latin script to Eastern Arabic script 677
27.38. Getting 4 letter CLDR script code using a native-language name of the script 677
27.39. Normalizing a date 680
27.40. Normalizing a date by locale 681
27.41. Normalizing a date with time 681
27.42. Normalizing a userdefined date 681
27.43. Automatic correction of input dates 682
27.44. Date testing 683
27.45. Normalize an unknown time 683
27.46. Testing a time 684
28.1. Logging with `Zend_Controller_Front` 701
28.2. Logging without `Zend_Controller_Front` 701
29.1. Simple E-Mail with Zend_Mail 707

29.2. Passing additional parameters to the Zend_Mail_Transport_Sendmail transport 708
29.3. Sending E-Mail via SMTP 708
29.4. Sending Multiple Mails per SMTP Connection 709
29.5. Manually controlling the transport connection 710
29.6. Using Different Transports 710
29.7. Sending HTML E-Mail 711
29.8. E-Mail Messages with Attachments 711
29.9. Changing the MIME Boundary 713
29.10. Adding E-Mail Message Headers 713
29.11. Enabling authentication within Zend_Mail_Transport_Smtp 714
29.12. Enabling a secure connection within Zend_Mail_Transport_Smtp 714
30.1. Converting measurements 727
30.2. The `meter` measurement 727
30.3. Creation using integer and floating values 728
30.4. Creation using strings 728
30.5. Arbitrary text input containing measurements 729
30.6. Localized string 729
30.7. Automatic output 730
30.8. Output a value 730
30.9. Outputting units 731
30.10. Convert 732
30.11. Adding units 732
30.12. Subtract 733
30.13. Different measurements 733
30.14. Identical measurements 733
30.15. Difference 734
30.16. Changing a value 734
30.17. Changing the type 735
31.1. Using Zend_Memory component 739
33.1. The Simple OpenID Login form 752
33.2. The Authentication Request Handler 752
33.3. The Authentication Response Verifier 753
33.4. The Complete OpenID Login Script 754
33.5. Authentication Request for Specified Realm 755
33.6. Immediate Check without Interaction 755
33.7. Databse Storage 756
33.8. Sending Requests with a Simple Registration Extension 760
33.9. Verifying Responses with a Simple Registration Extension 760
33.10. Zend_Auth Adapter for OpenID 762
33.11. The Identity 764
33.12. Simple Identity Provider 764
33.13. Simple Login Screen 765
33.14. Simple Trust Screen 766
33.15. All together 767
33.16. Identity with Profile 768
33.17. Provider with SREG 769
35.1. Create new or load existing PDF document. 779
35.2. Requesting specified revision of the PDF document. 779
35.3. Save PDF document. 780
35.4. PDF document pages management. 781
35.5. Cloning existing page. 781
35.6. Draw a string on the page 785
35.7. Draw a UTF-8-encoded string on the page 786
35.8. Create a standard font 786

35.9. Create a TrueType font 787
35.10. Create a TrueType font, but do not embed it in the PDF document. 788
35.11. Do not throw an exception for fonts that cannot be embedded. 788
35.12. Do not compress an embedded font. 788
35.13. Combining font embedding options. 789
35.14. Extracting fonts from a loaded document. 789
35.15. Extracting font from a loaded document by specifying font name. 790
35.16. Image drawing 791
35.17. Zend_Pdf module usage demo 802
36.1. Example of set() method 805
36.2. Example of get() method 805
36.3. Example of iterating over the registry 806
36.4. Example of constructing a registry 806
36.5. Example of initializing the static registry 806
36.6. Example of array access 807
36.7. Example of object access 807
36.8. Example of isRegistered() method 808
36.9. Example of isset() method 808
36.10. Example of specifying the static registry's class name 808
36.11. Example of _unsetInstance() method 809
37.1. A basic REST request 810
37.2. Response Status 811
37.3. Using Technorati's Rest Service 811
37.4. Example Technorati Response 812
37.5. Setting Request Arguments 813
37.6. Basic Zend_Rest_Server Usage - Classes 813
37.7. Basic Zend_Rest_Server Usage - Functions 814
37.8. Returning Custom Status 815
37.9. Return Custom XML 816
38.1. Custom text Analyzer. 849
40.1. isSpam() Usage 870
40.2. submitSpam() Usage 870
40.3. submitHam() Usage 871
40.4. Search Amazon Using the Traditional API 873
40.5. Search Amazon Using the Query API 873
40.6. Choosing an Amazon Web Service Country 873
40.7. Looking up a Specific Amazon Item by ASIN 874
40.8. Performing Amazon Item Searches 874
40.9. Using the `ResponseGroup` Option 874
40.10. Search Amazon Using the Alternative Query API 875
40.11. Retrieving User Profile Information 882
40.12. Retrieving a User's Weekly Artist Chart 883
40.13. Retrieving Related Artists 884
40.14. Get all posts 885
40.15. Accessing post lists 887
40.16. Filtering a Post List with Specific Tags 887
40.17. Filtering a Post List by URL 888
40.18. Post editing 888
40.19. Method call chaining 888
40.20. Deleting posts 889
40.21. Adding a post 889
40.22. Tags 890
40.23. Bundles 890
40.24. Retrieving public data 891

40.25. Changing the HTTP client of `Zend_Rest_Client` ... 891
40.26. Configuring your HTTP client to keep connections alive ... 892
40.27. Simple Flickr Photo Search ... 892
40.28. Finding a Flickr User's Public Photos by E-Mail Address ... 893
40.29. Retrieving a Group's Pool Photos by Group ID ... 893
40.30. Retrieving Flickr Image Details ... 894
40.31. Querying Links ... 902
40.32. Modifying Links ... 903
40.33. Working With Tags ... 904
40.34. Working With Notes ... 905
40.35. Retrieving Watchlists ... 906
40.36. Sending your first query ... 921
40.37. Refining your query ... 922
40.38. Sending multiple queries with the same `Zend_Service_Technorati` instance ... 922
40.39. Consuming a result set object ... 923
40.40. Seeking a specific result set object ... 923
40.41. Consuming a standalone result object ... 924
40.42. Handling a Query Exception ... 924
40.43. Getting API key daily usage information ... 925
40.44. Cosmos Query ... 926
40.45. Search Query ... 926
40.46. Tag Query ... 927
40.47. DailyCounts Query ... 927
40.48. TopTags Query ... 928
40.49. BlogInfo Query ... 928
40.50. BlogPostTags Query ... 929
40.51. GetInfo Query ... 929
40.52. Iterating result objects from a resultset collection ... 931
40.53. Searching the Web with Yahoo! ... 934
40.54. Finding Images with Yahoo! ... 934
40.55. Finding videos with Yahoo! ... 934
40.56. Finding Local Businesses and Services with Yahoo! ... 935
40.57. Searching Yahoo! News ... 935
40.58. Searching Yahoo! Site Explorer Inbound Links ... 935
40.59. Searching Yahoo! Site Explorer's PageData ... 936
41.1. Counting Page Views ... 943
41.2. New Way: Namespaces Avoid Collisions ... 944
41.3. Old Way: PHP Session Access ... 944
41.4. Session Iteration ... 944
41.5. Accessing Session Data ... 945
41.6. Starting the Global Session ... 945
41.7. Locking Session Namespaces ... 947
41.8. Expiration Examples ... 947
41.9. Namespaced Sessions for Controllers with Automatic Expiration ... 948
41.10. Limiting Session Namespace Access to a Single Instance ... 949
41.11. Modifying Array Data with a Session Namespace ... 950
41.12. Building Arrays Prior to Session Storage ... 950
41.13. Workaround: Reassign a Modified Array ... 950
41.14. Workaround: store array containing reference ... 951
41.15. PHPUnit Testing Code Dependent on Zend_Session ... 952
41.16. Using Zend_Config to Configure Zend_Session ... 954
41.17. Session Fixation ... 958
41.18. Basic Setup ... 961

41.19. Using a Multi-Column Primary Key ... 962
43.1. Application Login TestCase example ... 979
43.2. Testing a UserController ... 986
44.1. Using Zend_Text_Figlet ... 992
46.1. Example of single-language PHP code ... 1004
46.2. Example of multi-lingual PHP code ... 1004
46.3. Example TMX file ... 1009
46.4. Example CSV file ... 1010
46.5. Example CSV file two ... 1010
46.6. Example INI file ... 1011
46.7. Using translation options ... 1011
46.8. Handling languages with adapters ... 1013
46.9. How automatically language detection works ... 1014
46.10. Scanning a directory structure for sources ... 1015
46.11. Directory scanning for languages ... 1016
46.12. Filename scanning for languages ... 1017
46.13. Checking if a text is translatable ... 1019
46.14. Handling languages with adapters ... 1020
47.1. Creating a New URI with `Zend_Uri::factory()` ... 1021
47.2. Manipulating an Existing URI with `Zend_Uri::factory()` ... 1021
47.3. URI Validation with `Zend_Uri::check()` ... 1022
47.4. Getting the Scheme from a `Zend_Uri_*` Object ... 1022
47.5. Getting the Entire URI from a `Zend_Uri_*` Object ... 1022
47.6. Validating a `Zend_Uri_*` Object ... 1023
48.1. Creating a Simple Validation Class ... 1035
48.2. Writing a Validation Class having Dependent Conditions ... 1036
48.3. Validation with Independent Conditions, Multiple Reasons for Failure ... 1038
49.1. Example of compareVersion() method ... 1040
50.1. Basic Usage of Action View Helper ... 1058
50.2. Basic Usage of Partials ... 1059
50.3. Using PartialLoop to Render Iterable Models ... 1061
50.4. Rendering Partials in Other Modules ... 1062
50.5. Basic Usage of Placeholders ... 1062
50.6. Using Placeholders to Aggregate Content ... 1063
50.7. Using Placeholders to Capture Content ... 1064
50.8. Doctype Helper Basic Usage ... 1066
50.9. Retrieving the Doctype ... 1066
50.10. HeadLink Helper Basic Usage ... 1067
50.11. HeadMeta Helper Basic Usage ... 1069
50.12. HeadScript Helper Basic Usage ... 1071
50.13. Capturing Scripts Using the HeadScript Helper ... 1072
50.14. HeadStyle Helper Basic Usage ... 1073
50.15. Capturing Style Declarations Using the HeadStyle Helper ... 1074
50.16. HeadTitle Helper Basic Usage ... 1075
50.17. Flash helper ... 1076
50.18. Customizing the object by passing additional arguments ... 1077
50.19. Registered instance ... 1079
50.20. Within the view ... 1079
50.21. Direct usage ... 1079
50.22. Single parameter ... 1080
50.23. List of parameters ... 1080
50.24. Array of parameters ... 1080
50.25. Change locale dynamically ... 1081

50.26. Change locale statically 1081
50.27. Get the currently set locale 1081
52.1. XML-RPC Method Call 1086
52.2. XML-RPC Method Call with Parameters 1087
52.3. Proxy the Default Namespace 1089
52.4. Proxy Any Namespace 1090
52.5. Handling HTTP Errors 1090
52.6. Handling XML-RPC Faults 1091
52.7. Processing Request to Response 1092

Chapter 1. Introduction to Zend Framework

Overview

Zend Framework (ZF) is an open source framework for developing web applications and services with PHP 5. ZF is implemented using 100% object-oriented code. The component structure of ZF is somewhat unique; each component is designed with few dependencies on other components. This loosely coupled architecture allows developers to use components individually. We often call this a "use-at-will" design.

While they can be used separately, Zend Framework components in the standard library form a powerful and extensible web application framework when combined. ZF offers a robust and performant MVC implementation, a database abstraction that is simple to use, and a forms component that implements HTML 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_Auth and Zend_Acl, provide user authentication and authorization against all common credential stores. Still others implement client libraries to simply access to the most popular web services available. Whatever your application needs are, you're likely to find a Zend Framework component that can be used to dramatically reduce development time with a thoroughly tested foundation.

The principal sponsor of the Zend Framework project is Zend Technologies [http://www.zend.com], 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 that they wish to make available to Zend Framework developers.

Zend Framework could not deliver and support all of these features without the help of the vibrant ZF community. Community members, including contributors, make themselves available on mailing lists [http://framework.zend.com/archives], IRC channels [http://www.zftalk.com], and other forums. Whatever question you have about ZF, the community is always available to address it.

Installation

Zend Framework requires PHP 5.1.4 or higher, although Zend strongly recommended 5.2.3 or higher as there were some critical security and performance enhancements introduces between these two versions. You can find more details in the requirements appendix.

Installing Zend Framework is extremely simple. Once you have downloaded and extracted the framework, you should add the /library folder in the distribution to the beginning of your include path. You may also want to move the library folder to another- possibly shared- location on your filesystem.

- Download the latest stable release. [http://framework.zend.com/download] This version, available in both `.zip` and `.tar.gz` formats, is a good choice for those who are new to Zend Framework.
- Download the latest nightly snapshot. [http://framework.zend.com/download/snapshot] For those who would brave the cutting edge, the nightly snapshots represent the latest progress of Zend Framework development. Snapshots are bundled with documentation either in English only or in all available languages. If you anticipate working with the latest Zend Framework developments, consider using a Subversion (SVN) client.

- Using a Subversion [http://subversion.tigris.org] (SVN) client. Zend Framework is open source software, and the Subversion repository used for its development is publicly available. Consider using SVN to get the Zend Framework if you already use SVN for your application development, want to contribute back to the framework, or need to upgrade your framework version more often than releases occur.

  Exporting [http://svnbook.red-bean.com/nightly/en/svn.ref.svn.c.export.html] is useful if you want to get a particular framework revision without the `.svn` directories as created in a working copy.

  Checking out a working copy [http://svnbook.red-bean.com/nightly/en/svn.ref.svn.c.checkout.html] is good when you might contribute to Zend Framework, and a working copy can be updated any time with `svn update` [http://svnbook.red-bean.com/nightly/en/svn.ref.svn.c.update.html].

  An externals definition [http://svnbook.red-bean.com/nightly/en/svn.advanced.externals.html] is highly convenient for developers already using SVN to manage their application working copies.

  The URL for the trunk of the Zend Framework SVN repository is: http://framework.zend.com/svn/framework/standard/trunk

Once you have a copy of the Zend Framework available, your application needs to be able to access the framework classes. Though there are several ways to achieve this [http://www.php.net/manual/en/configuration.changes.php], your PHP `include_path` [http://www.php.net/manual/en/ini.core.php#ini.include-path] needs to contain the path to the Zend Framework library.

Zend provides a QuickStart [http://framework.zend.com/docs/quickstart] to get you up and running as quickly as possible. This is an excellent way to begin learning about the framework with an emphasis on real world examples that you can built upon.

Since Zend Framework components are loosely coupled, you may use a somewhat unique combination of them in your own applications. The following chapters provide a comprehensive reference to Zend Framework on a component-by- component basis.

Chapter 2. Zend_Acl

Introduction

Zend_Acl 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.

About Resources

Creating a resource in Zend_Acl is very simple. Zend_Acl provides the resource, `Zend_Acl_Resource_Interface`, to facilitate creating resources in an application. A class need only implement this interface, which consists of a single method, `getResourceId()`, so Zend_Acl to recognize the object as a resource. Additionally, `Zend_Acl_Resource` is provided by Zend_Acl as a basic resource implementation for developers to extend as needed.

Zend_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 thse 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_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_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.

About Roles

As with resources, creating a role is also very simple. All roles must implement `Zend_Acl_Role_Interface`. This interface consists of a single method, `getRoleId()`, Additionally, `Zend_Acl_Role` is provided by Zend_Acl as a basic role implementation for developers to extend as needed.

In Zend_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_Acl solves it.

Example 2.1. Multiple Inheritance amoung 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_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`"):

```
$acl = new Zend_Acl();

$acl->addRole(new Zend_Acl_Role('guest'))
    ->addRole(new Zend_Acl_Role('member'))
    ->addRole(new Zend_Acl_Role('admin'));

$parents = array('guest', 'member', 'admin');
$acl->addRole(new Zend_Acl_Role('someUser'), $parents);

$acl->add(new Zend_Acl_Resource('someResource'));

$acl->deny('guest', 'someResource');
$acl->allow('member', 'someResource');

echo $acl->isAllowed('someUser', 'someResource') ? 'allowed' : 'denied';
```

Since there is no rule specifically defined for the "`someUser`" role and "`someResource`", Zend_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_Acl finds that there is a rule specifying that "`member`" is allowed access to "`someResource`".

If Zend_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_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 (ACL)

An 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:

```
$acl = new Zend_Acl();
```

Note

Until a developer specifies an "allow" rule, Zend_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 and backup/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 2.1. 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_Acl_Role` is used, but any object that implements `Zend_Acl_Role_Interface` is acceptable. These groups can be added to the role registry as follows:

```
$acl = new Zend_Acl();

// Add groups to the Role registry using Zend_Acl_Role
// Guest does not inherit access controls
$roleGuest = new Zend_Acl_Role('guest');
$acl->addRole($roleGuest);

// Staff inherits from guest
$acl->addRole(new Zend_Acl_Role('staff'), $roleGuest);

/*
Alternatively, the above could be written:
```

```
$acl->addRole(new Zend_Acl_Role('staff'), 'guest');
*/

// Editor inherits from staff
$acl->addRole(new Zend_Acl_Role('editor'), 'staff');

// Administrator does not inherit access controls
$acl->addRole(new Zend_Acl_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_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_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:

```
$acl = new Zend_Acl();

$roleGuest = new Zend_Acl_Role('guest');
$acl->addRole($roleGuest);
$acl->addRole(new Zend_Acl_Role('staff'), $roleGuest);
$acl->addRole(new Zend_Acl_Role('editor'), 'staff');
$acl->addRole(new Zend_Acl_Role('administrator'));

// Guest may only view content
$acl->allow($roleGuest, null, 'view');

/*
Alternatively, the above could be written:
$acl->allow('guest', null, 'view');
//*/

// Staff inherits view privilege from guest, but also needs additional
// privileges
$acl->allow('staff', null, array('edit', 'submit', 'revise'));

// Editor inherits view, edit, submit, and revise privileges from
// staff, but also needs additional privileges
$acl->allow('editor', null, array('publish', 'archive', 'delete'));

// Administrator inherits nothing, but is allowed all privileges
$acl->allow('administrator');
```

The `null` values in the above `allow()` calls are used to indicate that the allow rules apply to all resources.

Querying the ACL

We now have a flexible ACL that can be used to determine whether requesters have permission to perform functions throughout the web application. Performing queries is quite simple using the `isAllowed()` method:

```
echo $acl->isAllowed('guest', null, 'view') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', null, 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('staff', null, 'revise') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('editor', null, 'view') ?
     "allowed" : "denied";
// allowed because of inheritance from guest

echo $acl->isAllowed('editor', null, 'update') ?
     "allowed" : "denied";
// denied because no allow rule for 'update'

echo $acl->isAllowed('administrator', null, 'view') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator', null, 'update') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges
```

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_Acl allows to you accomplish these refinements in a straightforward and flexible manner.

For the example CMS, it has been determined that whilst the 'staff' group covers the needs of the vast majority of users, there is a need for a new 'marketing' group that requires access to the newsletter and latest news in the CMS. The group is fairly self-sufficient and will have the ability to publish and archive both newsletters and the latest news.

In addition, it has also been requested that the 'staff' group be allowed to view news stories but not to revise the latest news. Finally, it should be impossible for anyone (administrators included) to archive any 'announcement' news stories since they only have a lifespan of 1-2 days.

First we revise the role registry to reflect these changes. We have determined that the 'marketing' group has the same basic permissions as 'staff', so we define 'marketing' in such a way that it inherits permissions from 'staff':

```
// The new marketing group inherits permissions from staff
$acl->addRole(new Zend_Acl_Role('marketing'), 'staff');
```

Next, note that the above access controls refer to specific resources (e.g., "newsletter", "latest news", "announcement news"). Now we add these resources:

```
// Create Resources for the rules

// newsletter
$acl->add(new Zend_Acl_Resource('newsletter'));

// news
$acl->add(new Zend_Acl_Resource('news'));

// latest news
$acl->add(new Zend_Acl_Resource('latest'), 'news');

// announcement news
$acl->add(new Zend_Acl_Resource('announcement'), 'news');
```

Then it is simply a matter of defining these more specific rules on the target areas of the ACL:

```
// Marketing must be able to publish and archive newsletters and the
// latest news
$acl->allow('marketing',
            array('newsletter', 'latest'),
            array('publish', 'archive'));

// Staff (and marketing, by inheritance), are denied permission to
// revise the latest news
$acl->deny('staff', 'latest', 'revise');
```

```
// Everyone (including administrators) are denied permission to
// archive news announcements
$acl->deny(null, 'announcement', 'archive');
```

We can now query the ACL with respect to the latest changes:

```
echo $acl->isAllowed('staff', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', 'latest', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('editor', 'announcement', 'archive') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('administrator', 'announcement', 'archive') ?
     "allowed" : "denied";
// denied
```

Removing Access Controls

To remove one or more access rules from the ACL, simply use the available `removeAllow()` or `removeDeny()` methods. As with `allow()` and `deny()`, you may provide a `null` value to indicate application to all roles, resources, and/or privileges:

```
// Remove the denial of revising latest news to staff (and marketing,
// by inheritance)
$acl->removeDeny('staff', 'latest', 'revise');
```

```
echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// allowed

// Remove the allowance of publishing and archiving newsletters to
// marketing
$acl->removeAllow('marketing',
                  'newsletter',
                  array('publish', 'archive'));

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'archive') ?
     "allowed" : "denied";
// denied
```

Privileges may be modified incrementally as indicated above, but a `null` value for the privileges overrides such incremental changes:

```
// Allow marketing all permissions upon the latest news
$acl->allow('marketing', 'latest');

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'anything') ?
     "allowed" : "denied";
// allowed
```

Advanced Usage

Storing ACL Data for Persistence

Zend_Acl 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_Acl with relative ease and flexibility. Many situations require some form of interactive maintenance of the ACL, and Zend_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_Acl is serializable, ACL objects may be serialized with PHP's serialize() [http://php.net/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_Acl has built-in support for implementing rules based on whatever conditions the developer needs.

Zend_Acl provides support for conditional rules with Zend_Acl_Assert_Interface. In order to use the rule assertion interface, a developer writes a class that implements the assert() method of the interface:

```
class CleanIPAssertion implements Zend_Acl_Assert_Interface
{
    public function assert(Zend_Acl $acl,
                           Zend_Acl_Role_Interface $role = null,
                           Zend_Acl_Resource_Interface $resource = null,
                           $privilege = null)
    {
        return $this->_isCleanIP($_SERVER['REMOTE_ADDR']);
    }

    protected function _isCleanIP($ip)
    {
        // ...
    }
}
```

Once an assertion class is available, the developer must supply an instance of the assertion class when assigning conditional rules. A rule that is created with an assertion only applies when the assertion method returns true.

```
$acl = new Zend_Acl();
$acl->allow(null, null, null, new CleanIPAssertion());
```

The above code creates a conditional allow rule that allows access to all privileges on everything by everyone, except when the requesting IP is "blacklisted." If a request comes in from an IP that is not considered "clean," then the allow rule does not apply. Since the rule applies to all roles, all resources, and all privileges, an "unclean" IP would result in a denial of access. This is a special case, however, and it should be understood that in all other cases (i.e., where a specific role, resource, or privilege is specified for the rule), a failed assertion results in the rule not applying, and other rules would be used to determine whether access is allowed or denied.

The `assert()` method of an assertion object is passed the ACL, role, resource, and privilege to which the authorization query (i.e., `isAllowed()`) applies, in order to provide a context for the assertion class to determine its conditions where needed.

Chapter 3. Zend_Auth

Introduction

Zend_Auth provides an API for authentication and includes concrete authentication adapters for common use case scenarios.

Zend_Auth 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_Auth. For more information about authorization and access control with the Zend Framework, please see Zend_Acl.

Note

The `Zend_Auth` class implements the Singleton pattern - only one instance of the class is available - through its static `getInstance()` method. This means that using the `new` operator and the `clone` keyword will not work with the `Zend_Auth` class; use `Zend_Auth::getInstance()` instead.

Adapters

A Zend_Auth adapter is 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_Auth adapters.

Each Zend_Auth adapter class implements `Zend_Auth_Adapter_Interface`. 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:

```
class MyAuthAdapter implements Zend_Auth_Adapter_Interface
{
    /**
     * Sets username and password for authentication
     *
     * @return void
     */
    public function __construct($username, $password)
    {
        // ...
    }
    /**
     * Performs an authentication attempt
     *
     * @throws Zend_Auth_Adapter_Exception If authentication cannot
     *                                     be performed
     * @return Zend_Auth_Result
     */
    public function authenticate()
    {
        // ...
    }
}
```

As indicated in its docblock, `authenticate()` must return an instance of `Zend_Auth_Result` (or of a class derived from `Zend_Auth_Result`). If for some reason performing an authentication query is impossible, `authenticate()` should throw an exception that derives from `Zend_Auth_Adapter_Exception`.

Results

Zend_Auth adapters return an instance of `Zend_Auth_Result` with `authenticate()` in order to represent the results of an authentication attempt. Adapters populate the `Zend_Auth_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_Auth adapters:

- `isValid()` - returns true if and only if the result represents a successful authentication attempt
- `getCode()` - returns a `Zend_Auth_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:

```
Zend_Auth_Result::SUCCESS
Zend_Auth_Result::FAILURE
Zend_Auth_Result::FAILURE_IDENTITY_NOT_FOUND
Zend_Auth_Result::FAILURE_IDENTITY_AMBIGUOUS
Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID
Zend_Auth_Result::FAILURE_UNCATEGORIZED
```

The following example illustrates how a developer may branch on the result code:

```
// inside of AuthController / loginAction
$result = $this->_auth->authenticate($adapter);

switch ($result->getCode()) {

    case Zend_Auth_Result::FAILURE_IDENTITY_NOT_FOUND:
        /** do stuff for nonexistent identity **/
        break;

    case Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID:
        /** do stuff for invalid credential **/
        break;

    case Zend_Auth_Result::SUCCESS:
        /** do stuff for successful authentication **/
        break;

    default:
        /** do stuff for other failure **/
        break;
}
```

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_Auth` provides persistent storage of the identity from a successful authentication attempt using the PHP session. Upon a successful authentication attempt, `Zend_Auth::authenticate()` stores the identity from the authentication result into persistent storage. Unless configured otherwise, `Zend_Auth` uses a storage class named `Zend_Auth_Storage_Session`, which, in turn, uses Zend_Session. A custom class may instead be used by providing an object that implements `Zend_Auth_Storage_Interface` to `Zend_Auth::setStorage()`.

Note

If automatic persistent storage of the identity is not appropriate for a particular use case, then developers may forgo using the `Zend_Auth` class altogether, instead using an adapter class directly.

Example 3.1. Modifying the Session Namespace

`Zend_Auth_Storage_Session` uses a session namespace of `'Zend_Auth'`. This namespace may be overridden by passing a different value to the constructor of `Zend_Auth_Storage_Session`, and this value is internally passed along to the constructor of `Zend_Session_Namespace`. This should occur before authentication is attempted, since `Zend_Auth::authenticate()` performs the automatic storage of the identity.

```
// Save a reference to the Singleton instance of Zend_Auth
$auth = Zend_Auth::getInstance();

// Use 'someNamespace' instead of 'Zend_Auth'
$auth->setStorage(new Zend_Auth_Storage_Session('someNamespace'));

/**
 * @todo Set up the auth adapter, $authAdapter
 */

// Authenticate, saving the result, and persisting the identity on
// success
$result = $auth->authenticate($authAdapter);
```

Implementing Customized Storage

Sometimes developers may need to use different identity persistence behavior than that provided by `Zend_Auth_Storage_Session`. For such cases developers may simply implement `Zend_Auth_Storage_Interface` and supply an instance of the class to `Zend_Auth::setStorage()`.

Example 3.2. Using a Custom Storage Class

In order to use an identity persistence storage class other than `Zend_Auth_Storage_Session`, a developer implements `Zend_Auth_Storage_Interface`:

```
class MyStorage implements Zend_Auth_Storage_Interface
{
    /**
     * Returns true if and only if storage is empty
     *
     * @throws Zend_Auth_Storage_Exception If it is impossible to
     *                                     determine whether storage
     *                                     is empty
     * @return boolean
     */
    public function isEmpty()
    {
        /**
         * @todo implementation
         */
    }

    /**
     * Returns the contents of storage
     *
     * Behavior is undefined when storage is empty.
     *
     * @throws Zend_Auth_Storage_Exception If reading contents from
     *                                     storage is impossible
     * @return mixed
     */
    public function read()
    {
        /**
         * @todo implementation
         */
    }

    /**
     * Writes $contents to storage
     *
     * @param  mixed $contents
     * @throws Zend_Auth_Storage_Exception If writing $contents to
     *                                     storage is impossible
     * @return void
     */
    public function write($contents)
    {
        /**
         * @todo implementation
         */
    }
    /**
     * Clears contents from storage
     *
     * @throws Zend_Auth_Storage_Exception If clearing contents from
     *                                     storage is impossible
     * @return void
     */
    public function clear()
    {
        /**
         * @todo implementation
         */
    }
}
```

In order to use this custom storage class, `Zend_Auth::setStorage()` is invoked before an authentication query is attempted:

```
// Instruct Zend_Auth to use the custom storage class
Zend_Auth::getInstance()->setStorage(new MyStorage());

/**
 * @todo Set up the auth adapter, $authAdapter
 */

// Authenticate, saving the result, and persisting the identity on
// success
$result = Zend_Auth::getInstance()->authenticate($authAdapter);
```

Using Zend_Auth

There are two provided ways to use Zend_Auth adapters:

1. indirectly, through `Zend_Auth::authenticate()`
2. directly, through the adapter's `authenticate()` method

The following example illustrates how to use a Zend_Auth adapter indirectly, through the use of the `Zend_Auth` class:

```
// Get a reference to the singleton instance of Zend_Auth
$auth = Zend_Auth::getInstance();

// Set up the authentication adapter
$authAdapter = new MyAuthAdapter($username, $password);

// Attempt authentication, saving the result
$result = $auth->authenticate($authAdapter);
```

```
if (!$result->isValid()) {
    // Authentication failed; print the reasons why
    foreach ($result->getMessages() as $message) {
        echo "$message\n";
    }
} else {
    // Authentication succeeded; the identity ($username) is stored
    // in the session
    // $result->getIdentity() === $auth->getIdentity()
    // $result->getIdentity() === $username
}
```

Once authentication has been attempted in a request, as in the above example, it is a simple matter to check whether a successfully authenticated identity exists:

```
$auth = Zend_Auth::getInstance();
if ($auth->hasIdentity()) {
    // Identity exists; get it
    $identity = $auth->getIdentity();
}
```

To remove an identity from persistent storage, simply use the `clearIdentity()` method. This typically would be used for implementing an application "logout" operation:

```
Zend_Auth::getInstance()->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_Auth` 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 `MyAuthAdapter`:

```
// Set up the authentication adapter
$authAdapter = new MyAuthAdapter($username, $password);

// Attempt authentication, saving the result
$result = $authAdapter->authenticate();

if (!$result->isValid()) {
    // Authentication failed; print the reasons why
    foreach ($result->getMessages() as $message) {
        echo "$message\n";
    }
} else {
    // Authentication succeeded
    // $result->getIdentity() === $username
}
```

Database Table Authentication

Introduction

Zend_Auth_Adapter_DbTable provides the ability to authenticate against credentials stored in a database table. Because Zend_Auth_Adapter_DbTable requires an instance of Zend_Db_Adapter_Abstract 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.

Example 3.3. Basic Usage

As explained in the introduction, the `Zend_Auth_Adapter_DbTable` constructor requires an instance of `Zend_Db_Adapter_Abstract` 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:

```
// Create an in-memory SQLite database connection
$dbAdapter = new Zend_Db_Adapter_Pdo_Sqlite(array('dbname' =>
                                                  ':memory:'));

// Build a simple table creation query
$sqlCreate = 'CREATE TABLE [users] ('
           . '[id] INTEGER  NOT NULL PRIMARY KEY, '
           . '[username] VARCHAR(50) UNIQUE NOT NULL, '
           . '[password] VARCHAR(32) NULL, '
           . '[real_name] VARCHAR(150) NULL)';

// Create the authentication credentials table
$dbAdapter->query($sqlCreate);

// Build a query to insert a row for which authentication may succeed
$sqlInsert = "INSERT INTO users (username, password, real_name) "
           . "VALUES ('my_username', 'my_password', 'My Real Name')";

// Insert the data
$dbAdapter->query($sqlInsert);
```

With the database connection and table data available, an instance of `Zend_Auth_Adapter_DbTable` may be created. Configuration option values may be passed to the constructor or deferred as parameters to setter methods after instantiation:

```
// Configure the instance with constructor parameters...
$authAdapter = new Zend_Auth_Adapter_DbTable(
    $dbAdapter,
    'users',
    'username',
    'password'
);

// ...or configure the instance with setter methods
$authAdapter = new Zend_Auth_Adapter_DbTable($dbAdapter);

$authAdapter
    ->setTableName('users')
    ->setIdentityColumn('username')
    ->setCredentialColumn('password')
```

```
;
```

At this point, the authentication adapter instance is ready to accept authentication queries. In order to formulate an authentication query, the input credential values are passed to the adapter prior to calling the `authenticate()` method:

```
// Set the input credential values (e.g., from a login form)
$authAdapter
    ->setIdentity('my_username')
    ->setCredential('my_password')
;

// Perform the authentication query, saving the result
$result = $authAdapter->authenticate();
```

In addition to the availability of the `getIdentity()` method upon the authentication result object, `Zend_Auth_Adapter_DbTable` also supports retrieving the table row upon authentication success:

```
// Print the identity
echo $result->getIdentity() . "\n\n";

// Print the result row
print_r($authAdapter->getResultRowObject());

/* Output:
my_username

Array
(
    [id] => 1
    [username] => my_username
    [password] => my_password
    [real_name] => My Real Name
)
*/
```

Since the table row contains the credential value, it is important to secure the values against unintended access.

Advanced Use: Persisting a DbTable Result Object

By default, `Zend_Auth_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_Auth` an identity object containing other useful information, is solved by using the `getResultRowObject()` method to return a `stdClass` object. The following code snippet illustrates its use:

```
// authenticate with Zend_Auth_Adapter_DbTable
$result = $this->_auth->authenticate($adapter);

if ($result->isValid()) {
    // store the identity as an object where only the username and
    // real_name have been returned
    $storage = $this->_auth->getStorage();
    $storage->write($adapter->getResultRowObject(array(
        'username',
        'real_name',
    )));

    // store the identity as an object where the password column has
    // been omitted
    $storage->write($adapter->getResultRowObject(
        null,
        'password'
    ));

    /* ... */

} else {

    /* ... */

}
```

Advanced Usage By Example

While the primary purpose of Zend_Auth (and consequently Zend_Auth_Adapter_DbTable) is primarily **authentication** and not **authorization**, there exist a few instances and problems that toe the line upon which domain the 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 bit of a disclaimer out of the way, Zend_Auth_Adapter_DbTable has some built in mechanisms that can be leveraged to add additional checks at authentication time to solve some common user problems.

```
// The status field value of an account is not equal to "compromised"
$adapter = new Zend_Auth_Adapter_DbTable(
    $db,
    'users',
    'username',
    'password',
    'MD5(?) AND status != "compromised"'
);

// The active field value of an account is equal to "TRUE"
$adapter = new Zend_Auth_Adapter_DbTable(
    $db,
    'users',
    'username',
    'password',
    'MD5(?) AND active = "TRUE"'
);
```

Another scenario can be the implementation of a salting mechanism. Salting is a term referring to a technique which can highly improve your application's security. It's based on the idea that concatenating a random string to every password makes it impossible to accomplish a successful brute force attack on the database using precomputed hash values from a dictionary.

Therefore we need to modify our table to store our salt string:

```
$sqlAlter = "ALTER TABLE [users] "
          . "ADD COLUMN [password_salt] "
          . "AFTER [password]";

$dbAdapter->query($sqlAlter);
```

Here's a simple way to generate a salt string for every user at registration:

```
for ($i = 0; $i < 50; $i++)
{
    $dynamicSalt .= chr(rand(33, 126));
}
```

And now let's build the adapter:

```
$adapter = new Zend_Auth_Adapter_DbTable(
    $db,
    'users',
    'username',
    'password',
    "MD5(CONCAT('"
    . Zend_Registry::get('staticSalt')
    . "', ?, password_salt))"
);
```

Note

You can improve security even more by using a static salt value hardcoded 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.

Digest Authentication

Introduction

Digest authentication [http://en.wikipedia.org/wiki/Digest_access_authentication] is a method of HTTP authentication that improves upon Basic authentication [http://en.wikipedia.org/wiki/Basic_authentication_scheme] 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_Auth_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_Auth_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.

```
$adapter = new Zend_Auth_Adapter_Digest($filename,
                                        $realm,
                                        $username,
                                        $password);

$result = $adapter->authenticate();
```

```
$identity = $result->getIdentity();

print_r($identity);

/*
Array
(
    [realm] => Some Realm
    [username] => someUser
)
*/
```

HTTP Authentication Adapter

Introduction

`Zend_Auth_Adapter_Http` provides a mostly-compliant implementation of RFC-2617 [http://tools.ietf.org/html/rfc2617], Basic [http://en.wikipedia.org/wiki/Basic_authentication_scheme] and Digest [http://en.wikipedia.org/wiki/Digest_access_authentication] 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_Auth_Adapter_Http` class requires a configuration array passed to its constructor. There are several configuration options available, and some are required:

Table 3.1. Configuration Options

Option Name	Required	Description
`accept_schemes`	Yes	Determines which authentication schemes the adapter will accept from the client. Must be a space-separated list containing `'basic'` and/or `'digest'`.
`realm`	Yes	Sets the authentication realm; usernames should be unique within a given realm.
`digest_domains`	Yes, when `'accept_schemes'` contains `'digest'`	Space-separated list of URIs for which the same authentication information is valid. The URIs need not all point to the same server.
`nonce_timeout`	Yes, when `'accept_schemes'` contains `'digest'`	Sets the number of seconds for which the nonce is valid. See notes below.
`proxy_auth`	No	Disabled by default. Enable to perform Proxy authentication, instead of normal origin server authentication.

Note

The current implementation of the `nonce_timeout` has some interesting side effects. This setting is supposed to determine the valid lifetime of a given nonce, or effectively how long a client's authentication information is accepted. Currently, if it's set to 3600 (for example), it will cause the adapter to prompt the client for new credentials every hour, on the hour. This will be resolved in a future release, once nonce tracking and stale support are implemented.

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_Auth_Adapter_Http` relies on objects implementing `Zend_Auth_Adapter_Http_Resolver_Interface`. 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 the Base64 encoding of the user's password. In Digest authentication, it should be the MD5 hash described above.

There are two equally easy ways to create a File resolver:

```
$path     = 'files/passwd.txt';
$resolver = new Zend_Auth_Adapter_Http_Resolver_File($path);
```

or

```
$path     = 'files/passwd.txt';
$resolver = new Zend_Auth_Adapter_Http_Resolver_File();
$resolver->setFile($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:

```
$config = array(
    'accept_schemes' => 'basic digest',
    'realm'          => 'My Web Site',
    'digest_domains' => '/members_only /my_account',
    'nonce_timeout'  => 3600,
);
```

This array will cause the adapter to accept either Basic or Digest authentication, and will require authenticated access to all the areas of the site under `/members_only` and `/my_account`. The realm value is usually displayed by the browser in the password dialog box. The `nonce_timeout`, of course, behaves as described above.

Next, create the Zend_Auth_Adapter_Http object:

```
$adapter = new Zend_Auth_Adapter_Http($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:

```
$basicResolver = new Zend_Auth_Adapter_Http_Resolver_File();
$basicResolver->setFile('files/basicPasswd.txt');

$digestResolver = new Zend_Auth_Adapter_Http_Resolver_File();
$digestResolver->setFile('files/digestPasswd.txt');

$adapter->setBasicResolver($basicResolver);
$adapter->setDigestResolver($digestResolver);
```

Finally, we perform the authentication. The adapter needs a reference to both the Request and Response objects in order to do its job:

```
assert($request instanceof Zend_Controller_Request_Http);
assert($response instanceof Zend_Controller_Response_Http);

$adapter->setRequest($request);
$adapter->setResponse($response);

$result = $adapter->authenticate();
if (!$result->isValid()) {
    // Bad userame/password, or canceled password prompt
}
```

LDAP Authentication

Introduction

`Zend_Auth_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 [http://www.microsoft.com/windowsserver2003/technologies/directory/activedirectory/] and OpenLDAP [http://www.openldap.org/], but it should also work with other LDAP service providers.

This documentation includes a guide on using `Zend_Auth_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_Auth_Adapter_Ldap` authentication into your application quickly, even if you're not using `Zend_Controller`, the meat of your code should look something like the following:

```
$username = $this->_request->getParam('username');
```

```
$password = $this->_request->getParam('password');

$auth = Zend_Auth::getInstance();

$config = new Zend_Config_Ini('../application/config/config.ini',
                              'production');
$log_path = $config->ldap->log_path;
$options = $config->ldap->toArray();
unset($options['log_path']);

$adapter = new Zend_Auth_Adapter_Ldap($options, $username,
                                      $password);

$result = $auth->authenticate($adapter);

if ($log_path) {
    $messages = $result->getMessages();

    $logger = new Zend_Log();
    $logger->addWriter(new Zend_Log_Writer_Stream($log_path));
    $filter = new Zend_Log_Filter_Priority(Zend_Log::DEBUG);
    $logger->addFilter($filter);

    foreach ($messages as $i => $message) {
        if ($i-- > 1) { // $messages[2] and up are log messages
            $message = str_replace("\n", "\n  ", $message);
            $logger->log("Ldap: $i: $message", Zend_Log::DEBUG);
        }
    }
}
```

Of course the logging code is optional, but it is highly recommended that you use a logger. `Zend_Auth_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_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 `application/config/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_Ini` requires that any values with equals characters (=) will need to be quoted (like the DNs shown below).

```
[production]

ldap.log_path = /tmp/ldap.log

; Typical options for OpenLDAP
ldap.server1.host = s0.foo.net
ldap.server1.accountDomainName = foo.net
ldap.server1.accountDomainNameShort = FOO
ldap.server1.accountCanonicalForm = 3
ldap.server1.username = "CN=user1,DC=foo,DC=net"
ldap.server1.password = pass1
ldap.server1.baseDn = "OU=Sales,DC=foo,DC=net"
ldap.server1.bindRequiresDn = true

; Typical options for Active Directory
ldap.server2.host = dc1.w.net
ldap.server2.useSsl = true
ldap.server2.accountDomainName = w.net
ldap.server2.accountDomainNameShort = W
ldap.server2.accountCanonicalForm = 3
ldap.server2.baseDn = "CN=Users,DC=w,DC=net"
```

The above configuration will instruct `Zend_Auth_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_Auth_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 options. Even if you will be using only one LDAP server, the options must still be within another array.

Below is `print_r()` [http://php.net/print_r] output of an example options parameter containing two sets of server options for LDAP servers `s0.foo.net` and `dc1.w.net` (same options as the above INI representation):

```
Array
(
    [server2] => Array
        (
            [host] => dc1.w.net
            [useSsl] => 1
            [accountDomainName] => w.net
            [accountDomainNameShort] => W
            [accountCanonicalForm] => 3
            [baseDn] => CN=Users,DC=w,DC=net
        )

    [server1] => Array
        (
            [host] => s0.foo.net
            [accountDomainName] => foo.net
            [accountDomainNameShort] => FOO
            [accountCanonicalForm] => 3
            [username] => CN=user1,DC=foo,DC=net
            [password] => pass1
            [baseDn] => OU=Sales,DC=foo,DC=net
            [bindRequiresDn] => 1
        )

)
```

The information provided in each set of options above is different mainly because AD does not require a username be in DN form when binding (see the `bindRequiresDn` option in the *Server Options* section below), which means we can omit the a number of options associated with retrieving the DN for a username being authenticated.

What is a DN?

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`, 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.

The Gory Details - What exactly 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` instance and calls the `Zend_Ldap::bind()` method with the username and password being authenticated. The `Zend_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 it does not match either of the server's domain names (*foo.net* or *FOO*), a special exception is thrown and caught by `Zend_Auth_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` proceeds to try to bind with the supplied credentials. If the bind is not successful, `Zend_Ldap` throws a `Zend_Ldap_Exception` which is caught by `Zend_Auth_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_Auth_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_Auth_Adapter_Ldap* consists of the following options, which are passed, largely unmodified, to `Zend_Ldap::setOptions()`:

Table 3.2. 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.
useSsl	If `true`, this value indicates that the LDAP client should use SSL / TLS 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. 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 - a 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` 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 **acountFilterFormat** used when searching for accounts. See the **accountFilterFormat** option.
baseDn	The DN under which all accounts being authenticated are located. This option is required. If you are uncertain about the correct **baseDn** value, it should be sufficient to derive it from the user's DNS domain using *DC=* components. For example, if the user's principal name is *alice@foo.net*, a **baseDn** of *DC=foo,DC=net* should work. A more precise location (e.g., *OU=Sales,DC=foo,DC=net*) will be more efficient, however.
accountCanonicalForm	A value of 2, 3 or 4 indicating the form to which account names should be canonicalized after successful authentication. Values are as follows: 2 for traditional username style names (e.g., *alice*), 3 for backslash-style names (e.g., *FOO\alice*) or 4 for principal style usernames (e.g., *alice@foo.net*). The default value is 4 (e.g., *alice@foo.net*). For example, with a value of 3, the identity returned by `Zend_Auth_Result::getIdentity()` (and `Zend_Auth::getIdentity()`, if `Zend_Auth` was used) will always be *FOO\alice*, regardless of what form Alice supplied, whether it be *alice*, *alice@foo.net*, *FOO\alice*, *FoO\aLicE*, *foo.net\alice*, etc. See the *Account Name Canonicalization* section in the `Zend_Ldap` documentation for details. Note that when using multiple sets of server options it is recommended, but not required, that the same **accountCanonicalForm** be used with all server options so that the resulting usernames are always canonicalized to the same form (e.g., if you canonicalize to *EXAMPLE\username* with an AD server but to *username@example.com* with an OpenLDAP server, that may be awkward for the application's high-level logic).

Name	Description
accountDomainName	The FQDN domain name for which the target LDAP server is an authority (e.g., `example.com`). This option is used to canonicalize names so that the username supplied by the user can be converted as necessary for binding. It is also used to determine if the server is an authority for the supplied username (e.g., if **accountDomainName** is *foo.net* and the user supplies *bob@bar.net*, the server will not be queried, and a failure will result). This option is not required, but if it is not supplied, usernames in principal name form (e.g., *alice@foo.net*) are not supported. It is strongly recommended that you supply this option, as there are many use-cases that require generating the principal name form.
accountDomainNameShort	The 'short' domain for which the target LDAP server is an authority (e.g., *FOO*). Note that there is a 1:1 mapping between the **accountDomainName** and **accountDomainNameShort**. This option should be used to specify the NetBIOS domain name for Windows networks but may also be used by non-AD servers (e.g., for consistency when multiple sets of server options with the backslash style **accountCanonicalForm**). This option is not required but if it is not supplied, usernames in backslash form (e.g., *FOO\alice*) are not supported.
accountFilterFormat	The LDAP search filter used to search for accounts. This string is a `printf()` [http://php.net/printf]-style expression that must contain one '`%s`' to accomodate 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))`'. For example, if for some reason you wanted to use `bindRequiresDn = true` with AD you would need to set `accountFilterFormat = '(&(objectClass=user)(sAMAccountName=%s))`'.

Note

If you enable `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_Auth_Adapter_Ldap` collects debugging information within its `authenticate()` method. This information is stored in the `Zend_Auth_Result` object as messages. The array returned by `Zend_Auth_Result::getMessages()` is described as follows:

Table 3.3. 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 3.4. Options for Active Directory

Name	Additional Notes
host	As with all servers, this option is required.
useSsl	For the sake of security, this should be `true` if the server has the necessary certificate installed.
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 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_Auth_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 3.5. Options for OpenLDAP

Name	Additional Notes
host	As with all servers, this option is required.
useSsl	For the sake of security, this should be `true` if the server has the necessary certificate installed.
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*).

Open ID Authentication

Introduction

`Zend_Auth_Adapter_OpenId` allows authenticate user using remote OpenID server. Such authentication process assumes that user submits to web application only their OpenID identity. Then they are redirected to their OpenID providers to prove the identity ownership using password or some other method. This password is never known to local web application.

The OpenID identity is just an HTTP URL that points to some web page with suitable information about the user and special tags which describes which server to use and which identity to submit there. You can read more about OpenID at OpenID official site [http://www.openid.net/].

The `Zend_Auth_Adapter_OpenId` class is a wrapper on top of `Zend_OpenId_Consumer` component which implements the OpenID authentication protocol itself.

Note

`Zend_OpenId` takes advantage of the GMP extension [http://php.net/gmp], where available. Consider enabling the GMP extension for better performance when using `Zend_Auth_Adapter_OpenId`.

Specifics

As any other `Zend_Auth` adapter the `Zend_Auth_Adapter_OpenId` class implements `Zend_Auth_Adapter_Interface`, which defines one method - `authenticate()`. This method performs the authentication itself, but the object must be prepared prior to calling it. Such adapter preparation includes setting up OpenID identity and some other `Zend_OpenId` specific options.

However in opposite to other `Zend_Auth` adapters it performs authentication on external server and it is done in two separate HTTP requests. So the `Zend_Auth_Adapter_OpenId::authenticate()` must be called twice. First time the method won't return, but will redirect user to their OpenID server. Then after authentication on server they will be redirected back and the script for this second request must call `Zend_Auth_Adapter_OpenId::authenticate()` again to verify signature which come with redirected request from the server and complete the authentication process. This time the method will return `Zend_Auth_Result` object as expected.

The following example shows the usage of `Zend_Auth_Adapter_OpenId`. As was said before the `Zend_Auth_Adapter_OpenId::authenticate()` is called two times. First time - after submitting of HTML form when `$_POST['openid_action']` is set to `"login"`, and the second time after HTTP redirection from OpenID server when `$_GET['openid_mode']` or `$_POST['openid_mode']` is set.

```
<?php
$status = "";
$auth = Zend_Auth::getInstance();
if ((isset($_POST['openid_action']) &&
     $_POST['openid_action'] == "login" &&
     !empty($_POST['openid_identifier'])) ||
    isset($_GET['openid_mode']) ||
    isset($_POST['openid_mode'])) {
    $result = $auth->authenticate(
        new Zend_Auth_Adapter_OpenId(@$_POST['openid_identifier']));
    if ($result->isValid()) {
        $status = "You are logged in as "
                . $auth->getIdentity()
                . "<br>\n";
    } else {
        $auth->clearIdentity();
        foreach ($result->getMessages() as $message) {
            $status .= "$message<br>\n";
        }
    }
} else if ($auth->hasIdentity()) {
    if (isset($_POST['openid_action']) &&
        $_POST['openid_action'] == "logout") {
        $auth->clearIdentity();
    } else {
        $status = "You are logged in as "
                . $auth->getIdentity()
                . "<br>\n";
    }
}
?>
```

```
<html><body>
<?php echo htmlspecialchars($status);?>
<form method="post"><fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier" value="">
<input type="submit" name="openid_action" value="login">
<input type="submit" name="openid_action" value="logout">
</fieldset></form></body></html>
*/
```

It is allowed customize the OpenID authentication process with: receiving redirection from the OpenID server on separate page, specifying the "root" of web site. In this case, using custom `Zend_OpenId_Consumer_Storage` or custom `Zend_Controller_Response`. It is also possible to use Simple Registration Extension to retrieve information about user from the OpenID server. All these possibilities described in more details in `Zend_OpenId_Consumer` reference.

Chapter 4. Zend_Cache

Introduction

`Zend_Cache` provides a generic way to cache any data.

Caching in Zend Framework is operated by frontends while cache records are stored through backend adapters (`File`, `Sqlite`, `Memcache`...) through a flexible system of IDs and tags. Using those, it is easy to delete specific types of records afterwards (for example: "delete all cache records marked with a given tag").

The core of the module (`Zend_Cache_Core`) is generic, flexible and configurable. Yet, for your specific needs there are cache frontends that extend `Zend_Cache_Core` for convenience: `Output`, `File`, `Function` and `Class`.

Example 4.1. Getting a frontend with `Zend_Cache::factory()`

`Zend_Cache::factory()` instantiates correct objects and ties them together. In this first example, we will use `Core` frontend together with `File` backend.

```
$frontendOptions = array(
   'lifetime' => 7200, // cache lifetime of 2 hours
   'automatic_serialization' => true
);

$backendOptions = array(
    'cache_dir' => './tmp/' // Directory where to put the cache files
);

// getting a Zend_Cache_Core object
$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);
```

Frontends and Backends consisting of multiple words

Some frontends and backends are named using multiple words, such as 'ZendPlatform'. When specifying them to the factory, separate them using a word separator, such as a space (' '), hyphen ('-'), or period ('.').

Example 4.2. Caching a database query result

Now that we have a frontend, we can cache any type of data (we turned on serialization). For example, we can cache a result from a very expensive database query. After it is cached, there is no need to even connect to the database; records are fetched from cache and unserialized.

```
// $cache initialized in previous example

// see if a cache already exists:
if(!$result = $cache->load('myresult')) {

    // cache miss; connect to the database

    $db = Zend_Db::factory( [...] );

    $result = $db->fetchAll('SELECT * FROM huge_table');

    $cache->save($result, 'myresult');

} else {

    // cache hit! shout so that we know
    echo "This one is from cache!\n\n";

}

print_r($result);
```

Example 4.3. Caching output with `Zend_Cache` output frontend

We 'mark up' sections in which we want to cache output by adding some conditional logic, encapsulating the section within `start()` and `end()` methods (this resembles the first example and is the core strategy for caching).

Inside, output your data as usual - all output will be cached when execution hits the `end()` method. On the next run, the whole section will be skipped in favor of fetching data from cache (as long as the cache record is valid).

```
$frontendOptions = array(
   'lifetime' => 30,                   // cache lifetime of 30 seconds
   'automatic_serialization' => false  // this is the default anyways
);

$backendOptions = array('cache_dir' => './tmp/');

$cache = Zend_Cache::factory('Output',
                             'File',
                             $frontendOptions,
                             $backendOptions);

// we pass a unique identifier to the start() method
if(!$cache->start('mypage')) {
    // output as usual:

    echo 'Hello world! ';
    echo 'This is cached ('.time().') ';

    $cache->end(); // the output is saved and sent to the browser
}

echo 'This is never cached ('.time().').';
```

Notice that we output the result of `time()` twice; this is something dynamic for demonstration purposes. Try running this and then refreshing several times; you will notice that the first number doesn't change while second changes as time passes. That is because the first number was output in the cached section and is saved among other output. After half a minute (we've set lifetime to 30 seconds) the numbers should match again because the cache record expired -- only to be cached again. You should try this in your browser or console.

Note

When using Zend_Cache, pay attention to the important cache identifier (passed to `save()` and `start()`). It must be unique for every resource you cache, otherwise unrelated cache records may wipe each other or, even worse, be displayed in place of the other.

The theory of caching

There are three key concepts in Zend_Cache. One is the unique identifier (a string) that is used to identify cache records. The second one is the `'lifetime'` directive as seen in the examples; it defines for how long the cached resource is considered 'fresh'. The third key concept is conditional execution so that parts of your code can be skipped entirely, boosting performance. The main frontend function (eg. `Zend_Cache_Core::get()`) is always designed to return false for a cache miss if that makes sense for the nature of a frontend. That enables end-users to wrap parts of the code they would like to cache (and skip) in `if(){ ... }` statements where the condition is a Zend_Cache method itself. On the end if these blocks you must save what you've generated, however (eg. `Zend_Cache_Core::save()`).

Note

The conditional execution design of your generating code is not necessary in some frontends (`Function`, for an example) when the whole logic is implemented inside the frontend.

Note

'Cache hit' is a term for a condition when a cache record is found, is valid and is 'fresh' (in other words hasn't expired yet). 'Cache miss' is everything else. When a cache miss happens, you must generate your data (as you would normally do) and have it cached. When you have a cache hit, on the other hand, the backend automatically fetches the record from cache transparently.

The `Zend_Cache` factory method

A good way to build a usable instance of a `Zend_Cache` Frontend is given in the following example :

```
// We choose a backend (for example 'File' or 'Sqlite'...)
$backendName = '[...]';

// We choose a frontend (for example 'Core', 'Output', 'Page'...)
$frontendName = '[...]';

// We set an array of options for the choosen frontend
$frontendOptions = array([...]);

// We set an array of options for the choosen backend
$backendOptions = array([...]);

// We create an instance of Zend_Cache
// (of course, the two last arguments are optional)
$cache = Zend_Cache::factory($frontendName, $backendName, $frontendOptions, $backe
```

In the following examples we will assume that the `$cache` variable holds a valid, instantiated frontend as shown and that you understand how to pass parameters to your chosen backends.

Note

Always use `Zend_Cache::factory()` to get frontend instances. Instantiating frontends and backends yourself will not work as expected.

Tagging records

Tags are a way to categorize cache records. When you save a cache with the `save()` method, you can set an array of tags to apply for this record. Then you will be able to clean all cache records tagged with a given tag (or tags):

```
$cache->save($huge_data, 'myUniqueID', array('tagA', 'tagB', 'tagC'));
```

Note

note than the `save()` method accepts an optional fourth argument : `$specificLifetime` (if != false, it sets a specific lifetime for this particular cache record)

Cleaning the cache

To remove/invalidate in particular cache id, you can use the `remove()` method :

```
$cache->remove('idToRemove');
```

To remove/invalidate several cache ids in one operation, you can use the `clean()` method. For example to remove all cache records :

```
// clean all records
$cache->clean(Zend_Cache::CLEANING_MODE_ALL);

// clean only outdated
$cache->clean(Zend_Cache::CLEANING_MODE_OLD);
```

If you want to remove cache entries matching the tags 'tagA' and 'tagC':

```
$cache->clean(Zend_Cache::CLEANING_MODE_MATCHING_TAG, array('tagA', 'tagC'));
```

Available cleaning modes are: `CLEANING_MODE_ALL`, `CLEANING_MODE_OLD`, `CLEANING_MODE_MATCHING_TAG` and `CLEANING_MODE_NOT_MATCHING_TAG`. The latter are, as their names suggest, combined with an array of tags in cleaning operations.

Zend_Cache frontends

Zend_Cache_Core

Introduction

`Zend_Cache_Core` is a special frontend because it is the core of the module. It is a generic cache frontend and is extended by other classes.

Note

All frontends inherit from `Zend_Cache_Core` so that its methods and options (described below) would also be available in other frontends, therefore they won't be documented there.

Available options

These options are passed to the factory method as demonstrated in previous examples.

Table 4.1. Core frontend options

Option	Data Type	Default Value	Description
`caching`	`boolean`	`true`	enable / disable caching (can be very useful for the debug of cached scripts)
`cache_id_prefix`	`string`	`null`	A prefix for all cache ids, if set to `null`, no cache id prefix will be used. The cache id prefix essentially creates a namespace in the cache, allowing multiple applications or websites to use a shared cache. Each application or website can use a different cache id prefix so specific cache ids can be used more than once.
`lifetime`	`int`	`3600`	cache lifetime (in seconds), if set to `null`, the cache is valid forever.
`logging`	`boolean`	`false`	if set to true, logging through `Zend_Log` is activated (but the system is slower)
`write_control`	`boolean`	`true`	Enable / disable write control (the cache is read just after writing to detect corrupt entries), enabling write_control will lightly slow the cache writing but not the cache reading (it can detect some corrupt cache files but it's not a perfect control)
`automatic_serialization`	`boolean`	`false`	Enable / disable automatic serialization, it can be used to save directly datas which aren't strings (but it's slower)
`automatic_cleaning_factor`	`int`	`10`	Disable / Tune the automatic cleaning process (garbage collector): 0 means no automatic cache cleaning, 1 means systematic cache cleaning and x > 1 means automatic random cleaning 1 times in x write operations.
`ignore_user_abort`	`boolean`	`false`	if set to true, the core will set the ignore_user_abort PHP flag inside the save() method to avoid cache corruptions in some cases

Examples

An example is given in the manual at the very beginning.

If you store only strings into cache (because with "automatic_serialization" option, it's possible to store some booleans), you can use a more compact construction like:

```
// we assume you already have $cache

$id = 'myBigLoop'; // cache id of "what we want to cache"

if (!($data = $cache->load($id))) {
    // cache miss

    $data = '';
    for ($i = 0; $i < 10000; $i++) {
        $data = $data . $i;
    }

    $cache->save($data);

}

// [...] do something with $data (echo it, pass it on etc.)
```

If you want to cache multiple blocks or data instances, the idea is the same:

```
// make sure you use unique identifiers:
$id1 = 'foo';
$id2 = 'bar';

// block 1
if (!($data = $cache->load($id1))) {
    // cache missed

    $data = '';
    for ($i=0;$i<10000;$i++) {
        $data = $data . $i;
    }

    $cache->save($data);

}
echo($data);

// this isn't affected by caching
echo('NEVER CACHED! ');

// block 2
if (!($data = $cache->load($id2))) {
    // cache missed

    $data = '';
    for ($i=0;$i<10000;$i++) {
        $data = $data . '!';
    }

    $cache->save($data);

}
echo($data);
```

If you want to cache special values (boolean with "automatic_serialization" option) or empty strings you can't use the compact construction given above. You have to test formally the cache record.

```
// the compact construction
// (not good if you cache empty strings and/or booleans)
if (!($data = $cache->load($id))) {

    // cache missed

    // [...] we make $data

    $cache->save($data);

}

// we do something with $data

// [...]

// the complete construction (works in any case)
if (!($cache->test($id))) {

    // cache missed

    // [...] we make $data

    $cache->save($data);

} else {

    // cache hit

    $data = $cache->load($id);

}

// we do something with $data
```

Zend_Cache_Frontend_Output

Introduction

`Zend_Cache_Frontend_Output` is an output-capturing frontend. It utilizes output buffering in PHP to capture everything between its `start()` and `end()` methods.

Available options

This frontend doesn't have any specific options other than those of `Zend_Cache_Core`.

Examples

An example is given in the manual at the very beginning. Here it is with minor changes:

```
// if it is a cache miss, output buffering is triggered
if (!($cache->start('mypage'))) {

    // output everything as usual
    echo 'Hello world! ';
    echo 'This is cached ('.time().') ';

    $cache->end(); // output buffering ends

}

echo 'This is never cached ('.time().').';
```

Using this form it is fairly easy to set up output caching in your already working project with little or no code refactoring.

Zend_Cache_Frontend_Function

Introduction

`Zend_Cache_Frontend_Function` caches the results of function calls. It has a single main method named `call()` which takes a function name and parameters for the call in an array.

Available options

Table 4.2. Function frontend options

Option	Data Type	Default Value	Description
`cache_by_default`	`boolean`	`true`	if true, function calls will be cached by default
`cached_functions`	`array`		function names which will always be cached
`non_cached_functions`	`array`		function names which must never be cached

Examples

Using the `call()` function is the same as using `call_user_func_array()` in PHP:

```
$cache->call('veryExpensiveFunc', $params);

// $params is an array
// For example to call veryExpensiveFunc(1, 'foo', 'bar') with
// caching, you can use
// $cache->call('veryExpensiveFunc', array(1, 'foo', 'bar'))
```

`Zend_Cache_Frontend_Function` is smart enough to cache both the return value of the function and its internal output.

Note

You can pass any built in or user defined function with the exception of `array()`, `echo()`, `empty()`, `eval()`, `exit()`, `isset()`, `list()`, `print()` and `unset()`.

Zend_Cache_Frontend_Class

Introduction

`Zend_Cache_Frontend_Class` is different from `Zend_Cache_Frontend_Function` because it allows caching of object and static method calls.

Available options

Table 4.3. Class frontend options

Option	Data Type	Default Value	Description
`cached_entity` (required)	`mixed`		if set to a class name, we will cache an abstract class and will use only static calls; if set to an object, we will cache this object methods
`cache_by_default`	`boolean`	`true`	if true, calls will be cached by default
`cached_methods`	`array`		method names which will always be cached
`non_cached_methods`	`array`		method names which must never be cached

Examples

For example, to cache static calls :

```
class Test {

    // Static method
    public static function foobar($param1, $param2) {
        echo "foobar_output($param1, $param2)";
        return "foobar_return($param1, $param2)";
    }

}

// [...]
$frontendOptions = array(
    'cached_entity' => 'Test' // The name of the class
);
// [...]

// The cached call
$result = $cache->foobar('1', '2');
```

To cache classic method calls :

```
class Test {

    private $_string = 'hello !';

    public function foobar2($param1, $param2) {
        echo($this->_string);
        echo "foobar2_output($param1, $param2)";
        return "foobar2_return($param1, $param2)";
    }

}

// [...]
$frontendOptions = array(
    'cached_entity' => new Test() // An instance of the class
);
// [...]

// The cached call
$result = $cache->foobar2('1', '2');
```

Zend_Cache_Frontend_File

Introduction

`Zend_Cache_Frontend_File` is a frontend driven by the modification time of a "master file". It's really interesting for examples in configuration or templates issues.

For instance, you have an XML configuration file which is parsed by a function which returns a "config object" (like with `Zend_Config`). With `Zend_Cache_Frontend_File`, you can store the "config object" into cache (to avoid the parsing of the XML config file at each time) but with a sort of strong dependency on the "master file". So, if the XML config file is modified, the cache is immediately invalidated.

Available options

Table 4.4. File frontend options

Option	Data Type	Default Value	Description
`master_file (mandatory)`	`string`		the complete path and name of the master file

Examples

Use of this frontend is the same than of `Zend_Cache_Core`. There is no need of a specific example - the only thing to do is to define the `master_file` when using the factory.

Zend_Cache_Frontend_Page

Introduction

`Zend_Cache_Frontend_Page` is like `Zend_Cache_Frontend_Output` but designed for a complete page. It's impossible to use `Zend_Cache_Frontend_Page` for caching only a single block.

On the other hand, the "cache id" is calculated automatically with `$_SERVER['REQUEST_URI']` and (depending on options) `$_GET`, `$_POST`, `$_SESSION`, `$_COOKIE`, `$_FILES`. More over, you have only one method to call (`start()`) because the `end()` call is fully automatic when the page is ended.

For the moment, it's not implemented but we plan to add a HTTP conditional system to save bandwidth (the system will send a HTTP 304 Not Modified if the cache is hit and if the browser has already the good version).

Available options (for this frontend in Zend_Cache factory)

Table 4.5. Page frontend options

Option	Data Type	Default Value	Description
`http_conditional`	`boolean`	`false`	use the http_conditional system (not implemented for the moment)
`debug_header`	`boolean`	`false`	if true, a debug text is added before each cached pages
`default_options`	`array`	`array(...see below...)`	an associative array of default options : • `(boolean, true by default) cache` : cache is on if true • `(boolean, false by default) cache_with_get_variables` : if true, cache is still on even if there are some variables in `$_GET` array • `(boolean, false by default) cache_with_post_variables` : if true, cache is still on even if there are some variables in `$_POST` array • `(boolean, false by default) cache_with_session_variables` : if true, cache is still on even if there are some variables in `$_SESSION` array • `(boolean, false by default) cache_with_files_variables` : if true, cache is still on even if there are some variables in `$_FILES` array • `(boolean, false by default) cache_with_cookie_variables` : if true, cache is still on even if there are some variables in `$_COOKIE` array • `(boolean, true by default) make_id_with_get_variables` : if true, the cache id will be dependent of the content of the `$_GET` array • `(boolean, true by default) make_id_with_post_variables` : if true, the cache id will be dependent of the content of the `$_POST` array • `(boolean, true by default) make_id_with_session_variables` : if true, the cache id will be dependent of the content of the `$_SESSION` array • `(boolean, true by default) make_id_with_files_variables` : if true, the cache id will be dependent of the content of the `$_FILES` array • `(boolean, true by default) make_id_with_cookie_variables` : if true, the cache id will be dependent of the content of the `$_COOKIE` array

Option	Data Type	Default Value	Description
`regexps`	`array`	`array()`	an associative array to set options only for some REQUEST_URI, keys are (PCRE) regexps, values are associative arrays with specific options to set if the regexp matchs on $_SERVER['REQUEST_URI'] (see default_options for the list of available options) ; if several regexps match the $_SERVER['REQUEST_URI'], only the last one will be used
`memorize_headers`	`array`	`array()`	an array of strings corresponding to some HTTP headers name. Listed headers will be stored with cache datas and "replayed" when the cache is hit

Examples

Use of Zend_Cache_Frontend_Page is really trivial :

```
// [...] // require, configuration and factory

$cache->start();
// if the cache is hit, the result is sent to the browser and the script stop here

// rest of the page ...
```

a more complex example which shows a way to get a centralized cache management in a bootstrap file (for using with Zend_Controller for example)

```
/*
 * You should avoid putting too many lines before the cache section.
 * For example, for optimal performances, "require_once" or
 * "Zend_Loader::loadClass" should be after the cache section.
 */

$frontendOptions = array(
   'lifetime' => 7200,
   'debug_header' => true, // for debugging
   'regexps' => array(
       // cache the whole IndexController
       '^/$' => array('cache' => true),

       // cache the whole IndexController
       '^/index/' => array('cache' => true),

       // we don't cache the ArticleController...
       '^/article/' => array('cache' => false),

       // ... but we cache the "view" action of this ArticleController
       '^/article/view/' => array(
           'cache' => true,
            // and we cache even there are some variables in $_POST
            'cache_with_post_variables' => true,

            // but the cache will be dependent on the $_POST array
            'make_id_with_post_variables' => true
        )
    )
);

$backendOptions = array(
    'cache_dir' => '/tmp/'
);

// getting a Zend_Cache_Frontend_Page object
$cache = Zend_Cache::factory('Page',
                             'File',
                             $frontendOptions,
                             $backendOptions);

$cache->start();
// if the cache is hit, the result is sent to the browser and the
// script stop here

// [...] the end of the bootstrap file
// these lines won't be executed if the cache is hit
```

The specific cancel method

Because of design issues, in some cases (for example when using non HTTP/200 return codes), you could need to cancel the current cache process. So we introduce for this particular frontend, the cancel() method.

```
// [...] // require, configuration and factory

$cache->start();

// [...]

if ($someTest) {
    $cache->cancel();
    // [...]
}

// [...]
```

Zend_Cache backends

Zend_Cache_Backend_File

This backends stores cache records into files (in a choosen directory).

Available options are :

Table 4.6. File backend options

Option	Data Type	Default Value	Description
`cache_dir`	`string`	`'/tmp/'`	Directory where to store cache files
`file_locking`	`boolean`	`true`	Enable / disable file_locking : Can avoid cache corruption under bad circumstances but it doesn't help on multithread webservers or on NFS filesystems...
`read_control`	`boolean`	`true`	Enable / disable read control : if enabled, a control key is embedded in the cache file and this key is compared with the one calculated after the reading.
`read_control_type`	`string`	`'crc32'`	Type of read control (only if read control is enabled). Available values are : 'md5' (best but slowest), 'crc32' (lightly less safe but faster, better choice), 'adler32' (new choice, faster than crc32), 'strlen' for a length only test (fastest).
`hashed_directory_level`	`int`	`0`	Hashed directory structure level : 0 means "no hashed directory structure", 1 means "one level of directory", 2 means "two levels"... This option can speed up the cache only when you have many thousands of cache files. Only specific benchs can help you to choose the perfect value for you. Maybe, 1 or 2 is a good start.
`hashed_directory_umask`	`int`	`0700`	Umask for the hashed directory structure
`file_name_prefix`	`string`	`'zend_cache'`	prefix for cache files ; be really careful with this option because a too generic value in a system cache dir (like /tmp) can cause disasters when cleaning the cache
`cache_file_umask`	`int`	`0700`	umask for cache files
`metatadatas_array_max_size`	`int`	`100`	internal max size for the metadatas array (don't change this value unless you know what you are doing)

Zend_Cache_Backend_Sqlite

This backends stores cache records into a SQLite database.

Available options are :

Table 4.7. Sqlite backend options

Option	Data Type	Default Value	Description
`cache_db_complete_path (mandatory)`	`string`	`null`	The complete path (filename included) of the SQLite database
`automatic_vacuum_factor`	`int`	`10`	Disable / Tune the automatic vacuum process. The automatic vacuum process defragment the database file (and make it smaller) when a clean() or delete() is called : 0 means no automatic vacuum ; 1 means systematic vacuum (when delete() or clean() methods are called) ; x (integer) > 1 => automatic vacuum randomly 1 times on x clean() or delete().

Zend_Cache_Backend_Memcached

This backends stores cache records into a memcached server. memcached [http://www.danga.com/memcached/] is a high-performance, distributed memory object caching system. To use this backend, you need a memcached daemon and the memcache PECL extension [http://pecl.php.net/package/memcache].

Be careful : with this backend, "tags" are not supported for the moment as the "doNotTestCacheValidity=true" argument.

Available options are :

Table 4.8. Memcached backend options

Option	Data Type	Default Value	Description
`servers`	`array`	`array(array('host' => 'localhost','port' => 11211, 'persistent' => true))`	An array of memcached servers ; each memcached server is described by an associative array : 'host' => (string) : the name of the memcached server, 'port' => (int) : the port of the memcached server, 'persistent' => (bool) : use or not persistent connections to this memcached server
`compression`	`boolean`	`false`	true if you want to use on-the-fly compression

Zend_Cache_Backend_Apc

This backends stores cache records in shared memory through the APC [http://pecl.php.net/package/APC] (Alternative PHP Cache) extension (which is of course need for using this backend).

Be careful : with this backend, "tags" are not supported for the moment as the "doNotTestCacheValidity=true" argument.

There is no option for this backend.

Zend_Cache_Backend_Xcache

This backends stores cache records in shared memory through the XCache [http://xcache.lighttpd.net/] extension (which is of course need for using this backend).

Be careful : with this backend, "tags" are not supported for the moment as the "doNotTestCacheValidity=true" argument.

Available options are :

Table 4.9. Xcache backend options

Option	Data Type	Default Value	Description
`user`	`string`	`null`	xcache.admin.user, necessary for the clean() method
`password`	`string`	`null`	xcache.admin.pass (in clear form, not MD5), necessary for the clean() method

Zend_Cache_Backend_ZendPlatform

This backend uses content caching API of the Zend Platform [http://www.zend.com/products/platform] product. Naturally, to use this backend you need to have Zend Platform installed.

This backend supports tags, but does not support `CLEANING_MODE_NOT_MATCHING_TAG` cleaning mode.

Specify this backend using a word separator -- '-', '.', ' ', or '_' -- between the words 'Zend' and 'Platform' when using the `Zend_Cache::factory()` method:

```
$cache = Zend_Cache::factory('Core', 'Zend Platform');
```

There are no options for this backend.

Chapter 5. Zend_Captcha

Introduction

CAPTCHA [http://en.wikipedia.org/wiki/Captcha] stands for "Completely Automated 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 desire to prevent spam submissions.

Captchas can take variety of forms, including asking logic questions, presenting skewed fonts, and presenting images and asking how they relate. Zend_Captcha aims to provide a variety of backends that may be utilized either standalone or in conjunction with `Zend_Form`.

Captcha Operation

All concrete CAPTCHA objects implement `Zend_Captcha_Adapter`, which looks like the following:

```
interface Zend_Captcha_Adapter extends Zend_Validate_Interface
{
    public function generate();

    public function render(Zend_View $view, $element = null);

    public function setName($name);

    public function getName();

    public function getDecorator();

    // Additionally, to satisfy Zend_Validate_Interface:
    public function isValid($value);

    public function getMessages();

    public function getErrors();
}
```

The name mutator and accessor are used to specify and retrieve the captcha identifier. `getDecorator()` can be used to specify a Zend_Form decorator either by name or returning an actual decorator object. The true keys to usage, however, lie in `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 in image, a figlet, a logic problem, etc.

A typical use case might look like the following:

```
// Creating a Zend_View instance
```

```
$view = new Zend_View();

// Originating request:
$captcha = new Zend_Captcha_Figlet(array(
    'name' => 'foo',
    'wordLen' => 6,
    'timeout' => 300,
));
$id = $captcha->generate();
echo $captcha->render($view);

// On subsequent request:
// Assume captcha setup as before, and $value is the submitted value:
if ($captcha->isValid($_POST['foo'], $_POST)) {
    // Validated!
}
```

Captcha Adapters

The following adapters are shipped with Zend Framework by default.

Zend_Captcha_Word

Zend_Captcha_Word is an abstract adapter that serves as the basis for the Dumb, Figlet, and Image adapters. It provides mutators for specifying word length, session TTL, the session namespace object to use, and the session namespace class to use for persistence if you do not wish to use Zend_Session_Namespace. Additionally, it encapsulates all validation logic.

By default, the word length is 8 characters, the session timeout is 5 minutes, and Zend_Session_Namespace is used for persistence (using the namespace "Zend_Form_Captcha_<captcha ID>").

In addition to the standard methods required by the `Zend_Captcha_Adapter` interface, `Zend_Captcha_Word` 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. `$ttl` should be specified in seconds.
- `setSessionClass($class)` and `getSessionClass()` allow you to specify an alternate `Zend_Session_Namespace` implementation to use to persist the captcha token, as well as 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 it for you if none has been generated yet.
- `setSession(Zend_Session_Namespace $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 to the constructor, or, alternately, pass them to `setOptions()` (or pass a `Zend_Config` object to `setConfig()`). 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

Remeber, Word is an abstract class and may not be instantiated directly.

Zend_Captcha_Dumb

The Dumb adapter is mostly self-describing. It provides a random string that needs to be typed in reverse to validate. As such, it's not a good CAPTCHA solution, and should only be used either for testing or as a last resort. It extends `Zend_Captcha_Word`.

Zend_Captcha_Figlet

The Figlet adapter utilizes Zend_Text_Figlet to present a Figlet to the user. Figlet captchas are limited to characters only.

Options passed to the constructor will also be passed to the Zend_Text_Figlet object the adapter utilizes; see that documentation for details on what configuration options may be utilized.

Zend_Captcha_Image

The Image adapter takes the word generated and renders it as an image, performing various skewing permutations on it to make it difficult to automatically decipher. To perform its work, it requires the GD extension [http://php.net/gd] compiled with TrueType or Freetype support. Currently, the Image adapter can only generate PNG images.

`Zend_Captcha_Image` extends `Zend_Captcha_Word`, and additionally exposes the following methods:

- `setExpiration($expiration)` and `getExpiration()` allow you to specify a maximum lifetime a captcha image may reside on the filesystem. This is typically a longer duration than the session lifetime. Garbage collection is run periodically each time the captcha object is invoked, and images that have expired will be cleaned up. Expiration values are in seconds.
- `setGcFreq($gcFreq)` and `getGcFreg()` allow you to specify how frequently garbage collection should run. Garbage collection will run every `1/$gcFreq` calls (default is 100).
- `setFont($font)` and `getFont()` allow you to specify the font you wish to utilize. It should be a fully qualified path to the font file to utilize. If you do not set this value, the captcha will throw an exception during generation; the font is mandatory.
- `setFontSize($fsize)` and `getFontSize()` allow you to specify the font size, in pixels, to use when generating the captcha. This defaults to 24px.
- `setHeight($height)` and `getHeight()` allow you to specify the height, in pixels, of the generated captcha image. This defaults to 50px.
- `setWidth($width)` and `getWidth()` allow you to specify the width, in pixels, of the generated captcha image. This defaults to 200px.

- `setImgDir($imgDir)` and `getImgDir()` allow you to specify the directory in which captcha images are stored. This defaults to "./images/captcha/", which should look relative to the bootstrap script.
- `setImgUrl($imgUrl)` and `getImgUrl()` allow you to specify the relative path to a captcha image to use for the HTML markup. This defaults to "/images/captcha/".
- `setSuffix($suffix)` and `getSuffix()` allows you to specify the filename suffix to utilize. This defaults to ".png". Note: changing this will not change the image type generated.

All of the above options may be passed as options 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 ReCaptcha adapter utilizes Zend_Service_ReCaptcha to generate and validate captchas. It exposes the following methods:

- `setPrivKey($key)` and `getPrivKey()` allow you to specify the private key you use with the ReCaptcha service. This must be specified during construction, though it may be overridden at any point.
- `setPubKey($key)` and `getPubKey()` allow you to specify the public key you use with the ReCaptcha service. This must be specified during construction, though it may be overridden at any point.
- `setService(Zend_Service_ReCaptcha $service)` and `getService()` allow you to specify and interact with the ReCaptcha service object.

Chapter 6. Zend_Config

Introduction

Zend_Config is designed to simplify access to and use of configuration data within applications. It provides a nested object property based user interface for accessing such configuration data within application code. The configuration data may come from a variety of media supporting hierarchical data storage. Currently Zend_Config provides adapters for configuration data that are stored in text files with Zend_Config_Ini and Zend_Config_Xml.

Example 6.1. Using Zend_Config Per Se

Normally it is expected that users would use one of the adapter classes such as Zend_Config_Ini or Zend_Config_Xml, but if configuration data are available in a PHP array, one may simply pass the data to the Zend_Config constructor in order to utilize a simple object-oriented interface:

```
// Given an array of configuration data
$configArray = array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        )
    )
);

// Create the object-oriented wrapper upon the configuration data
$config = new Zend_Config($configArray);

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;

// Use the configuration data to connect to the database
$db = Zend_Db::factory($config->database->adapter,
                       $config->database->params->toArray());

// Alternative usage: simply pass the Zend_Config object.
// The Zend_Db factory knows how to interpret it.
$db = Zend_Db::factory($config->database);
```

As illustrated in the example above, Zend_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 also has get() which will return the supplied default value if the data element doesn't exist. For example:

```
$host = $config->database->get('host', 'localhost');
```

Example 6.2. Using Zend_Config with a PHP Configuration File

It is often desirable to use a pure PHP-based configuration file. The following code illustrates how easily this can be accomplished:

```
// config.php
return array(
    'webhost'  => 'www.example.com',
    'database' => array(
        'adapter' => 'pdo_mysql',
        'params'  => array(
            'host'     => 'db.example.com',
            'username' => 'dbuser',
            'password' => 'secret',
            'dbname'   => 'mydatabase'
        )
    )
);
```

```
// Configuration consumption
$config = new Zend_Config(require 'config.php');

// Print a configuration datum (results in 'www.example.com')
echo $config->webhost;
```

Theory of Operation

Configuration data are made accessible to the `Zend_Config` constructor through an associative array, which may be multidimensional, in order to support organizing the data from general to specific. Concrete adapter classes function to adapt configuration data from storage to produce the associative array for the `Zend_Config` constructor. User scripts may provide such arrays directly to the `Zend_Config` constructor, without using an adapter class, since it may be appropriate to do so in certain situations.

Each configuration data array value becomes a property of the `Zend_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` 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` implements the `Countable` and `Iterator` interfaces in order to facilitate simple access to configuration data. Thus, one may use the `count()` [http://php.net/count] function and PHP constructs such as `foreach` [http://php.net/foreach] upon `Zend_Config` objects.

By default, configuration data made available through `Zend_Config` are read-only, and an assignment (e.g., `$config->database->host = 'example.com'`) results in a thrown exception. This default behavior may be overridden through the constructor, however, to allow modification of data values. Also, when modifications are allowed, `Zend_Config` supports unsetting of values (i.e. `unset($config->database->host);`).

Note

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`. Third-party open source solutions are readily available for the purpose of creating and modifying configuration data for various storage media.

Adapter classes inherit from the `Zend_Config` class since they utilize its functionality.

The `Zend_Config` family of classes enables configuration data to be organized into sections. `Zend_Config` adapter objects may be loaded with a single specified section, multiple specified sections, or all sections (if none are specified).

`Zend_Config` adapter classes support a single inheritance model that enables configuration data to be inherited from one section of configuration data into another. This is provided in order to reduce or eliminate the need for duplicating configuration data for different purposes. An inheriting section may also override the values that it inherits through its parent section. Like PHP class inheritance, a section may inherit from a parent section, which may inherit from a grandparent section, and so on, but multiple inheritance (i.e., section C inheriting directly from parent sections A and B) is not supported.

If you have two `Zend_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`.

Zend_Config_Ini

`Zend_Config_Ini` enables developers to store configuration data in a familiar INI format and read them in the application by using nested object property syntax. The INI format is specialized to provide both the ability to have a hierarchy of configuration data keys and inheritance between configuration data sections. Configuration data hierarchies are supported by separating the keys with the dot or period character (`.`). A section may extend or inherit from another section by following the section name with a colon character (`:`) and the name of the section from which data are to be inherited.

parse_ini_file

`Zend_Config_Ini` utilizes the `parse_ini_file()` [http://php.net/parse_ini_file] PHP function. Please review this documentation to be aware of its specific behaviors, which propagate to `Zend_Config_Ini`, such as how the special values of `true`, `false`, `yes`, `no`, and `null` are handled.

Key Separator

By default, the key separator character is the period character (`.`). This can be changed, however, by changing the `$options` key `'nestSeparator'` when constructing the `Zend_Config_Ini` object. For example:

```
$options['nestSeparator'] = ':';
$config = new Zend_Config_Ini('/path/to/config.ini',
                              'staging',
                              $options);
```

Example 6.3. Using Zend_Config_Ini

This example illustrates a basic use of `Zend_Config_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. Because the staging system configuration data are very similar to those for production, the staging section inherits from the production section. In this case, the decision is arbitrary and could have been written conversely, with the production section inheriting from the staging section, though this may not be the case for more complex situations. Suppose, then, that the following configuration data are contained in `/path/to/config.ini`:

```
; Production site configuration data
[production]
webhost                  = www.example.com
database.adapter         = pdo_mysql
database.params.host     = db.example.com
database.params.username = dbuser
database.params.password = secret
database.params.dbname   = dbname

; Staging site configuration data inherits from production and
; overrides values as necessary
[staging : production]
database.params.host     = dev.example.com
database.params.username = devuser
database.params.password = devsecret
```

Next, assume that the application developer needs the staging configuration data from the INI file. It is a simple matter to load these data by specifying the INI file and the staging section:

```
$config = new Zend_Config_Ini('/path/to/config.ini', 'staging');

echo $config->database->params->host;   // prints "dev.example.com"
echo $config->database->params->dbname; // prints "dbname"
```

Note

Table 6.1. Zend_Config_Ini Constructor parameters

Parameter	**Notes**
`$filename`	The INI file to load.
`$section`	The [section] within the ini file that is to be loaded. Setting this parameter to null will load all sections. Alternatively, an array of section names may be supplied to load multiple sections.
`$options = false`	Options array. The following keys are supported: • *allowModifications*: Set to true to allow subsequent modification of loaded file. Defaults to false • *nestSeparator*: Set to the character to be used as the nest separator. Defaults to "."

Zend_Config_Xml

`Zend_Config_Xml` enables developers to store configuration data in a simple XML format and read them via nested object property syntax. The root element of the XML file is irrelevant and may be named arbitrarily. The first level of XML elements correspond with configuration data sections. The XML format supports hierarchical organization through nesting of XML elements below the section-level elements. The content of a leaf-level XML element corresponds to the value of a configuration datum. Section inheritance is supported by a special XML attribute named `extends`, and the value of this attribute corresponds with the section from which data are to be inherited by the extending section.

Return type

Configuration data read into `Zend_Config_Xml` are always returned as strings. Conversion of data from strings to other types is left to developers to suit their particular needs.

Example 6.4. Using Zend_Config_Xml

This example illustrates a basic use of `Zend_Config_Xml` for loading configuration data from an XML file. In this example there are configuration data for both a production system and for a staging system. Because the staging system configuration data are very similar to those for production, the staging section inherits from the production section. In this case, the decision is arbitrary and could have been written conversely, with the production section inheriting from the staging section, though this may not be the case for more complex situations. Suppose, then, that the following configuration data are contained in `/path/to/config.xml`:

```
<?xml version="1.0"?>
<configdata>
    <production>
        <webhost>www.example.com</webhost>
        <database>
            <adapter>pdo_mysql</adapter>
            <params>
                <host>db.example.com</host>
                <username>dbuser</username>
                <password>secret</password>
                <dbname>dbname</dbname>
            </params>
        </database>
    </production>
    <staging extends="production">
        <database>
            <params>
                <host>dev.example.com</host>
                <username>devuser</username>
                <password>devsecret</password>
            </params>
        </database>
    </staging>
</configdata>
```

Next, assume that the application developer needs the staging configuration data from the XML file. It is a simple matter to load these data by specifying the XML file and the staging section:

```
$config = new Zend_Config_Xml('/path/to/config.xml', 'staging');

echo $config->database->params->host;   // prints "dev.example.com"
echo $config->database->params->dbname; // prints "dbname"
```

Example 6.5. Using tag attributes in Zend_Config_Xml

Zend_Config_Xml also supports two additional ways of defining nodes in the configuration. Both make use of attributes. Since the `extends` and the `value` attributes are reserved keywords (the latter one by the the second way of using attributes), they may not be used. The first way of making usage of attributes is to add attributes in a parent nodes, which then will be translated into children of that node:

```
<?xml version="1.0"?>
<configdata>
    <production webhost="www.example.com">
        <database adapter="pdo_mysql">
            <params host="db.example.com" username="dbuser" password="secret" dbna
        </database>
    </production>
    <staging extends="production">
        <database>
            <params host="dev.example.com" username="devuser" password="devsecret"
        </database>
    </staging>
</configdata>
```

The other way does not really shorten the config, but keeps it easier to maintain since you don't have to write the tag-name twice. You simply create an empty tag, which contains it's value withing the `value` attribute:

```
<?xml version="1.0"?>
<configdata>
    <production>
        <webhost>www.example.com</webhost>
        <database>
            <adapter value="pdo_mysql"/>
            <params>
                <host value="db.example.com"/>
                <username value="dbuser"/>
                <password value="secret"/>
                <dbname value="dbname"/>
            </params>
        </database>
    </production>
    <staging extends="production">
        <database>
            <params>
                <host value="dev.example.com"/>
                <username value="devuser"/>
                <password value="devsecret"/>
            </params>
        </database>
    </staging>
</configdata>
```

Chapter 7. Zend_Console_Getopt

Introduction to Getopt

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 -1str**" uses a cluster of four short flags. This command is equivalent to "**ls -1 -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.

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.

Example 7.1. Using the Short Syntax

```
$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.

Example 7.2. Using the Long Syntax

```
$opts = new Zend_Console_Getopt(
  array(
    'apple|a'    => 'apple option, with no parameter',
    'banana|b=i' => 'banana option, with required integer parameter',
    'pear|p-s'   => 'pear option, with optional string parameter'
  )
);
```

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_Getopt_Exception`. 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.

Example 7.3. Catching Getopt Exceptions

```
try {
    $opts = new Zend_Console_Getopt('abp:');
    $opts->parse();
} catch (Zend_Console_Getopt_Exception $e) {
    echo $e->getUsageMessage();
    exit;
}
```

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`.

Example 7.4. Using getOption()

```
$opts = new Zend_Console_Getopt('abp:');
$b = $opts->getOption('b');
$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.

Example 7.5. Using __get() and __isset() magic methods

```
$opts = new Zend_Console_Getopt('abp:');
if (isset($opts->b)) {
    echo "I got the b option.\n";
}
$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.

Example 7.6. Using getRemainingArgs()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setArguments(array('-p', 'p_parameter', 'filename'));
$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.

Example 7.7. Using addRules()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->addRules(
  array(
    'verbose|v' => 'Print verbose output'
  )
);
```

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.

Example 7.8. Using setHelp()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setHelp(
    array(
        'a' => 'apple option, with no parameter',
        'b' => 'banana option, with required integer parameter',
        'p' => 'pear option, with optional string parameter'
    )
);
```

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.

Example 7.9. Using setAliases()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setAliases(
    array(
        'a' => 'apple',
        'a' => 'apfel',
        'p' => 'pear'
    )
);
```

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.

Example 7.10. Using addArguments() and setArguments()

```
// By default, the constructor uses $_SERVER['argv']
$opts = new Zend_Console_Getopt('abp:');

// Append an array to the existing arguments
$opts->addArguments(array('-a', '-p', 'p_parameter', 'non_option_arg'));

// Substitute a new array for the existing arguments
$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.

Clarifying the term "option"

The term "option" is used for configuration of the `Zend_Console_Getopt` class to match terminology used elsewhere in the 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.

Example 7.11. Using setOption()

```
$opts = new Zend_Console_Getopt('abp:');
$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.

Example 7.12. Using setOptions()

```
$opts = new Zend_Console_Getopt('abp:');
$opts->setOptions(
    array(
        'ignoreCase' => true,
        'dashDash'   => false
    )
);
```

Chapter 8. Zend_Controller

Zend_Controller Quick Start

Introduction

Zend_Controller is the heart of Zend Framework's MVC system. MVC stands for Model-View-Controller [http://en.wikipedia.org/wiki/Model-view-controller] and is a design pattern targeted at separating application logic from display logic. Zend_Controller_Front implements a Front Controller [http://www.martinfowler.com/eaaCatalog/frontController.html] pattern, in which all requests are intercepted by the front controller and dispatched to individual Action Controllers based on the URL requested.

The Zend_Controller system was built with extensibility in mind, either by subclassing the existing classes, writing new classes that implement the various interfaces and abstract classes that form the foundation of the controller family of classes, or writing plugins or action helpers to augment or manipulate the functionality of the system.

Quick Start

If you need more in-depth information, see the following sections. If you just want to get up and running quickly, read on.

Create your filesystem layout

The first step is to create your file system layout. The typical layout is as follows:

```
application/
    controllers/
        IndexController.php
    models/
    views/
        scripts/
            index/
                index.phtml
        helpers/
        filters/
html/
    .htaccess
    index.php
```

Set your document root

In your web server, point your document root to the html directory of the above file system layout.

Create your rewrite rules

Edit the html/.htaccess file above to read as follows:

```
RewriteEngine on
RewriteRule !\.(js|ico|gif|jpg|png|css)$ index.php
```

The above rules will route any non-resource (images, stylesheets) requests to the front controller. If there are other extensions you wish to exclude from the front controller (PDFs, text files, etc), add their extensions to the switch, or create your own rewrite rules.

Note

The above rewrite rules are for Apache; for examples of rewrite rules for other web servers, see the router documentation.

Create your bootstrap file

The bootstrap file is the page all requests are routed through -- `html/index.php` in this case. Open up `html/index.php` in the editor of your choice and add the following:

```
Zend_Controller_Front::run('/path/to/app/controllers');
```

This will instantiate and dispatch the front controller, which will route requests to action controllers.

Create your default action controller

Before discussing action controllers, you should first understand how requests are routed in Zend Framework. By default, the first segment of a URL path maps to a controller, and the second to an action. For example, given the URL `http://framework.zend.com/roadmap/components`, the path is `/roadmap/components`, which will map to the controller `roadmap` and the action `components`. If no action is provided, the action `index` is assumed, and if no controller is provided, the controller `index` is assumed (following the Apache convention that maps a `DirectoryIndex` automatically).

`Zend_Controller`'s dispatcher then takes the controller value and maps it to a class. By default, it Title-cases the controller name and appends the word `Controller`. Thus, in our example above, the controller `roadmap` is mapped to the class `RoadmapController`.

Similarly, the action value is mapped to a method of the controller class. By default, the value is lower-cased, and the word `Action` is appended. Thus, in our example above, the action `components` becomes `componentsAction`, and the final method called is `RoadmapController::componentsAction()`.

So, moving on, let's now create a default action controller and action method. As noted earlier, the default controller and action called are both `index`. Open the file `application/controllers/IndexController.php`, and enter the following:

```
/** Zend_Controller_Action */
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
    }
}
```

By default, the ViewRenderer action helper is enabled. What this means is that by simply defining an action method and a corresponding view script, you will immediately get content rendered. By default, `Zend_View` is used as the View layer in the MVC. The `ViewRenderer` does some magic, and uses the controller name (e.g., `index`) and the current action name (e.g., `index`) to determine what template to pull. By default, templates end in the `.phtml` extension, so this means that, in the above example, the template `index/index.phtml` will be rendered. Additionally, the `ViewRenderer` automatically assumes that the directory `views` at the same level as the controller directory will be the base view directory, and that the actual view scripts will be in the `views/scripts/` subdirectory. Thus, the template rendered will be found in `application/views/scripts/index/index.phtml`.

Create your view script

As mentioned in the previous section, view scripts are found in `application/views/scripts/`; the view script for the default controller and action is in `application/views/scripts/index/index.phtml`. Create this file, and type in some HTML:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>My first Zend Framework App</title>
</head>
<body>
    <h1>Hello, World!</h1>
</body>
</html>
```

Create your error controller

By default, the error handler plugin is registered. This plugin expects that a controller exists to handle errors. By default, it assumes an `ErrorController` in the default module with an `errorAction` method:

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
    }
}
```

Assuming the already discussed directory layout, this file will go in `application/controllers/ErrorController.php`. You will also need to create a view script in `application/views/scripts/error/error.phtml`; sample content might look like:

```
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Error</title>
</head>
<body>
    <h1>An error occurred</h1>
    <p>An error occurred; please try again later.</p>
</body>
</html>
```

View the site!

With your first controller and view under your belt, you can now fire up your browser and browse to the site. Assuming `example.com` is your domain, any of the following URLs will get to the page we've just created:

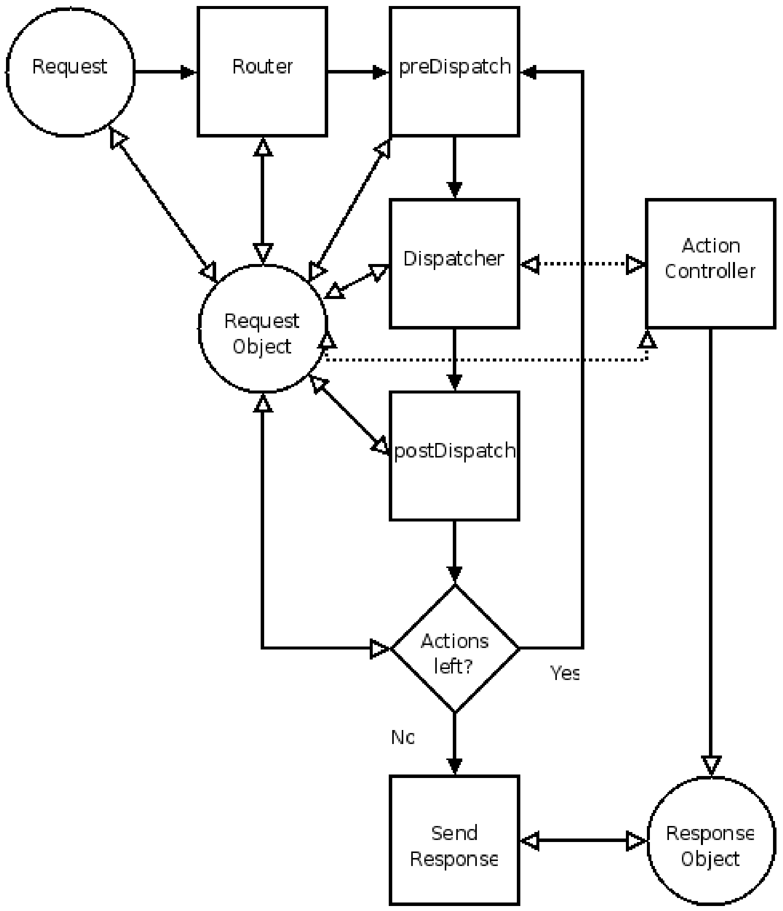
- `http://example.com/`
- `http://example.com/index`
- `http://example.com/index/index`

You're now ready to start creating more controllers and action methods. Congratulations!

Zend_Controller Basics

The `Zend_Controller` system is designed to be lightweight, modular, and extensible. It is a minimalist design to permit flexibility and some freedom to users while providing enough structure so that systems built around `Zend_Controller` share some common conventions and similar code layout.

The following diagram depicts the workflow, and the narrative following describes in detail the interactions:



The `Zend_Controller` workflow is implemented by several components. While it is not necessary to completely understand the underpinnings of all of these components to use the system, having a working knowledge of the process is helpful.

- `Zend_Controller_Front` orchestrates the entire workflow of the `Zend_Controller` system. It is an interpretation of the FrontController pattern. `Zend_Controller_Front` processes all requests received by the server and is ultimately responsible for delegating requests to ActionControllers (`Zend_Controller_Action`).
- `Zend_Controller_Request_Abstract` (often referred to as the `Request Object`) represents the request environment and provides methods for setting and retrieving the controller and action names and any request parameters. Additionally it keeps track of whether or not the action it contains has been dispatched by `Zend_Controller_Dispatcher`. Extensions to the abstract request object can be used to encapsulate the entire request environment, allowing routers to pull information from the request environment in order to set the controller and action names.

  By default, `Zend_Controller_Request_Http` is used, which provides access to the entire HTTP request environment.
- `Zend_Controller_Router_Interface` is used to define routers. Routing is the process of examining the request environment to determine which controller, and action of that controller, should receive the request. This controller, action, and optional parameters are then set in the request object to be processed by `Zend_Controller_Dispatcher_Standard`. Routing occurs only once: when the request is initially received and before the first controller is dispatched.

The default router, `Zend_Controller_Router_Rewrite`, takes a URI endpoint as specified in `Zend_Controller_Request_Http` and decomposes it into a controller, action, and parameters based on the path information in the url. As an example, the URL `http://localhost/foo/bar/key/value` would be decoded to use the `foo` controller, `bar` action, and specify a parameter `key` with a value of `value`.

`Zend_Controller_Router_Rewrite` can also be used to match arbitrary paths; see the router documentation for more information.

- `Zend_Controller_Dispatcher_Interface` is used to define dispatchers. Dispatching is the process of pulling the controller and action from the request object and mapping them to a controller file/class and action method in the controller class. If the controller or action do not exist, it handles determining default controllers and actions to dispatch.

  The actual dispatching process consists of instantiating the controller class and calling the action method in that class. Unlike routing, which occurs only once, dispatching occurs in a loop. If the request object's dispatched status is reset at any point, the loop will be repeated, calling whatever action is currently set in the request object. The first time the loop finishes with the request object's dispatched status set (boolean true), it will finish processing.

  The default dispatcher is `Zend_Controller_Dispatcher_Standard`. It defines controllers as MixedCasedClasses ending in the word Controller, and action methods as camelCasedMethods ending in the word Action: `FooController::barAction()`. In this case, the controller would be referred to as `foo` and the action as `bar`.

  ### CaseNamingConventions

  Since humans are notoriously inconsistent at maintaining case sensitivity when typing links, Zend Framework actually normalizes path information to lowercase. This, of course, will affect how you name your controller and actions... or refer to them in links.

  If you wish to have your controller class or action method name have multiple MixedCasedWords or camelCasedWords, you will need to separate those words on the url with either a '-' or '.' (though you can configure the character used).

  As an example, if you were going to the action in `FooBarController::bazBatAction()`, you'd refer to it on the url as `/foo-bar/baz-bat` or `/foo.bar/baz.bat`.

- `Zend_Controller_Action` is the base action controller component. Each controller is a single class that extends the `Zend_Controller_Action class` and should contain one or more action methods.

- `Zend_Controller_Response_Abstract` defines a base response class used to collect and return responses from the action controllers. It collects both headers and body content.

  The default response class is `Zend_Controller_Response_Http`, which is suitable for use in an HTTP environment.

The workflow of `Zend_Controller` is relatively simple. A request is received by `Zend_Controller_Front`, which in turn calls `Zend_Controller_Router_Rewrite` to determine which controller (and action in that controller) to dispatch. `Zend_Controller_Router_Rewrite` decomposes the URI in order to set the controller and action names in the request. `Zend_Controller_Front` then enters a dispatch loop. It calls `Zend_Controller_Dispatcher_Standard`, passing it the request, to dispatch to the controller and action specified in the request (or use defaults). After the controller has finished, control returns to `Zend_Controller_Front`. If the controller has indicated that another controller should be dispatched by resetting the dispatched status of the request, the loop continues and another dispatch is performed. Otherwise, the process ends.

The Front Controller

Overview

`Zend_Controller_Front` implements a Front Controller pattern [http://www.martinfowler.com/eaaCatalog/frontController.html] used in Model-View-Controller (MVC) [http://en.wikipedia.org/wiki/Model-view-controller] applications. Its purpose is to initialize the request environment, route the incoming request, and then dispatch any discovered actions; it aggregates any responses and returns them when the process is complete.

`Zend_Controller_Front` also implements the Singleton pattern [http://en.wikipedia.org/wiki/Singleton_pattern], meaning only a single instance of it may be available at any given time. This allows it to also act as a registry on which the other objects in the dispatch process may draw.

`Zend_Controller_Front` registers a plugin broker with itself, allowing various events it triggers to be observed by plugins. In most cases, this gives the developer the opportunity to tailor the dispatch process to the site without the need to extend the front controller to add functionality.

At a bare minimum, the front controller needs one or more paths to directories containing action controllers in order to do its work. A variety of methods may also be invoked to further tailor the front controller environment and that of its helper classes.

Default Behaviour

By default, the front controller loads the ErrorHandler plugin, as well as the ViewRenderer action helper plugin. These are to simplify error handling and view renderering in your controllers, respectively.

To disable the `ErrorHandler`, perform the following at any point prior to calling `dispatch()`:

```
// Disable the ErrorHandler plugin:
$front->setParam('noErrorHandler', true);
```

To disable the `ViewRenderer`, do the following prior to calling `dispatch()`:

```
// Disable the ViewRenderer helper:
$front->setParam('noViewRenderer', true);
```

Primary Methods

The front controller has several accessors for setting up its environment. However, there are three primary methods key to the front controller's functionality:

getInstance()

`getInstance()` is used to retrieve a front controller instance. As the front controller implements a Singleton pattern, this is also the only means possible for instantiating a front controller object.

```
$front = Zend_Controller_Front::getInstance();
```

setControllerDirectory() and addControllerDirectory

`setControllerDirectory()` is used to tell the dispatcher where to look for action controller class files. It accepts either a single path or an associative array of module/path pairs.

As some examples:

```
// Set the default controller directory:
$front->setControllerDirectory('../application/controllers');

// Set several module directories at once:
$front->setControllerDirectory(array(
    'default' => '../application/controllers',
    'blog'    => '../modules/blog/controllers',
    'news'    => '../modules/news/controllers',
));

// Add a 'foo' module directory:
$front->addControllerDirectory('../modules/foo/controllers', 'foo');
```

Note

If you use `addControllerDirectory()` without a module name, it will set the directory for the `default` module -- overwriting it if it already exists.

You can get the current settings for the controller directory using `getControllerDirectory()`; this will return an array of module/directory pairs.

addModuleDirectory() and getModuleDirectory()

One aspect of the front controller is that you may define a modular directory structure for creating standalone components; these are called "modules".

Each module should be in its own directory and mirror the directory structure of the default module -- i.e., it should have a "controllers" subdirectory at the minimum, and typically a "views" subdirectory and other application subdirectories.

`addModuleDirectory()` allows you to pass the name of a directory containing one or more module directories. It then scans it and adds them as controller directories to the front controller.

Later, if you want to determine the path to a particular module or the current module, you can call `getModuleDirectory()`, optionally passing a module name to get that specific module directory.

dispatch()

`dispatch(Zend_Controller_Request_Abstract $request = null, Zend_Controller_Response_Abstract $response = null)` does the heavy work of the front controller. It may optionally take a request object and/or a response object, allowing the developer to pass in custom objects for each.

If no request or response object are passed in, `dispatch()` will check for previously registered objects and use those or instantiate default versions to use in its process (in both cases, the HTTP flavor will be used as the default).

Similarly, `dispatch()` checks for registered router and dispatcher objects, instantiating the default versions of each if none is found.

The dispatch process has three distinct events:

- Routing
- Dispatching
- Response

Routing takes place exactly once, using the values in the request object when `dispatch()` is called. Dispatching takes place in a loop; a request may either indicate multiple actions to dispatch, or the controller or a plugin may reset the request object to force additional actions to dispatch. When all is done, the front controller returns a response.

run()

`Zend_Controller_Front::run($path)` is a static method taking simply a path to a directory containing controllers. It fetches a front controller instance (via getInstance(), registers the path provided via setControllerDirectory(), and finally dispatches.

Basically, `run()` is a convenience method that can be used for site setups that do not require customization of the front controller environment.

```
// Instantiate front controller, set controller directory, and dispatch in one
// easy step:
Zend_Controller_Front::run('../application/controllers');
```

Environmental Accessor Methods

In addition to the methods listed above, there are a number of accessor methods that can be used to affect the front controller environment -- and thus the environment of the classes to which the front controller delegates.

- `resetInstance()` can be used to clear all current settings. Its primary purpose is for testing, but it can also be used for instances where you wish to chain together multiple front controllers.

- `(set|get)DefaultControllerName()` let you specify a different name to use for the default controller ('index' is used otherwise) and retrieve the current value. They proxy to the dispatcher.
- `(set|get)DefaultAction()` let you specify a different name to use for the default action ('index' is used otherwise) and retrieve the current value. They proxy to the dispatcher.
- `(set|get)Request()` let you specify the request class or object to use during the dispatch process and to retrieve the current object. When setting the request object, you may pass in a request class name, in which case the method will load the class file and instantiate it.
- `(set|get)Router()` let you specify the router class or object to use during the dispatch process and to retrieve the current object. When setting the router object, you may pass in a router class name, in which case the method will load the class file and instantiate it.

  When retrieving the router object, it first checks to see if one is present, and if not, instantiates the default router (rewrite router).
- `(set|get)BaseUrl()` let you specify the base URL to strip when routing requests and to retrieve the current value. The value is provided to the request object just prior to routing.
- `(set|get)Dispatcher()` let you specify the dispatcher class or object to use during the dispatch process and retrieve the current object. When setting the dispatcher object, you may pass in a dispatcher class name, in which case the method will load the class file and instantiate it.

  When retrieving the dispatcher object, it first checks to see if one is present, and if not, instantiates the default dispatcher.
- `(set|get)Response()` let you specify the response class or object to use during the dispatch process and to retrieve the current object. When setting the response object, you may pass in a response class name, in which case the method will load the class file and instantiate it.
- `registerPlugin(Zend_Controller_Plugin_Abstract $plugin, $stackIndex = null)` allows you to register a plugin objects. By setting the optional `$stackIndex`, you can control the order in which plugins will execute.
- `unregisterPlugin($plugin)` let you unregister plugin objects. `$plugin` may be either a plugin object or a string denoting the class of plugin to unregister.
- `throwExceptions($flag)` is used to turn on/off the ability to throw exceptions during the dispatch process. By default, exceptions are caught and placed in the response object; turning on `throwExceptions()` will override this behaviour.

  For more information, read the section called “MVC Exceptions”.
- `returnResponse($flag)` is used to tell the front controller whether to return the response (`true`) from `dispatch()`, or if the response should be automatically emitted (`false`). By default, the response is automatically emitted (by calling `Zend_Controller_Response_Abstract::sendResponse()`); turning on `returnResponse()` will override this behaviour.

  Reasons to return the response include a desire to check for exceptions prior to emitting the response, needing to log various aspects of the response (such as headers), etc.

Front Controller Parameters

In the introduction, we indicated that the front controller also acts as a registry for the various controller components. It does so through a family of "param" methods. These methods allow you to register arbitrary data -- objects and variables -- with the front controller to be retrieved at any time in the dispatch chain. These values are passed on to the router, dispatcher, and action controllers. The methods include:

- `setParam($name, $value)` allows you to set a single parameter of `$name` with value `$value`.
- `setParams(array $params)` allows you to set multiple parameters at once using an associative array.
- `getParam($name)` allows you to retrieve a single parameter at a time, using `$name` as the identifier.
- `getParams()` allows you to retrieve the entire list of parameters at once.
- `clearParams()` allows you to clear a single parameter (by passing a string identifier), multiple named parameters (by passing an array of string identifiers), or the entire parameter stack (by passing nothing).

There are several pre-defined parameters that may be set that have specific uses in the dispatch chain:

- `useDefaultControllerAlways` is used to hint to the dispatcher to use the default controller in the default module for any request that is not dispatchable (i.e., the module, controller, and/or action do not exist). By default, this is off.

  See the section called “MVC Exceptions You May Encounter” for more detailed information on using this setting.
- `disableOutputBuffering` is used to hint to the dispatcher that it should not use output buffering to capture output generated by action controllers. By default, the dispatcher captures any output and appends it to the response object body content.
- `noViewRenderer` is used to disable the ViewRenderer. Set this parameter to true to disable it.
- `noErrorHandler` is used to disable the Error Handler plugin. Set this parameter to true to disable it.

Subclassing the Front Controller

To subclass the Front Controller, at the very minimum you will need to override the `getInstance()` method:

```
class My_Controller_Front extends Zend_Controller_Front
{
    public static function getInstance()
    {
        if (null === self::$_instance) {
            self::$_instance = new self();
        }

        return self::$_instance;
    }
}
```

Overriding the `getInstance()` method ensures that subsequent calls to `Zend_Controller_Front::getInstance()` will return an instance of your new subclass instead of a `Zend_Controller_Front` instance -- this is particularly useful for some of the alternate routers and view helpers.

Typically, you will not need to subclass the front controller unless you need to add new functionality (for instance, a plugin autoloader, or a way to specify action helper paths). Some points where you may want to alter behaviour may include modifying how controller directories are stored, or what default router or dispatcher are used.

The Request Object

Introduction

The request object is a simple value object that is passed between `Zend_Controller_Front` and the router, dispatcher, and controller classes. It packages the names of the requested module, controller, action, and optional parameters, as well as the rest of the request environment, be it HTTP, the CLI, or PHP-GTK.

- The module name is accessed by `getModuleName()` and `setModuleName()`.
- The controller name is accessed by `getControllerName()` and `setControllerName()`.
- The name of the action to call within that controller is accessed by `getActionName()` and `setActionName()`.
- Parameters to be accessible by the action are an associative array of key/value pairs that are retrieved by `getParams()` and set with `setParams()`, or individually by `getParam()` and `setParam()`.

Based on the type of request, there may be more methods available. The default request used, `Zend_Controller_Request_Http`, for instance, has methods for retrieving the request URI, path information, `$_GET` and `$_POST` parameters, etc.

The request object is passed to the front controller, or if none is provided, it is instantiated at the beginning of the dispatch process, before routing occurs. It is passed through to every object in the dispatch chain.

Additionally, the request object is particularly useful in testing. The developer may craft the request environment, including module, controller, action, parameters, URI, etc, and pass the request object to the front controller to test application flow. When paired with the response object, elaborate and precise unit testing of MVC applications becomes possible.

HTTP Requests

Accessing Request Data

`Zend_Controller_Request_Http` encapsulates access to relevant values such as the key name and value for the controller and action router variables, and all additional parameters parsed from the URI. It additionally allows access to values contained in the superglobals as public members, and manages the current Base URL and Request URI. Superglobal values cannot be set on a request object, instead use the setParam/getParam methods to set or retrieve user parameters.

Superglobal data

When accessing superglobal data through `Zend_Controller_Request_Http` as public member properties, it is necessary to keep in mind that the property name (superglobal array key) is matched to a superglobal in a specific order of precedence: 1. GET, 2. POST, 3. COOKIE, 4. SERVER, 5. ENV.

Specific superglobals can be accessed using a public method as an alternative. For example, the raw value of `$_POST['user']` can be accessed by calling `getPost('user')` on the request object. These include `getQuery()` for retrieving `$_GET` elements, and `getHeader()` for retrieving request headers.

GET and POST data

Be cautious when accessing data from the request object as it is not filtered in any way. The router and dispatcher validate and filter data for use with their tasks, but leave the data untouched in the request object.

Retrieve the Raw POST Data, Too!

As of 1.5.0, you can also retrieve the raw post data via the `getRawBody()` method. This method returns false if no data was submitted in that fashion, but the full body of the post otherwise.

This is primarily useful for accepting content when developing a RESTful MVC application.

You may also set user parameters in the request object using `setParam()` and retrieve these later using `getParam()`. The router makes use of this functionality to set parameters matched in the request URI into the request object.

getParam() retrieves more than user params

In order to do some of its work, `getParam()` actually retrieves from several sources. In order of priority, these include: user parameters set via `setParam()`, `GET` parameters, and finally `POST` parameters. Be aware of this when pulling data via this method.

If you wish to pull only from parameters you set via `setParam()`, use the `getUserParam()`.

Additionally, as of 1.5.0, you can lock down which parameter sources will be searched. `setParamSources()` allows you to specify an empty array or an array with one or more of the values '_GET' or '_POST' indicating which parameter sources are allowed (by default, both are allowed); if you wish to restrict access to only '_GET' specify `setParamSources(array('_GET'))`.

Apache Quirks

If you are using Apache's 404 handler to pass incoming requests to the front controller, or using a PT flag with rewrite rules, `$_SERVER['REDIRECT_URL']` contains the URI you need, not `$_SERVER['REQUEST_URI']`. If you are using such a setup and getting invalid routing, you should use the `Zend_Controller_Request_Apache404` class instead of the default Http class for your request object:

```
$request = new Zend_Controller_Request_Apache404();
$front->setRequest($request);
```

This class extends the `Zend_Controller_Request_Http` class and simply modifies the autodiscovery of the request URI. It can be used as a drop-in replacement.

Base Url and Subdirectories

`Zend_Controller_Request_Http` allows Zend_Controller_Router_Rewrite to be used in subdirectories. Zend_Controller_Request_Http will attempt to automatically detect your base URL and set it accordingly.

For example, if you keep your `index.php` in a webserver subdirectory named `/projects/myapp/index.php`, base URL (rewrite base) should be set to `/projects/myapp`. This string will then be stripped from the beginning of the path before calculating any route matches. This frees one from the necessity of prepending it to any of your routes. A route of `'user/:username'` will match URIs like `http://localhost/projects/myapp/user/martel` and `http://example.com/user/martel`.

URL detection is case sensitive

Automatic base URL detection is case sensitive, so make sure your URL will match a subdirectory name in a filesystem (even on Windows machines). If it doesn't, an exception will be raised.

Should base URL be detected incorrectly you can override it with your own base path with the help of the `setBaseUrl()` method of either the `Zend_Controller_Request_Http` class, or the `Zend_Controller_Front` class. The easiest method is to set it in `Zend_Controller_Front`, which will proxy it into the request object. Example usage to set a custom base URL:

```
/**
 * Dispatch Request with custom base URL with Zend_Controller_Front.
 */
$router     = new Zend_Controller_Router_Rewrite();
$controller = Zend_Controller_Front::getInstance();
$controller->setControllerDirectory('./application/controllers')
           ->setRouter($router)
           ->setBaseUrl('/projects/myapp'); // set the base url!
$response   = $controller->dispatch();
```

Determining the Request Method

`getMethod()` allows you to determine the HTTP request method used to request the current resource. Additionally, a variety of methods exist that allow you to get boolean responses when asking if a specific type of request has been made:

- `isGet()`
- `isPost()`
- `isPut()`
- `isDelete()`

- `isHead()`
- `isOptions()`

The primary use case for these is for creating RESTful MVC architectures.

Detecting AJAX Requests

`Zend_Controller_Request_Http` has a rudimentary method for detecting AJAX requests: `isXmlHttpRequest()`. This method looks for an HTTP request header `X-Requested-With` with the value 'XMLHttpRequest'; if found, it returns true.

Currently, this header is known to be passed by default with the following JS libraries:

- Prototype/Scriptaculous (and libraries derived from Prototype)
- Yahoo! UI Library
- jQuery
- MochiKit

Most AJAX libraries allow you to send custom HTTP request headers; if your library does not send this header, simply add it as a request header to ensure the `isXmlHttpRequest()` method works for you.

Subclassing the Request Object

The base request class used for all request objects is the abstract class `Zend_Controller_Request_Abstract`. At its most basic, it defines the following methods:

```
abstract class Zend_Controller_Request_Abstract
{
    /**
     * @return string
     */
    public function getControllerName();

    /**
     * @param string $value
     * @return self
     */
    public function setControllerName($value);

    /**
     * @return string
     */
    public function getActionName();

    /**
     * @param string $value
     * @return self
     */
    public function setActionName($value);
```

```
/**
 * @return string
 */
public function getControllerKey();

/**
 * @param string $key
 * @return self
 */
public function setControllerKey($key);

/**
 * @return string
 */
public function getActionKey();

/**
 * @param string $key
 * @return self
 */
public function setActionKey($key);

/**
 * @param string $key
 * @return mixed
 */
public function getParam($key);

/**
 * @param string $key
 * @param mixed $value
 * @return self
 */
public function setParam($key, $value);

/**
 * @return array
 */
 public function getParams();

/**
 * @param array $array
 * @return self
 */
public function setParams(array $array);

/**
 * @param boolean $flag
 * @return self
 */
public function setDispatched($flag = true);

/**
```

```
     * @return boolean
     */
    public function isDispatched();
}
```

The request object is a container for the request environment. The controller chain really only needs to know how to set and retrieve the controller, action, optional parameters, and dispatched status. By default, the request will search its own parameters using the controller or action keys in order to determine the controller and action.

Extend this class, or one of its derivatives, when you need the request class to interact with a specific environment in order to retrieve data for use in the above tasks. Examples include the HTTP environment, a CLI environment, or a PHP-GTK environment.

The Standard Router

Introduction

`Zend_Controller_Router_Rewrite` is the standard framework router. Routing is the process of taking a URI endpoint (that part of the URI which comes after the base URL) and decomposing it into parameters to determine which module, controller, and action of that controller should receive the request. This values of the module, controller, action and other parameters are packaged into a `Zend_Controller_Request_Http` object which is then processed by `Zend_Controller_Dispatcher_Standard`. Routing occurs only once: when the request is initially received and before the first controller is dispatched.

`Zend_Controller_Router_Rewrite` is designed to allow for mod_rewrite-like functionality using pure php structures. It is very loosely based on Ruby on Rails routing and does not require any prior knowledge of webserver URL rewriting. It is designed to work with a single Apache mod_rewrite rule (one of):

```
RewriteEngine on
RewriteRule !\.(js|ico|gif|jpg|png|css)$ index.php
```

or:

```
RewriteEngine on
RewriteCond %{SCRIPT_FILENAME} !-f
RewriteCond %{SCRIPT_FILENAME} !-d
RewriteRule ^(.*)$ index.php/$1
```

The rewrite router can also be used with the IIS webserver if Isapi_Rewrite [http://www.isapirewrite.com] has been installed as an Isapi extension with the following rewrite rule:

```
RewriteRule ^[\w/\%]*(?:\.(?!(?:js|ico|gif|jpg|png|css)$)[\w\%]*$)? /index.php [I]
```

IIS Isapi_Rewrite

When using IIS, `$_SERVER['REQUEST_URI']` will either not exist, or be set as an empty string. In this case, `Zend_Controller_Request_Http` will attempt to use the `$_SERVER['HTTP_X_REWRITE_URL']` value set by the Isapi_Rewrite extension.

If using Lighttpd, the following rewrite rule is valid:

```
url.rewrite-once = (
    ".*\?(.*)$" => "/index.php?$1",
    ".*\.(js|ico|gif|jpg|png|css)$" => "$0",
    "" => "/index.php"
)
```

Using a router

To properly use the rewrite router you have to instantiate it, add some user defined routes and inject it into the controller. The following code illustrates the procedure:

```
// Create a router

$router = $ctrl->getRouter(); // returns a rewrite router by default
$router->addRoute(
    'user',
    new Zend_Controller_Router_Route('user/:username',
                                     array('controller' => 'user',
                                           'action' => 'info'))
);
```

Basic Rewrite Router operation

The heart of the RewriteRouter is the definition of user defined routes. Routes are added by calling the addRoute method of RewriteRouter and passing in a new instance of a class implementing `Zend_Controller_Router_Route_Interface`. Eg.:

```
$router->addRoute('user',
                  new Zend_Controller_Router_Route('user/:username'));
```

Rewrite Router comes with four basic types of routes (one of which is special):

- the section called “Zend_Controller_Router_Route”

- the section called “Zend_Controller_Router_Route_Static”
- the section called “Zend_Controller_Router_Route_Regex”
- the section called “Default routes” *

Routes may be used numerous times to create a chain or user defined application routing schema. You may use any number of routes in any configuration, with the exception of the Module route, which should rather be used once and probably as the most generic route (i.e., as a default). Each route will be described in greater detail later on.

The first parameter to addRoute is the name of the route. It is used as a handle for getting the routes out of the router (e.g., for URL generation purposes). The second parameter being the route itself.

Note

The most common use of the route name is through the means of Zend_View url helper:

```
<a href=
"<?= $this->url(array('username' => 'martel'), 'user') ?>">Martel</a>
```

Which would result in the href: `user/martel`.

Routing is a simple process of iterating through all provided routes and matching its definitions to current request URI. When a positive match is found, variable values are returned from the Route instance and are injected into the `Zend_Controller_Request` object for later use in the dispatcher as well as in user created controllers. On a negative match result, the next route in the chain is checked.

Reverse matching

Routes are matched in reverse order so make sure your most generic routes are defined first.

Returned values

Values returned from routing come from URL parameters or user defined route defaults. These variables are later accessible through the `Zend_Controller_Request::getParam()` or `Zend_Controller_Action::_getParam()` methods.

There are three special variables which can be used in your routes - 'module', 'controller' and 'action'. These special variables are used by Zend_Controller_Dispatcher to find a controller and action to dispatch to.

Special variables

The names of these special variables may be different if you choose to alter the defaults in `Zend_Controller_Request_Http` by means of the `setControllerKey` and `setActionKey` methods.

Default routes

Zend_Controller_Router_Rewrite comes preconfigured with a default route, which will match URIs in the shape of `controller/action`. Additionally, a module name may be specified as the first path element, allowing URIs of the form `module/controller/action`. Finally, it will also match any additional parameters appended to the URI by default - `controller/action/var1/value1/var2/value2`.

Some examples of how such routes are matched:

```
// Assuming the following:
$ctrl->setControllerDirectory(
    array(
        'default' => '/path/to/default/controllers',
        'news'    => '/path/to/news/controllers',
        'blog'    => '/path/to/blog/controllers'
    )
);

Module only:
http://example/news
    module == news

Invalid module maps to controller name:
http://example/foo
    controller == foo

Module + controller:
http://example/blog/archive
    module     == blog
    controller == archive

Module + controller + action:
http://example/blog/archive/list
    module     == blog
    controller == archive
    action     == list

Module + controller + action + params:
http://example/blog/archive/list/sort/alpha/date/desc
    module     == blog
    controller == archive
    action     == list
    sort       == alpha
    date       == desc
```

The default route is simply a `Zend_Controller_Router_Route_Module` object stored under the name (index) of 'default' in RewriteRouter. It's created more-or-less like below:

```
$compat = new Zend_Controller_Router_Route_Module(array(),
                                                  $dispatcher,
                                                  $request);
$this->addRoute('default', $compat);
```

If you do not want this particular default route in your routing schema, you may override it by creating your own 'default' route (i.e., storing it under the name of 'default') or removing it altogether by using `removeDefaultRoutes()`:

```
// Remove any default routes
$router->removeDefaultRoutes();
```

Base URL and subdirectories

The rewrite router can be used in subdirectories (e.g., `http://domain.com/~user/application-root/`) in which case the base URL of the application (`/~user/application-root`) should be automatically detected by `Zend_Controller_Request_Http` and used accordingly.

Should the base URL be detected incorrectly you can override it with your own base path by using `Zend_Controller_Request_Http` and calling the `setBaseUrl()` method (see the section called “Base Url and Subdirectories”):

```
$request->setBaseUrl('/~user/application-root/');
```

Route Types

Zend_Controller_Router_Route

`Zend_Controller_Router_Route` is the standard framework route. It combines ease of use with flexible route definition. Each route consists primarily of URL mapping (of static and dynamic parts (variables)) and may be initialized with defaults as well as with variable requirements.

Let's imagine our fictional application will need some informational page about the content authors. We want to be able to point our web browsers to `http://domain.com/author/martel` to see the information about this "martel" guy. And the route for such functionality could look like:

```
$route = new Zend_Controller_Router_Route(
    'author/:username',
    array(
        'controller' => 'profile',
        'action'     => 'userinfo'
    )
);

$router->addRoute('user', $route);
```

The first parameter in the `Zend_Controller_Router_Route` constructor is a route definition that will be matched to a URL. Route definitions consist of static and dynamic parts separated by the slash ('/')

character. Static parts are just simple text: `author`. Dynamic parts, called variables, are marked by prepending a colon to the variable name: `:username`.

Character usage

The current implementation allows you to use any character (except a slash) as a variable identifier, but it is strongly recommended that one uses only characters that are valid for PHP variable identifiers. Future implementations may alter this behaviour, which could result in hidden bugs in your code.

This example route should be matched when you point your browser to `http://domain.com/author/martel`, in which case all its variables will be injected to the `Zend_Controller_Request` object and will be accessible in your `ProfileController`. Variables returned by this example may be represented as an array of the following key and value pairs:

```
$values = array(
    'username'   => 'martel',
    'controller' => 'profile',
    'action'     => 'userinfo'
);
```

Later on, `Zend_Controller_Dispatcher_Standard` should invoke the `userinfoAction()` method of your `ProfileController` class (in the default module) based on these values. There you will be able to access all variables by means of the `Zend_Controller_Action::_getParam()` or `Zend_Controller_Request::getParam()` methods:

```
public function userinfoAction()
{
    $request = $this->getRequest();
    $username = $request->getParam('username');

    $username = $this->_getParam('username');
}
```

Route definition can contain one more special character - a wildcard - represented by '*' symbol. It is used to gather parameters similarly to the default Module route (var => value pairs defined in the URI). The following route more-or-less mimics the Module route behavior:

```
$route = new Zend_Controller_Router_Route(
    ':module/:controller/:action/*',
    array('module' => 'default')
);
$router->addRoute('default', $route);
```

Variable defaults

Every variable in the route can have a default and this is what the second parameter of the `Zend_Controller_Router_Route` constructor is used for. This parameter is an array with keys representing variable names and with values as desired defaults:

```
$route = new Zend_Controller_Router_Route(
    'archive/:year',
    array('year' => 2006)
);
$router->addRoute('archive', $route);
```

The above route will match URLs like `http://domain.com/archive/2005` and `http://example.com/archive`. In the latter case the variable year will have an initial default value of 2006.

This example will result in injecting a year variable to the request object. Since no routing information is present (no controller and action parameters are defined), the application will be dispatched to the default controller and action method (which are both defined in `Zend_Controller_Dispatcher_Abstract`). To make it more usable, you have to provide a valid controller and a valid action as the route's defaults:

```
$route = new Zend_Controller_Router_Route(
    'archive/:year',
    array(
        'year'       => 2006,
        'controller' => 'archive',
        'action'     => 'show'
    )
);
$router->addRoute('archive', $route);
```

This route will then result in dispatching to the method `showAction()` of the class `ArchiveController`.

Variable requirements

One can add a third parameter to the `Zend_Controller_Router_Route` constructor where variable requirements may be set. These are defined as parts of a regular expression:

```
$route = new Zend_Controller_Router_Route(
    'archive/:year',
    array(
        'year'       => 2006,
        'controller' => 'archive',
        'action'     => 'show'
    ),
    array('year' => '\d+')
);
```

```
$router->addRoute('archive', $route);
```

With a route defined like above, the router will match it only when the year variable will contain numeric data, eg. `http://domain.com/archive/2345`. A URL like `http://example.com/archive/test` will not be matched and control will be passed to the next route in the chain instead.

Hostname routing

You can also use the hostname for route matching. For simple matching there is a static hostname option:

```
$route = new Zend_Controller_Router_Route(
    array(
        'host' => 'blog.mysite.com',
        'path' => 'archive'
    ),
    array(
        'module'     => 'blog',
        'controller' => 'archive',
        'action'     => 'index'
    )
);
$router->addRoute('archive', $route);
```

If you want to match parameters in the hostname, there is a regex option. In the following example, the subdomain is used as username parameter for the action controller. When assembling the route, you simply give the username as parameter, as you would with the other path parameters:

```
$route = new Zend_Controller_Router_Route(
    array(
        'host' => array(
            'regex'   => '([a-z]+).mysite.com',
            'reverse' => '%s.mysite.com'
            'params'  => array(
                1 => 'username'
            )
        ),
        'path' => ''
    ),
    array(
        'module'     => 'users',
        'controller' => 'profile',
        'action'     => 'index'
    )
);
$router->addRoute('profile', $route);
```

Zend_Controller_Router_Route_Static

The examples above all use dynamic routes -- routes that contain patterns to match against. Sometimes, however, a particular route is set in stone, and firing up the regular expression engine would be an overkill. The answer to this situation is to use static routes:

```
$route = new Zend_Controller_Router_Route_Static(
    'login',
    array('controller' => 'auth', 'action' => 'login')
);
$router->addRoute('login', $route);
```

Above route will match a URL of `http://domain.com/login`, and dispatch to `AuthController::loginAction()`.

Zend_Controller_Router_Route_Regex

In addition to the default and static route types, a Regular Expression route type is available. This route offers more power and flexibility over the others, but at a slight cost of complexity. At the same time, it should be faster than the standard Route.

Like the standard route, this route has to be initialized with a route definition and some defaults. Let's create an archive route as an example, similar to the previously defined one, only using the Regex route this time:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array(
        'controller' => 'archive',
        'action'     => 'show'
    )
);
$router->addRoute('archive', $route);
```

Every defined regex subpattern will be injected to the request object. With our above example, after successful matching `http://domain.com/archive/2006`, the resulting value array may look like:

```
$values = array(
    1            => '2006',
    'controller' => 'archive',
    'action'     => 'show'
);
```

Note

Leading and trailing slashes are trimmed from the URL in the Router prior to a match. As a result, matching the URL `http://domain.com/foo/bar/`, would involve a regex of `foo/bar`, and not `/foo/bar`.

Note

Line start and line end anchors ('^' and '$', respectively) are automatically pre- and appended to all expressions. Thus, you should not use these in your regular expressions, and you should match the entire string.

Note

This route class uses the # character for a delimiter. This means that you will need to escape hash characters ('#') but not forward slashes ('/') in your route definitions. Since the '#' character (named anchor) is rarely passed to the webserver, you will rarely need to use that character in your regex.

You can get the contents of the defined subpatterns the usual way:

```
public function showAction()
{
    $request = $this->getRequest();
    $year    = $request->getParam(1); // $year = '2006';
}
```

Note

Notice the key is an integer (1) instead of a string ('1').

This route will not yet work exactly the same as its standard route counterpart since the default for 'year' is not yet set. And what may not yet be evident is that we will have a problem with a trailing slash even if we declare a default for the year and make the subpattern optional. The solution is to make the whole year part optional along with the slash but catch only the numeric part:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive(?:/(\d+))?',
    array(
        1            => '2006',
        'controller' => 'archive',
        'action'     => 'show'
    )
);
$router->addRoute('archive', $route);
```

Now let's get to the problem you have probably noticed on your own by now. Using integer based keys for parameters is not an easily manageable solution and may be potentially problematic in the long run.

And that's where the third parameter comes in. This parameter is an associative array that represents a map of regex subpatterns to parameter named keys. Let's work on our easier example:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array(
        'controller' => 'archive',
        'action' => 'show'
    ),
    array(
        1 => 'year'
    )
);
$router->addRoute('archive', $route);
```

This will result in following values injected into Request:

```
$values = array(
    'year'       => '2006',
    'controller' => 'archive',
    'action'     => 'show'
);
```

The map may be defined in either direction to make it work in any environment. Keys may contain variable names or subpattern indexes:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array( ... ),
    array(1 => 'year')
);

// OR

$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array( ... ),
    array('year' => 1)
);
```

Note

Subpattern keys have to be represented by integers.

Notice that the numeric index in Request values is now gone and a named variable is shown in its place. Of course you can mix numeric and named variables if you wish:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)/page/(\d+)',
    array( ... ),
    array('year' => 1)
);
```

Which will result in mixed values available in the Request. As an example, the URL `http://domain.com/archive/2006/page/10` will result in following values:

```
$values = array(
    'year'       => '2006',
    2            => 10,
    'controller' => 'archive',
    'action'     => 'show'
);
```

Since regex patterns are not easily reversed, you will need to prepare a reverse URL if you wish to use a URL helper or even an assemble method of this class. This reversed path is represented by a string parsable by sprintf() and is defined as a fourth construct parameter:

```
$route = new Zend_Controller_Router_Route_Regex(
    'archive/(\d+)',
    array( ... ),
    array('year' => 1),
    'archive/%s'
);
```

All of this is something which was already possible by the means of a standard route object, so where's the benefit in using the Regex route, you ask? Primarily, it allows you to describe any type of URL without any restrictions. Imagine you have a blog and wish to create URLs like: `http://domain.com/blog/archive/01-Using_the_Regex_Router.html`, and have it decompose the last path element, `01-Using_the_Regex_Router.html`, into an article ID and article title/description; this is not possible with the standard route. With the Regex route, you can do something like the following solution:

```
$route = new Zend_Controller_Router_Route_Regex(
    'blog/archive/(\d+)-(.+)\.html',
    array(
        'controller' => 'blog',
        'action'     => 'view'
    ),
    array(
        1 => 'id',
        2 => 'description'
    ),
    'blog/archive/%d-%s.html'
);
$router->addRoute('blogArchive', $route);
```

As you can see, this adds a tremendous amount of flexibility over the standard route.

Using Zend_Config with the RewriteRouter

Sometimes it is more convenient to update a configuration file with new routes than to change the code. This is possible via the `addConfig()` method. Basically, you create a Zend_Config-compatible configuration, and in your code read it in and pass it to the RewriteRouter.

As an example, consider the following INI file:

```
[production]
routes.archive.route = "archive/:year/*"
routes.archive.defaults.controller = archive
routes.archive.defaults.action = show
routes.archive.defaults.year = 2000
routes.archive.reqs.year = "\d+"

routes.news.type = "Zend_Controller_Router_Route_Static"
routes.news.route = "news"
routes.news.defaults.controller = "news"
routes.news.defaults.action = "list"

routes.archive.type = "Zend_Controller_Router_Route_Regex"
routes.archive.route = "archive/(\d+)"
routes.archive.defaults.controller = "archive"
routes.archive.defaults.action = "show"
routes.archive.map.1 = "year"
; OR: routes.archive.map.year = 1
```

The above INI file can then be read into a `Zend_Config` object as follows:

```
$config = new Zend_Config_Ini('/path/to/config.ini', 'production');
$router = new Zend_Controller_Router_Rewrite();
$router->addConfig($config, 'routes');
```

In the above example, we tell the router to use the 'routes' section of the INI file to use for its routes. Each first-level key under that section will be used to define a route name; the above example defines the routes 'archive' and 'news'. Each route then requires, at minimum, a 'route' entry and one or more 'defaults' entries; optionally one or more 'reqs' (short for 'required') may be provided. All told, these correspond to the three arguments provided to a `Zend_Controller_Router_Route_Interface` object. An option key, 'type', can be used to specify the route class type to use for that particular route; by default, it uses `Zend_Controller_Router_Route`. In the example above, the 'news' route is defined to use `Zend_Controller_Router_Route_Static`.

Subclassing the Router

The standard rewrite router should provide most functionality you may need; most often, you will only need to create a new route type in order to provide new or modified functionality over the provided routes.

That said, you may at some point find yourself wanting to use a different routing paradigm. The interface `Zend_Controller_Router_Interface` provides the minimal information required to create a router, and consists of a single method.

```
interface Zend_Controller_Router_Interface
{
  /**
   * @param  Zend_Controller_Request_Abstract $request
   * @throws Zend_Controller_Router_Exception
   * @return Zend_Controller_Request_Abstract
   */
  public function route(Zend_Controller_Request_Abstract $request);
}
```

Routing only occurs once: when the request is first received into the system. The purpose of the router is to determine the controller, action, and optional parameters based on the request environment, and then set them in the request. The request object is then passed to the dispatcher. If it is not possible to map a route to a dispatch token, the router should do nothing to the request object.

The Dispatcher

Overview

Dispatching is the process of taking the request object, `Zend_Controller_Request_Abstract`, extracting the module name, controller name, action name, and optional parameters contained in it, and then instantiating a controller and calling an action of that controller. If any of the module, controller, or action are not found, it will use default values for them. `Zend_Controller_Dispatcher_Standard` specifies `index` for each of the controller and action defaults and `default` for the module default value, but allows the developer to change the default values for each using the `setDefaultController()`, `setDefaultAction()`, and `setDefaultModule()` methods, respectively.

Default Module

When creating modular applications, you may find that you want your default module namespaced as well (the default configuration is that the default module is *not* namespaced). As of 1.5.0, you can now do so by specifying the `prefixDefaultModule` as true in either the front controller or your dispatcher:

```
// In your front controller:
$front->setParam('prefixDefaultModule', true);

// In your dispatcher:
$dispatcher->setParam('prefixDefaultModule', true);
```

This allows you to re-purpose an existing module to be the default module for an application.

Dispatching happens in a loop in the front controller. Before dispatching occurs, the front controller routes the request to find user specified values for the module, controller, action, and optional parameters. It then enters a dispatch loop, dispatching the request.

At the beginning of each iteration, it sets a flag in the request object indicating that the action has been dispatched. If an action or pre/postDispatch plugin resets that flag, the dispatch loop will continue and attempt to dispatch the new request. By changing the controller and/or action in the request and resetting the dispatched flag, the developer may define a chain of requests to perform.

The action controller method that controls such dispatching is `_forward()`; call this method from any of the pre/postDispatch() or action methods, providing an action, controller, module, and optionally any additional parameters you may wish to send to the new action:

```
public function fooAction()
{
    // forward to another action in the current controller and module:
    $this->_forward('bar', null, null, array('baz' => 'bogus'));
}

public function barAction()
{
    // forward to an action in another controller:
    // FooController::bazAction(),
    // in the current module:
    $this->_forward('baz', 'foo', null, array('baz' => 'bogus'));
}

public function bazAction()
{
    // forward to an action in another controller in another module,
    // Foo_BarController::bazAction():
    $this->_forward('baz', 'bar', 'foo', array('baz' => 'bogus'));
}
```

Subclassing the Dispatcher

`Zend_Controller_Front` will first call the router to determine the first action in the request. It then enters a dispatch loop, which calls on the dispatcher to dispatch the action.

The dispatcher needs a variety of data in order to do its work - it needs to know how to format controller and action names, where to look for controller class files, whether or not a provided module name is valid, and an API for determining if a given request is even dispatchable based on the other information available.

`Zend_Controller_Dispatcher_Interface` defines the following methods as required for any dispatcher implementation:

```
interface Zend_Controller_Dispatcher_Interface
{
    /**
     * Format a string into a controller class name.
     *
     * @param string $unformatted
     * @return string
     */
    public function formatControllerName($unformatted);

    /**
     * Format a string into an action method name.
     *
     * @param string $unformatted
     * @return string
     */
    public function formatActionName($unformatted);

    /**
     * Determine if a request is dispatchable
     *
     * @param  Zend_Controller_Request_Abstract $request
     * @return boolean
     */
    public function isDispatchable(
        Zend_Controller_Request_Abstract $request
    );

    /**
     * Set a user parameter (via front controller, or for local use)
     *
     * @param string $name
     * @param mixed $value
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function setParam($name, $value);

    /**
     * Set an array of user parameters
     *
     * @param array $params
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function setParams(array $params);

    /**
```

```
     * Retrieve a single user parameter
     *
     * @param string $name
     * @return mixed
     */
    public function getParam($name);

    /**
     * Retrieve all user parameters
     *
     * @return array
     */
    public function getParams();

    /**
     * Clear the user parameter stack, or a single user parameter
     *
     * @param null|string|array single key or array of keys for
     *        params to clear
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function clearParams($name = null);

    /**
     * Set the response object to use, if any
     *
     * @param Zend_Controller_Response_Abstract|null $response
     * @return void
     */
    public function setResponse(
        Zend_Controller_Response_Abstract $response = null
    );

    /**
     * Retrieve the response object, if any
     *
     * @return Zend_Controller_Response_Abstract|null
     */
    public function getResponse();

    /**
     * Add a controller directory to the controller directory stack
     *
     * @param string $path
     * @param string $args
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function addControllerDirectory($path, $args = null);

    /**
     * Set the directory (or directories) where controller files are
     * stored
     *
     * @param string|array $dir
     * @return Zend_Controller_Dispatcher_Interface
     */
    public function setControllerDirectory($path);

    /**
     * Return the currently set directory(ies) for controller file
     * lookup
     *
     * @return array
     */
    public function getControllerDirectory();

    /**
     * Dispatch a request to a (module/)controller/action.
     *
     * @param  Zend_Controller_Request_Abstract $request
     * @param  Zend_Controller_Response_Abstract $response
     * @return Zend_Controller_Request_Abstract|boolean
     */
    public function dispatch(
        Zend_Controller_Request_Abstract $request,
        Zend_Controller_Response_Abstract $response
    );

    /**
     * Whether or not a given module is valid
     *
     * @param string $module
     * @return boolean
     */
    public function isValidModule($module);

    /**
     * Retrieve the default module name
     *
     * @return string
     */
    public function getDefaultModule();

    /**
     * Retrieve the default controller name
     *
     * @return string
     */
    public function getDefaultControllerName();

    /**
     * Retrieve the default action
     *
     * @return string
     */
    public function getDefaultAction();
}
```

In most cases, however, you should simply extend the abstract class `Zend_Controller_Dispatcher_Abstract`, in which each of these have already been defined, or `Zend_Controller_Dispatcher_Standard` to modify functionality of the standard dispatcher.

Possible reasons to subclass the dispatcher include a desire to use a different class or method naming schema in your action controllers, or a desire to use a different dispatching paradigm such as dispatching to action files under controller directories (instead of dispatching to class methods).

Action Controllers

Introduction

`Zend_Controller_Action` is an abstract class you may use for implementing Action Controllers for use with the Front Controller when building a website based on the Model-View-Controller (MVC) pattern.

To use `Zend_Controller_Action`, you will need to subclass it in your actual action controller classes (or subclass it to create your own base class for action controllers). The most basic operation is to subclass it, and create action methods that correspond to the various actions you wish the controller to handle for your site. Zend_Controller's routing and dispatch handling will autodiscover any methods ending in 'Action' in your class as potential controller actions.

For example, let's say your class is defined as follows:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // do something
    }

    public function bazAction()
    {
        // do something
    }
}
```

The above `FooController` class (controller `foo`) defines two actions, `bar` and `baz`.

There's much more that can be accomplished than this, such as custom initialization actions, default actions to call should no action (or an invalid action) be specified, pre- and post-dispatch hooks, and a variety of helper methods. This chapter serves as an overview of the action controller functionality

Default Behaviour

By default, the front controller enables the ViewRenderer action helper. This helper takes care of injecting the view object into the controller, as well as automatically rendering views. You may disable it within your action controller via one of the following methods:

```
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        // Local to this controller only; affects all actions,
        // as loaded in init:
        $this->_helper->viewRenderer->setNoRender(true);

        // Globally:
        $this->_helper->removeHelper('viewRenderer');

        // Also globally, but would need to be in conjunction with the
        // local version in order to propagate for this controller:
        Zend_Controller_Front::getInstance()
            ->setParam('noViewRenderer', true);
    }
}
```

`initView()`, `getViewScript()`, `render()`, and `renderScript()` each proxy to the `ViewRenderer` unless the helper is not in the helper broker or the `noViewRenderer` flag has been set.

You can also simply disable rendering for an individual view by setting the `ViewRenderer`'s `noRender` flag:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // disable autorendering for this action only:
        $this->_helper->viewRenderer->setNoRender();
    }
}
```

The primary reasons to disable the `ViewRenderer` are if you simply do not need a view object or if you are not rendering via view scripts (for instance, when using an action controller to serve web service protocols such as SOAP, XML-RPC, or REST). In most cases, you will never need to globally disable the `ViewRenderer`, only selectively within individual controllers or actions.

Object initialization

While you can always override the action controller's constructor, we do not recommend this. Zend_Controller_Action::__construct() performs some important tasks, such as registering the request and response objects, as well as any custom invocation arguments passed in from the front controller. If you must override the constructor, be sure to call `parent::__construct($request, $response, $invokeArgs)`.

The more appropriate way to customize instantiation is to use the `init()` method, which is called as the last task of `__construct()`. For example, if you want to connect to a database at instantiation:

```
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        $this->db = Zend_Db::factory('Pdo_Mysql', array(
            'host'     => 'myhost',
            'username' => 'user',
            'password' => 'XXXXXXX',
            'dbname'   => 'website'
        ));
    }
}
```

Pre- and Post-Dispatch Hooks

`Zend_Controller_Action` specifies two methods that may be called to bookend a requested action, `preDispatch()` and `postDispatch()`. These can be useful in a variety of ways: verifying authentication and ACLs prior to running an action (by calling `_forward()` in `preDispatch()`, the action will be skipped), for instance, or placing generated content in a sitewide template (`postDispatch()`).

Accessors

A number of objects and variables are registered with the object, and each has accessor methods.

- *Request Object*: `getRequest()` may be used to retrieve the request object used to call the action.
- *Response Object*: `getResponse()` may be used to retrieve the response object aggregating the final response. Some typical calls might look like:

  ```
  $this->getResponse()->setHeader('Content-Type', 'text/xml');
  $this->getResponse()->appendBody($content);
  ```

- *Invocation Arguments*: the front controller may push parameters into the router, dispatcher, and action controller. To retrieve these, use `getInvokeArg($key)`; alternatively, fetch the entire list using `getInvokeArgs()`.
- *Request parameters*: The request object aggregates request parameters, such as any _GET or _POST parameters, or user parameters specified in the URL's path information. To retrieve these, use `_getParam($key)` or `_getAllParams()`. You may also set request parameters using `_setParam()`; this is useful when forwarding to additional actions.

  To test whether or not a parameter exists (useful for logical branching), use `_hasParam($key)`.

Note

`_getParam()` may take an optional second argument containing a default value to use if the parameter is not set or is empty. Using it eliminates the need to call `_hasParam()` prior to retrieving a value:

```
// Use default value of 1 if id is not set
$id = $this->_getParam('id', 1);

// Instead of:
if ($this->_hasParam('id') {
    $id = $this->_getParam('id');
} else {
    $id = 1;
}
```

View Integration

`Zend_Controller_Action` provides a rudimentary and flexible mechanism for view integration. Two methods accomplish this, `initView()` and `render()`; the former method lazy-loads the `$view` public property, and the latter renders a view based on the current requested action, using the directory hierarchy to determine the script path.

View Initialization

`initView()` initializes the view object. `render()` calls `initView()` in order to retrieve the view object, but it may be initialized at any time; by default it populates the `$view` property with a `Zend_View` object, but any class implementing `Zend_View_Interface` may be used. If `$view` is already initialized, it simply returns that property.

The default implementation makes the following assumption of the directory structure:

```
applicationOrModule/
    controllers/
        IndexController.php
    views/
        scripts/
            index/
                index.phtml
        helpers/
        filters/
```

In other words, view scripts are assumed to be in the `views/scripts/` subdirectory, and the `views` subdirectory is assumed to contain sibling functionality (helpers, filters). When determining the view script name and path, the `views/scripts/` directory will be used as the base path, with a directories named after the individual controllers providing a hierarchy of view scripts.

Rendering Views

`render()` has the following signature:

```
string render(string $action = null,
              string $name = null,
              bool $noController = false);
```

`render()` renders a view script. If no arguments are passed, it assumes that the script requested is `[controller]/[action].phtml` (where `.phtml` is the value of the `$viewSuffix` property). Passing a value for `$action` will render that template in the `[controller]` subdirectory. To override using the `[controller]` subdirectory, pass a true value for `$noController`. Finally, templates are rendered into the response object; if you wish to render to a specific named segment in the response object, pass a value to `$name`.

Note

Since controller and action names may contain word delimiter characters such as '_', '.', and '-', render() normalizes these to '-' when determining the script name. Internally, it uses the dispatcher's word and path delimiters to do this normalization. Thus, a request to `/foo.bar/baz-bat` will render the script `foo-bar/baz-bat.phtml`. If your action method contains camelCasing, please remember that this will result in '-' separated words when determining the view script file name.

Some examples:

```
class MyController extends Zend_Controller_Action
{
    public function fooAction()
    {
        // Renders my/foo.phtml
        $this->render();

        // Renders my/bar.phtml
        $this->render('bar');

        // Renders baz.phtml
        $this->render('baz', null, true);

        // Renders my/login.phtml to the 'form' segment of the
        // response object
        $this->render('login', 'form');

        // Renders site.phtml to the 'page' segment of the response
        // object; does not use the 'my/' subirectory
        $this->render('site', 'page', true);
    }

    public function bazBatAction()
    {
        // Renders my/baz-bat.phtml
        $this->render();
    }
}
```

Utility Methods

Besides the accessors and view integration methods, `Zend_Controller_Action` has several utility methods for performing common tasks from within your action methods (or from pre-/post-dispatch).

- `_forward($action, $controller = null, $module = null, array $params = null)`: perform another action. If called in `preDispatch()`, the currently requested action will be skipped in favor of the new one. Otherwise, after the current action is processed, the action requested in _forward() will be executed.
- `_redirect($url, array $options = array())`: redirect to another location. This method takes a URL and an optional set of options. By default, it performs an HTTP 302 redirect.

  The options may include one or more of the following:

  - *exit:* whether or not to exit immediately. If requested, it will cleanly close any open sessions and perform the redirect.

    You may set this option globally within the controller using the `setRedirectExit()` accessor.

  - *prependBase:* whether or not to prepend the base URL registered with the request object to the URL provided.

    You may set this option globally within the controller using the `setRedirectPrependBase()` accessor.

  - *code:* what HTTP code to utilize in the redirect. By default, an HTTP 302 is utilized; any code between 301 and 306 may be used.

    You may set this option globally within the controller using the `setRedirectCode()` accessor.

Subclassing the Action Controller

By design, `Zend_Controller_Action` must be subclassed in order to create an action controller. At the minimum, you will need to define action methods that the controller may call.

Besides creating useful functionality for your web applications, you may also find that you're repeating much of the same setup or utility methods in your various controllers; if so, creating a common base controller class that extends `Zend_Controller_Action` could solve such redundancy.

Example 8.1. How to Handle Non-Existent Actions

If a request to a controller is made that includes an undefined action method, `Zend_Controller_Action::__call()` will be invoked. `__call()` is, of course, PHP's magic method for method overloading.

By default, this method throws a `Zend_Controller_Action_Exception` indicating the requested method was not found in the controller. If the method requested ends in 'Action', the assumption is that an action was requested and does not exist; such errors result in an exception with a code of 404. All other methods result in an exception with a code of 500. This allows you to easily differentiate between page not found and application errors in your error handler.

You should override this functionality if you wish to perform other operations. For instance, if you wish to display an error message, you might write something like this:

```
class MyController extends Zend_Controller_Action
{
    public function __call($method, $args)
    {
        if ('Action' == substr($method, -6)) {
            // If the action method was not found, render the error
            // template
            return $this->render('error');
        }

        // all other methods throw an exception
        throw new Exception('Invalid method "'
                            . $method
                            . '" called',
                            500);
    }
}
```

Another possibility is that you may want to forward on to a default controller page:

```
class MyController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->render();
    }

    public function __call($method, $args)
    {
        if ('Action' == substr($method, -6)) {
            // If the action method was not found, forward to the
            // index action
            return $this->_forward('index');
        }

        // all other methods throw an exception
        throw new Exception('Invalid method "'
                            . $method
                            . '" called',
                            500);
    }
}
```

Besides overriding `__call()`, each of the initialization, utility, accessor, view, and dispatch hook methods mentioned previously in this chapter may be overridden in order to customize your controllers. As an example, if you are storing your view object in a registry, you may want to modify your `initView()` method with code resembling the following:

```
abstract class My_Base_Controller extends Zend_Controller_Action
{
    public function initView()
    {
        if (null === $this->view) {
            if (Zend_Registry::isRegistered('view')) {
                $this->view = Zend_Registry::get('view');
            } else {
                $this->view = new Zend_View();
                $this->view->setBasePath(dirname(__FILE__) . '/../views');
            }
        }

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

Hopefully, from the information in this chapter, you can see the flexibility of this particular component and how you can shape it to your application's or site's needs.

Action Helpers

Introduction

Action Helpers allow developers to inject runtime and/or on-demand functionality into any Action Controllers that extend Zend_Controller_Action. Action Helpers aim to minimize the necessity to extend the abstract Action Controller in order to inject common Action Controller functionality.

There are a number of ways to use Action Helpers. Action Helpers employ the use of a brokerage system, similar to the types of brokerage you see in Zend_View_Helper, and that of Zend_Controller_Plugin. Action Helpers (like `Zend_View_Helper`) may be loaded and called on demand, or they may be instantiated at request time (bootstrap) or action controller creation time (init()). To understand this more fully, please see the usage section below.

Helper Initialization

A helper can be initialized in several different ways, based on your needs as well as the functionality of that helper.

The helper broker is stored as the `$_helper` member of `Zend_Controller_Action`; use the broker to retrieve or call on helpers. Some methods for doing so include:

- Explicitly using `getHelper()`. Simply pass it a name, and a helper object is returned:

  ```
  $flashMessenger = $this->_helper->getHelper('FlashMessenger');
  $flashMessenger->addMessage('We did something in the last request');
  ```

- Use the helper broker's `__get()` functionality and retrieve the helper as if it were a member property of the broker:

  ```
  $flashMessenger = $this->_helper->FlashMessenger;
  $flashMessenger->addMessage('We did something in the last request');
  ```

- Finally, most action helpers implement the method `direct()` which will call a specific, default method in the helper. In the example of the `FlashMessenger`, it calls `addMessage()`:

  ```
  $this->_helper->FlashMessenger('We did something in the last request');
  ```

Note

All of the above examples are functionally equivalent.

You may also instantiate helpers explicitly. You may wish to do this if using the helper outside of an action controller, or if you wish to pass a helper to the helper broker for use by any action. Instantiation is as per any other PHP class.

The Helper Broker

`Zend_Controller_Action_HelperBroker` handles the details of registering helper objects and helper paths, as well as retrieving helpers on-demand.

To register a helper with the broker, use `addHelper`:

```
Zend_Controller_Action_HelperBroker::addHelper($helper);
```

Of course, instantiating and passing helpers to the broker is a bit time and resource intensive, so two methods exists to automate things slightly: `addPrefix()` and `addPath()`.

- `addPrefix()` takes a class prefix and uses it to determine a path where helper classes have been defined. It assumes the prefix follows Zend Framework class naming conventions.

  ```
  // Add helpers prefixed with My_Action_Helpers in My/Action/Helpers/
  Zend_Controller_Action_HelperBroker::addPrefix('My_Action_Helpers');
  ```

- `addPath()` takes a directory as its first argument and a class prefix as the second argument (defaulting to 'Zend_Controller_Action_Helper'). This allows you to map your own class prefixes to specific directories.

  ```
  // Add helpers prefixed with Helper in Plugins/Helpers/
  Zend_Controller_Action_HelperBroker::addPath('./Plugins/Helpers',
                                               'Helper');
  ```

Since these methods are static, they may be called at any point in the controller chain in order to dynamically add helpers as needed.

To determine if a helper exists in the helper broker, use `hasHelper($name)`, where `$name` is the short name of the helper (minus the prefix):

```
// Check if 'redirector' helper is registered with the broker:
if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
    echo 'Redirector helper registered';
}
```

There are also two static methods for retrieving helpers from the helper broker: `getExistingHelper()` and `getStaticHelper()`. `getExistingHelper()` will retrieve a helper only if it has previously been invoked by or explicitly registered with the helper broker; it will throw an exception if not. `getStaticHelper()` does the same as `getExistingHelper()`, but will attempt to instantiate the helper if has not yet been registered with the helper stack. `getStaticHelper()` is a good choice for retrieving helpers which you wish to configure.

Both methods take a single argument, `$name`, which is the short name of the helper (minus the prefix).

```
// Check if 'redirector' helper is registered with the broker, and fetch:
if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
    $redirector =
        Zend_Controller_Action_HelperBroker::getExistingHelper('redirector');
}
```

```
// Or, simply retrieve it, not worrying about whether or not it was
// previously registered:
$redirector =
    Zend_Controller_Action_HelperBroker::getStaticHelper('redirector');
}
```

Finally, to delete a registered helper from the broker, use `removeHelper($name)`, where `$name` is the short name of the helper (minus the prefix):

```
// Conditionally remove the 'redirector' helper from the broker:
if (Zend_Controller_Action_HelperBroker::hasHelper('redirector')) {
    Zend_Controller_Action_HelperBroker::removeHelper('redirector')
}
```

Built-in Action Helpers

Zend Framework includes several action helpers by default: `AutoComplete` for automating responses for AJAX autocompletion; `ContextSwitch` and `AjaxContext` for serving alternate response formats for your actions; a `FlashMessenger` for handling session flash messages; `Json` for encoding and sending JSON responses; a `Redirector`, to provide different implementations for redirecting to internal and external pages from your application; and a `ViewRenderer` to automate the process of setting up the view object in your controllers and rendering views.

ActionStack

The `ActionStack` helper allows you to push requests to the ActionStack front controller plugin, effectively helping you create a queue of actions to execute during the request. The helper allows you to add actions either by specifying new request objects or action/controller/module sets.

Invoking ActionStack helper initializes ActionStack Plugin

Invoking the `ActionStack` helper implicitly registers the `ActionStack` plugin -- which means you do not need to explicitly register the `ActionStack` plugin to use this functionality.

Example 8.2. Adding a task using action, controller and module names

Often, it's simplest to simply specify the action, controller, and module (and optional request parameters), much as you would when calling `Zend_Controller_Action::_forward()`:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Add two actions to the stack
        // Add call to /foo/baz/bar/baz
        // (FooController::bazAction() with request var bar == baz)
        $this->_helper->actionStack('baz',
                                    'foo',
                                    'default',
                                    array('bar' => 'baz'));

        // Add call to /bar/bat
        // (BarController::batAction())
        $this->_helper->actionStack('bat', 'bar');
    }
}
```

Example 8.3. Adding a task using a request object

Sometimes the OOP nature of a request object makes most sense; you can pass such an object to the `ActionStack` helper as well.

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Add two actions to the stack
        // Add call to /foo/baz/bar/baz
        // (FooController::bazAction() with request var bar == baz)
        $request = clone $this->getRequest();
        // Don't set controller or module; use current values
        $request->setActionName('baz')
                ->setParams(array('bar' => 'baz'));
        $this->_helper->actionStack($request);

        // Add call to /bar/bat
        // (BarController::batAction())
        $request = clone $this->getRequest();
        // don't set module; use current value
        $request->setActionName('bat')
                ->setControllerName('bar');
        $this->_helper->actionStack($request);
    }
}
```

AutoComplete

Many AJAX javascript libraries offer functionality for providing autocompletion whereby a selectlist of potentially matching results is displayed as the user types. The `AutoComplete` helper aims to simplify returning acceptable responses to such methods.

Since not all JS libraries implement autocompletion in the same way, the `AutoComplete` helper provides some abstract base functionality necessary to many libraries, and concrete implementations for individual libraries. Return types are generally either JSON arrays of strings, JSON arrays of arrays (with each member array being an associative array of metadata used to create the selectlist), or HTML.

Basic usage for each implementation is the same:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Perform some logic...

        // Encode and send response;
        $this->_helper->autoCompleteDojo($data);
        // Or explicitly:
        $response = $this->_helper->autoCompleteDojo
                                  ->sendAutoCompletion($data);

        // Or simply prepare autocompletion response:
        $response = $this->_helper->autoCompleteDojo
                                  ->prepareAutoCompletion($data);
    }
}
```

By default, autocompletion does the following:

- Disables layouts and ViewRenderer.
- Sets appropriate response headers.
- Sets response body with encoded/formatted autocompletion data.
- Sends response.

Available methods of the helper include:

- `disableLayouts()` can be used to disable layouts and the ViewRenderer. Typically, this is called within `prepareAutoCompletion()`.
- `encodeJson($data, $keepLayouts = false)` will encode data to JSON, optionally enabling or disabling layouts. Typically, this is called within `prepareAutoCompletion()`.
- `prepareAutoCompletion($data, $keepLayouts = false)` is used to prepare data in the response format necessary for the concrete implementation, optionally enabling or disabling layouts. The return value will vary based on the implementation.
- `sendAutoCompletion($data, $keepLayouts = false)` is used to send data in the response format necessary for the concrete implementation. It calls `prepareAutoCompletion()`, and then sends the response.
- `direct($data, $sendNow = true, $keepLayouts = false)` is used when calling the helper as a method of the helper broker. The `$sendNow` flag is used to determine whether to call `sendAutoCompletion()` or `prepareAutoCompletion()`, respectively.

Currently, `AutoComplete` supports the Dojo and Scriptaculous AJAX libraries.

AutoCompletion with Dojo

Dojo does not have an AutoCompletion widget per se, but has two widgets that can perform AutoCompletion: ComboBox and FilteringSelect. In both cases, they require a data store that implements the QueryReadStore; for more information on these topics, see the dojo.data [http://dojotoolkit.org/book/dojo-book-0-9/part-3-programmatic-dijit-and-dojo/data-retrieval-dojo-data-0] documentation.

In Zend Framework, you can pass a simple indexed array to the AutoCompleteDojo helper, and it will return a JSON response suitable for use with such a store:

```
// within a controller action:
$this->_helper->autoCompleteDojo($data);
```

Example 8.4. AutoCompletion with Dojo Using Zend MVC

AutoCompletion with Dojo via the Zend MVC requires several things: generating a form object for the ComboBox on which you want AutoCompletion, a controller action for serving the AutoCompletion results, creating a custom QueryReadStore to connect to the AutoCompletion action, and generation of the javascript to use to initialize AutoCompletion on the server side.

First, let's look at the javascript necessary. Dojo offers a complete framework for creating OOP javascript, much as Zend Framework does for PHP. Part of that is the ability to create pseudo-namespaces using the directory hierarchy. We'll create a 'custom' directory at the same level as the Dojo directory that's part of the Dojo distribution. Inside that directory, we'll create a javascript file, TestNameReadStore.js, with the following contents:

```
dojo.provide("custom.TestNameReadStore");
dojo.declare("custom.TestNameReadStore", dojox.data.QueryReadStore, {
    fetch:function (request) {
        request.serverQuery = { test:request.query.name };
        return this.inherited("fetch", arguments);
    }
});
```

This class is simply an extension of Dojo's own QueryReadStore, which is itself an abstract class. We simply define a method by which to request, and assigning it to the 'test' element.

Next, let's create the form element for which we want AutoCompletion:

```
class TestController extends Zend_Controller_Action
{
    protected $_form;

    public function getForm()
    {
        if (null === $this->_form) {
            $this->_form = new Zend_Form();
            $this->_form->setMethod('get')
                ->setAction(
                    $this->getRequest()->getBaseUrl() . '/test/process'
                )
                ->addElements(array(
                    'test' => array('type' => 'text', 'options' => array(
                        'filters'        => array('StringTrim'),
                        'dojoType'       => array('dijit.form.ComboBox'),
                        'store'          => 'testStore',
                        'autoComplete'   => 'false',
                        'hasDownArrow'   => 'true',
                        'label' => 'Your input:',
                    )),
                    'go' => array('type' => 'submit',
                                  'options' => array('label' => 'Go!'))
                ));
        }
        return $this->_form;
    }
}
```

Here, we simply create a form with 'test' and 'go' methods. The 'test' method adds several special, Dojo-specific attributes: dojoType, store, autoComplete, and hasDownArrow. The dojoType is used to indicate that we are creating a ComboBox, and we will link it to a data store (key 'store') of 'testStore' -- more on that later. Specifying 'autoComplete' as false tells Dojo not to automatically select the first match, but instead show a list of matches. Finally, 'hasDownArrow' creates a down arrow similar to a select box so we can show and hide the matches.

Let's add a method to display the form, as well as an end point for processing AutoCompletion:

```
class TestController extends Zend_Controller_Action
{
    // ...

    /**
     * Landing page
     */
    public function indexAction()
    {
        $this->view->form = $this->getForm();
    }

    public function autocompleteAction()
    {
        if ('ajax' != $this->_getParam('format', false)) {
            return $this->_helper->redirector('index');
        }
        if ($this->getRequest()->isPost()) {
            return $this->_helper->redirector('index');
        }

        $match = trim($this->getRequest()->getQuery('test', ''));

        $matches = array();
        foreach ($this->getData() as $datum) {
            if (0 === strpos($datum, $match)) {
                $matches[] = $datum;
            }
        }
        $this->_helper->autoCompleteDojo($matches);
    }
}
```

In our `autocompleteAction()` we do a number of things. First, we look to make sure we have a post request, and that there is a 'format' parameter set to the value 'ajax'; these are simply to help reduce spurious queries to the action. Next, we check for a 'test' parameter, and compare it against our data. (I purposely leave out the implementation of `getData()` here -- it could be any sort of data source.) Finally, we send our matches to our AutoCompletion helper.

Now that we have all the pieces on the backend, let's look at what we need to deliver in our view script for the landing page. First, we need to setup our data store, then render our form, and finally ensure that the appropriate Dojo libraries -- including our custom data store -- get loaded. Let's look at the view script, which comments the steps:

```
<? // setup our data store: ?>
<div dojoType="custom.TestNameReadStore" jsId="testStore"
    url="<?= $this->baseUrl() ?>/unit-test/autocomplete/format/ajax"
    requestMethod="get"></div>

<? // render our form: ?>
<?= $this->form ?>

<? // setup Dojo-related CSS to load in HTML head: ?>
<? $this->headStyle()->captureStart() ?>
@import "<?= $this->baseUrl() ?>/javascript/dijit/themes/tundra/tundra.css";
@import "<?= $this->baseUrl() ?>/javascript/dojo/resources/dojo.css";
<? $this->headStyle()->captureEnd() ?>

<? // setup javascript to load in HTML head, including all required
   // Dojo libraries: ?>
<? $this->headScript()
        ->setAllowArbitraryAttributes(true)
        ->appendFile($this->baseUrl() . '/javascript/dojo/dojo.js',
            'text/javascript',
            array('djConfig' => 'parseOnLoad: true'))
        ->captureStart() ?>
djConfig.usePlainJson=true;
dojo.registerModulePath("custom","../custom");
dojo.require("dojo.parser");
dojo.require("dojox.data.QueryReadStore");
dojo.require("dijit.form.ComboBox");
dojo.require("custom.TestNameReadStore");
<? $this->headScript()->captureEnd() ?>
```

Note the calls to view helpers such as headStyle and headScript; these are placeholders, which we can then render in the HTML head section of our layout view script.

We now have all the pieces to get Dojo AutoCompletion working.

AutoCompletion with Scriptaculous

Scriptaculous [http://wiki.script.aculo.us/scriptaculous/show/Ajax.Autocompleter] expects an HTML response in a specific format.

The helper to use with this library is 'AutoCompleteScriptaculous'. Simply provide it an array of data, and the helper will create an HTML response compatible with Ajax.Autocompleter.

ContextSwitch and AjaxContext

The `ContextSwitch` action helper is intended for facilitating returning different response formats on request. The `AjaxContext` helper is a specialized version of `ContextSwitch` that facilitates returning responses to XmlHttpRequests.

To enable either one, you must provide hinting in your controller as to what actions can respond to which contexts. If an incoming request indicates a valid context for the given action, the helper will then:

- Disable layouts, if enabled.
- Set an alternate view suffix, effectively requiring a separate view script for the context.
- Send appropriate response headers for the context desired.
- Optionally, call specified callbacks to setup the context and/or perform post-processing.

As an example, let's consider the following controller:

```
class NewsController extends Zend_Controller_Action
{
    /**
     * Landing page; forwards to listAction()
     */
    public function indexAction()
    {
        $this->_forward('list');
    }

    /**
     * List news items
     */
    public function listAction()
    {
    }

    /**
     * View a news item
     */
    public function viewAction()
    {
    }
}
```

Let's say that we want the `listAction()` to also be available in an XML format. Instead of creating a different action, we can hint that it can return an XML response:

```
class NewsController extends Zend_Controller_Action
```

```
{
    public function init()
    {
        $contextSwitch = $this->_helper->getHelper('contextSwitch');
        $contextSwitch->addActionContext('list', 'xml')
                      ->initContext();
    }

    // ...
}
```

What this will do is:

- Set the 'Content-Type' response header to 'text/xml'.
- Change the view suffix to 'xml.phtml' (or, if you use an alternate view suffix, 'xml.[your suffix]').

Now, you'll need to create a new view script, 'news/list.xml.phtml', which will create and render the XML.

To determine if a request should initiate a context switch, the helper checks for a token in the request object. By default, it looks for the 'format' parameter, though this may be configured. This means that, in most cases, to trigger a context switch, you can add a 'format' parameter to your request:

- Via URL parameter: `/news/list/format/xml` (recall, the default routing schema allows for arbitrary key/value pairs following the action)
- Via GET parameter: `/news/list?format=xml`

`ContextSwitch` allows you to specify arbitrary contexts, including what suffix change will occur (if any), any response headers that should be sent, and arbitrary callbacks for initialization and post processing.

Default Contexts Available

By default, two contexts are available to the `ContextSwitch` helper: json and xml.

- *JSON*. The JSON context sets the 'Content-Type' response header to 'application/json', and the view script suffix to 'json.phtml'.

  By default, however, no view script is required. It will simply serialize all view variables, and emit the JSON response immediately.

  This behaviour can be disabled by turning off auto-JSON serialization:

  ```
  $this->_helper->contextSwitch()->setAutoJsonSerialization(false);
  ```

- *XML*. The XML context sets the 'Content-Type' response header to 'text/xml', and the view script suffix to 'xml.phtml'. You will need to create a new view script for the context.

Creating Custom Contexts

Sometimes, the default contexts are not enough. For instance, you may wish to return YAML, or serialized PHP, an RSS or ATOM feed, etc. `ContextSwitch` allows you to do so.

The easiest way to add a new context is via the `addContext()` method. This method takes two arguments, the name of the context, and an array specification. The specification should include one or more of the following:

- *suffix*: the suffix to prepend to the default view suffix as registered in the ViewRenderer.
- *headers*: an array of header/value pairs you wish sent as part of the response.
- *callbacks*: an array containing one or more of the keys 'init' or 'post', pointing to valid PHP callbacks that can be used for context initialization and post processing.

  Initialization callbacks occur when the context is detected by `ContextSwitch`. You can use it to perform arbitrary logic that should occur. As an example, the JSON context uses a callback to disable the ViewRenderer when auto-JSON serialization is on.

  Post processing occurs during the action's `postDispatch()` routine, and can be used to perform arbitrary logic. As an example, the JSON context uses a callback to determine if auto-JSON serialization is on; if so, it serializes the view variables to JSON and sends the response, but if not, it re-enables the ViewRenderer.

There are a variety of methods for interacting with contexts:

- `addContext($context, array $spec)`: add a new context. Throws an exception if the context already exists.
- `setContext($context, array $spec)`: add a new context or overwrite an existing context. Uses the same specification as `addContext()`.
- `addContexts(array $contexts)`: add many contexts at once. The `$contexts` array should be an array of context/specification pairs. If any of the contexts already exists, it will throw an exception.
- `setContexts(array $contexts)`: add new contexts and overwrite existing ones. Uses the same specification as `addContexts()`.
- `hasContext($context)`: returns true if the context exists, false otherwise.
- `getContext($context)`: retrieve a single context by name. Returns an array following the specification used in `addContext()`.
- `getContexts()`: retrieve all contexts. Returns an array of context/specification pairs.
- `removeContext($context)`: remove a single context by name. Returns true if successful, false if the context was not found.
- `clearContexts()`: remove all contexts.

Setting Contexts Per Action

There are two mechanisms for setting available contexts. You can either manually create arrays in your controller, or use several methods in `ContextSwitch` to assemble them.

The principle method for adding action/context relations is `addActionContext()`. It expects two arguments, the action to which the context is being added, and either the name of a context or an array of contexts. As an example, consider the following controller class:

```
class FooController extends Zend_Controller_Action
{
    public function listAction()
    {
    }

    public function viewAction()
    {
    }

    public function commentsAction()
    {
    }

    public function updateAction()
    {
    }
}
```

Let's say we wanted to add an XML context to the 'list' action, and XML and JSON contexts to the 'comments' action. We could do so in the `init()` method:

```
class FooController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->contextSwitch()
             ->addActionContext('list', 'xml')
             ->addActionContext('comments', array('xml', 'json'))
             ->initContext();
    }
}
```

Alternately, you could simply define the array property `$contexts`:

```
class FooController extends Zend_Controller_Action
{
    public $contexts = array(
        'list'     => array('xml'),
        'comments' => array('xml', 'json')
    );

    public function init()
    {
        $this->_helper->contextSwitch()->initContext();
    }
}
```

The above is less overhead, but also prone to potential errors.

The following methods can be used to build the context mappings:

- `addActionContext($action, $context)`: marks one or more contexts as available to an action. If mappings already exists, simply appends to those mappings. `$context` may be a single context, or an array of contexts.

  A value of `true` for the context will mark all available contexts as available for the action.

  An empty value for $context will disable all contexts for the given action.

- `setActionContext($action, $context)`: marks one or more contexts as available to an action. If mappings already exists, it replaces them with those specified. `$context` may be a single context, or an array of contexts.

- `addActionContexts(array $contexts)`: add several action/context pairings at once. `$contexts` should be an associative array of action/context pairs. It proxies to `addActionContext()`, meaning that if pairings already exist, it appends to them.

- `setActionContexts(array $contexts)`: acts like `addActionContexts()`, but overwrites existing action/context pairs.

- `hasActionContext($action, $context)`: determine if a particular action has a given context.

- `getActionContexts($action = null)`: returns either all contexts for a given action, or all action/context pairs.

- `removeActionContext($action, $context)`: remove one or more contexts from a given action. `$context` may be a single context or an array of contexts.

- `clearActionContexts($action = null)`: remove all contexts from a given action, or from all actions with contexts.

Initializing Context Switching

To initialize context switching, you need to call `initContext()` in your action controller:

```
class NewsController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->contextSwitch()->initContext();
    }
}
```

In some cases, you may want to force the context used; for instance, you may only want to allow the XML context if context switching is activated. You can do so by passing the context to `initContext()`:

```
$contextSwitch->initContext('xml');
```

Additional Functionality

A variety of methods can be used to alter the behaviour of the `ContextSwitch` helper. These include:

- `setAutoJsonSerialization($flag)`: By default, JSON contexts will serialize any view variables to JSON notation and return this as a response. If you wish to create your own response, you should turn this off; this needs to be done prior to the call to `initContext()`.

  ```
  $contextSwitch->setAutoJsonSerialization(false);
  $contextSwitch->initContext();
  ```

  You can retrieve the value of the flag with `getAutoJsonSerialization()`.

- `setSuffix($context, $suffix, $prependViewRendererSuffix)`: With this method, you can specify a different suffix to use for a given context. The third argument is used to indicate whether or not to prepend the current ViewRenderer suffix with the new suffix; this flag is enabled by default.

  Passing an empty value to the suffix will cause only the ViewRenderer suffix to be used.

- `addHeader($context, $header, $content)`: Add a response header for a given context. `$header` is the header name, and `$content` is the value to pass for that header.

  Each context can have multiple headers; `addHeader()` adds additional headers to the context's header stack.

  If the `$header` specified already exists for the context, an exception will be thrown.

- `setHeader($context, $header, $content)`: `setHeader()` acts just like `addHeader()`, except it allows you to overwrite existing context headers.

- `addHeaders($context, array $headers)`: Add multiple headers at once to a given context. Proxies to `addHeader()`, so if the header already exists, an exception will be thrown. `$headers` is an array of header/context pairs.

- `setHeaders($context, array $headers.)`: like `addHeaders()`, except it proxies to `setHeader()`, allowing you to overwrite existing headers.

- `getHeader($context, $header)`: retrieve the value of a header for a given context. Returns null if not found.

- `removeHeader($context, $header)`: remove a single header for a given context.

- `clearHeaders($context, $header)`: remove all headers for a given context.

- `setCallback($context, $trigger, $callback)`: set a callback at a given trigger for a given context. Triggers may be either 'init' or 'post' (indicating callback will be called at either context initialization or postDispatch). `$callback` should be a valid PHP callback.
- `setCallbacks($context, array $callbacks)`: set multiple callbacks for a given context. `$callbacks` should be trigger/callback pairs. In actuality, the most callbacks that can be registered are two, one for initialization and one for post processing.
- `getCallback($context, $trigger)`: retrieve a callback for a given trigger in a given context.
- `getCallbacks($context)`: retrieve all callbacks for a given context. Returns an array of trigger/callback pairs.
- `removeCallback($context, $trigger)`: remove a callback for a given trigger and context.
- `clearCallbacks($context)`: remove all callbacks for a given context.
- `setContextParam($name)`: set the request parameter to check when determining if a context switch has been requested. The value defaults to 'format', but this accessor can be used to set an alternate value.

  `getContextParam()` can be used to retrieve the current value.
- `setAutoDisableLayout($flag)`: By default, layouts are disabled when a context switch occurs; this is because typically layouts will only be used for returning normal responses, and have no meaning in alternate contexts. However, if you wish to use layouts (perhaps you may have a layout for the new context), you can change this behaviour by passing a false value to `setAutoDisableLayout()`. You should do this *before* calling `initContext()`.

  To get the value of this flag, use the accessor `getAutoDisableLayout()`.
- `getCurrentContext()` can be used to determine what context was detected, if any. This returns null if no context switch occurred, or if called before `initContext()` has been invoked.

AjaxContext Functionality

The `AjaxContext` helper extends `ContextSwitch`, so all of the functionality listed for `ContextSwitch` is available to it. There are a few key differences, however.

First, it uses a different action controller property for determining contexts, `$ajaxable`. This is so you can have different contexts used for AJAX versus normal HTTP requests. The various `*ActionContext*()` methods of `AjaxContext` will write to this property.

Second, it will only trigger if an XmlHttpRequest has occurred, as determined by the request object's `isXmlHttpRequest()` method. Thus, if the context parameter ('format') is passed in the request, but the request was not made as an XmlHttpRequest, no context switch will trigger.

Third, `AjaxContext` adds an additional context, HTML. In this context, it sets the suffix to 'ajax.phtml' in order to differentiate the context from a normal request. No additional headers are returned.

Example 8.5. Allowing Actions to Respond To Ajax Requests

In this following example, we're allowing requests to the actions 'view', 'form', and 'process' to respond to AJAX requests. In the first two cases, 'view' and 'form', we'll return HTML snippets with which to update the page; in the latter, we'll return JSON.

```
class CommentController extends Zend_Controller_Action
{
    public function init()
    {
        $ajaxContext = $this->_helper->getHelper('AjaxContext');
        $ajaxContext->addActionContext('view', 'html')
                    ->addActionContext('form', 'html')
                    ->addActionContext('process', 'json')
                    ->initContext();
    }

    public function viewAction()
    {
        // Pull a single comment to view.
        // When AjaxContext detected, uses the comment/view.ajax.phtml
        // view script.
    }

    public function formAction()
    {
        // Render the "add new comment" form.
        // When AjaxContext detected, uses the comment/form.ajax.phtml
        // view script.
    }

    public function processAction()
    {
        // Process a new comment
        // Return the results as JSON; simply assign the results as
        // view variables, and JSON will be returned.
    }
}
```

On the client end, your AJAX library will simply request the endpoints '/comment/view', '/comment/form', and '/comment/process', and pass the 'format' parameter: '/comment/view/format/html', '/comment/form/format/html', '/comment/process/format/json'. (Or you can pass the parameter via query string: e.g., "?format=json".)

Assuming your library passes the 'X-Requested-With: XmlHttpRequest' header, these actions will then return the appropriate response format.

FlashMessenger

Introduction

The `FlashMessenger` helper allows you to pass messages that the user may need to see on the next request. To accomplish this, `FlashMessenger` uses `Zend_Session_Namespace` to store messages for future or next request retrieval. It is generally a good idea that if you plan on using `Zend_Session` or `Zend_Session_Namespace`, that you initialize with `Zend_Session::start()` in your bootstrap file. (See the Zend_Session documentation for more details on its usage.)

Basic Usage Example

The usage example below shows the use of the flash messenger at its most basic. When the action `/some/my` is called, it adds the flash message "Record Saved!" A subsequent request to the action `/some/my-next-request` will retrieve it (and thus delete it as well).

```
class SomeController extends Zend_Controller_Action
{
    /**
     * FlashMessenger
     *
     * @var Zend_Controller_Action_Helper_FlashMessenger
     */
    protected $_flashMessenger = null;

    public function init()
    {
        $this->_flashMessenger =
            $this->_helper->getHelper('FlashMessenger');
        $this->initView();
    }

    public function myAction()
    {
        /**
         * default method of getting
         * Zend_Controller_Action_Helper_FlashMessenger instance
         * on-demand
         */
        $this->_flashMessenger->addMessage('Record Saved!');
    }

    public function myNextRequestAction()
    {
        $this->view->messages = $this->_flashMessenger->getMessages();
        $this->render();
    }
}
```

JSON

JSON responses are rapidly becoming the response of choice when dealing with AJAX requests that expect dataset responses; JSON can be immediately parsed on the client-side, leading to quick execution.

The JSON action helper does several things:

- Disables layouts if currently enabled.
- Disables the ViewRenderer if currently enabled.
- Sets the 'Content-Type' response header to 'application/json'.
- By default, immediately returns the response, without waiting for the action to finish execution.

Usage is simple: either call it as a method of the helper broker, or call one of the methods `encodeJson()` or `sendJson()`:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // do some processing...
        // Send the JSON response:
        $this->_helper->json($data);

        // or...
        $this->_helper->json->sendJson($data);

        // or retrieve the json:
        $json = $this->_helper->json->encodeJson($data);
    }
}
```

Keeping Layouts

If you have a separate layout for JSON responses -- perhaps to wrap the JSON response in some sort of context -- each method in the JSON helper accepts a second, optional argument: a flag to enable or disable layouts. Passing a boolean `true` value will keep layouts enabled:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // Retrieve the json, keeping layouts:
        $json = $this->_helper->json->encodeJson($data, true);
    }
}
```

Redirector

Introduction

The `Redirector` helper allows you to use a redirector object to fulfill your application's needs for redirecting to a new URL. It provides numerous benefits over the `_redirect()` method, such as being able to preconfigure sitewide behavior into the redirector object or using the built in `gotoSimple($action, $controller, $module, $params)` interface similar to that of `Zend_Controller_Action::_forward()`.

The `Redirector` has a number of methods that can be used to affect the behaviour at redirect:

- `setCode()` can be used to set the HTTP response code to use during the redirect.
- `setExit()` can be used to force an `exit()` following a redirect. By default this is true.
- `setGotoSimple()` can be used to set a default URL to use if none is passed to `gotoSimple()`. Uses the API of `Zend_Controller_Action::_forward()`: setGotoSimple($action, $controller = null, $module = null, array $params = array());
- `setGotoRoute()` can be used to set a URL based on a registered route. Pass in an array of key/value pairs and a route name, and it will assemble the URL according to the route type and definition.
- `setGotoUrl()` can be used to set a default URL to use if none is passed to `gotoUrl()`. Accepts a single URL string.
- `setPrependBase()` can be used to prepend the request object's base URL to a URL specified with `setGotoUrl()`, `gotoUrl()`, or `gotoUrlAndExit()`.
- `setUseAbsoluteUri()` can be used to force the `Redirector` to use absolute URIs when redirecting. When this option is set, it uses the value of `$_SERVER['HTTP_HOST']`, `$_SERVER['SERVER_PORT']`, and `$_SERVER['HTTPS']` to form a full URI to the URL specified by one of the redirect methods. This option is off by default, but may be enabled by default in later releases.

Additionally, there are a variety of methods in the redirector for performing the actual redirects:

- `gotoSimple()` uses `setGotoSimple()` (`_forward()-like API`) to build a URL and perform a redirect.
- `gotoRoute()` uses `setGotoRoute()` (`route-assembly`) to build a URL and perform a redirect.
- `gotoUrl()` uses `setGotoUrl()` (`URL string`) to build a URL and perform a redirect.

Finally, you can determine the current redirect URL at any time using `getRedirectUrl()`.

Basic Usage Examples

Example 8.6. Setting Options

This example overrides several options, including setting the HTTP status code to use in the redirect ('303'), not defaulting to exit on redirect, and defining a default URL to use when redirecting.

```
class SomeController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');

        // Set the default options for the redirector
        // Since the object is registered in the helper broker, these
        // become relevant for all actions from this point forward
        $this->_redirector->setCode('303')
                          ->setExit(false)
                          ->setGotoSimple("this-action",
                                          "some-controller");
    }

    public function myAction()
    {
        /* do some stuff */

        // Redirect to a previously registered URL, and force an exit
        // to occur when done:
        $this->_redirector->redirectAndExit();
        return; // never reached
    }
}
```

Example 8.7. Using Defaults

This example assumes that the defaults are used, which means that any redirect will result in an immediate `exit()`.

```
// ALTERNATIVE EXAMPLE
class AlternativeController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }

    public function myAction()
    {
        /* do some stuff */

        $this->_redirector
            ->gotoUrl('/my-controller/my-action/param1/test/param2/test2');
        return; // never reached since default is to goto and exit
    }
}
```

Example 8.8. Using goto()'s _forward() API

`gotoSimple()`'s API mimics that of `Zend_Controller_Action::_forward()`. The primary difference is that it builds a URL from the parameters passed, and using the default `:module/:controller/:action/*` format of the default router. It then redirects instead of chaining the action.

```
class ForwardController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }

    public function myAction()
    {
        /* do some stuff */

        // Redirect to 'my-action' of 'my-controller' in the current
        // module, using the params param1 => test and param2 => test2
        $this->_redirector->gotoSimple('my-action',
                                       'my-controller',
                                       null,
                                       array('param1' => 'test',
                                             'param2' => 'test2'
                                             )
                                       );
    }
}
```

Example 8.9. Using route assembly with gotoRoute()

The following example uses the router's `assemble()` method to create a URL based on an associative array of parameters passed. It assumes the following route has been registered:

```
$route = new Zend_Controller_Router_Route(
    'blog/:year/:month/:day/:id',
    array('controller' => 'archive',
          'module' => 'blog',
          'action' => 'view')
);
$router->addRoute('blogArchive', $route);
```

Given an array with year set to 2006, month to 4, day to 24, and id to 42, it would then build the URL `/blog/2006/4/24/42`.

```
class BlogAdminController extends Zend_Controller_Action
{
    /**
     * Redirector - defined for code completion
     *
     * @var Zend_Controller_Action_Helper_Redirector
     */
    protected $_redirector = null;

    public function init()
    {
        $this->_redirector = $this->_helper->getHelper('Redirector');
    }

    public function returnAction()
    {
        /* do some stuff */

        // Redirect to blog archive. Builds the following URL:
        // /blog/2006/4/24/42
        $this->_redirector->gotoRoute(
            array('year' => 2006,
                  'month' => 4,
                  'day' => 24,
                  'id' => 42),
            'blogArchive'
        );
    }
}
```

ViewRenderer

Introduction

The ViewRenderer helper is designed to satisfy the following goals:

- Eliminate the need to instantiate view objects within controllers; view objects will be automatically registered with the controller.
- Automatically set view script, helper, and filter paths based on the current module, and automatically associate the current module name as a class prefix for helper and filter classes.
- Create a globally available view object for all dispatched controllers and actions.
- Allow the developer to set default view rendering options for all controllers.
- Add the ability to automatically render a view script with no intervention.
- Allow the developer to create her own specifications for the view base path and for view script paths.

Note

If you perform a `_forward()`, redirect, or `render` manually, autorendering will not occur, as by performing any of these actions you are telling the `ViewRenderer` that you are determining your own output.

Note

The `ViewRenderer` is enabled by default. You may disable it via the front controller `noViewRenderer` param (`$front->setParam('noViewRenderer', true)`) or removing the helper from the helper broker stack (`Zend_Controller_Action_HelperBroker::removeHelper('viewRenderer')`).

If you wish to modify settings of the `ViewRenderer` prior to dispatching the front controller, you may do so in one of two ways:

- Instantiate and register your own `ViewRenderer` object and pass it to the helper broker:

```
$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer();
$viewRenderer->setView($view)
             ->setViewSuffix('php');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

- Initialize and/or retrieve a `ViewRenderer` object on demand via the helper broker:

```
$viewRenderer = Zend_Controller_Action_HelperBroker::getStaticHelper('viewRend
$viewRenderer->setView($view)
             ->setViewSuffix('php');
```

API

At its most basic usage, you simply instantiate the `ViewRenderer` and pass it to the action helper broker. The easiest way to instantiate it and register in one go is to use the helper broker's `getStaticHelper()` method:

```
Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
```

The first time an action controller is instantiated, it will trigger the `ViewRenderer` to instantiate a view object. Each time a controller is instantiated, the `ViewRenderer`'s `init()` method is called, which will cause it to set the view property of the action controller, and call `addScriptPath()` with a path relative to the current module; this will be called with a class prefix named after the current module, effectively namespacing all helper and filter classes you define for the module.

Each time `postDispatch()` is called, it will call `render()` for the current action.

As an example, consider the following class:

```
// A controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    // Render bar/index.phtml by default; no action required
    public function indexAction()
    {
    }

    // Render bar/populate.phtml with variable 'foo' set to 'bar'.
    // Since view object defined at preDispatch(), it's already available.
    public function populateAction()
    {
        $this->view->foo = 'bar';
    }
}

...

// in one of your view scripts:
$this->foo(); // call Foo_View_Helper_Foo::foo()
```

The `ViewRenderer` also defines a number of accessors to allow setting and retrieving view options:

- `setView($view)` allows you to set the view object for the `ViewRenderer`. It gets set as the public class property `$view`.
- `setNeverRender($flag = true)` can be used to disable or enable autorendering globally, i.e., for all controllers. If set to true, `postDispatch()` will not automatically call `render()` in the current controller. `getNeverRender()` retrieves the current value.

- `setNoRender($flag = true)` can be used to disable or enable autorendering. If set to true, `postDispatch()` will not automatically call `render()` in the current controller. This setting is reset each time `preDispatch()` is called (i.e., you need to set this flag for each controller for which you don't want autorenderering to occur). `getNoRender()` retrieves the current value.
- `setNoController($flag = true)` can be used to tell `render()` not to look for the action script in a subdirectory named after the controller (which is the default behaviour). `getNoController()` retrieves the current value.
- `setNeverController($flag = true)` is analogous to `setNoController()`, but works on a global level -- i.e., it will not be reset for each dispatched action. `getNeverController()` retrieves the current value.
- `setScriptAction($name)` can be used to specify the action script to render. `$name` should be the name of the script minus the file suffix (and without the controller subdirectory, unless `noController` has been turned on). If not specified, it looks for a view script named after the action in the request object. `getScriptAction()` retrieves the current value.
- `setResponseSegment($name)` can be used to specify which response object named segment to render into. If not specified, it renders into the default segment. `getResponseSegment()` retrieves the current value.
- `initView($path, $prefix, $options)` may be called to specify the base view path, class prefix for helper and filter scripts, and `ViewRenderer` options. You may pass any of the following flags: `neverRender`, `noRender`, `noController`, `scriptAction`, and `responseSegment`.
- `setRender($action = null, $name = null, $noController = false)` allows you to set any of `scriptAction`, `responseSegment`, and `noController` in one pass. `direct()` is an alias to this method, allowing you to call this method easily from your controller:

```
// Render 'foo' instead of current action script
$this->_helper->viewRenderer('foo');

// render form.phtml to the 'html' response segment, without using a
// controller view script subdirectory:
$this->_helper->viewRenderer('form', 'html', true);
```

Note

`setRender()` and `direct()` don't actually render the view script, but instead set hints that `postDispatch()` and `render()` will use to render the view.

The constructor allows you to optionally pass the view object and `ViewRenderer` options; it accepts the same flags as `initView()`:

```
$view    = new Zend_View(array('encoding' => 'UTF-8'));
$options = array('noController' => true, 'neverRender' => true);
$viewRenderer =
    new Zend_Controller_Action_Helper_ViewRenderer($view, $options);
```

There are several additional methods for customizing path specifications used for determining the view base path to add to the view object, and the view script path to use when autodetermining the view script to render. These methods each take one or more of the following placeholders:

- `:moduleDir` refers to the current module's base directory (by convention, the parent directory of the module's controller directory).
- `:module` refers to the current module name.
- `:controller` refers to the current controller name.
- `:action` refers to the current action name.
- `:suffix` refers to the view script suffix (which may be set via `setViewSuffix()`).

The methods for controlling path specifications are:

- `setViewBasePathSpec($spec)` allows you to change the path specification used to determine the base path to add to the view object. The default specification is `:moduleDir/views`. You may retrieve the current specification at any time using `getViewBasePathSpec()`.
- `setViewScriptPathSpec($spec)` allows you to change the path specification used to determine the path to an individual view script (minus the base view script path). The default specification is `:controller/:action.:suffix`. You may retrieve the current specification at any time using `getViewScriptPathSpec()`.
- `setViewScriptPathNoControllerSpec($spec)` allows you to change the path specification used to determine the path to an individual view script when `noController` is in effect (minus the base view script path). The default specification is `:action.:suffix`. You may retrieve the current specification at any time using `getViewScriptPathNoControllerSpec()`.

For fine-grained control over path specifications, you may use Zend_Filter_Inflector. Under the hood, the `ViewRenderer` uses an inflector to perform path mappings already. To interact with the inflector -- either to set your own for use, or to modify the default inflector, the following methods may be used:

- `getInflector()` will retrieve the inflector. If none exists yet in the `ViewRenderer`, it creates one using the default rules.

  By default, it uses static rule references for the suffix and module directory, as well as a static target; this allows various `ViewRenderer` properties the ability to dynamically modify the inflector.
- `setInflector($inflector, $reference)` allows you to set a custom inflector for use with the `ViewRenderer`. If `$reference` is true, it will set the suffix and module directory as static references to `ViewRenderer` properties, as well as the target.

Default Lookup Conventions

The `ViewRenderer` does some path normalization to make view script lookups easier. The default rules are as follows:

- `:module`: MixedCase and camelCasedWords are separated by dashes, and the entire string cast to lowercase. E.g.: "FooBarBaz" becomes "foo-bar-baz".

  Internally, the inflector uses the filters `Zend_Filter_Word_CamelCaseToDash` and `Zend_Filter_StringToLower`.

  - `:controller`: MixedCase and camelCasedWords are separated by dashes; underscores are converted to directory separators, and the entire string cast to lower case. Examples: "FooBar" becomes "foo-bar"; "FooBar_Admin" becomes "foo-bar/admin".

    Internally, the inflector uses the filters `Zend_Filter_Word_CamelCaseToDash`, `Zend_Filter_Word_UnderscoreToSeparator`, and `Zend_Filter_StringToLower`.

  - `:action`: MixedCase and camelCasedWords are separated by dashes; non-alphanumeric characters are translated to dashes, and the entire string cast to lower case. Examples: "fooBar" becomes "foo-bar"; "foo-barBaz" becomes "foo-bar-baz".

    Internally, the inflector uses the filters `Zend_Filter_Word_CamelCaseToDash`, `Zend_Filter_PregReplace`, and `Zend_Filter_StringToLower`.

The final items in the `ViewRenderer` API are the methods for actually determining view script paths and rendering views. These include:

- `renderScript($script, $name)` allows you to render a script with a path you specify, optionally to a named path segment. When using this method, the `ViewRenderer` does no autodetermination of the script name, but instead directly passes the `$script` argument directly to the view object's `render()` method.

  Note

  Once the view has been rendered to the response object, it sets the `noRender` to prevent accidentally rendering the same view script multiple times.

  Note

  By default, `Zend_Controller_Action::renderScript()` proxies to the `ViewRenderer`'s `renderScript()` method.

- `getViewScript($action, $vars)` creates the path to a view script based on the action passed and/or any variables passed in `$vars`. Keys for this array may include any of the path specification keys ('moduleDir', 'module', 'controller', 'action', and 'suffix'). Any variables passed will be used; otherwise, values based on the current request will be utlized.

  `getViewScript()` will use either the `viewScriptPathSpec` or `viewScriptPathNoControllerSpec` based on the setting of the `noController` flag.

  Word delimiters occurring in module, controller, or action names will be replaced with dashes ('-'). Thus, if you have the controller name 'foo.bar' and the action 'baz:bat', using the default path specification will result in a view script path of 'foo-bar/baz-bat.phtml'.

  Note

  By default, `Zend_Controller_Action::getViewScript()` proxies to the `ViewRenderer`'s `getViewScript()` method.

- `render($action, $name, $noController)` checks first to see if either `$name` or `$noController` have been passed, and if so, sets the appropriate flags (responseSegment and noController, respectively) in the ViewRenderer. It then passes the `$action` argument, if any, on to `getViewScript()`. Finally, it passes the calculated view script path to `renderScript()`.

Note

Be aware of the side-effects of using render(): the values you pass for the response segment name and for the noController flag will persist in the object. Additionally, noRender will be set after rendering is completed.

Note

By default, `Zend_Controller_Action::render()` proxies to the `ViewRenderer`'s `render()` method.

- `renderBySpec($action, $vars, $name)` allows you to pass path specification variables in order to determine the view script path to create. It passes `$action` and `$vars` to `getScriptPath()`, and then passes the resulting script path and `$name` on to `renderScript()`.

Basic Usage Examples

Example 8.10. Basic Usage

At its most basic, you simply initialize and register the `ViewRenderer` helper with the helper broker in your bootstrap, and then set variables in your action methods.

```
// In your bootstrap:
Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');

...

// 'foo' module, 'bar' controller:
class Foo_BarController extends Zend_Controller_Action
{
    // Render bar/index.phtml by default; no action required
    public function indexAction()
    {
    }

    // Render bar/populate.phtml with variable 'foo' set to 'bar'.
    // Since view object defined at preDispatch(), it's already available.
    public function populateAction()
    {
        $this->view->foo = 'bar';
    }

    // Renders nothing as it forwards to another action; the new action
    // will perform any rendering
    public function bazAction()
    {
        $this->_forward('index');
    }

    // Renders nothing as it redirects to another location
    public function batAction()
    {
        $this->_redirect('/index');
    }
}
```

Naming Conventions: Word delimiters in controller and action names

If your controller or action name is composed of several words, the dispatcher requires that these are separated on the URL by specific path and word delimiter characters. The `ViewRenderer` replaces any path delimiter found in the controller name with an actual path delimiter ('/'), and any word delimiter found with a dash ('-') when creating paths. Thus, a call to the action `/foo.bar/baz.bat` would dispatch to `FooBarController::bazBatAction()` in FooBarController.php, which would render `foo-bar/baz-bat.phtml`; a call to the action

`/bar_baz/baz-bat` would dispatch to `Bar_BazController::bazBatAction()` in `Bar/BazController.php` (note the path separation) and render `bar/baz/baz-bat.phtml`.

Note that the in the second example, the module is still the default module, but that, because of the existence of a path separator, the controller receives the name `Bar_BazController`, in `Bar/BazController.php`. The ViewRenderer mimics the controller directory hierarchy.

Example 8.11. Disabling autorender

For some actions or controllers, you may want to turn off the autorendering -- for instance, if you're wanting to emit a different type of output (XML, JSON, etc), or if you simply want to emit nothing. You have two options: turn off all cases of autorendering (`setNeverRender()`), or simply turn it off for the current action (`setNoRender()`).

```
// Baz controller class, bar module:
class Bar_BazController extends Zend_Controller_Action
{
    public function fooAction()
    {
        // Don't auto render this action
        $this->_helper->viewRenderer->setNoRender();
    }
}

// Bat controller class, bar module:
class Bar_BatController extends Zend_Controller_Action
{
    public function preDispatch()
    {
        // Never auto render this controller's actions
        $this->_helper->viewRenderer->setNoRender();
    }
}
```

Note

In most cases, it makes no sense to turn off autorendering globally (ala `setNeverRender()`), as the only thing you then gain from `ViewRenderer` is the autosetup of the view object.

Example 8.12. Choosing a different view script

Some situations require that you render a different script than one named after the action. For instance, if you have a controller that has both add and edit actions, they may both display the same 'form' view, albeit with different values set. You can easily change the script name used with either `setScriptAction()`, `setRender()`, or calling the helper as a method, which will invoke `setRender()`.

```
// Bar controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    public function addAction()
    {
        // Render 'bar/form.phtml' instead of 'bar/add.phtml'
        $this->_helper->viewRenderer('form');
    }

    public function editAction()
    {
        // Render 'bar/form.phtml' instead of 'bar/edit.phtml'
        $this->_helper->viewRenderer->setScriptAction('form');
    }

    public function processAction()
    {
        // do some validation...
        if (!$valid) {
            // Render 'bar/form.phtml' instead of 'bar/process.phtml'
            $this->_helper->viewRenderer->setRender('form');
            return;
        }

        // otherwise continue processing...
    }

}
```

Example 8.13. Modifying the registered view

What if you need to modify the view object -- for instance, change the helper paths, or the encoding? You can do so either by modifying the view object set in your controller, or by grabbing the view object out of the `ViewRenderer`; both are references to the same object.

```
// Bar controller class, foo module:
class Foo_BarController extends Zend_Controller_Action
{
    public function preDispatch()
    {
        // change view encoding
        $this->view->setEncoding('UTF-8');
    }

    public function bazAction()
    {
        // Get view object and set escape callback to 'htmlspecialchars'
        $view = $this->_helper->viewRenderer->view;
        $view->setEscape('htmlspecialchars');
    }
}
```

Advanced Usage Examples

Example 8.14. Changing the path specifications

In some circumstances, you may decide that the default path specifications do not fit your site's needs. For instance, you may want to have a single template tree to which you may then give access to your designers (this is very typical when using Smarty [http://smarty.php.net/], for instance). In such a case, you may want to hardcode the view base path specification, and create an alternate specification for the action view script paths themselves.

For purposes of this example, let's assume that the base path to views should be '/opt/vendor/templates', and that you wish for view scripts to be referenced by ':moduleDir/:controller/:action.:suffix'; if the noController flag has been set, you want to render out of the top level instead of in a subdirectory (':action.:suffix'). Finally, you want to use 'tpl' as the view script filename suffix.

```
/**
 * In your bootstrap:
 */

// Different view implementation
$view = new ZF_Smarty();

$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
$viewRenderer->setViewBasePathSpec('/opt/vendor/templates')
             ->setViewScriptPathSpec(':module/:controller/:action.:suffix')
             ->setViewScriptPathNoControllerSpec(':action.:suffix')
             ->setViewSuffix('tpl');
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

Example 8.15. Rendering multiple view scripts from a single action

At times, you may need to render multiple view scripts from a single action. This is very straightforward -- simply make multiple calls to `render()`:

```
class SearchController extends Zend_Controller_Action
{
    public function resultsAction()
    {
        // Assume $this->model is the current model
        $this->view->results =
            $this->model->find($this->_getParam('query', '');

        // render() by default proxies to the ViewRenderer
        // Render first the search form and then the results
        $this->render('form');
        $this->render('results');
    }

    public function formAction()
    {
        // do nothing; ViewRenderer autorenders the view script
    }
}
```

Writing Your Own Helpers

Action helpers extend `Zend_Controller_Action_Helper_Abstract`, an abstract class that provides the basic interface and functionality required by the helper broker. These include the following methods:

- `setActionController()` is used to set the current action controller.
- `init()`, triggered by the helper broker at instantiation, can be used to trigger initialization in the helper; this can be useful for resetting state when multiple controllers use the same helper in chained actions.
- `preDispatch()`, is triggered prior to a dispatched action.
- `postDispatch()` is triggered when a dispatched action is done -- even if a `preDispatch()` plugin has skipped the action. Mainly useful for cleanup.
- `getRequest()` retrieves the current request object.
- `getResponse()` retrieves the current response object.
- `getName()` retrieves the helper name. It retrieves the portion of the class name following the last underscore character, or the full class name otherwise. As an example, if the class is named `Zend_Controller_Action_Helper_Redirector`, it will return `Redirector`; a class named `FooMessage` will simply return itself.

You may optionally include a `direct()` method in your helper class. If defined, it allows you to treat the helper as a method of the helper broker, in order to allow easy, one-off usage of the helper. As an example, the redirector defines `direct()` as an alias of `goto()`, allowing use of the helper like this:

```
// Redirect to /blog/view/item/id/42
$this->_helper->redirector('item', 'view', 'blog', array('id' => 42));
```

Internally, the helper broker's `__call()` method looks for a helper named `redirector`, then checks to see if that helper has a defined `direct` class, and calls it with the arguments provided.

Once you have created your own helper class, you may provide access to it as described in the sections above.

The Response Object

Usage

The response object is the logical counterpart to the request object. Its purpose is to collate content and/or headers so that they may be returned en masse. Additionally, the front controller will pass any caught exceptions to the response object, allowing the developer to gracefully handle exceptions. This functionality may be overridden by setting `Zend_Controller_Front::throwExceptions(true)`:

```
$front->throwExceptions(true);
```

To send the response output, including headers, use `sendResponse()`.

```
$response->sendResponse();
```

Note

By default, the front controller calls `sendResponse()` when it has finished dispatching the request; typically you will never need to call it. However, if you wish to manipulate the response or use it in testing, you can override this behaviour by setting the `returnResponse` flag with `Zend_Controller_Front::returnResponse(true)`:

```
$front->returnResponse(true);
$response = $front->dispatch();

// do some more processing, such as logging...
// and then send the output:
$response->sendResponse();
```

Developers should make use of the response object in their action controllers. Instead of directly rendering output and sending headers, push them to the response object:

```
// Within an action controller action:
// Set a header
$this->getResponse()
    ->setHeader('Content-Type', 'text/html')
    ->appendBody($content);
```

By doing this, all headers get sent at once, just prior to displaying the content.

Note

If using the action controller view integration, you do not need to set the rendered view script content in the response object, as `Zend_Controller_Action::render()` does this by default.

Should an exception occur in an application, check the response object's `isException()` flag, and retrieve the exception using `getException()`. Additionally, one may create custom response objects that redirect to error pages, log exception messages, do pretty formatting of exception messages (for development environments), etc.

You may retrieve the response object following the front controller dispatch(), or request the front controller to return the response object instead of rendering output.

```
// retrieve post-dispatch:
$front->dispatch();
$response = $front->getResponse();
if ($response->isException()) {
    // log, mail, etc...
}

// Or, have the front controller dispatch() process return it
$front->returnResponse(true);
$response = $front->dispatch();

// do some processing...

// finally, echo the response
$response->sendResponse();
```

By default, exception messages are not displayed. This behaviour may be overridden by calling `renderExceptions()`, or enabling the front controller to throwExceptions(), as shown above:

```
$response->renderExceptions(true);
$front->dispatch($request, $response);

// or:
```

```
$front->returnResponse(true);
$response = $front->dispatch();
$response->renderExceptions();
$response->sendResponse();

// or:
$front->throwExceptions(true);
$front->dispatch();
```

Manipulating Headers

As stated previously, one aspect of the response object's duties is to collect and emit HTTP response headers. A variety of methods exist for this:

- `canSendHeaders()` is used to determine if headers have already been sent. It takes an optional flag indicating whether or not to throw an exception if headers have already been sent. This can be overridden by setting the property `headersSentThrowsException` to `false`.
- `setHeader($name, $value, $replace = false)` is used to set an individual header. By default, it does not replace existing headers of the same name in the object; however, setting `$replace` to true will force it to do so.

  Before setting the header, it checks with `canSendHeaders()` to see if this operation is allowed at this point, and requests that an exception be thrown.
- `setRedirect($url, $code = 302)` sets an HTTP Location header for a redirect. If an HTTP status code has been provided, it will use that status code.

  Internally, it calls `setHeader()` with the `$replace` flag on to ensure only one such header is ever sent.
- `getHeaders()` returns an array of all headers. Each array element is an array with the keys 'name' and 'value'.
- `clearHeaders()` clears all registered headers.
- `setRawHeader()` can be used to set headers that are not key/value pairs, such as an HTTP status header.
- `getRawHeaders()` returns any registered raw headers.
- `clearRawHeaders()` clears any registered raw headers.
- `clearAllHeaders()` clears both regular key/value headers as well as raw headers.

In addition to the above methods, there are accessors for setting and retrieving the HTTP response code for the current request, `setHttpResponseCode()` and `getHttpResponseCode()`.

Named Segments

The response object has support for "named segments". This allows you to segregate body content into different segments and order those segments so output is returned in a specific order. Internally, body content is saved as an array, and the various accessor methods can be used to indicate placement and names within that array.

As an example, you could use a `preDispatch()` hook to add a header to the response object, then have the action controller add body content, and a `postDispatch()` hook add a footer:

```
// Assume that this plugin class is registered with the front controller
class MyPlugin extends Zend_Controller_Plugin_Abstract
{
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $response = $this->getResponse();
        $view = new Zend_View();
        $view->setBasePath('../views/scripts');

        $response->prepend('header', $view->render('header.phtml'));
    }

    public function postDispatch(Zend_Controller_Request_Abstract $request)
    {
        $response = $this->getResponse();
        $view = new Zend_View();
        $view->setBasePath('../views/scripts');

        $response->append('footer', $view->render('footer.phtml'));
    }
}

// a sample action controller
class MyController extends Zend_Controller_Action
{
    public function fooAction()
    {
        $this->render();
    }
}
```

In the above example, a call to `/my/foo` will cause the final body content of the response object to have the following structure:

```
array(
    'header'  => ..., // header content
    'default' => ..., // body content from MyController::fooAction()
    'footer'  => ...  // footer content
);
```

When this is rendered, it will render in the order in which elements are arranged in the array.

A variety of methods can be used to manipulate the named segments:

- `setBody()` and `appendBody()` both allow you to pass a second value, `$name`, indicating a named segment. In each case, if you provide this, it will overwrite that specific named segment or create it if it does not exist (appending to the array by default). If no named segment is passed to `setBody()`, it resets the entire body content array. If no named segment is passed to appendBody(), the content is appended to the value in the 'default' name segment.
- `prepend($name, $content)` will create a segment named `$name` and place it at the beginning of the array. If the segment exists already, it will be removed prior to the operation (i.e., overwritten and replaced).
- `append($name, $content)` will create a segment named `$name` and place it at the end of the array. If the segment exists already, it will be removed prior to the operation (i.e., overwritten and replaced).
- `insert($name, $content, $parent = null, $before = false)` will create a segment named `$name`. If provided with a `$parent` segment, the new segment will be placed either before or after that segment (based on the value of `$before`) in the array. If the segment exists already, it will be removed prior to the operation (i.e., overwritten and replaced).
- `clearBody($name = null)` will remove a single named segment if a `$name` is provided (and the entire array otherwise).
- `getBody($spec = false)` can be used to retrieve a single array segment if `$spec` is the name of a named segment. If `$spec` is false, it returns a string formed by concatenating all named segments in order. If `$spec` is true, it returns the body content array.

Testing for Exceptions in the Response Object

As mentioned earlier, by default, exceptions caught during dispatch are registered with the response object. Exceptions are registered in a stack, which allows you to keep all exceptions thrown -- application exceptions, dispatch exceptions, plugin exceptions, etc. Should you wish to check for particular exceptions or to log exceptions, you'll want to use the response object's exception API:

- `setException(Exception $e)` allows you to register an exception.
- `isException()` will tell you if an exception has been registered.
- `getException()` returns the entire exception stack.
- `hasExceptionOfType($type)` allows you to determine if an exception of a particular class is in the stack.
- `hasExceptionOfMessage($message)` allows you to determine if an exception with a specific message is in the stack.
- `hasExceptionOfCode($code)` allows you to determine if an exception with a specific code is in the stack.
- `getExceptionByType($type)` allows you to retrieve all exceptions of a specific class from the stack. It will return false if none are found, and an array of exceptions otherwise.
- `getExceptionByMessage($message)` allows you to retrieve all exceptions with a specific message from the stack. It will return false if none are found, and an array of exceptions otherwise.

- `getExceptionByCode($code)` allows you to retrieve all exceptions with a specific code from the stack. It will return false if none are found, and an array of exceptions otherwise.
- `renderExceptions($flag)` allows you to set a flag indicating whether or not exceptions should be emitted when the response is sent.

Subclassing the Response Object

The purpose of the response object is to collect headers and content from the various actions and plugins and return them to the client; secondarily, it also collects any errors (exceptions) that occur in order to process them, return them, or hide them from the end user.

The base response class is `Zend_Controller_Response_Abstract`, and any subclass you create should extend that class or one of its derivatives. The various methods available have been listed in the previous sections.

Reasons to subclass the response object include modifying how output is returned based on the request environment (e.g., not sending headers for CLI or PHP-GTK requests), adding functionality to return a final view based on content stored in named segments, etc.

Plugins

Introduction

The controller architecture includes a plugin system that allows user code to be called when certain events occur in the controller process lifetime. The front controller uses a plugin broker as a registry for user plugins, and the plugin broker ensures that event methods are called on each plugin registered with the front controller.

The event methods are defined in the abstract class `Zend_Controller_Plugin_Abstract`, from which user plugin classes inherit:

- `routeStartup()` is called before `Zend_Controller_Front` calls on the router to evaluate the request against the registered routes.
- `routeShutdown()` is called after the router finishes routing the request.
- `dispatchLoopStartup()` is called before `Zend_Controller_Front` enters its dispatch loop.
- `preDispatch()` is called before an action is dispatched by the dispatcher. This callback allows for proxy or filter behavior. By altering the request and resetting its dispatched flag (via `Zend_Controller_Request_Abstract::setDispatched(false)`), the current action may be skipped and/or replaced.
- `postDispatch()` is called after an action is dispatched by the dispatcher. This callback allows for proxy or filter behavior. By altering the request and resetting its dispatched flag (via `Zend_Controller_Request_Abstract::setDispatched(false)`), a new action may be specified for dispatching.
- `dispatchLoopShutdown()` is called after `Zend_Controller_Front` exits its dispatch loop.

Writing Plugins

In order to write a plugin class, simply include and extend the abstract class `Zend_Controller_Plugin_Abstract`:

```
class MyPlugin extends Zend_Controller_Plugin_Abstract
{
    // ...
}
```

None of the methods of `Zend_Controller_Plugin_Abstract` are abstract, and this means that plugin classes are not forced to implement any of the available event methods listed above. Plugin writers may implement only those methods required by their particular needs.

`Zend_Controller_Plugin_Abstract` also makes the request and response objects available to controller plugins via the `getRequest()` and `getResponse()` methods, respectively.

Using Plugins

Plugin classes are registered with `Zend_Controller_Front::registerPlugin()`, and may be registered at any time. The following snippet illustrates how a plugin may be used in the controller chain:

```
class MyPlugin extends Zend_Controller_Plugin_Abstract
{
    public function routeStartup(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()->appendBody("<p>routeStartup() called</p>\n");
    }

    public function routeShutdown(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()->appendBody("<p>routeShutdown() called</p>\n");
    }

    public function dispatchLoopStartup(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()->appendBody("<p>dispatchLoopStartup() called</p>\n");
    }

    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()->appendBody("<p>preDispatch() called</p>\n");
    }

    public function postDispatch(Zend_Controller_Request_Abstract $request)
    {
        $this->getResponse()->appendBody("<p>postDispatch() called</p>\n");
    }
    public function dispatchLoopShutdown()
    {
        $this->getResponse()->appendBody("<p>dispatchLoopShutdown() called</p>\n")
    }
}

$front = Zend_Controller_Front::getInstance();
$front->setControllerDirectory('/path/to/controllers')
      ->setRouter(new Zend_Controller_Router_Rewrite())
      ->registerPlugin(new MyPlugin());
$front->dispatch();
```

Assuming that no actions called emit any output, and only one action is called, the functionality of the above plugin would still create the following output:

```
<p>routeStartup() called</p>
<p>routeShutdown() called</p>
<p>dispatchLoopStartup() called</p>
<p>preDispatch() called</p>
<p>postDispatch() called</p>
<p>dispatchLoopShutdown() called</p>
```

Note

Plugins may be registered at any time during the front controller execution. However, if an event has passed for which the plugin has a registered event method, that method will not be triggered.

Retrieving and Manipulating Plugins

On occasion, you may need to unregister or retrieve a plugin. The following methods of the front controller allow you to do so:

- `getPlugin($class)` allows you to retrieve a plugin by class name. If no plugins match, it returns false. If more than one plugin of that class is registered, it returns an array.
- `getPlugins()` retrieves the entire plugin stack.
- `unregisterPlugin($plugin)` allows you to remove a plugin from the stack. You may pass a plugin object, or the class name of the plugin you wish to unregister. If you pass the class name, any plugins of that class will be removed.

Plugins Included in the Standard Distribution

Zend Framework includes a plugin for error handling in its standard distribution.

ActionStack

The `ActionStack` plugin allows you to manage a stack of requests, and operates as a `postDispatch` plugin. If a forward (i.e., a call to another action) is already detected in the current request object, it does nothing. However, if not, it checks its stack and pulls the topmost item off it and forwards to the action specified in that request. The stack is processed in LIFO order.

You can retrieve the plugin from the front controller at any time using `Zend_Controller_Front::getPlugin('Zend_Controller_Plugin_ActionStack')`. Once you have the plugin object, there are a variety of mechanisms you can use to manipulate it.

- `getRegistry()` and `setRegistry()`. Internally, `ActionStack` uses a `Zend_Registry` instance to store the stack. You can substitute a different registry instance or retrieve it with these accessors.
- `getRegistryKey()` and `setRegistryKey()`. These can be used to indicate which registry key to use when pulling the stack. Default value is 'Zend_Controller_Plugin_ActionStack'.
- `getStack()` allows you to retrieve the stack of actions in its entirety.
- `pushStack()` and `popStack()` allow you to add to and pull from the stack, respectively. `pushStack()` accepts a request object.

An additional method, `forward()`, expects a request object, and sets the state of the current request object in the front controller to the state of the provided request object, and markes it as undispatched (forcing another iteration of the dispatch loop).

Zend_Controller_Plugin_ErrorHandler

`Zend_Controller_Plugin_ErrorHandler` provides a drop-in plugin for handling exceptions thrown by your application, including those resulting from missing controllers or actions; it is an alternative to the methods listed in the MVC Exceptions section.

The primary targets of the plugin are:

- Intercept exceptions raised due to missing controllers or action methods
- Intercept exceptions raised within action controllers

In other words, the `ErrorHandler` plugin is designed to handle HTTP 404-type errors (page missing) and 500-type errors (internal error). It is not intended to catch exceptions raised in other plugins or routing.

By default, `Zend_Controller_Plugin_ErrorHandler` will forward to `ErrorController::errorAction()` in the default module. You may set alternate values for these by using the various accessors available to the plugin:

- `setErrorHandlerModule()` sets the controller module to use.
- `setErrorHandlerController()` sets the controller to use.
- `setErrorHandlerAction()` sets the controller action to use.
- `setErrorHandler()` takes an associative array, which may contain any of the keys 'module', 'controller', or 'action', with which it will set the appropriate values.

Additionally, you may pass an optional associative array to the constructor, which will then proxy to `setErrorHandler()`.

`Zend_Controller_Plugin_ErrorHandler` registers a `postDispatch()` hook and checks for exceptions registered in the response object. If any are found, it attempts to forward to the registered error handler action.

If an exception occurs dispatching the error handler, the plugin will tell the front controller to throw exceptions, and rethrow the last exception registered with the response object.

Using the ErrorHandler as a 404 Handler

Since the `ErrorHandler` plugin captures not only application errors, but also errors in the controller chain arising from missing controller classes and/or action methods, it can be used as a 404 handler. To do so, you will need to have your error controller check the exception type.

Exceptions captured are logged in an object registered in the request. To retrieve it, use `Zend_Controller_Action::_getParam('error_handler')`:

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');
    }
}
```

Once you have the error object, you can get the type via `$errors->type`. It will be one of the following:

- `Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER`, indicating the controller was not found.
- `Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION`, indicating the requested action was not found.
- `Zend_Controller_Plugin_ErrorHandler::EXCEPTION_OTHER`, indicating other exceptions.

You can then test for either of the first two types, and, if so, indicate a 404 page:

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
                // 404 error -- controller or action not found
                $this->getResponse()
                     ->setRawHeader('HTTP/1.1 404 Not Found');

                // ... get some output to display...
                break;
            default:
                // application error; display error page, but don't
                // change status code
                break;
        }
    }
}
```

Finally, you can retrieve the exception that triggered the error handler by grabbing the `exception` property of the `error_handler` object:

```
public function errorAction()
{
        $errors = $this->_getParam('error_handler');


        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
                // 404 error -- controller or action not found
                $this->getResponse()
                     ->setRawHeader('HTTP/1.1 404 Not Found');

                // ... get some output to display...
                break;
            default:
                // application error; display error page, but don't change
                // status code

                // ...

                // Log the exception:
                $exception = $errors->exception;
                $log = new Zend_Log(
                    new Zend_Log_Writer_Stream(
                        '/tmp/applicationException.log'
                    )
                );
                $log->debug($exception->getMessage() . "\n" .
                            $exception->getTraceAsString());
                break;
        }
}
```

Handling Previously Rendered Output

If you dispatch multiple actions in a request, or if your action makes multiple calls to `render()`, its possible that the response object already has content stored within it. This can lead to rendering a mixture of expected content and error content.

If you wish to render errors inline in such pages, no changes will be necessary. If you do not wish to render such content, you should clear the response body prior to rendering any views:

```
$this->getResponse()->clearBody();
```

Plugin Usage Examples

Example 8.16. Standard usage

```
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new Zend_Controller_Plugin_ErrorHandler());
```

Example 8.17. Setting a different error handler

```
$front = Zend_Controller_Front::getInstance();
$front->registerPlugin(new Zend_Controller_Plugin_ErrorHandler(array(
    'module'     => 'mystuff',
    'controller' => 'static',
    'action'     => 'error'
)));
```

Example 8.18. Using accessors

```
$plugin = new Zend_Controller_Plugin_ErrorHandler();
$plugin->setErrorHandlerModule('mystuff')
       ->setErrorHandlerController('static')
       ->setErrorHandlerAction('error');

$front = Zend_Controller_Front::getInstance();
$front->registerPlugin($plugin);
```

Error Controller Example

In order to use the Error Handler plugin, you need an error controller. Below is a simple example.

```
class ErrorController extends Zend_Controller_Action
{
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');

        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
                // 404 error -- controller or action not found
                $this->getResponse()->setRawHeader('HTTP/1.1 404 Not Found');

                $content =<<<EOH
<h1>Error!</h1>
<p>The page you requested was not found.</p>
EOH;
                break;
            default:
                // application error
                $content =<<<EOH
<h1>Error!</h1>
<p>An unexpected error occurred. Please try again later.</p>
EOH;
                break;
        }

        // Clear previous content
        $this->getResponse()->clearBody();

        $this->view->content = $content;
    }
}
```

Using a Conventional Modular Directory Structure

Introduction

The Conventional Modular directory structure allows you to separate different MVC applications into self-contained units, and re-use them with different front controllers. To illustrate such a directory structure:

```
docroot/
    index.php
application/
    default/
        controllers/
```

```
            IndexController.php
            FooController.php
        models/
        views/
            scripts/
                index/
                foo/
            helpers/
            filters/
    blog/
        controllers/
            IndexController.php
        models/
        views/
            scripts/
                index/
            helpers/
            filters/
    news/
        controllers/
            IndexController.php
            ListController.php
        models/
        views/
            scripts/
                index/
                list/
            helpers/
            filters/
```

In this paradigm, the module name serves as a prefix to the controllers it contains. The above example contains three module controllers, 'Blog_IndexController', 'News_IndexController', and 'News_ListController'. Two global controllers, 'IndexController' and 'FooController' are also defined; neither of these will be namespaced. This directory structure will be used for examples in this chapter.

No namespacing in the default module

Note that in the default module, controllers do not need a namespace prefix. Thus, in the example above, the controllers in the default module do not need a prefix of 'Default_' -- they are simply dispatched according to their base controller name: 'IndexController' and 'FooController'. A namespace prefix is used in all other modules, however.

So, how do you implement such a directory layout using the Zend Framework MVC components?

Specifying Module Controller Directories

The first step to making use of modules is to modify how you specify the controller directory list in the front controller. In the basic MVC series, you pass either an array or a string to `setControllerDirectory()`, or a path to `addControllerDirectory()`. When using modules, you need to alter your calls to these methods slightly.

With `setControllerDirectory()`, you will need to pass an associative array and specify key/value pairs of module name/directory paths. The special key `default` will be used for global controllers (those not needing a module namespace). All entries should contain a string key pointing to a single path, and the `default` key must be present. As an example:

```
$front->setControllerDirectory(array(
    'default' => '/path/to/application/controllers',
    'blog'    => '/path/to/application/blog/controllers'
));
```

`addControllerDirectory()` will take an optional second argument. When using modules, pass the module name as the second argument; if not specified, the path will be added to the `default` namespace. As an example:

```
$front->addControllerDirectory('/path/to/application/news/controllers',
                               'news');
```

Saving the best for last, the easiest way to specify module directories is to do so en masse, with all modules under a common directory and sharing the same structure. This can be done with `addModuleDirectory()`:

```
/**
 * Assuming the following directory structure:
 * application/
 *     modules/
 *         default/
 *             controllers/
 *         foo/
 *             controllers/
 *         bar/
 *             controllers/
 */
$front->addModuleDirectory('/path/to/application/modules');
```

The above example will define the `default`, `foo`, and `bar` modules, each pointing to the `controllers` subdirectory of their respective module.

You may customize the controller subdirectory to use within your modules by using `setModuleControllerDirectoryName()`:

```
/**
 * Change the controllers subdirectory to be 'con'
 * application/
 *     modules/
 *         default/
 *             con/
 *         foo/
 *             con/
 *         bar/
 *             con/
 */
$front->setModuleControllerDirectoryName('con');
$front->addModuleDirectory('/path/to/application/modules');
```

Note

You can indicate that no controller subdirectory be used for your modules by passing an empty value to `setModuleControllerDirectoryName()`.

Routing to modules

The default route in `Zend_Controller_Router_Rewrite` is an object of type `Zend_Controller_Router_Route_Module`. This route expects one of the following routing schemas:

- `:module/:controller/:action/*`
- `:controller/:action/*`

In other words, it will match a controller and action by themselves or with a preceding module. The rules for matching specify that a module will only be matched if a key of the same name exists in the controller directory array passed to the front controller and dispatcher.

Module or Global Default Controller

In the default router, if a controller was not specified in the URL, a default controller is used (`IndexController`, unless otherwise requested). With modular controllers, if a module has been specified but no controller, the dispatcher first looks for this default controller in the module path, and then falls back on the default controller found in the 'default', global, namespace.

If you wish to always default to the global namespace, set the `useDefaultControllerAlways` parameter in the front controller:

```
$front->setParam('useDefaultControllerAlways', true);
```

MVC Exceptions

Introduction

The MVC components in Zend Framework utilize a Front Controller, which means that all requests to a given site will go through a single entry point. As a result, all exceptions bubble up to the Front Controller eventually, allowing the developer to handle them in a single location.

However, exception messages and backtrace information often contain sensitive system information, such as SQL statements, file locations, and more. To help protect your site, by default `Zend_Controller_Front` catches all exceptions and registers them with the response object; in turn, by default, the response object does not display exception messages.

How can you handle exceptions?

Several mechanisms are built in to the MVC components already to allow you to handle exceptions.

- By default, the error handler plugin is registered and active. This plugin was designed to handle:

  - Errors due to missing controllers or actions

  - Errors occurring within action controllers

  It operates as a `postDispatch()` plugin, and checks to see if a dispatcher, action controller, or other exception has occurred. If so, it forwards to an error handler controller.

  This handler will cover most exceptional situations, and handle missing controllers and actions gracefully.

- `Zend_Controller_Front::throwExceptions()`

  By passing a boolean true value to this method, you can tell the front controller that instead of aggregating exceptions in the response object or using the error handler plugin, you'd rather handle them yourself. As an example:

  ```
  $front->throwExceptions(true);
  try {
      $front->dispatch();
  } catch (Exception $e) {
      // handle exceptions yourself
  }
  ```

  This method is probably the easiest way to add custom exception handling covering the full range of possible exceptions to your front controller application.

- `Zend_Controller_Response_Abstract::renderExceptions()`

  By passing a boolean true value to this method, you tell the response object that it should render an exception message and backtrace when rendering itself. In this scenario, any exception raised by your application will be displayed. This is only recommended for non-production environments.

- `Zend_Controller_Front::returnResponse()` and `Zend_Controller_Response_Abstract::isException()`.

  By passing a boolean true to `Zend_Controller_Front::returnResponse()`, `Zend_Controller_Front::dispatch()` will not render the response, but instead return it. Once you have the response, you may then test to see if any exceptions were trapped using its `isException()` method, and retrieving the exceptions via the `getException()` method. As an example:

  ```
  $front->returnResponse(true);
  ```

```
$response = $front->dispatch();
if ($response->isException()) {
    $exceptions = $response->getException();
    // handle exceptions ...
} else {
    $response->sendHeaders();
    $response->outputBody();
}
```

The primary advantage this method offers over `Zend_Controller_Front::throwExceptions()` is to allow you to conditionally render the response after handling the exception. This will catch any exception in the controller chain, unlike the error handler plugin.

MVC Exceptions You May Encounter

The various MVC components -- request, router, dispatcher, action controller, and response objects -- may each throw exceptions on occasion. Some exceptions may be conditionally overridden, and others are used to indicate the developer may need to consider their application structure.

As some examples:

- `Zend_Controller_Dispatcher::dispatch()` will, by default, throw an exception if an invalid controller is requested. There are two recommended ways to deal with this.
  - Set the `useDefaultControllerAlways` parameter.

    In your front controller, or your dispatcher, add the following directive:

    ```
    $front->setParam('useDefaultControllerAlways', true);

    // or

    $dispatcher->setParam('useDefaultControllerAlways', true);
    ```

    When this flag is set, the dispatcher will use the default controller and action instead of throwing an exception. The disadvantage to this method is that any typos a user makes when accessing your site will still resolve and display your home page, which can wreak havoc with search engine optimization.
  - The exception thrown by `dispatch()` is a `Zend_Controller_Dispatcher_Exception` containing the text 'Invalid controller specified'. Use one of the methods outlined in the previous section to catch the exception, and then redirect to a generic error page or the home page.
- `Zend_Controller_Action::__call()` will throw a `Zend_Controller_Action_Exception` if it cannot dispatch a non-existent action to a method. Most likely, you will want to use some default action in the controller in cases like this. Ways to achieve this include:
  - Subclass `Zend_Controller_Action` and override the `__call()` method. As an example:

    ```
    class My_Controller_Action extends Zend_Controller_Action
    ```

```
{
    public function __call($method, $args)
    {
        if ('Action' == substr($method, -6)) {
            $controller = $this->getRequest()->getControllerName();
            $url = '/' . $controller . '/index';
            return $this->_redirect($url);
        }

        throw new Exception('Invalid method');
    }
}
```

The example above intercepts any undefined action method called and redirects it to the default action in the controller.

- Subclass `Zend_Controller_Dispatcher` and override the `getAction()` method to verify the action exists. As an example:

```
class My_Controller_Dispatcher extends Zend_Controller_Dispatcher
{
    public function getAction($request)
    {
        $action = $request->getActionName();
        if (empty($action)) {
            $action = $this->getDefaultAction();
            $request->setActionName($action);
            $action = $this->formatActionName($action);
        } else {
            $controller = $this->getController();
            $action     = $this->formatActionName($action);
            if (!method_exists($controller, $action)) {
                $action = $this->getDefaultAction();
                $request->setActionName($action);
                $action = $this->formatActionName($action);
            }
        }

        return $action;
    }
}
```

The above code checks to see that the requested action exists in the controller class; if not, it resets the action to the default action.

This method is nice because you can transparently alter the action prior to final dispatch. However, it also means that typos in the URL may still dispatch correctly, which is not great for search engine optimization.

- Use `Zend_Controller_Action::preDispatch()` or `Zend_Controller_Plugin_Abstract::preDispatch()` to identify invalid actions.

  By subclassing `Zend_Controller_Action` and modifying `preDispatch()`, you can modify all of your controllers to forward to another action or redirect prior to actually dispatching the action. The code for this will look similar to the code for overriding `__call()`, above.

  Alternatively, you can check this information in a global plugin. This has the advantage of being action controller independent; if your application consists of a variety of action controllers, and not all of them inherit from the same class, this method can add consistency in handling your various classes.

  As an example:

```
class My_Controller_PreDispatchPlugin extends Zend_Controller_Plugin_Abstract
{
    public function preDispatch(Zend_Controller_Request_Abstract $request)
    {
        $dispatcher =
            Zend_Controller_Front::getInstance()->getDispatcher();
        $controller = $dispatcher->getController($request);
        if (!$controller) {
            $controller =
                $dispatcher->getDefaultControllerName($request);
        }
        $action     = $dispatcher->getAction($request);

        if (!method_exists($controller, $action)) {
            $defaultAction = $dispatcher->getDefaultAction();
            $controllerName = $request->getControllerName();
            $response =
                Zend_Controller_Front::getInstance()->getResponse();
            $response->setRedirect('/' . $controllerName .
                                  '/' . $defaultAction);
            $response->sendHeaders();
            exit;
        }
    }
}
```

  In this example, we check to see if the action requested is available in the controller. If not, we redirect to the default action in the controller, and exit script execution immediately.

Migrating from Previous Versions

The API of the MVC components has changed over time. If you started using Zend Framework in an early version, follow the guidelines below to migrate your scripts to use the new architecture.

Migrating from 1.5.x to 1.6.0 or newer

Dispatcher Interface changes

Users brought to our attention the fact that `Zend_Controller_Front` and `Zend_Controller_Router_Route_Module` were each using methods of the dispatcher that were not in the dispatcher interface. We have now added the following three methods to ensure that custom dispatchers will continue to work with the shipped implementations:

- `getDefaultModule()`: should return the name of the default module.
- `getDefaultControllerName()`: should return the name of the default controller.
- `getDefaultAction()`: should return the name of the default action.

Migrating from 1.0.x to 1.5.0 or newer

Though most basic functionality remains the same, and all documented functionality remains the same, there is one particular *undocumented* "feature" that has changed.

When writing URLs, the documented way to write camelCased action names is to use a word separator; these are '.' or '-' by default, but may be configured in the dispatcher. The dispatcher internally lowercases the action name, and uses these word separators to re-assemble the action method using camelCasing. However, because PHP functions are not case sensitive, you *could* still write URLs using camelCasing, and the dispatcher would resolve these to the same location. For example, 'camel-cased' would become 'camelCasedAction' by the dispatcher, whereas 'camelCased' would become 'camelcasedAction'; however, due to the case insensitivity of PHP, both will execute the same method.

This causes issues with the ViewRenderer when resolving view scripts. The canonical, documented way is that all word separators are converted to dashes, and the words lowercased. This creates a semantic tie between the actions and view scripts, and the normalization ensures that the scripts can be found. However, if the action 'camelCased' is called and actually resolves, the word separator is no longer present, and the ViewRenderer attempts to resolve to a different location -- 'camelcased.phtml' instead of 'camel-cased.phtml'.

Some developers relied on this "feature", which was never intended. Several changes in the 1.5.0 tree, however, made it so that the ViewRenderer no longer resolves these paths; the semantic tie is now enforced. First among these, the dispatcher now enforces case sensitivity in action names. What this means is that referring to your actions on the url using camelCasing will no longer resolve to the same method as using word separators (i.e., 'camel-casing'). This leads to the ViewRenderer now only honoring the word-separated actions when resolving view scripts.

If you find that you were relying on this "feature", you have several options:

- Best option: rename your view scripts. Pros: forward compatibility. Cons: if you have many view scripts that relied on the former, unintended behavior, you will have a lot of renaming to do.
- Second best option: The ViewRenderer now delegates view script resolution to `Zend_Filter_Inflector`; you can modify the rules of the inflector to no longer separate the words of an action with a dash:

```
$viewRenderer =
    Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
$inflector = $viewRenderer->getInflector();
```

```
$inflector->setFilterRule(':action', array(
    new Zend_Filter_PregReplace(
        '#[^a-z0-9' . preg_quote(DIRECTORY_SEPARATOR, '#') . ']+#i',
        ''
    ),
    'StringToLower'
));
```

The above code will modify the inflector to no longer separate the words with dash; you may also want to remove the 'StringToLower' filter if you *do* want the actual view script names camelCased as well.

If renaming your view scripts would be too tedious or time consuming, this is your best option until you can find the time to do so.

- Least desirable option: You can force the dispatcher to dispatch camelCased action names with a new front controller flag, 'useCaseSensitiveActions':

  ```
  $front->setParam('useCaseSensitiveActions', true);
  ```

  This will allow you to use camelCasing on the url and still have it resolve to the same action as when you use word separators. However, this will mean that the original issues will cascade on through; you will likely need to use the second option above in addition to this for things to work at all reliably.

  Note, also, that usage of this flag will raise a notice that this usage is deprecated.

Migrating from 0.9.3 to 1.0.0RC1 or newer

The principal changes introduced in 1.0.0RC1 are the introduction of and default enabling of the ErrorHandler plugin and the ViewRenderer action helper. Please read the documentation to each thoroughly to see how they work and what effect they may have on your applications.

The `ErrorHandler` plugin runs during `postDispatch()` checking for exceptions, and forwarding to a specified error handler controller. You should include such a controller in your application. You may disable it by setting the front controller parameter `noErrorHandler`:

```
$front->setParam('noErrorHandler', true);
```

The `ViewRenderer` action helper automates view injection into action controllers as well as autorendering of view scripts based on the current action. The primary issue you may encounter is if you have actions that do not render view scripts and neither forward or redirect, as the `ViewRenderer` will attempt to render a view script based on the action name.

There are several strategies you can take to update your code. In the short term, you can globally disable the `ViewRenderer` in your front controller bootstrap prior to dispatching:

```
// Assuming $front is an instance of Zend_Controller_Front
$front->setParam('noViewRenderer', true);
```

However, this is not a good long term strategy, as it means most likely you'll be writing more code.

When you're ready to start using the `ViewRenderer` functionality, there are several things to look for in your controller code. First, look at your action methods (the methods ending in 'Action'), and determine what each is doing. If none of the following is happening, you'll need to make changes:

- Calls to `$this->render()`
- Calls to `$this->_forward()`
- Calls to `$this->_redirect()`
- Calls to the `Redirector` action helper

The easiest change is to disable auto-rendering for that method:

```
$this->_helper->viewRenderer->setNoRender();
```

If you find that none of your action methods are rendering, forwarding, or redirecting, you will likely want to put the above line in your `preDispatch()` or `init()` methods:

```
public function preDispatch()
{
    // disable view script autorendering
    $this->_helper->viewRenderer->setNoRender()
    // .. do other things...
}
```

If you are calling `render()`, and you're using the Conventional Modular directory structure, you'll want to change your code to make use of autorendering:

- If you're rendering multiple view scripts in a single action, you don't need to change a thing.
- If you're simply calling `render()` with no arguments, you can remove such lines.
- If you're calling `render()` with arguments, and not doing any processing afterwards or rendering multiple view scripts, you can change these calls to read `$this->_helper->viewRenderer()`.

If you're not using the conventional modular directory structure, there are a variety of methods for setting the view base path and script path specifications so that you can make use of the `ViewRenderer`. Please read the ViewRenderer documentation for information on these methods.

If you're using a view object from the registry, or customizing your view object, or using a different view implementation, you'll want to inject the `ViewRenderer` with this object. This can be done easily at any time.

- Prior to dispatching a front controller instance:

```
// Assuming $view has already been defined
$viewRenderer = new Zend_Controller_Action_Helper_ViewRenderer($view);
Zend_Controller_Action_HelperBroker::addHelper($viewRenderer);
```

- Any time during the bootstrap process:

```
$viewRenderer =
    Zend_Controller_Action_HelperBroker::getStaticHelper('viewRenderer');
$viewRenderer->setView($view);
```

There are many ways to modify the `ViewRenderer`, including setting a different view script to render, specifying replacements for all replaceable elements of a view script path (including the suffix), choosing a response named segment to utilize, and more. If you aren't using the conventional modular directory structure, you can even associate different path specifications with the `ViewRenderer`.

We encourage you to adapt your code to use the `ErrorHandler` and `ViewRenderer` as they are now core functionality.

Migrating from 0.9.2 to 0.9.3 or newer

0.9.3 introduces action helpers. As part of this change, the following methods have been removed as they are now encapsulated in the redirector action helper:

- `setRedirectCode()`; use `Zend_Controller_Action_Helper_Redirector::setCode()`.
- `setRedirectPrependBase()`; use `Zend_Controller_Action_Helper_Redirector::setPrependBase()`.
- `setRedirectExit()`; use `Zend_Controller_Action_Helper_Redirector::setExit()`.

Read the action helpers documentation for more information on how to retrieve and manipulate helper objects, and the redirector helper documentation for more information on setting redirect options (as well as alternate methods for redirecting).

Migrating from 0.6.0 to 0.8.0 or newer

Per previous changes, the most basic usage of the MVC components remains the same:

```
Zend_Controller_Front::run('/path/to/controllers');
```

However, the directory structure underwent an overhaul, several components were removed, and several others either renamed or added. Changes include:

- `Zend_Controller_Router` was removed in favor of the rewrite router.
- `Zend_Controller_RewriteRouter` was renamed to `Zend_Controller_Router_Rewrite`, and promoted to the standard router shipped with the framework; `Zend_Controller_Front` will use it by default if no other router is supplied.
- A new route class for use with the rewrite router was introduced, `Zend_Controller_Router_Route_Module`; it covers the default route used by the MVC, and has support for controller modules.
- `Zend_Controller_Router_StaticRoute` was renamed to `Zend_Controller_Router_Route_Static`.
- `Zend_Controller_Dispatcher` was renamed `Zend_Controller_Dispatcher_Standard`.
- `Zend_Controller_Action::_forward()`'s arguments have changed. The signature is now:

  ```
  final protected function _forward($action,
                                    $controller = null,
                                    $module = null,
                                    array $params = null);
  ```

  `$action` is always required; if no controller is specified, an action in the current controller is assumed. `$module` is always ignored unless `$controller` is specified. Finally, any `$params` provided will be appended to the request object. If you do not require the controller or module, but still need to pass parameters, simply specify null for those values.

Migrating from 0.2.0 or before to 0.6.0

The most basic usage of the MVC components has not changed; you can still do each of the following:

```
Zend_Controller_Front::run('/path/to/controllers');
```

```
/* -- create a router -- */
$router = new Zend_Controller_RewriteRouter();
$router->addRoute('user',
                  'user/:username',
                  array('controller' => 'user', 'action' => 'info')
);

/* -- set it in a controller -- */
$ctrl = Zend_Controller_Front::getInstance();
$ctrl->setRouter($router);

/* -- set controller directory and dispatch -- */
$ctrl->setControllerDirectory('/path/to/controllers');
$ctrl->dispatch();
```

We encourage use of the Response object to aggregate content and headers. This will allow for more flexible output format switching (for instance, JSON or XML instead of XHTML) in your applications. By default, `dispatch()` will render the response, sending both headers and rendering any content. You may also have the front controller return the response using `returnResponse()`, and then render the response using your own logic. A future version of the front controller may enforce use of the response object via output buffering.

There are many additional features that extend the existing API, and these are noted in the documentation.

The main changes you will need to be aware of will be found when subclassing the various components. Key amongst these are:

- `Zend_Controller_Front::dispatch()` by default traps exceptions in the response object, and does not render them, in order to prevent sensitive system information from being rendered. You can override this in several ways:
  - Set `throwExceptions()` in the front controller:

    ```
    $front->throwExceptions(true);
    ```

  - Set `renderExceptions()` in the response object:

    ```
    $response->renderExceptions(true);
    $front->setResponse($response);
    $front->dispatch();

    // or:
    $front->returnResponse(true);
    $response = $front->dispatch();
    $response->renderExceptions(true);
    echo $response;
    ```

- `Zend_Controller_Dispatcher_Interface::dispatch()` now accepts and returns a the section called “The Request Object” object instead of a dispatcher token.
- `Zend_Controller_Router_Interface::route()` now accepts and returns a the section called “The Request Object” object instead of a dispatcher token.
- `Zend_Controller_Action` changes include:
  - The constructor now accepts exactly three arguments, `Zend_Controller_Request_Abstract $request`, `Zend_Controller_Response_Abstract $response`, and `array $params (optional)`. `Zend_Controller_Action::__construct()` uses these to set the request, response, and invokeArgs properties of the object, and if overriding the constructor, you should do

so as well. Better yet, use the `init()` method to do any instance configuration, as this method is called as the final action of the constructor.

- `run()` is no longer defined as final, but is also no longer used by the front controller; it's sole purpose is for using the class as a page controller. It now takes two optional arguments, a `Zend_Controller_Request_Abstract $request` and a `Zend_Controller_Response_Abstract $response`.

- `indexAction()` no longer needs to be defined, but is encouraged as the default action. This allows using the RewriteRouter and action controllers to specify different default action methods.

- `__call()` should be overridden to handle any undefined actions automatically.

- `_redirect()` now takes an optional second argument, the HTTP code to return with the redirect, and an optional third argument, `$prependBase`, that can indicate that the base URL registered with the request object should be prepended to the url specified.

- The `_action` property is no longer set. This property was a `Zend_Controller_Dispatcher_Token`, which no longer exists in the current incarnation. The sole purpose of the token was to provide information about the requested controller, action, and URL parameters. This information is now available in the request object, and can be accessed as follows:

```
// Retrieve the requested controller name
// Access used to be via: $this->_action->getControllerName().
// The example below uses getRequest(), though you may also directly
// access the $_request property; using getRequest() is recommended as
// a parent class may override access to the request object.
$controller = $this->getRequest()->getControllerName();

// Retrieve the requested action name
// Access used to be via: $this->_action->getActionName().
$action = $this->getRequest()->getActionName();

// Retrieve the request parameters
// This hasn't changed; the _getParams() and _getParam() methods simply
// proxy to the request object now.
$params = $this->_getParams();
// request 'foo' parameter, using 'default' as default value if not found
$foo = $this->_getParam('foo', 'default');
```

- `noRouteAction()` has been removed. The appropriate way to handle non-existent action methods should you wish to route them to a default action is using `__call()`:

```
public function __call($method, $args)
{
    // If an unmatched 'Action' method was requested, pass on to the
    // default action method:
    if ('Action' == substr($method, -6)) {
        return $this->defaultAction();
    }
```

```
        throw new Zend_Controller_Exception('Invalid method called');
    }
```

- `Zend_Controller_RewriteRouter::setRewriteBase()` has been removed. Use `Zend_Controller_Front::setBaseUrl()` instead (or Zend_Controller_Request_Http::setBaseUrl(), if using that request class).
- `Zend_Controller_Plugin_Interface` was replaced by `Zend_Controller_Plugin_Abstract`. All methods now accept and return a the section called “The Request Object” object instead of a dispatcher token.

Chapter 9. Zend_Currency

Introduction to Zend_Currency

Zend_Currency is part of the I18n core of the Zend_Framework. It handles all issues related to currency, money representation and formating. And it also provides additional informational methods which include localized informations on currencies, informations about which currency is used in which region and more.

Why should Zend_Currency be used ?

Zend_Currency offers the following functions for handling currency and money related work.

- **Complete Locale support**

  Zend_Currency works with all available locales and has therefor the ability to know about over 100 different localized currency informations. This includes for example currency names, abbreviations, money signs and more.

- **Reusable currency definitions**

  Zend_Currency does not include the value of the currency. This is the reason why it's functionality is not included in Zend_Locale_Format. But this gives the advantage that already defined currency representations can be reused.

- **Fluent interface**

  Zend_Currency includes fluent interfaces where possible.

- **Additional informational methods**

  Zend_Currency includes additional methods which offers informations about in which regions a currency is used or which currency is known to be used in a defined region.

How to work with currencies

To use Zend_Currency within the own application just create an instance of it without any parameter. This will create an instance of Zend_Currency with the actual locale, and defines the currency which has to be used for this locale.

Example 9.1. Creating an instance of Zend_Currency from the actual locale

Expect you have 'en_US' set as actual locale through the users or your environment. By using no parameter while creating the instance you say Zend_Currency to use the actual currency from the locale 'en_US'. This leads to an instance with US Dollar set as actual currency with the formatting rules from 'en_US'.

```
$currency = new Zend_Currency();
```

Since Zend Framework 1.6 Zend_Currency does also support the usage of an application wide locale. You can simply set a Zend_Locale instance to the registry like shown below. With this notation you can forget about setting the locale manually with each instance when you want to use the same locale multiple times.

```
// in your bootstrap file
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// somewhere in your application
$currency = new Zend_Currency();
```

Note

Be aware, that if your system has no default locale, or if the locale of your system can not be detected automatically, `Zend_Currency` will throw an exception. If you have this behaviour you should set the locale you manually.

Of course, depending on your needs, several parameters can be given at creation. Each of this parameters is optional and can be suppressed. Even the order of the parameters can be switched. The meaning of each parameter is described in this list:

- **currency**:

  A locale can include several currencies. Therefor the first parameter **'currency'** can define which currency should be used by giving the short name or full name of that currency. If that currency in not known in any locale an exception will be thrown. Currency short names are always 3 lettered and written uppercase. Well known currency shortnames are for example `USD` or `EUR`. For a list of all known currencies see the informational methods of `Zend_Currency`.

- **locale**:

  The third parameter **'locale'** defines which locale should be used for formatting the currency. The given locale will also be used to get the standard script and currency of this currency if these parameters are not given.

  ### Note

  Note that Zend_Currency only accepts locales which include a region. This means that all given locale which only include the language will throw an exception. For example the locale **en** will throw an exception whereas the locale **en_US** will return **USD** as currency.

Example 9.2. Other examples for creating an instance of Zend_Currency

```
// expect standard locale 'de_AT'

// creates an instance from 'en_US' using 'USD' which is default
// currency for 'en_US'
$currency = new Zend_Currency('en_US');

// creates an instance from the actual locale ('de_AT') using 'EUR' as
// currency
$currency = new Zend_Currency();

// creates an instance using 'EUR' as currency, 'en_US' for number
// formating
$currency = new Zend_Currency('en_US', 'EUR');
```

So you can supress any of these parameters if you want to use the default ones. This has no negative effect on handling the currencies. It can be useful f.e. if you don't know the default currency for a region.

Note

For many countries there are several known currencies. One currency will actually be used and maybe several ancient currencies. If the '**currency**' parameter is suppressed the actual currency will be used. The region '**de**' for example knows the currencies '**EUR**' and '**DEM**'... '**EUR**' is the actual one and will be used if the parameter is suppressed.

Create output from an currency

To get an existing value converted to a currency formatted output the method **toCurrency()** can be used. It takes a value which should be converted. The value itself can be any normalized number.

If you have a localized number you will have to convert it first to an normalized number with Zend_Locale_Format::getNumber(). Afterwards it can be used with `toCurrency()` to create an currency output.

`toCurrency(array $options)` accepts an array with options which can be used to temporary set another format or currency representation. For details about which options can be used see Changing the format of a currency.

Example 9.3. Creating output for an currency

```
// creates an instance with 'en_US' using 'USD' which is the default
// values for 'en_US'
$currency = new Zend_Currency('en_US');

// prints '$ 1,000.00'
echo $currency->toCurrency(1000);

// prints '$ 1.000,00'
echo $currency->toCurrency(1000, array('format' => 'de_AT'));

// prints '$         '
echo $currency->toCurrency(1000, array('script' => 'Arab'));
```

Changing the format of a currency

The format which is used by creation of a `Zend_Currency` instance is of course the standard format. But sometimes it is necessary to change this format for own purposes.

The format of an currency output includes the following parts:

- **Currency symbol, shortname or name**:

  The symbol of the currency is normally displayed within an currency output. It can be suppressed when needed or even overwritten.

- **Currency position**:

  The position of the currency sign is normally automatically defined by the locale. It can be changed if necessary.

- **Script**:

  The script which shall be used to display digits. Detailed information about scripts and their usage can be found in the documentation of `Zend_Locale` in supported number scripts.

- **Number formatting**:

  The amount of currency (formally known as value of money) is formatted by the usage of formatting rules within the locale. For example is in English the ',' sign used as separator for thousands, and in German the '.' sign.

So if you are in need to change the format, you can use the **setFormat()** method. It takes an array which includes all options which you want to change. The `options` array supports the following settings:

- **position**: Defines the position at which the currency description should be displayed. The supported position can be found in this table.

- **script**: Defined which script has to be used for displaying digits. The default script for most locales is **'Latn'**, which includes the digits 0 to 9. Also other scripts like 'Arab' (Arabian) can be used. All supported scripts can be found in this table.

- **format**: Defines which locale has to be used for displaying numbers. This number-format includes for example the thousand separator. If no format is set the locale from the `Zend_Currency` object will be used.
- **display**: Defines which part of the currency has to be used for displaying the currency representation. There are 4 representations which can be used and which are all described in this table.
- **precision**: Defines the precision which has to be used for the currency representation. The default value is **2**.
- **name**: Defines the full currency name which has to be displayed. This option overwrites any currency name which is set through the creation of `Zend_Currency`.
- **currency**: Defines the international abbreviation which has to be displayed. This option overwrites any abbreviation which is set through the creation of `Zend_Currency`.
- **symbol**: Defines the currency symbol which has to be displayed. This option overwrites any symbol which is set through the creation of `Zend_Currency`.

Table 9.1. Constants for the selecting the currency description

constant	description
NO_SYMBOL	Do not display any currency representation
USE_SYMBOL	Display the currency symbol
USE_SHORTNAME	Display the 3 lettered international currency abbreviation
USE_NAME	Display the full currency name

Table 9.2. Constants for the selecting the currency position

constant	description
STANDARD	Set the standard position as defined within the locale
RIGHT	Display the currency representation at the right side of the value
LEFT	Display the currency representation at the left side of the value

Example 9.4. Changing the displayed format of a currency

```
// creates an instance with 'en_US' using 'USD', 'Latin' and 'en_US' as
// these are the default values from 'en_US'
$currency = new Zend_Currency('en_US');

// prints 'US$ 1,000.00'
echo $currency->toCurrency(1000);

$currency->setFormat('display' => Zend_Currency::USE_NAME,
                     'position' => Zend_Currency::RIGHT);

// prints '1.000,00 US Dollar'
echo $currency->toCurrency(1000);

$currency->setFormat('name' => 'American Dollar');
// prints '1.000,00 American Dollar'
echo $currency->toCurrency(1000);
```

Informational methods for Zend_Currency

Of course, `Zend_Currency` supports also methods to get informations about any existing and many ancient currencies from `Zend_Locale`. The supported methods are:

- **getSymbol()**:

  Returns the known sign of the actual currency or a given currency. For example **$** for the US Dollar within the locale '**en_US**.

- **getShortName()**:

  Returns the abbreviation of the actual currency or a given currency. For example **USD** for the US Dollar within the locale '**en_US**.

- **getName()**:

  Returns the full name of the actual currency of a given currency. For example **US Dollar** for the US Dollar within the locale '**en_US**.

- **getRegionList()**:

  Returns a list of regions where the actual currency or a given one is known to be used. It is possible that a currency is used within several regions therefor the return value is always an array.

- **getCurrencyList()**:

  Returns a list of currencies which are known to be used in the given region.

The function `getSymbol()`, `getShortName()` and `getName()` accept two optional parameters. If no parameter is given the expected data will be returned from the actual set currency. The first parameter takes the shortname of a currency. Short names are always three lettered, for example EUR for euro or

USD for US Dollar. The second parameter defines from which locale the data should be read. If no locale is given, the actual set locale is used.

Example 9.5. Getting informations from currencies

```
// creates an instance with 'en_US' using 'USD', 'Latin' and 'en_US'
// as these are the default values from 'en_US'
$currency = new Zend_Currency('en_US');

// prints '$'
echo $currency->getSymbol();

// prints 'EUR'
echo $currency->getShortName('EUR');

// prints 'Österreichische Schilling'
echo $currency->getName('ATS', 'de_AT');

// returns an array with all regions where USD is used
print_r($currency->getRegionList();

// returns an array with all currencies which were ever used in this
// region
print_r($currency->getCurrencyList('de_AT');
```

Settings new default values

The method `setLocale` allows to set a new locale for `Zend_Currency`. When calling this function also all default values for the currency will be overwritten. This includes currency name, abbreviation and symbol.

Example 9.6. Setting a new locale

```
// get the currency for US
$currency = new Zend_Currency('en_US');
print $currency->toCurrency(1000);

// get the currency for AT
$currency->setLocale('de_AT');
print $currency->toCurrency(1000);
```

Speed up Zend_Currency

The work of `Zend_Currency` can be speed up by the usage of `Zend_Cache`. By using the static method `Zend_Currency::setCache($cache)` which accepts one option, an `Zend_Cache` adapter. When you set it, the localization data of the methods from Zend_Currency are cached. For convenience there is also static method `Zend_Currency::getCache()`.

Example 9.7. Caching currencies

```
// creating a cache object
$cache = Zend_Cache::factory('Core',
                             'File',
                             array('lifetime' => 120,
                                   'automatic_serialization' => true),
                             array('cache_dir'
                                       => dirname(__FILE__) . '/_files/'));
Zend_Currency::setCache($cache);
```

Migrating from Previous Versions

The API of `Zend_Currency` has changed in the past to increase the usability. If you started using `Zend_Currency` with a version which is mentioned in this chapter follow the guidelines below to migrate your scripts to the new API.

Migrating from 1.0.2 to 1.0.3 or newer

Creating an object of `Zend_Currency` has become simpler. You now do not longer have to give a script or set it to null. The optional script parameter is now an option which can be set through the `setFormat()` method.

```
$currency = new Zend_Currency($currency, $locale);
```

The `setFormat()` method takes now an array of options which can be set. These options are set permanent and override all previous set values. Also a new option 'precision' has been integrated. The following options has been integrated:

- **position**: Replacement for the old 'rules' parameter.

  script: Replacement for the old 'script' parameter.

  format: Replacement for the old 'locale' parameter which does not set new currencies but only the number format.

  display: Replacement for the old 'rules' parameter.

  precision: New parameter.

  name: Replacement for the ole 'rules' parameter. Sets the full currencies name.

  currency: New parameter.

  symbol: New parameter.

```
$currency->setFormat(array $options);
```

The `toCurrency()` method does no longer support the optional 'script' and 'locale' parameters. Instead it takes an options array which can contain the same keys as for the `setFormat` method.

```
$currency->toCurrency($value, array $options);
```

The methods `getSymbol()`, `getShortName()`, `getName()`, `getRegionList()` and `getCurrencyList()` are no longer static and can be called from within the object. They return the set values of the object if no parameter has been set.

Chapter 10. Zend_Date

Introduction

The Zend_Date component offers a detailed, but simple API for manipulating dates and times. Its methods accept a wide variety of types of information, including date parts, in numerous combinations yielding many features and possibilities above and beyond the existing PHP date related functions. For the very latest manual updates, please see our online manual (frequently synced to Subversion) [http://framework.zend.com/wiki/display/ZFDOCDEV/Home] .

Although simplicity remains the goal, working with localized dates and times while modifying, combining, and comparing parts involves some unavoidable complexity. Dates, as well as times, are often written differently in different locales. For example, some place the month first, while other write the year first when expressing calendar dates. For more information about handling localization and normalization, please refer to Zend_Locale .

Zend_Date also supports abbreviated names of months in many languages. Zend_Locale facilitates the normalization of localized month and weekday names to timestamps, which may, in turn, be shown localized to other regions.

Always Set a Default Timezone

Before using any date related functions in PHP or the Zend Framework, first make certain your application has a correct default timezone, by either setting the TZ environment variable, using the date.timezone php.ini setting, or using date_default_timezone_set() [http://php.net/date_default_timezone_set] . In PHP, we can adjust all date and time related functions to work for a particular user by setting a default timezone according to the user's expectations. For a complete list of timezone settings, see the CLDR Timezone Identifier List [http://unicode.org/cldr/data/diff/supplemental/territory_containment_un_m_49.html] .

Example 10.1. Setting a default timezone

```
// timezone for an American in California
date_default_timezone_set('America/Los_Angeles');
// timezone for a German in Germany
date_default_timezone_set('Europe/Berlin');
```

When creating Zend_Date instances, their timezone will automatically become the current default timezone! Thus, the timezone setting will account for any Daylight Savings Time (DST) in effect, eliminating the need to explicitly specify DST.

Keep in mind that the timezones **UTC** and **GMT** do not include Daylight Saving Time. This means that even if you define per hand that Zend_Date should work with DST, it would automatically be switched back for the instances of Zend_Date which have been set to UTC or GMT.

Why Use Zend_Date?

Zend_Date offers the following features, which extend the scope of PHP date functions:

- Simple API

  `Zend_Date` offers a very simple API, which combines the best of date/time functionality from four programming languages. It is possible, for example, to add or compare two times within a single row.

- Completely internationalized

  All full and abbreviated names of months and weekdays are supported for more than 130 languages. Methods support both input and the output of dates using the localized names of months and weekdays, in the conventional format associated with each locale.

- Unlimited timestamps

  Although PHP 5.2 docs state, "The valid range of a timestamp is typically from Fri, 13 Dec 1901 20:45:54 GMT to Tue, 19 Jan 2038 03:14:07 GMT," `Zend_Date` supports a nearly unlimited range, with the help of the BCMath extension. If BCMath is not available, then Zend_Date will have reduced support only for timestamps within the range of the `float` type supported by your server. "The size of a float is platform-dependent, although a maximum of ~1.8e308 with a precision of roughly 14 decimal digits is a common value (that's 64 bit IEEE format)." [http://www.php.net/float]. Additionally, inherent limitations of float data types, and rounding error of float numbers may introduce errors into calculations. To avoid these problems, the ZF I18n components use BCMath extension, if available.

- Support for ISO_8601 date specifications

  ISO_8601 date specifications are supported. Even partially compliant ISO_8601 date specifications will be identified. These date formats are particularly useful when working with databases. For example, even though MsSQL and MySQL [http://dev.mysql.com/doc/refman/5.0/en/date-and-time-functions.html] differ a little from each other, both are supported by `Zend_Date` using the Zend_Date::ISO_8601 format specification constant. When date strings conform to "Y/m/d" or "Y-m-d H:i:s", according to PHP date() format tokens, use Zend_Date's built-in support for ISO 8601 formatted dates.

- Calculate sunrise and sunset

  For any place and day, the times for sunrise and sunset can be displayed, so that you won't miss a single daylight second for working on your favorite PHP project :)

Theory of Operation

Why is there only one class `Zend_Date` for handling dates and times in the Zend Framework?

Many languages split the handling of times and calendar dates into two classes. However, the Zend Framework strives for extreme simplicity, and forcing the developer to manage different objects with different methods for times and dates becomes a burden in many situations. Since `Zend_Date` methods support working with ambiguous dates that might not include all parts (era, year, month, day, hour, minute, second, timezone), developers enjoy the flexibility and ease of using the same class and the same methods to perform the same manipulations (e.g. addition, subtraction, comparison, merging of date parts, etc.). Splitting the handling of these date fragments into multiple classes would create complications when smooth interoperation is desired with a small learning curve. A single class reduces code duplication for similar operations, without the need for a complex inheritance hierarchy.

Internals

- UNIX Timestamp

All dates and times, even ambiguous ones (e.g. no year), are represented internally as absolute moments in time, represented as a UNIX timestamp expressing the difference between the desired time and January 1st, 1970 00:00:00 GMT/UTC. This was only possible, because `Zend_Date` is not limited to UNIX timestamps nor integer values. The BCMath extension is required to support extremely large dates outside of the range Fri, 13 Dec 1901 20:45:54 GMT to Tue, 19 Jan 2038 03:14:07 GMT. Additional, tiny math errors may arise due to the inherent limitations of float data types and rounding, unless using the BCMath extension.

- Date parts as timestamp offsets

  Thus, an instance object representing three hours would be expressed as three hours after January 1st, 1970 00:00:00 GMT/UTC -i.e. 0 + 3 * 60 * 60 = 10800.

- PHP functions

  Where possible, `Zend_Date` usually uses PHP functions to improve performance.

Basic Methods

The following sections show basic usage of `Zend_Date` primarily by example. For this manual, "dates" always imply a calendar date with a time, even when not explicitly mentioned, and vice-versa. The part not specified defaults to an internal representation of "zero". Thus, adding a date having no calendar date and a time value of 12 hours to another date consisting only of a calendar date would result in a date having that calendar date and a time of "noon".

Setting only a specific date, with no time part, implies a time set to 00:00:00. Conversely, setting only a specific time implies a date internally set to 01.01.1970 plus the number of seconds equal to the elapsed hours, minutes, and seconds identified by the time. Normally, people measure things from a starting point, such as the year 0 A.D. However, many software systems use the first second of the year 1970 as the starting point, and denote times as a timestamp offset counting the number of seconds elapsed from this starting point.

The current date

Without any arguments, constructing an instance returns an object in the default locale with the current, local date using PHP's `time()` function to obtain the UNIX timestamp [http://en.wikipedia.org/wiki/Unix_Time] for the object. Make sure your PHP environment has the correct default timezone .

Example 10.2. Creating the current date

```
$date = new Zend_Date();

// Output of the current timestamp
print $date;
```

Zend_Date by Example

Reviewing basic methods of `Zend_Date` is a good place to start for those unfamiliar with date objects in other languages or frameworks. A small example will be provided for each method below.

Output a Date

The date in a `Zend_Date` object may be obtained as a localized integer or string using the `get()` method. There are many available options, which will be explained in later sections.

Example 10.3. get() - output a date

```
$date = new Zend_Date();

// Output of the desired date
print $date->get();
```

Setting a Date

The `set()` method alters the date stored in the object, and returns the final date value as a timestamp (not an object). Again, there are many options which will be explored in later sections.

Example 10.4. set() - set a date

```
$date = new Zend_Date();

// Setting of a new time
$date->set('13:00:00',Zend_Date::TIMES);
print $date->get(Zend_Date::W3C);
```

Adding and Subtracting Dates

Adding two dates with `add()` usually involves adding a real date in time with an artificial timestramp representing a date part, such as 12 hours, as shown in the example below. Both `add()` and `sub()` use the same set of options as `set()`, which will be explained later.

Example 10.5. add() - adding dates

```
$date = new Zend_Date();

// changes $date by adding 12 hours
$date->add('12:00:00', Zend_Date::TIMES);

echo "Date via get() = ", $date->get(Zend_Date::W3C), "\n";

// use magic __toString() method to call Zend_Date's toString()
echo "Date via toString() = ", $date, "\n";
```

Comparison of dates

All basic `Zend_Date` methods can operate on entire dates contained in the objects, or can operate on date parts, such as comparing the minutes value in a date to an absolute value. For example, the current minutes in the current time may be compared with a specific number of minutes using `compare()`, as in the example below.

Example 10.6. compare() - compare dates

```
$date = new Zend_Date();

// Comparation of both times
if ($date->compare(10, Zend_Date::MINUTE) == -1) {
    print "This hour is less than 10 minutes old";
} else {
    print "This hour is at least 10 minutes old";
}
```

For simple equality comparisons, use `equals()`, which returns a boolean.

Example 10.7. equals() - identify a date or date part

```
$date = new Zend_Date();

// Comparation of the two dates
if ($date->equals(10, Zend_Date::HOUR)) {
    print "It's 10 o'clock. Time to get to work.";
} else {
    print "It is not 10 o'clock. You can keep sleeping.";
}
```

Zend_Date API Overview

While the `Zend_Date` API remains simplistic and unitary, its design remains flexible and powerful through the rich permutations of operations and operands.

Zend_Date Options

Selecting the date format type

Several methods use date format strings, in a way similar to PHP's `date()`. If you are more comfortable with PHP's date format specifier than with ISO format specifiers, then you can use `Zend_Date::setOptions(array('format_type' => 'php'))`. Afterward, use PHP's date format specifiers for all functions which accept a `$format` parameter. Use `Zend_Date::setOptions(array('format_type' => 'iso'))` to switch back to the default mode of supporting only ISO date format tokens. For a list of supported format codes, see the section called “Self-defined OUTPUT formats using PHP's date() format specifiers”

DST and Date Math

When dates are manipulated, sometimes they cross over a DST change, normally resulting in the date losing or gaining an hour. For exmaple, when adding months to a date before a DST change, if the resulting date is after the DST change, then the resulting date will appear to lose or gain an hour, resulting in the time value of the date changing. For boundary dates, such as midnight of the first or last day of a month, adding enough months to cross a date boundary results in the date losing an hour and becoming the last hour of the preceding month, giving the appearance of an "off by 1" error. To avoid this situation, the DST change ignored by using the `fix_dst` option. When crossing the Summer/Winter DST boundary, normally an hour is substracted or added depending on the date. For example, date math crossing the Spring DST leads to a date having a day value one less than expected, if the time part of the date was originally 00:00:00. Since Zend_Date is based on timestamps, and not calendar dates with a time component, the timestamp loses an hour, resulting in the date having a calendar day value one less than expected. To prevent such problems use the option `fix_dst`, which defaults to true, causing DST to have no effect on date "math" (`addMOnth(), subMonth()`). Use `Zend_Date::setOptions(array('fix_dst' => false))` to enable the subtraction or addition of the DST adjustment when performing date "math".

If your actual timezone within the instance of `Zend_Date` is set to UTC or GMT the option `'fix_dst'` will not be used because these two timezones do not work with DST. When you change the timezone for this instance again to a timezone which is not UTC or GMT the previous set 'fix_dst' option will be used again for date "math".

Month Calculations

When adding or substracting months from an existing date, the resulting value for the day of the month might be unexpected, if the original date fell on a day close to the end of the month. For example, when adding one month to January 31st, people familiar with SQL will expect February 28th as the result. On the other side, people familiar with Excel and OpenOffice will expect March 3rd as the result. The problem only occurs, if the resulting month does not have the day, which is set in the original date. For ZF developers, the desired behavior is selectable using the `extend_month` option to choose either the SQL behaviour, if set to false, or the spreadsheet behaviour when set to true. The default behaviour for `extend_month` is false, providing behavior compatible to SQL. By default, `Zend_Date` computes month calculations by truncating dates to the end of the month (if necessary), without wrapping into the next month when the original date designates a day of the month exceeding the number of days in the resulting month. Use `Zend_Date::setOptions(array('extend_month' => true));` to make month calculations work like popular spreadsheet programs.

Speed up date localization and normalization with Zend_Cache

You can speed up `Zend_Date` by using an `Zend_Cache` adapter. This speeds up all methods of `Zend_Date` when you are using localized data. For example all methods which accept `Zend_Date::DATE` and `Zend_Date::TIME` constants would benefit from this. To set an `Zend_Cache` adapter to `Zend_Date` just use `Zend_Date::setOptions(array('cache' => $adapter));`.

Receiving syncronised timestamps with Zend_TimeSync

Normally the clocks from servers and computers differ from each other. `Zend_Date` is able to handle such problems with the help of `Zend_TimeSync`. You can set a timeserver with `Zend_Date::setOptions(array('timesync' => $timeserver));` which will set the offset between the own actual timestamp and the real actual timestamp for all instances of Zend_Date. Using this option does not change the timestamp of existing instances. So best usage is to set it within the bootstrap file.

Working with Date Values

Once input has been normalized via the creation of a `Zend_Date` object, it will have an associated timezone, but an internal representation using standard UNIX timestamps [http://en.wikipedia.org/wiki/Unix_Time] . In order for a date to be rendered in a localized manner, a timezone must be known first. The default timezone is always GMT/UTC. To examine an object's timezone use `getTimeZone())`. To change an object's timezone, use `setTimeZone())`. All manipulations of these objects are assumed to be relative to this timezone.

Beware of mixing and matching operations with date parts between date objects for different timezones, which generally produce undesireable results, unless the manipulations are only related to the timestamp. Operating on `Zend_Date` objects having different timezones generally works, except as just noted, since dates are normalized to UNIX timestamps on instantiation of `Zend_Date`.

Most methods expect a constant selecting the desired `$part` of a date, such as `Zend_Date::HOUR`. These constants are valid for all of the functions below. A list of all available constants is provided in the section called "List of All Constants" . If no `$part` is specified, then `Zend_Date::TIMESTAMP` is assumed. Alternatively, a user-specified format may be used for `$part`, using the same underlying mechanism and format codes as `Zend_Locale_Format::getDate()` . If a date object is constructed using an obviously invalid date (e.g. a month number greater than 12), then `Zend_Date` will throw an exception, unless no specific date format has been selected -i.e. `$part` is either `null` or `Zend_Date::DATES` (a "loose" format).

Example 10.8. User-specified input date format

```
$date1 = new Zend_Date('Feb 31, 2007', null, 'en_US');
echo $date1, "\n"; // outputs "Mar 3, 2007 12:00:00 AM"

$date2 = new Zend_Date('Feb 31, 2007', Zend_Date::DATES, 'en_US');
echo $date2, "\n"; // outputs "Mar 3, 2007 12:00:00 AM"

// strictly restricts interpretation to specified format
$date3 = new Zend_Date('Feb 31, 2007', 'MM.dd.yyyy');
echo $date3, "\n"; // outputs "Mar 3, 2007 12:00:00 AM"
```

If the optional `$locale` parameter is provided, then the `$locale` disambiguates the `$date` operand by replacing month and weekday names for string `$date` operands, and even parsing date strings expressed according to the conventions of that locale (see `Zend_Locale_Format::getDate()`). The automatic normalization of localized `$date` operands of a string type occurs when `$part` is one of the `Zend_Date::DATE*` or `Zend_Date::TIME*` constants. The locale identifies which language should be used to parse month names and weekday names, if the `$date` is a string containing a date. If there is no `$date` input parameter, then the `$locale` parameter specifies the locale to use for localizing output (e.g. the date format for a string representation). Note that the `$date` input parameter might actually have a type name instead (e.g. `$hour` for `addHour()`), although that does not prevent the use of `Zend_Date` objects as arguments for that parameter. If no `$locale` was specified, then the locale of the current object is used to interpret `$date`, or select the localized format for output.

Since Zend Framework 1.6 `Zend_Date` does also support the usage of an application wide locale. You can simply set a `Zend_Locale` instance to the registry like shown below. With this notation you can forget about setting the locale manually with each instance when you want to use the same locale multiple times.

```
// in your bootstrap file
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// somewhere in your application
$date = new Zend_Date('31.Feb.2007');
```

Basic `Zend_Date` Operations Common to Many Date Parts

The methods `add(), sub(), compare(), get(), and set()` operate generically on dates. In each case, the operation is performed on the date held in the instance object. The `$date` operand is required for all of these methods, except `get()`, and may be a `Zend_Date` instance object, a numeric string, or an integer. These methods assume `$date` is a timestamp, if it is not an object. However, the `$part` operand controls which logical part of the two dates are operated on, allowing operations on parts of the object's date, such as year or minute, even when `$date` contains a long form date string, such as, "December 31, 2007 23:59:59". The result of the operation changes the date in the object, except for `compare()`, and `get()`.

Example 10.9. Operating on Parts of Dates

```
$date = new Zend_Date(); // $date's timestamp === time()

// changes $date by adding 12 hours
$date->add('12', Zend_Date::HOUR);
print $date;
```

Convenience methods exist for each combination of the basic operations and several common date parts as shown in the tables below. These convenience methods help us lazy programmers avoid having to type out the date part constants when using the general methods above. Conveniently, they are named by combining a prefix (name of a basic operation) with a suffix (type of date part), such as `addYear()`. In the list below, all combinations of "Date Parts" and "Basic Operations" exist. For example, the operation "add" exists for each of these date parts, including `addDay()`, `addYear()`, etc.

These convenience methods have the same equivalent functionality as the basic operation methods, but expect string and integer `$date` operands containing only the values representing the type indicated by the suffix of the convenience method. Thus, the names of these methods (e.g. "Year" or "Minute") identify the units of the `$date` operand, when `$date` is a string or integer.

List of Date Parts

Table 10.1. Date Parts

Date Part	Explanation
Timestamp [http://en.wikipedia.org/wiki/Unix_Time]	UNIX timestamp, expressed in seconds elapsed since January 1st, 1970 00:00:00 GMT/UTC.
Year [http://en.wikipedia.org/wiki/Gregorian_calendar]	Gregorian calendar year (e.g. 2006)
Month [http://en.wikipedia.org/wiki/Month#Julian_and_Gregorian_calendars]	Gregorian calendar month (1-12, localized names supported)
24 hour clock [http://en.wikipedia.org/wiki/24-hour_clock]	Hours of the day (0-23) denote the hours elapsed, since the start of the day.
minute [http://en.wikipedia.org/wiki/Minute]	Minutes of the hour (0-59) denote minutes elapsed, since the start of the hour.
Second [http://en.wikipedia.org/wiki/Second]	Seconds of the minute (0-59) denote the elapsed seconds, since the start of the minute.
millisecond [http://en.wikipedia.org/wiki/Millisecond]	Milliseconds denote thousandths of a second (0-999). `Zend_Date` supports two additional methods for working with time units smaller than seconds. By default, `Zend_Date` instances use a precision defaulting to milliseconds, as seen using `getFractionalPrecision()`. To change the precision use `setFractionalPrecision($precision)`. However, precision is limited practically to microseconds, since `Zend_Date` uses `microtime()` `[http://php.net/microtime]`.
Day [http://en.wikipedia.org/wiki/Day]	`Zend_Date::DAY_SHORT` is extracted from `$date` if the `$date` operand is an instance of `Zend_Date` or a numeric string. Otherwise, an attempt is made to extract the day according to the conventions documented for these constants: `Zend_Date::WEEKDAY_NARROW`, `Zend_Date::WEEKDAY_NAME`, `Zend_Date::WEEKDAY_SHORT`, `Zend_Date::WEEKDAY` (Gregorian calendar assumed)
Week [http://en.wikipedia.org/wiki/Week]	`Zend_Date::WEEK` is extracted from `$date` if the `$date` operand is an instance of `Zend_Date` or a numeric string. Otherwise an exception is raised. (Gregorian calendar assumed)

Date Part	**Explanation**
Date	`Zend_Date::DAY_MEDIUM` is extracted from `$date` if the `$date` operand is an instance of `Zend_Date`. Otherwise, an attempt is made to normalize the `$date` string into a Zend_Date::DATE_MEDIUM formatted date. The format of `Zend_Date::DAY_MEDIUM` depends on the object's locale.
Weekday	Weekdays are represented numerically as 0 (for Sunday) through 6 (for Saturday). `Zend_Date::WEEKDAY_DIGIT` is extracted from `$date`, if the `$date` operand is an instance of `Zend_Date` or a numeric string. Otherwise, an attempt is made to extract the day according to the conventions documented for these constants: `Zend_Date::WEEKDAY_NARROW`, `Zend_Date::WEEKDAY_NAME`, `Zend_Date::WEEKDAY_SHORT`, `Zend_Date::WEEKDAY` (Gregorian calendar assumed)
DayOfYear	In `Zend_Date`, the day of the year represents the number of calendar days elapsed since the start of the year (0-365). As with other units above, fractions are rounded down to the nearest whole number. (Gregorian calendar assumed)
Arpa [http://www.faqs.org/rfcs/rfc822.html]	Arpa dates (i.e. RFC 822 formatted dates) are supported. Output uses either a "GMT" or "Local differential hours+min" format (see section 5 of RFC 822). Before PHP 5.2.2, using the DATE_RFC822 constant with PHP date functions sometimes produces incorrect results [http://bugs.php.net/bug.php?id=40308]. Zend_Date's results are correct. Example: `Mon, 31 Dec 06 23:59:59 GMT`
Iso [http://en.wikipedia.org/wiki/ISO_8601]	Only complete ISO 8601 dates are supported for output. Example: `2009-02-14T00:31:30+01:00`

List of Date Operations

The basic operations below can be used instead of the convenience operations for specific date parts, if the appropriate constant is used for the `$part` parameter.

Table 10.2. Basic Operations

Basic Operation	Explanation
get()	**get($part = null, $locale = null)** Use `get($part)` to retrieve the date `$part` of this object's date localized to `$locale` as a formatted string or integer. When using the BCMath extension, numeric strings might be returned instead of integers for large values. **NOTE:** Unlike `get()`, the other get*() convenience methods only return instances of `Zend_Date` containing a date representing the selected or computed date/time.
set()	**set($date, $part = null, $locale = null)** Sets the `$part` of the current object to the corresponding value for that part found in the input `$date` having a locale `$locale`.
add()	**add($date, $part = null, $locale = null)** Adds the `$part` of `$date` having a locale `$locale` to the current object's date.
sub()	**sub($date, $part = null, $locale = null)** Subtracts the `$part` of `$date` having a locale `$locale` from the current object's date.
copyPart()	**copyPart($part, $locale = null)** Returns a cloned object, with only `$part` of the object's date copied to the clone, with the clone have its locale arbitrarily set to `$locale` (if specified).
compare()	**compare($date, $part = null, $locale = null)** compares `$part` of `$date` to this object's timestamp, returning 0 if they are equal, 1 if this object's part was more recent than $date's part, otherwise -1.

Comparing Dates

The following basic operations do not have corresponding convenience methods for the date parts listed in the section called “Zend_Date API Overview” .

Table 10.3. Date Comparison Methods

Method	Explanation
equals()	**equals($date, $part = null, $locale = null)** returns true, if `$part` of `$date` having locale `$locale` is the same as this object's date `$part`, otherwise false
isEarlier()	**isEarlier($date, $part = null, $locale = null)** returns true, if `$part` of this object's date is earlier than `$part` of `$date` having a locale `$locale`
isLater()	**isLater($date, $part = null, $locale = null)** returns true, if `$part` of this object's date is later than `$part` of `$date` having a locale `$locale`
isToday()	**isToday()** Tests if today's year, month, and day match this object's date value, using this object's timezone.
isTomorrow()	**isTomorrow()** Tests if tomorrow's year, month, and day match this object's date value, using this object's timezone.
isYesterday()	**isYesterday()** Tests if yesterday's year, month, and day match this object's date value, using this object's timezone.
isLeapYear()	**isLeapYear()** Use `isLeapYear()` to determine if the current object is a leap year, or use Zend_Date::checkLeapYear($year) to check $year, which can be a string, integer, or instance of `Zend_Date`. Is the year a leap year?
isDate()	**isDate($date, $format = null, $locale = null)** This method checks if a given date is a real date and returns true if all checks are ok. It works like php's checkdate() function but can also check for localized month names and for dates extending the range of checkdate() false

Getting Dates and Date Parts

Several methods support retrieving values related to a `Zend_Date` instance.

Table 10.4. Date Output Methods

Method	Explanation
toString()	**toString($format = null, $locale = null)** Invoke directly or via the magic method `__toString()`. The `toString()` method automatically formats the date object's value according to the conventions of the object's locale, or an optionally specified `$locale`. For a list of supported format codes, see the section called “Self-Defined OUTPUT Formats with ISO” .
toArray()	**toArray()** Returns an array representation of the selected date according to the conventions of the object's locale. The returned array is equivalent to PHP's getdate() [http://php.net/getdate] function and includes: • Number of day as '**day**' (`Zend_Date::DAY_SHORT`) • Number of month as '**month**' (`Zend_Date::MONTH_SHORT`) • Year as '**year**' (`Zend_Date::YEAR`) • Hour as '**hour**' (`Zend_Date::HOUR_SHORT`) • Minute as '**minute**' (`Zend_Date::MINUTE_SHORT`) • Second as '**second**' (`Zend_Date::SECOND_SHORT`) • Abbreviated timezone as '**timezone**' (`Zend_Date::TIMEZONE`) • Unix timestamp as '**timestamp**' (`Zend_Date::TIMESTAMP`) • Number of weekday as '**weekday**' (`Zend_Date::WEEKDAY_DIGIT`) • Day of year as '**dayofyear**' (`Zend_Date::DAY_OF_YEAR`) • Week as '**week**' (`Zend_Date::WEEK`) • Delay of timezone to GMT as '**gmtsecs**' (`Zend_Date::GMT_SECS`)
toValue()	**toValue($part = null)** Returns an integer representation of the selected date `$part` according to the conventions of the object's locale. Returns `false` when `$part` selects a non-numeric value, such as `Zend_Date::MONTH_NAME_SHORT`. **NOTE:** This method calls `get()` and casts the result to a PHP integer, which will give unpredictable results, if `get()` returns a numeric string containing a number too large for a PHP integer on your system. Use `get()` instead.
get()	**get($part = null, $locale = null)** This method returns the `$part` of object's date localized to `$locale` as a formatted string or integer. See the section called “List of Date Operations” for more information.
now()	**now($locale = null)** This convenience function is equivalent to `new Zend_Date()`. It returns the current date as a `Zend_Date` object, having `$locale`

Working with Fractions of Seconds

Several methods support retrieving values related to a `Zend_Date` instance.

Table 10.5. Date Output Methods

Method	Explanation
getFractionalPrecision()	Return the precision of the part seconds
setFractionalPrecision()	Set the precision of the part seconds

Sunrise / Sunset

Three methods provide access to geographically localized information about the Sun, including the time of sunrise and sunset.

Table 10.6. Miscellaneous Methods

Method	Explanation
getSunrise($location)	Return the date's time of sunrise
getSunset($location)	Return the date's time of sunset
getSunInfo($location)	Return an array with the date's sun dates

Creation of dates

`Zend_Date` provides several different ways to create a new instance of itself. As there are different needs the most convinient ways will be shown in this chapter.

Create the actual date

The simplest way of creating a date object is to create the actual date. You can either create a new instance with **new Zend_Date()** or use the convinient static method **Zend_Date::now()** which both will return the actual date as new instance of `Zend_Date`. The actual date always include the actual date and time for the actual set timezone.

Example 10.10. Date creation by instance

Date creation by creating a new instance means that you do not need to give an parameter. Of course there are several parameters which will be described later but normally this is the simplest and most used way to get the actual date as `Zend_Date` instance.

```
$date = new Zend_Date();
```

Example 10.11. Static date creation

Sometimes it is easier to use a static method for date creation. Therefor you can use the **`now()`** method. It returns a new instance of `Zend_Date` the same way as if you would use `new Zend_Date()`. But it will always return the actual date and can not be changed by giving optional parameters.

```
$date = Zend_Date::now();
```

Create a date from database

Databases are often used to store date values. But the problem is, that every database outputs it's date values in a different way. `MsSQL` databases use a quite different standard date output than `MySQL` databases. But for simplification `Zend_Date` makes it very easy to create a date from database date values.

Of course each database can be said to convert the output of a defined column to a special value. For example you could convert a `datetime` value to output a minute value. But this is time expensive and often you are in need of handling dates in an other way than expected when creating the database query.

So we have one quick and one convinient way of creating dates from database values.

Example 10.12. Quick creation of dates from database date values

All databases are known to handle queries as fast as possible. They are built to act and respond quick. The quickest way for handling dates is to get unix timestamps from the database. All databases store date values internal as timestamp (not unix timestamp). This means that the time for creating a timestamp through a query is much smaller than converting it to a specified format.

```
// SELECT UNIX_TIMESTAMP(my_datetime_column) FROM my_table
$date = new Zend_Date($unixtimestamp, Zend_Date::TIMESTAMP);
```

Example 10.13. Convenient creation of dates from database date values

The standard output of all databases is quite different even if it looks the same on the first eyecatch. But all are part of the `ISO` Standard and explained through it. So the easiest way of date creation is the usage of `Zend_Date::ISO_8601`. Databases which are known to be recognised by `Zend_Date::ISO_8601` are `MySQL`, `MsSQL` for example. But all databases are also able to return a `ISO 8601` representation of a date column. `ISO 8601` has the big advantage that it is human readable. The disadvantage is that `ISO 8601` needs more time for computation than a simple unix timestamp. But it should also be mentioned that unix timestamps are only supported for dates after 1 January 1970.

```
// SELECT datecolumn FROM my_table
$date = new Zend_Date($datecolumn, Zend_Date::ISO_8601);
```

Create dates from an array

Dates can also be created by the usage of an array. This is a simple and easy way. The used array keys are:

- **day**: day of the date as number
- **month**: month of the date as number
- **year**: full year of the date
- **hour**: hour of the date
- **minute**: minute of the date
- **second**: second of the date

Example 10.14. Date creation by array

Normally you will give a complete date array for creation of a new date instance. But when you do not give all values, the not given array values are zeroed. This means that if f.e. no hour is given the hour **0** is used.

```
$datearray = array('year' => 2006,
                   'month' => 4,
                   'day' => 18,
                   'hour' => 12,
                   'minute' => 3,
                   'second' => 10);
$date = new Zend_Date($datearray)
;
```

```
$datearray = array('year' => 2006, 'month' => 4, 'day' => 18);
$date = new Zend_Date($datearray);
```

Constants for General Date Functions

Whenever a `Zend_Date` method has a `$parts` parameter, one of the constants below can be used as the argument for that parameter, in order to select a specific part of a date or indicate the date format used or desired (e.g. RFC 822).

Using Constants

For example, the constant `Zend_Date::HOUR` can be used in the ways shown below. When working with days of the week, calendar dates, hours, minutes, seconds, and any other date parts that are expressed differently when in different parts of the world, the object's timezone will automatically be used to compute the correct value, even though the internal timestamp is the same for the same moment in time, regardless of the user's physical location in the world. Regardless of the units involved, output must be expressed either as GMT/UTC or localized to a locale. The example output below reflects localization to Europe/GMT+1 hour (e.g. Germany, Austria, France).

Table 10.7. Operations involving Zend_Date::HOUR

Function/input	Description	Original date	Effect/output
get(Zend_Date::HOUR)	Output of the hour	2009-02-13T14:53:27+01:00	14
set(12, Zend_Date::HOUR)	Set new hour	2009-02-13T14:53:27+01:00	2009-02-13T12:53:27+01:00
add(12, Zend_Date::HOUR)	Add hours	2009-02-13T14:53:27+01:00	2009-02-14T02:53:27+01:00
sub(12, Zend_Date::HOUR)	Subtract hours	2009-02-13T14:53:27+01:00	2009-02-13T02:53:27+01:00
compare(12, Zend_Date::HOUR)	Compare hour, returns 0, 1 or -1	2009-02-13T14:53:27+01:00	1 (if object > argument)
copy(Zend_Date::HOUR)	Copies only the hour part	2009-02-13T14:53:27+01:00	1970-01-01T14:00:00+01:00
equals(14, Zend_Date::HOUR)	Compares the hour, returns TRUE or FALSE	2009-02-13T14:53:27+01:00	TRUE
isEarlier(12, Zend_Date::HOUR)	Compares the hour, returns TRUE or FALSE	2009-02-13T14:53:27+01:00	TRUE
isLater(12, Zend_Date::HOUR)	Compares the hour, returns TRUE or FALSE	2009-02-13T14:53:27+01:00	FALSE

List of All Constants

Each part of a date/time has a unique constant in `Zend_Date`. All constants supported by `Zend_Date` are listed below.

Table 10.8. Day Constants

Constant	**Description**	**Date**	**Affected part/example**
Zend_Date::DAY	Day (as a number, two digit)	2009-02-06T14:53:27+01:00	2009-02-**06**T14:53:27+01:00 (06)
Zend_Date::DAY_SHORT	Day (as a number, one or two digit)	2009-02-06T14:53:27+01:00	2009-02-0**6**T14:53:27+01:00 (6)
Zend_Date::WEEKDAY	Weekday (Name of the day, localized, complete)	2009-02-06T14:53:27+01:00	**Friday**
Zend_Date::WEEKDAY_SHORT	Weekday (Name of the day, localized, abbreviated, the first three digits)	2009-02-06T14:53:27+01:00	**Fre** for Friday
Zend_Date::WEEKDAY_NAME	Weekday (Name of the day, localized, abbreviated, the first two digits)	2009-02-06T14:53:27+01:00	**Fr** for Friday
Zend_Date::WEEKDAY_NARROW	Weekday (Name of the day, localized, abbreviated, only the first digit)	2009-02-06T14:53:27+01:00	**F** for Friday
Zend_Date::WEEKDAY_DIGIT	Weekday (0 = Sunday, 6 = Saturday)	2009-02-06T14:53:27+01:00	**5** for Friday
Zend_Date::WEEKDAY_8601	Weekday according to ISO 8601 (1 = Monday, 7 = Sunday)	2009-02-06T14:53:27+01:00	**5** for Friday
Zend_Date::DAY_OF_YEAR	Day (as a number, one or two digit)	2009-02-06T14:53:27+01:00	**43**
Zend_Date::DAY_SUFFIX	English addendum for the day (st, nd, rd, th)	2009-02-06T14:53:27+01:00	**th**

Table 10.9. Week Constants

Constant	**Description**	**Date**	**Affected part/example**
Zend_Date::WEEK	Week (as a number, 1-53)	2009-02-06T14:53:27+01:00	**7**

Table 10.10. Month Constants

Constant	Description	Date	Affected part/example
Zend_Date::MONTH_NAME	Month (Name of the month, localized, complete)	2009-02-06T14:53:27+01:00	**February**
Zend_Date::MONTH_NAME_SHORT	Month (Name of the month, localized, abbreviated, three digit)	2009-02-06T14:53:27+01:00	**Feb**
Zend_Date::MONTH_NAME_NARROW	Month (Name of the month, localized, abbreviated, one digit)	2009-02-06T14:53:27+01:00	**F**
Zend_Date::MONTH	Month (Number of the month, two digit)	2009-02-06T14:53:27+01:00	2009-**02**-06T14:53:27+01:00 (02)
Zend_Date::MONTH_SHORT	Month (Number of the month, one or two digit)	2009-02-06T14:53:27+01:00	2009-0**2**-06T14:53:27+01:00 (2)
Zend_Date::MONTH_DAYS	Number of days for this month (number)	2009-02-06T14:53:27+01:00	**28**

Table 10.11. Year Constants

Constant	Description	Date	Affected part/example
Zend_Date::YEAR	Year (number)	2009-02-06T14:53:27+01:00	**2009**-02-06T14:53:27+01:00
Zend_Date::YEAR_8601	Year according to ISO 8601 (number)	2009-02-06T14:53:27+01:00	**2009**
Zend_Date::YEAR_SHORT	Year (number, two digit)	2009-02-06T14:53:27+01:00	20**09**-02-06T14:53:27+01:00
Zend_Date::YEAR_SHORT_8601	Year according to ISO 8601 (number, two digit)	2009-02-06T14:53:27+01:00	**09**
Zend_Date::LEAPYEAR	Is the year a leap year? (TRUE or FALSE)	2009-02-06T14:53:27+01:00	**FALSE**

Table 10.12. Time Constants

Constant	Description	Date	Affected part/example
Zend_Date::HOUR	Hour (00-23, two digit)	2009-02-06T14:53:27+01:00	**14**
Zend_Date::HOUR_SHORT	Hour (0-23, one or two digit)	2009-02-06T14:53:27+01:00	**14**
Zend_Date::HOUR_SHORT_AM	Hour (1-12, one or two digit)	2009-02-06T14:53:27+01:00	**2**
Zend_Date::HOUR_AM	Hour (01-12, two digit)	2009-02-06T14:53:27+01:00	**02**
Zend_Date::MINUTE	Minute (00-59, two digit)	2009-02-06T14:53:27+01:00	2009-02-06T14:**53**:27+01:00
Zend_Date::MINUTE_SHORT	Minute (0-59, one or two digit)	2009-02-06T14:03:27+01:00	2009-02-06T14:0**3**:27+01:00
Zend_Date::SECOND	Second (00-59, two digit)	2009-02-06T14:53:27+01:00	2009-02-06T14:53:**27**+01:00
Zend_Date::SECOND_SHORT	Second (0-59, one or two digit)	2009-02-06T14:53:07+01:00	2009-02-06T14:53:0**7**+01:00
Zend_Date::MILLISECOND	Millisecond (theoretically infinite)	2009-02-06T14:53:27.20546	2009-02-06T14:53:27.**20546**
Zend_Date::MERIDIEM	Time of day (forenoon/afternoon)	2009-02-06T14:53:27+01:00	**afternoon**
Zend_Date::SWATCH	Swatch Internet Time	2009-02-06T14:53:27+01:00	**620**

Table 10.13. Timezone Constants

Constant	Description	Date	Affected part/example
Zend_Date::TIMEZONE	Name der time zone (string, abbreviated)	2009-02-06T14:53:27+01:00	**CET**
Zend_Date::TIMEZONE_NAME	Name of the time zone (string, complete)	2009-02-06T14:53:27+01:00	**Europe/Paris**
Zend_Date::TIMEZONE_SECS	Difference of the time zone to GMT in seconds (integer)	2009-02-06T14:53:27+01:00	**3600** seconds to GMT
Zend_Date::GMT_DIFF	Difference to GMT in seconds (string)	2009-02-06T14:53:27+01:00	**+0100**
Zend_Date::GMT_DIFF_SEP	Difference to GMT in seconds (string, separated)	2009-02-06T14:53:27+01:00	**+01:00**
Zend_Date::DAYLIGHT	Summer time or Winter time? (TRUE or FALSE)	2009-02-06T14:53:27+01:00	**FALSE**

Table 10.14. Date Format Constants (formats include timezone)

Constant	Description	Date	Affected part/example
Zend_Date::ISO_8601	Date according to ISO 8601 (string, complete)	2009-02-13T14:53:27+01:00	**2009-02-13T14:53:27+01:00**
Zend_Date::RFC_2822	Date according to RFC 2822 (string)	2009-02-13T14:53:27+01:00	**Fri, 13 Feb 2009 14:53:27 +0100**
Zend_Date::TIMESTAMP	Unix time [http://en.wikipedia.org/wiki/Unix_Time] (seconds since 1.1.1970, mixed)	2009-02-13T14:53:27+01:00	**1234533207**
Zend_Date::ATOM	Date according to ATOM (string)	2009-02-13T14:53:27+01:00	**2009-02-13T14:53:27+01:00**
Zend_Date::COOKIE	Date for Cookies (string, for Cookies)	2009-02-13T14:53:27+01:00	**Friday, 13-Feb-09 14:53:27 Europe/Paris**
Zend_Date::RFC_822	Date according to RFC 822 (string)	2009-02-13T14:53:27+01:00	**Fri, 13 Feb 09 14:53:27 +0100**
Zend_Date::RFC_850	Date according to RFC 850 (string)	2009-02-13T14:53:27+01:00	**Friday, 13-Feb-09 14:53:27 Europe/Paris**
Zend_Date::RFC_1036	Date according to RFC 1036 (string)	2009-02-13T14:53:27+01:00	**Fri, 13 Feb 09 14:53:27 +0100**
Zend_Date::RFC_1123	Date according to RFC 1123 (string)	2009-02-13T14:53:27+01:00	**Fri, 13 Feb 2009 14:53:27 +0100**
Zend_Date::RSS	Date for RSS Feeds (string)	2009-02-13T14:53:27+01:00	**Fri, 13 Feb 2009 14:53:27 +0100**
Zend_Date::W3C	Date for HTML/HTTP according to W3C (string)	2009-02-13T14:53:27+01:00	**2009-02-13T14:53:27+01:00**

Especially note `Zend_Date::DATES`, since this format specifier has a unique property within `Zend_Date` as an **input** format specifier. When used as an input format for `$part`, this constant provides the most flexible acceptance of a variety of similar date formats. Heuristics are used to automatically extract dates from an input string and then "fix" simple errors in dates (if any), such as swapping of years, months, and days, when possible.

Table 10.15. Date and Time Formats (format varies by locale)

Constant	Description	Date	Affected part/example
Zend_Date::ERA	Epoch (string, localized, abbreviated)	2009-02-06T14:53:27+01:00	**AD** (anno Domini)
Zend_Date::ERA_NAME	Epoch (string, localized, complete)	2009-02-06T14:53:27+01:00	**anno domini** (anno Domini)
Zend_Date::DATES	Standard date (string, localized, default value).	2009-02-13T14:53:27+01:00	**13.02.2009**
Zend_Date::DATE_FULL	Complete date (string, localized, complete)	2009-02-13T14:53:27+01:00	**Friday, 13. February 2009**
Zend_Date::DATE_LONG	Long date (string, localized, long)	2009-02-13T14:53:27+01:00	**13. February 2009**
Zend_Date::DATE_MEDIUM	Normal date (string, localized, normal)	2009-02-13T14:53:27+01:00	**13.02.2009**
Zend_Date::DATE_SHORT	Abbreviated Date (string, localized, abbreviated)	2009-02-13T14:53:27+01:00	**13.02.09**
Zend_Date::TIMES	Standard time (string, localized, default value)	2009-02-13T14:53:27+01:00	**14:53:27**
Zend_Date::TIME_FULL	Complete time (string, localized, complete)	2009-02-13T14:53:27+01:00	**14:53 Uhr CET**
Zend_Date::TIME_LONG	Long time (string, localized, Long)	2009-02-13T14:53:27+01:00	**14:53:27 CET**
Zend_Date::TIME_MEDIUM	Normal time (string, localized, normal)	2009-02-13T14:53:27+01:00	**14:53:27**
Zend_Date::TIME_SHORT	Abbreviated time (string, localized, abbreviated)	2009-02-13T14:53:27+01:00	**14:53**

Self-Defined OUTPUT Formats with ISO

If you need a date format not shown above, then use a self-defined format composed from the ISO format token specifiers below. The following examples illustrate the usage of constants from the table below to create self-defined ISO formats. The format length is unlimited. Also, multiple usage of format constants is allowed.

The accepted format specifiers can be changed from ISO Format to PHP's date format if you are more comfortable with it. However, not all formats defined in the ISO norm are supported with PHP's date format specifiers. Use the `Zend_Date::setOptions(array('format_type' => 'php'))` method to switch Zend_Date methods from supporting ISO format specifiers to PHP date() type specifiers (see the section called “Self-defined OUTPUT formats using PHP's date() format specifiers” below).

Example 10.15. Example usage for self-defined ISO formats

```
$locale = new Zend_Locale('de_AT');
$date = new Zend_Date(1234567890, false, $locale);
print $date->toString("'Era:GGGG='GGGG, ' Date:yy.MMMM.dd'yy.MMMM.dd");
```

Table 10.16. Constants for ISO 8601 date output

Constant	Description	Corresponds best to	Affected part/example
G	Epoch, localized, abbreviated	Zend_Date::ERA	**AD**
GG	Epoch, localized, abbreviated	Zend_Date::ERA	**AD**
GGG	Epoch, localized, abbreviated	Zend_Date::ERA	**AD**
GGGG	Epoch, localized, complete	Zend_Date::ERA_NAME	**anno domini**
GGGGG	Epoch, localized, abbreviated	Zend_Date::ERA	**a**
y	Year, at least one digit	Zend_Date::YEAR	**9**
yy	Year, at least two digit	Zend_Date::YEAR_SHORT	**09**
yyy	Year, at least three digit	Zend_Date::YEAR	**2009**
yyyy	Year, at least four digit	Zend_Date::YEAR	**2009**
yyyyy	Year, at least five digit	Zend_Date::YEAR	**02009**
Y	Year according to ISO 8601, at least one digit	Zend_Date::YEAR_8601	**9**
YY	Year according to ISO 8601, at least two digit	Zend_Date::YEAR_SHORT_8601	**09**
YYY	Year according to ISO 8601, at least three digit	Zend_Date::YEAR_8601	**2009**
YYYY	Year according to ISO 8601, at least four digit	Zend_Date::YEAR_8601	**2009**
YYYYY	Year according to ISO 8601, at least five digit	Zend_Date::YEAR_8601	**02009**
M	Month, one or two digit	Zend_Date::MONTH_SHORT	**2**
MM	Month, two digit	Zend_Date::MONTH	**02**
MMM	Month, localized, abbreviated	Zend_Date::MONTH_NAME_SHORT	**Feb**
MMMM	Month, localized, complete	Zend_Date::MONTH_NAME	**February**
MMMMM	Month, localized, abbreviated, one digit	Zend_Date::MONTH_NAME_NARROW	**F**
w	Week, one or two digit	Zend_Date::WEEK	**5**
ww	Week, two digit	Zend_Date::WEEK	**05**
d	Day of the month, one or two digit	Zend_Date::DAY_SHORT	**9**
dd	Day of the month, two digit	Zend_Date::DAY	**09**
D	Day of the year, one, two or three digit	Zend_Date::DAY_OF_YEAR	**7**
DD	Day of the year, two or three digit	Zend_Date::DAY_OF_YEAR	**07**
DDD	Day of the year, three digit	Zend_Date::DAY_OF_YEAR	**007**
E	Day of the week, localized, abbreviated, one char	Zend_Date::WEEKDAY_NARROW	**M**
EE	Day of the week, localized, abbreviated, two char	Zend_Date::WEEKDAY_NAME	**Mo**

Constant	Description	Corresponds best to	Affected part/example
EEE	Day of the week, localized, abbreviated, three char	Zend_Date::WEEKDAY_SHORT	**Mon**
EEEE	Day of the week, localized, complete	Zend_Date::WEEKDAY	**Monday**
EEEEE	Day of the week, localized, abbreviated, one digit	Zend_Date::WEEKDAY_NARROW	**M**
e	Number of the day, one digit	Zend_Date::WEEKDAY_NARROW	**4**
ee	Number of the day, two digit	Zend_Date::WEEKDAY_NARROW	**04**
a	Time of day, localized	Zend_Date::MERIDIEM	**vorm.**
h	Hour, (1-12), one or two digit	Zend_Date::HOUR_SHORT_AM	**2**
hh	Hour, (01-12), two digit	Zend_Date::HOUR_AM	**02**
H	Hour, (0-23), one or two digit	Zend_Date::HOUR_SHORT	**2**
HH	Hour, (00-23), two digit	Zend_Date::HOUR	**02**
m	Minute, (0-59), one or two digit	Zend_Date::MINUTE_SHORT	**2**
mm	Minute, (00-59), two digit	Zend_Date::MINUTE	**02**
s	Second, (0-59), one or two digit	Zend_Date::SECOND_SHORT	**2**
ss	Second, (00-59), two digit	Zend_Date::SECOND	**02**
S	Millisecond	Zend_Date::MILLISECOND	**20536**
z	Time zone, localized, abbreviated	Zend_Date::TIMEZONE	**CET**
zz	Time zone, localized, abbreviated	Zend_Date::TIMEZONE	**CET**
zzz	Time zone, localized, abbreviated	Zend_Date::TIMEZONE	**CET**
zzzz	Time zone, localized, complete	Zend_Date::TIMEZONE_NAME	**Europe/Paris**
Z	Difference of time zone	Zend_Date::GMT_DIFF	**+0100**
ZZ	Difference of time zone	Zend_Date::GMT_DIFF	**+0100**
ZZZ	Difference of time zone	Zend_Date::GMT_DIFF	**+0100**
ZZZZ	Difference of time zone, separated	Zend_Date::GMT_DIFF_SEP	**+01:00**
A	Millisecond	Zend_Date::MILLISECOND	**20563**

Note

Note that the default ISO format differs from PHP's format which can be irritating if you have not used in previous. Especially the format specifiers for **Year and Minute** are often not used in the intended way.

For **year** there are two specifiers available which are often mistaken. The **Y** specifier for the ISO year and the **y** specifier for the real year. The difference is small but significant. **Y** calculates the ISO year, which is often used for calendar formats. See for example the 31. December 2007. The real year is 2007, but it is the first day of the first week in the week 1 of the year 2008. So, if you are using 'dd.MM.yyyy' you will get '31.December.2007' but if you use 'dd.MM.YYYY' you will get '31.December.2008'. As you see this is no bug but a expected behaviour depending on the used specifiers.

For **minute** the difference is not so big. ISO uses the specifier **m** for the minute, unlike PHP which uses **i**. So if you are getting no minute in your format check if you have used the right specifier.

Self-defined OUTPUT formats using PHP's date() format specifiers

If you are more comfortable with PHP's date format specifier than with ISO format specifiers, then you can use the `Zend_Date::setOptions(array('format' => 'php'))` method to switch Zend_Date methods from supporting ISO format specifiers to PHP date() type specifiers. Afterwards, all format parameters must be given with PHP's `date()` format specifiers [http://php.net/date] . The PHP date format lacks some of the formats supported by the ISO Format, and vice-versa. If you are not already comfortable with it, then use the standard ISO format instead. Also, if you have legacy code using PHP's date format, then either manually convert it to the ISO format using Zend_Locale_Format::convertPhpToIsoFormat() , or use `setOptions()`. The following examples illustrate the usage of constants from the table below to create self-defined formats.

Example 10.16. Example usage for self-defined formats with PHP specifier

```
$locale = new Zend_Locale('de_AT');
Zend_Date::setOptions(array('format_type' => 'php'));
$date = new Zend_Date(1234567890, false, $locale);

// outputs something like 'February 16, 2007, 3:36 am'
print $date->toString('F j, Y, g:i a');

print $date->toString("'Format:D M j G:i:s T Y='D M j G:i:s T Y");
```

The following table shows the list of PHP date format specifiers with their equivalent Zend_Date constants and CLDR/ISO equivalent format specifiers. In most cases, when the CLDR/ISO format does not have an equivalent format specifier, the PHP format specifier is not altered by `Zend_Locale_Format::convertPhpToIsoFormat()`, and the Zend_Date methods then recognize these "peculiar" PHP format specifiers, even when in the default "ISO" format mode.

Table 10.17. Constants for PHP date output

Constant	Description	Corresponds best to	closest CLDR equivalent	Affected part/example
d	Day of the month, two digit	Zend_Date::DAY	dd	**09**
D	Day of the week, localized, abbreviated, three digit	Zend_Date::WEEKDAY_SHORT	EEE	**Mon**
j	Day of the month, one or two digit	Zend_Date::DAY_SHORT	d	**9**
l (lowercase L)	Day of the week, localized, complete	Zend_Date::WEEKDAY	EEEE	**Monday**
N	Number of the weekday, one digit	Zend_Date::WEEKDAY_8601	e	**4**
S	English suffixes for day of month, two chars	no equivalent	no equivalent	**st**
w	Number of the weekday, 0=sunday, 6=saturday	Zend_Date::WEEKDAY_DIGIT	no equivalent	**4**
z	Day of the year, one, two or three digit	Zend_Date::DAY_OF_YEAR	D	**7**
W	Week, one or two digit	Zend_Date::WEEK	w	**5**
F	Month, localized, complete	Zend_Date::MONTH_NAME	MMMM	**February**
m	Month, two digit	Zend_Date::MONTH	MM	**02**
M	Month, localized, abbreviated	Zend_Date::MONTH_NAME_SHORT	MMM	**Feb**
n	Month, one or two digit	Zend_Date::MONTH_SHORT	M	**2**
t	Number of days per month, one or two digits	Zend_Date::MONTH_DAYS	no equivalent	**30**
L	Leapyear, boolean	Zend_Date::LEAPYEAR	no equivalent	**true**
o	Year according to ISO 8601, at least four digit	Zend_Date::YEAR_8601	YYYY	**2009**
Y	Year, at least four digit	Zend_Date::YEAR	yyyy	**2009**
y	Year, at least two digit	Zend_Date::YEAR_SHORT	yy	**09**

Constant	Description	Corresponds best to	closest CLDR equivalent	Affected part/example
a	Time of day, localized	Zend_Date::MERIDIEM	a (sort of, but likely to be uppercase)	**vorm.**
A	Time of day, localized	Zend_Date::MERIDIEM	a (sort of, but no guarantee that the format is uppercase)	**VORM.**
B	Swatch internet time	Zend_Date::SWATCH	no equivalent	**1463**
g	Hour, (1-12), one or two digit	Zend_Date::HOUR_SHORT_AM	h	**2**
G	Hour, (0-23), one or two digit	Zend_Date::HOUR_SHORT	H	**2**
h	Hour, (01-12), two digit	Zend_Date::HOUR_AM	hh	**02**
H	Hour, (00-23), two digit	Zend_Date::HOUR	HH	**02**
i	Minute, (00-59), two digit	Zend_Date::MINUTE	mm	**02**
s	Second, (00-59), two digit	Zend_Date::SECOND	ss	**02**
e	Time zone, localized, complete	Zend_Date::TIMEZONE_NAME	zzzz	**Europe/Paris**
I	Daylight	Zend_Date::DAYLIGHT	no equivalent	**1**
O	Difference of time zone	Zend_Date::GMT_DIFF	Z or ZZ or ZZZ	**+0100**
P	Difference of time zone, separated	Zend_Date::GMT_DIFF_SEP	ZZZZ	**+01:00**
T	Time zone, localized, abbreviated	Zend_Date::TIMEZONE	z or zz or zzz	**CET**
Z	Time zone offset in seconds	Zend_Date::TIMEZONE_SECS	no equivalent	**3600**
c	Standard Iso format output	Zend_Date::ISO_8601	no equivalent	**2004-02-12T15:19:21+00:00**
r	Standard Rfc 2822 format output	Zend_Date::RFC_2822	no equivalent	**Thu, 21 Dec 2000 16:01:07 +0200**
U	Unix timestamp	Zend_Date::TIMESTAMP	no equivalent	**15275422364**

Working examples

Within this chapter, we will describe several additional functions which are also available through `Zend_Date`. Of course all described functions have additional examples to show the expected working and the simple API for the proper using of them.

Checking dates

Probably most dates you will get as input are strings. But the problem with strings is that you can not be sure if the string is a real date. Therefor `Zend_Date` has spent an own static function to check date strings. `Zend_Locale` has an own function `getDate($date, $locale);` which parses a date and returns the proper and normalized date parts. A monthname for example will be recognised and returned just a month number. But as `Zend_Locale` does not know anything about dates because it is a normalizing and localizing class we have integrated an own function `isDate($date);` which checks this.

`isDate($date, $format, $locale);` can take up to 3 parameters and needs minimum one parameter. So what we need to verify a date is, of course, the date itself as string. The second parameter can be the format which the date is expected to have. If no format is given the standard date format from your locale is used. For details about how formats should look like see the chapter about self defined formats .

The third parameter is also optional as the second parameter and can be used to give a locale. We need the locale to normalize monthnames and daynames. So with the third parameter we are able to recognise dates like '01.Jänner.2000' or '01.January.2000' depending on the given locale.

`isDate();` of course checks if a date does exist. `Zend_Date` itself does not check a date. So it is possible to create a date like '31.February.2000' with `Zend_Date` because `Zend_Date` will automatically correct the date and return the proper date. In our case '03.March.2000'. `isDate()` on the other side does this check and will return false on '31.February.2000' because it knows that this date is impossible.

Example 10.17. Checking dates

```
// Checking dates
$date = '01.03.2000';
if (Zend_Date::isDate($date)) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}

// Checking localized dates
$date = '01 February 2000';
if (Zend_Date::isDate($date,'dd MMMM yyyy', 'en')) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}

// Checking impossible dates
$date = '30 February 2000';
if (Zend_Date::isDate($date,'dd MMMM yyyy', 'en')) {
    print "String $date is a date";
} else {
    print "String $date is NO date";
}
```

Sunrise and Sunset

`Zend_Date` has also functions integrated for getting informations from the sun. Often it is necessary to get the time for sunrise or sunset within a particularly day. This is quite easy with `Zend_Date` as just the expected day has to be given and additionally location for which the sunrise or sunset has to be calculated.

As most people do not know the location of their city we have also spent a helper class which provides the location data for about 250 capital and other big cities around the whole world. Most people could use cities near themself as the difference for locations situated to each other can only be measured within some seconds.

For creating a listbox and choosing a special city the function `Zend_Date_Cities::getCityList` can be used. It returns the names of all available predefined cities for the helper class.

Example 10.18. Getting all available cities

```
// Output the complete list of available cities
print_r (Zend_Date_Cities::getCityList());
```

The location itself can be received with the `Zend_Date_Cities::City()` function. It accepts the name of the city as returned by the `Zend_Date_Cities::getCityList()` function and optional as second parameter the horizon to set.

There are 4 defined horizons which can be used with locations to receive the exact time of sunset and sunrise. The '`horizon`' parameter is always optional in all functions. If it is not set, the '`effective`' horizon is used.

Table 10.18. Types of supported horizons for sunset and sunrise

Horizon	Description	Usage
effective	Standard horizon	Expects the world to be a ball. This horizon is always used if non is defined.
civil	Common horizon	Often used in common medias like TV or radio
nautic	Nautic horizon	Often used in sea navigation
astronomic	Astronomic horizon	Often used for calculation with stars

Of course also a self-defined location can be given and calculated with. Therefor a '`latitude`' and a '`longitude`' has to be given and optional the '`horizon`'.

Example 10.19. Getting the location for a city

```
// Get the location for a defined city
// uses the effective horizon as no horizon is defined
print_r (Zend_Date_Cities::City('Vienna'));

// use the nautic horizon
print_r (Zend_Date_Cities::City('Vienna', 'nautic'));

// self definition of a location
$mylocation = array('latitude' => 41.5, 'longitude' => 13.2446);
```

As now all needed data can be set the next is to create a `Zend_Date` object with the day where sunset or sunrise should be calculated. For the calculation there are 3 functions available. It is possible to calculate sunset with '`getSunset()`', sunrise with '`getSunrise()`' and all available informations related to the sun with '`getSunInfo()`'. After the calculation the `Zend_Date` object will be returned with the calculated time.

Example 10.20. Calculating sun informations

```
// Get the location for a defined city
$city = Zend_Date_Cities::City('Vienna');

// create a date object for the day for which the sun has to be calculated
$date = new Zend_Date('10.03.2007', Zend_Date::ISO_8601, 'de');

// calculate sunset
$sunset = $date->getSunset($city);
print $sunset->get(Zend_Date::ISO_8601);

// calculate all sun informations
$info = $date->getSunInfo($city);
foreach ($info as $sun) {
    print "\n" . $sun->get(Zend_Date::ISO_8601);
}
```

Timezones

Timezones are as important as dates themselves. There are several timezones depending on where in the world a user lives. So working with dates also means to set the proper timezone. This may sound complicated but it's easier as expected. As already mentioned in the first chapter of `Zend_Date` the default timezone has to be set. Either by `php.ini` or by definition within the bootstrap file.

A `Zend_Date` object of course also stores the actual timezone. Even if the timezone is changed after the creation of the object it remembers the original timezone and works with it. It is also not necessary to change the timezone within the code with php functions. `Zend_Date` has two built-in functions which makes it possible to handle this.

`getTimezone()` returns the actual set timezone of within the `Zend_Date` object. Remember that `Zend_Date` is not coupled with php internals. So the returned timezone is not the timezone of the php script but the timezone of the object. `setTimezone($zone)` is the second function and makes it possible to set new timezone for `Zend_Date`. A given timezone is always checked. If it does not exist an exception will be thrown. Additionally the actual scripts or systems timezone can be set to the date object by calling `setTimezone()` without the zone parameter. This is also done automatically when the date object is created.

Example 10.21. Working with timezones

```
// Set a default timezone... this has to be done within the bootstrap
// file or php.ini.
// We do this here just for having a complete example.
date_default_timezone_set('Europe/Vienna');

// create a date object
$date = new Zend_Date('10.03.2007', Zend_Date::DATES, 'de');

// view our date object
print $date->getIso();

// what timezone do we have ?
print $date->getTimezone();

// set another timezone
$date->setTimezone('America/Chicago');

// what timezone do we now have ?
print $date->getTimezone();

// see the changed date object
print $date->getIso();
```

`Zend_Date` always takes the actual timezone for object creation as shown in the first lines of the example. Changing the timezone within the created object also has an effect to the date itself. Dates are always related to a timezone. Changing the timezone for a `Zend_Date` object does not change the time of `Zend_Date`. Remember that internally dates are always stored as timestamps and in GMT. So the timezone means how much hours should be substracted or added to get the actual global time for the own timezone and region.

Having the timezone coupled within `Zend_Date` has another positive effect. It is possible to have several dates with different timezones.

Example 10.22. Multiple timezones

```
// Set a default timezone... this has to be done within the bootstrap
// file or php.ini.
// We do this here just for having a complete example.
date_default_timezone_set('Europe/Vienna');

// create a date object
$date = new Zend_Date('10.03.2007 00:00:00', Zend_Date::ISO_8601, 'de');

// view our date object
print $date->getIso();

// the date stays unchanged even after changeing the timezone
date_default_timezone_set('America/Chicago');
print $date->getIso();

$otherdate = clone $date;
$otherdate->setTimezone('Brazil/Acre');

// view our date object
print $otherdate->getIso();

// set the object to the actual systems timezone
$lastdate = clone $date;
$lastdate->setTimezone();

// view our date object
print $lastdate->getIso();
```

Chapter 11. Zend_Db

Zend_Db_Adapter

Zend_Db and its related classes provide a simple SQL database interface for Zend Framework. The Zend_Db_Adapter is the basic class you use to connect your PHP application to an RDBMS. There is a different Adapter class for each brand of RDBMS.

The Zend_Db Adapters create a bridge from the vendor-specific PHP extensions to a common interface, to help you write PHP applications once and deploy with multiple brands of RDBMS with very little effort.

The interface of the Adapter class is similar to the interface of the PHP Data Objects [http://www.php.net/pdo] extension. Zend_Db provides Adapter classes to PDO drivers for the following RDBMS brands:

- IBM DB2 and Informix Dynamic Server (IDS), using the pdo_ibm [http://www.php.net/pdo-ibm] PHP extension
- MySQL, using the pdo_mysql [http://www.php.net/pdo-mysql] PHP extension
- Microsoft SQL Server, using the pdo_mssql [http://www.php.net/pdo-mssql] PHP extension
- Oracle, using the pdo_oci [http://www.php.net/pdo-oci] PHP extension
- PostgreSQL, using the pdo_pgsql [http://www.php.net/pdo-pgsql] PHP extension
- SQLite, using the pdo_sqlite [http://www.php.net/pdo-sqlite] PHP extension

In addition, Zend_Db provides Adapter classes that utilize PHP database extensions for the following RDBMS brands:

- MySQL, using the mysqli [http://www.php.net/mysqli] PHP extension
- Oracle, using the oci8 [http://www.php.net/oci8] PHP extension
- IBM DB2, using the ibm_db2 [http://www.php.net/ibm_db2] PHP extension
- Firebird/Interbase, using the php_interbase [http://www.php.net/ibase] PHP extension

Note

Each Zend_Db Adapter uses a PHP extension. You must have the respective PHP extension enabled in your PHP environment to use a Zend_Db Adapter. For example, if you use any of the PDO Zend_Db Adapters, you need to enable both the PDO extension and the PDO driver for the brand of RDBMS you use.

Connecting to a Database using an Adapter

This section describes how to create an instance of a database Adapter. This corresponds to making a connection to your RDBMS server from your PHP application.

Using a Zend_Db Adapter Constructor

You can create an instance of an Adapter using its constructor. An Adapter constructor takes one argument, which is an array of parameters used to declare the connection.

Example 11.1. Using an Adapter constructor

```
$db = new Zend_Db_Adapter_Pdo_Mysql(array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

Using the Zend_Db Factory

As an alternative to using an Adapter constructor directly, you can create an instance of an Adapter using the static method `Zend_Db::factory()`. This method dynamically loads the Adapter class file on demand, using Zend_Loader::loadClass().

The first argument is a string that names the base name of the Adapter class. For example the string 'Pdo_Mysql' corresponds to the class Zend_Db_Adapter_Pdo_Mysql. The second argument is the same array of parameters you would have given to the Adapter constructor.

Example 11.2. Using the Adapter factory method

```
// We don't need the following statement because the
// Zend_Db_Adapter_Pdo_Mysql file will be loaded for us by the Zend_Db
// factory method.

// require_once 'Zend/Db/Adapter/Pdo/Mysql.php';

// Automatically load class Zend_Db_Adapter_Pdo_Mysql and create an instance of it
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
));
```

If you create your own class that extends Zend_Db_Adapter_Abstract, but you do not name your class with the "Zend_Db_Adapter" package prefix, you can use the `factory()` method to load your Adapter if you specify the leading portion of the adapter class with the 'adapterNamespace' key in the parameters array.

Example 11.3. Using the Adapter factory method for a custom adapter class

```
// We don't need to load the adapter class file
// because it will be loaded for us by the Zend_Db factory method.

// Automatically load class MyProject_Db_Adapter_Pdo_Mysql and create
// an instance of it.
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'             => '127.0.0.1',
    'username'         => 'webuser',
    'password'         => 'xxxxxxxx',
    'dbname'           => 'test',
    'adapterNamespace' => 'MyProject_Db_Adapter'
));
```

Using Zend_Config with the Zend_Db Factory

Optionally, you may specify either argument of the `factory()` method as an object of type Zend_Config.

If the first argument is a config object, it is expected to contain a property named `adapter`, containing the string naming the adapter class name base. Optionally, the object may contain a property named `params`, with subproperties corresponding to adapter parameter names. This is used only if the second argument of the `factory()` method is absent.

Example 11.4. Using the Adapter factory method with a Zend_Config object

In the example below, a Zend_Config object is created from an array. You can also load data from an external file, for example with Zend_Config_Ini or Zend_Config_Xml.

```
$config = new Zend_Config(
    array(
        'database' => array(
            'adapter' => 'Mysqli',
            'params' => array(
                'dbname' => 'test',
                'username' => 'webuser',
                'password' => 'secret',
            )
        )
    )
);

$db = Zend_Db::factory($config->database);
```

The second argument of the `factory()` method may be an associative array containing entries corresponding to adapter parameters. This argument is optional. If the first argument is of type Zend_Config, it is assumed to contain all parameters, and the second argument is ignored.

Adapter Parameters

The list below explains common parameters recognized by Zend_Db Adapter classes.

- **host**: a string containing a hostname or IP address of the database server. If the database is running on the same host as the PHP application, you may use 'localhost' or '127.0.0.1'.
- **username**: account identifier for authenticating a connection to the RDBMS server.
- **password**: account password credential for authenticating a connection to the RDBMS server.
- **dbname**: database instance name on the RDBMS server.
- **port**: some RDBMS servers can accept network connections on a administrator-specified port number. The port parameter allow you to specify the port to which your PHP application connects, to match the port configured on the RDBMS server.
- **options**: this parameter is an associative array of options that are generic to all Zend_Db_Adapter classes.
- **driver_options**: this parameter is an associative array of additional options that are specific to a given database extension. One typical use of this parameter is to set attributes of a PDO driver.
- **adapterNamespace**: names the initial part of the class name for the adapter, instead of 'Zend_Db_Adapter'. Use this if you need to use the `factory()` method to load a non-Zend database adapter class.

Example 11.5. Passing the case-folding option to the factory

You can specify this option by the constant `Zend_Db::CASE_FOLDING`. This corresponds to the `ATTR_CASE` attribute in PDO and IBM DB2 database drivers, adjusting the case of string keys in query result sets. The option takes values `Zend_Db::CASE_NATURAL` (the default), `Zend_Db::CASE_UPPER`, and `Zend_Db::CASE_LOWER`.

```
$options = array(
    Zend_Db::CASE_FOLDING => Zend_Db::CASE_UPPER
);

$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'options'        => $options
);

$db = Zend_Db::factory('Db2', $params);
```

Example 11.6. Passing the auto-quoting option to the factory

You can specify this option by the constant `Zend_Db::AUTO_QUOTE_IDENTIFIERS`. If the value is `true` (the default), identifiers like table names, column names, and even aliases are delimited in all SQL syntax generated by the Adapter object. This makes it simple to use identifiers that contain SQL keywords, or special characters. If the value is `false`, identifiers are not delimited automatically. If you need to delimit identifiers, you must do so yourself using the `quoteIdentifier()` method.

```
$options = array(
    Zend_Db::AUTO_QUOTE_IDENTIFIERS => false
);

$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'options'        => $options
);

$db = Zend_Db::factory('Pdo_Mysql', $params);
```

Example 11.7. Passing PDO driver options to the factory

```
$pdoParams = array(
    PDO::MYSQL_ATTR_USE_BUFFERED_QUERY => true
);

$params = array(
    'host'           => '127.0.0.1',
    'username'       => 'webuser',
    'password'       => 'xxxxxxxx',
    'dbname'         => 'test',
    'driver_options' => $pdoParams
);

$db = Zend_Db::factory('Pdo_Mysql', $params);

echo $db->getConnection()
        ->getAttribute(PDO::MYSQL_ATTR_USE_BUFFERED_QUERY);
```

Managing Lazy Connections

Creating an instance of an Adapter class does not immediately connect to the RDBMS server. The Adapter saves the connection parameters, and makes the actual connection on demand, the first time you need to execute a query. This ensures that creating an Adapter object is quick and inexpensive. You can create an instance of an Adapter even if you are not certain that you need to run any database queries during the current request your application is serving.

If you need to force the Adapter to connect to the RDBMS, use the `getConnection()` method. This method returns an object for the connection as represented by the respective PHP database extension. For example, if you use any of the Adapter classes for PDO drivers, then `getConnection()` returns the PDO object, after initiating it as a live connection to the specific database.

It can be useful to force the connection if you want to catch any exceptions it throws as a result of invalid account credentials, or other failure to connect to the RDBMS server. These exceptions are not thrown until the connection is made, so it can help simplify your application code if you handle the exceptions in one place, instead of at the time of the first query against the database.

Example 11.8. Handling connection exceptions

```
try {
    $db = Zend_Db::factory('Pdo_Mysql', $parameters);
    $db->getConnection();
} catch (Zend_Db_Adapter_Exception $e) {
    // perhaps a failed login credential, or perhaps the RDBMS is not running
} catch (Zend_Exception $e) {
    // perhaps factory() failed to load the specified Adapter class
}
```

The example database

In the documentation for Zend_Db classes, we use a set of simple tables to illustrate usage of the classes and methods. These example tables could store information for tracking bugs in a software development project. The database contains four tables:

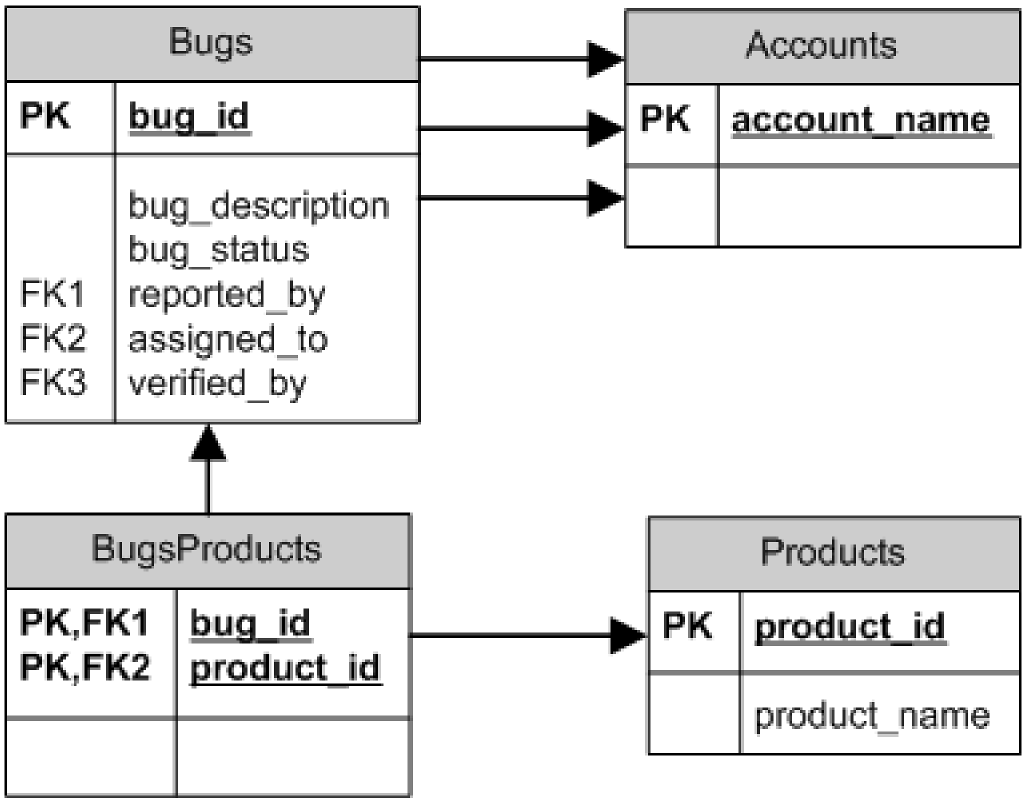
- **accounts** stores information about each user of the bug-tracking database.
- **products** stores information about each product for which a bug can be logged.
- **bugs** stores information about bugs, including that current state of the bug, the person who reported the bug, the person who is assigned to fix the bug, and the person who is assigned to verify the fix.
- **bugs_products** stores a relationship between bugs and products. This implements a many-to-many relationship, because a given bug may be relevant to multiple products, and of course a given product can have multiple bugs.

The following SQL data definition language pseudocode describes the tables in this example database. These example tables are used extensively by the automated unit tests for Zend_Db.

```
CREATE TABLE accounts (
  account_name      VARCHAR(100) NOT NULL PRIMARY KEY
);

CREATE TABLE products (
  product_id        INTEGER NOT NULL PRIMARY KEY,
  product_name      VARCHAR(100)
);

CREATE TABLE bugs (
  bug_id            INTEGER NOT NULL PRIMARY KEY,
  bug_description   VARCHAR(100),
  bug_status        VARCHAR(20),
  reported_by       VARCHAR(100) REFERENCES accounts(account_name),
  assigned_to       VARCHAR(100) REFERENCES accounts(account_name),
  verified_by       VARCHAR(100) REFERENCES accounts(account_name)
);

CREATE TABLE bugs_products (
  bug_id            INTEGER NOT NULL REFERENCES bugs,
  product_id        INTEGER NOT NULL REFERENCES products,
  PRIMARY KEY       (bug_id, product_id)
);
```

Also notice that the bugs table contains multiple foreign key references to the accounts table. Each of these foreign keys may reference a different row in the accounts table for a given bug.

The diagram below illustrates the physical data model of the example database.



Reading Query Results

This section describes methods of the Adapter class with which you can run SELECT queries and retrieve the query results.

Fetching a Complete Result Set

You can run a SQL SELECT query and retrieve its results in one step using the `fetchAll()` method.

The first argument to this method is a string containing a SELECT statement. Alternatively, the first argument can be an object of class Zend_Db_Select. The Adapter automatically converts this object to a string representation of the SELECT statement.

The second argument to `fetchAll()` is an array of values to substitute for parameter placeholders in the SQL statement.

Example 11.9. Using fetchAll()

```
$sql = 'SELECT * FROM bugs WHERE bug_id = ?';

$result = $db->fetchAll($sql, 2);
```

Changing the Fetch Mode

By default, `fetchAll()` returns an array of rows, each of which is an associative array. The keys of the associative array are the columns or column aliases named in the select query.

You can specify a different style of fetching results using the `setFetchMode()` method. The modes supported are identified by constants:

- **Zend_Db::FETCH_ASSOC**: return data in an array of associative arrays. The array keys are column names, as strings. This is the default fetch mode for Zend_Db_Adapter classes.

  Note that if your select-list contains more than one column with the same name, for example if they are from two different tables in a JOIN, there can be only one entry in the associative array for a given name. If you use the FETCH_ASSOC mode, you should specify column aliases in your SELECT query to ensure that the names result in unique array keys.

  By default, these strings are returned as they are returned by the database driver. This is typically the spelling of the column in the RDBMS server. You can specify the case for these strings, using the `Zend_Db::CASE_FOLDING` option. Specify this when instantiating the Adapter. See Example 11.5, “Passing the case-folding option to the factory”.

- **Zend_Db::FETCH_NUM**: return data in an array of arrays. The arrays are indexed by integers, corresponding to the position of the respective field in the select-list of the query.

- **Zend_Db::FETCH_BOTH**: return data in an array of arrays. The array keys are both strings as used in the FETCH_ASSOC mode, and integers as used in the FETCH_NUM mode. Note that the number of elements in the array is double that which would be in the array if you used either FETCH_ASSOC or FETCH_NUM.

- **Zend_Db::FETCH_COLUMN**: return data in an array of values. The value in each array is the value returned by one column of the result set. By default, this is the first column, indexed by 0.
- **Zend_Db::FETCH_OBJ**: return data in an array of objects. The default class is the PHP built-in class stdClass. Columns of the result set are available as public properties of the object.

Example 11.10. Using setFetchMode()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchAll('SELECT * FROM bugs WHERE bug_id = ?', 2);

// $result is an array of objects
echo $result[0]->bug_description;
```

Fetching a Result Set as an Associative Array

The `fetchAssoc()` method returns data in an array of associative arrays, regardless of what value you have set for the fetch mode.

Example 11.11. Using fetchAssoc()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchAssoc('SELECT * FROM bugs WHERE bug_id = ?', 2);

// $result is an array of associative arrays, in spite of the fetch mode
echo $result[0]['bug_description'];
```

Fetching a Single Column from a Result Set

The `fetchCol()` method returns data in an array of values, regardless of the value you have set for the fetch mode. This only returns the first column returned by the query. Any other columns returned by the query are discarded. If you need to return a column other than the first, see the section called “Fetching a Single Column from a Result Set”.

Example 11.12. Using fetchCol()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchCol(
    'SELECT bug_description, bug_id FROM bugs WHERE bug_id = ?', 2);

// contains bug_description; bug_id is not returned
echo $result[0];
```

Fetching Key-Value Pairs from a Result Set

The `fetchPairs()` method returns data in an array of key-value pairs, as an associative array with a single entry per row. The key of this associative array is taken from the first column returned by the SELECT query. The value is taken from the second column returned by the SELECT query. Any other columns returned by the query are discarded.

You should design the SELECT query so that the first column returned has unique values. If there are duplicates values in the first column, entries in the associative array will be overwritten.

Example 11.13. Using fetchPairs()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchPairs('SELECT bug_id, bug_status FROM bugs');

echo $result[2];
```

Fetching a Single Row from a Result Set

The `fetchRow()` method returns data using the current fetch mode, but it returns only the first row fetched from the result set.

Example 11.14. Using fetchRow()

```
$db->setFetchMode(Zend_Db::FETCH_OBJ);

$result = $db->fetchRow('SELECT * FROM bugs WHERE bug_id = 2');

// note that $result is a single object, not an array of objects
echo $result->bug_description;
```

Fetching a Single Scalar from a Result Set

The `fetchOne()` method is like a combination of `fetchRow()` with `fetchCol()`, in that it returns data only for the first row fetched from the result set, and it returns only the value of the first column in that row. Therefore it returns only a single scalar value, not an array or an object.

Example 11.15. Using fetchOne()

```
$result = $db->fetchOne('SELECT bug_status FROM bugs WHERE bug_id = 2');

// this is a single string value
echo $result;
```

Writing Changes to the Database

You can use the Adapter class to write new data or change existing data in your database. This section describes methods to do these operations.

Inserting Data

You can add new rows to a table in your database using the `insert()` method. The first argument is a string that names the table, and the second argument is an associative array, mapping column names to data values.

Example 11.16. Inserting to a table

```
$data = array(
    'created_on'      => '2007-03-22',
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);

$db->insert('bugs', $data);
```

Columns you exclude from the array of data are not specified to the database. Therefore, they follow the same rules that an SQL INSERT statement follows: if the column has a DEFAULT clause, the column takes that value in the row created, otherwise the column is left in a NULL state.

By default, the values in your data array are inserted using parameters. This reduces risk of some types of security issues. You don't need to apply escaping or quoting to values in the data array.

You might need values in the data array to be treated as SQL expressions, in which case they should not be quoted. By default, all data values passed as strings are treated as string literals. To specify that the value is an SQL expression and therefore should not be quoted, pass the value in the data array as an object of type Zend_Db_Expr instead of a plain string.

Example 11.17. Inserting expressions to a table

```
$data = array(
    'created_on'      => new Zend_Db_Expr('CURDATE()'),
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);

$db->insert('bugs', $data);
```

Retrieving a Generated Value

Some RDBMS brands support auto-incrementing primary keys. A table defined this way generates a primary key value automatically during an INSERT of a new row. The return value of the `insert()` method is *not* the last inserted ID, because the table might not have an auto-incremented column. Instead, the return value is the number of rows affected (usually 1).

If your table is defined with an auto-incrementing primary key, you can call the `lastInsertId()` method after the insert. This method returns the last value generated in the scope of the current database connection.

Example 11.18. Using lastInsertId() for an auto-increment key

```
$db->insert('bugs', $data);

// return the last value generated by an auto-increment column
$id = $db->lastInsertId();
```

Some RDBMS brands support a sequence object, which generates unique values to serve as primary key values. To support sequences, the `lastInsertId()` method accepts two optional string arguments. These arguments name the table and the column, assuming you have followed the convention that a sequence is named using the table and column names for which the sequence generates values, and a suffix "_seq". This is based on the convention used by PostgreSQL when naming sequences for SERIAL columns. For example, a table "bugs" with primary key column "bug_id" would use a sequence named "bugs_bug_id_seq".

Example 11.19. Using lastInsertId() for a sequence

```
$db->insert('bugs', $data);

// return the last value generated by sequence 'bugs_bug_id_seq'.
$id = $db->lastInsertId('bugs', 'bug_id');

// alternatively, return the last value generated by sequence 'bugs_seq'.
$id = $db->lastInsertId('bugs');
```

If the name of your sequence object does not follow this naming convention, use the `lastSequenceId()` method instead. This method takes a single string argument, naming the sequence literally.

Example 11.20. Using lastSequenceId()

```
$db->insert('bugs', $data);

// return the last value generated by sequence 'bugs_id_gen'.
$id = $db->lastSequenceId('bugs_id_gen');
```

For RDBMS brands that don't support sequences, including MySQL, Microsoft SQL Server, and SQLite, the arguments to the lastInsertId() method are ignored, and the value returned is the most recent value generated for any table by INSERT operations during the current connection. For these RDBMS brands, the lastSequenceId() method always returns `null`.

Why not use "SELECT MAX(id) FROM table"?

Sometimes this query returns the most recent primary key value inserted into the table. However, this technique is not safe to use in an environment where multiple clients are inserting records to the database. It is possible, and therefore is bound to happen eventually, that another client inserts another row in the instant between the insert performed by your client application and your query for the MAX(id) value. Thus the value returned does not identify the row you inserted, it identifies the row inserted by some other client. There is no way to know when this has happened.

Using a strong transaction isolation mode such as "repeatable read" can mitigate this risk, but some RDBMS brands don't support the transaction isolation required for this, or else your application may use a lower transaction isolation mode by design.

Furthermore, using an expression like "MAX(id)+1" to generate a new value for a primary key is not safe, because two clients could do this query simultaneously, and then both use the same calculated value for their next INSERT operation.

All RDBMS brands provide mechanisms to generate unique values, and to return the last value generated. These mechanisms necessarily work outside of the scope of transaction isolation, so there is no chance of two clients generating the same value, and there is no chance that the value generated by another client could be reported to your client's connection as the last value generated.

Updating Data

You can update rows in a database table using the `update()` method of an Adapter. This method takes three arguments: the first is the name of the table; the second is an associative array mapping columns to change to new values to assign to these columns.

The values in the data array are treated as string literals. See the section called “Inserting Data” for information on using SQL expressions in the data array.

The third argument is a string containing an SQL expression that is used as criteria for the rows to change. The values and identifiers in this argument are not quoted or escaped. You are responsible for ensuring that any dynamic content is interpolated into this string safely. See the section called “Quoting Values and Identifiers” for methods to help you do this.

The return value is the number of rows affected by the update operation.

Example 11.21. Updating rows

```
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);

$n = $db->update('bugs', $data, 'bug_id = 2');
```

If you omit the third argument, then all rows in the database table are updated with the values specified in the data array.

If you provide an array of strings as the third argument, these strings are joined together as terms in an expression separated by `AND` operators.

Example 11.22. Updating rows using an array of expressions

```
$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);

$where[] = "reported_by = 'goofy'";
$where[] = "bug_status = 'OPEN'";

$n = $db->update('bugs', $data, $where);

// Resulting SQL is:
//  UPDATE "bugs" SET "update_on" = '2007-03-23', "bug_status" = 'FIXED'
//  WHERE ("reported_by" = 'goofy') AND ("bug_status" = 'OPEN')
```

Deleting Data

You can delete rows from a database table using the `delete()` method. This method takes two arguments: the first is a string naming the table.

The second argument is a string containing an SQL expression that is used as criteria for the rows to delete. The values and identifiers in this argument are not quoted or escaped. You are responsible for ensuring that any dynamic content is interpolated into this string safely. See the section called “Quoting Values and Identifiers” for methods to help you do this.

The return value is the number of rows affected by the delete operation.

Example 11.23. Deleting rows

```
$n = $db->delete('bugs', 'bug_id = 3');
```

If you omit the second argument, the result is that all rows in the database table are deleted.

If you provide an array of strings as the second argument, these strings are joined together as terms in an expression separated by `AND` operators.

Quoting Values and Identifiers

When you form SQL queries, often it is the case that you need to include the values of PHP variables in SQL expressions. This is risky, because if the value in a PHP string contains certain symbols, such as the quote symbol, it could result in invalid SQL. For example, notice the imbalanced quote characters in the following query:

```
$name = "O'Reilly";
$sql = "SELECT * FROM bugs WHERE reported_by = '$name'";

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O'Reilly'
```

Even worse is the risk that such code mistakes might be exploited deliberately by a person who is trying to manipulate the function of your web application. If they can specify the value of a PHP variable through the use of an HTTP parameter or other mechanism, they might be able to make your SQL queries do things that you didn't intend them to do, such as return data to which the person should not have privilege to read. This is a serious and widespread technique for violating application security, known as "SQL Injection" (see http://en.wikipedia.org/wiki/SQL_Injection).

The Zend_Db Adapter class provides convenient functions to help you reduce vulnerabilities to SQL Injection attacks in your PHP code. The solution is to escape special characters such as quotes in PHP values before they are interpolated into your SQL strings. This protects against both accidental and deliberate manipulation of SQL strings by PHP variables that contain special characters.

Using `quote()`

The `quote()` method accepts a single argument, a scalar string value. It returns the value with special characters escaped in a manner appropriate for the RDBMS you are using, and surrounded by string value delimiters. The standard SQL string value delimiter is the single-quote (`'`).

Example 11.24. Using quote()

```
$name = $db->quote("O'Reilly");
echo $name;
// 'O\'Reilly'

$sql = "SELECT * FROM bugs WHERE reported_by = $name";

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

Note that the return value of `quote()` includes the quote delimiters around the string. This is different from some functions that escape special characters but do not add the quote delimiters, for example mysql_real_escape_string() [http://www.php.net/mysqli_real_escape_string].

Values may need to be quoted or not quoted according to the SQL datatype context in which they are used. For instance, in some RDBMS brands, an integer value must not be quoted as a string if it is compared to an integer-type column or expression. In other words, the following is an error in some SQL implementations, assuming `intColumn` has a SQL datatype of `INTEGER`

```
SELECT * FROM atable WHERE intColumn = '123'
```

You can use the optional second argument to the `quote()` method to apply quoting selectively for the SQL datatype you specify.

Example 11.25. Using quote() with a SQL type

```
$value = '1234';
$sql = 'SELECT * FROM atable WHERE intColumn = '
     . $db->quote($value, 'INTEGER');
```

Each Zend_Db_Adapter class has encoded the names of numeric SQL datatypes for the respective brand of RDBMS. You can also use the constants `Zend_Db::INT_TYPE`, `Zend_Db::BIGINT_TYPE`, and `Zend_Db::FLOAT_TYPE` to write code in a more RDBMS-independent way.

Zend_Db_Table specifies SQL types to `quote()` automatically when generating SQL queries that reference a table's key columns.

Using `quoteInto()`

The most typical usage of quoting is to interpolate a PHP variable into a SQL expression or statement. You can use the `quoteInto()` method to do this in one step. This method takes two arguments: the first argument is a string containing a placeholder symbol (`?`), and the second argument is a value or PHP variable that should be substituted for that placeholder.

The placeholder symbol is the same symbol used by many RDBMS brands for positional parameters, but the `quoteInto()` method only emulates query parameters. The method simply interpolates the value into the string, escapes special characters, and applies quotes around it. True query parameters maintain the separation between the SQL string and the parameters as the statement is parsed in the RDBMS server.

Example 11.26. Using quoteInto()

```
$sql = $db->quoteInto("SELECT * FROM bugs WHERE reported_by = ?", "O'Reilly");

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 'O\'Reilly'
```

You can use the optional third parameter of `quoteInto()` to specify the SQL datatype. Numeric datatypes are not quoted, and other types are quoted.

Example 11.27. Using quoteInto() with a SQL type

```
$sql = $db
    ->quoteInto("SELECT * FROM bugs WHERE bug_id = ?", '1234', 'INTEGER');

echo $sql;
// SELECT * FROM bugs WHERE reported_by = 1234
```

Using `quoteIdentifier()`

Values are not the only part of SQL syntax that might need to be variable. If you use PHP variables to name tables, columns, or other identifiers in your SQL statements, you might need to quote these strings too. By default, SQL identifiers have syntax rules like PHP and most other programming languages. For example, identifiers should not contain spaces, certain punctuation or special characters, or international characters. Also certain words are reserved for SQL syntax, and should not be used as identifiers.

However, SQL has a feature called *delimited identifiers*, which allows broader choices for the spelling of identifiers. If you enclose a SQL identifier in the proper types of quotes, you can use identifiers with spellings that would be invalid without the quotes. Delimited identifiers can contain spaces, punctuation, or international characters. You can also use SQL reserved words if you enclose them in identifier delimiters.

The `quoteIdentifier()` method works like `quote()`, but it applies the identifier delimiter characters to the string according to the type of Adapter you use. For example, standard SQL uses double-quotes (`"`) for identifier delimiters, and most RDBMS brands use that symbol. MySQL uses back-quotes (`` ` ``) by default. The `quoteIdentifier()` method also escapes special characters within the string argument.

Example 11.28. Using quoteIdentifier()

```
// we might have a table name that is an SQL reserved word
$tableName = $db->quoteIdentifier("order");

$sql = "SELECT * FROM $tableName";

echo $sql
// SELECT * FROM "order"
```

SQL delimited identifiers are case-sensitive, unlike unquoted identifiers. Therefore, if you use delimited identifiers, you must use the spelling of the identifier exactly as it is stored in your schema, including the case of the letters.

In most cases where SQL is generated within Zend_Db classes, the default is that all identifiers are delimited automatically. You can change this behavior with the option `Zend_Db::AUTO_QUOTE_IDENTIFIERS`. Specify this when instantiating the Adapter. See Example 11.6, “Passing the auto-quoting option to the factory”.

Controlling Database Transactions

Databases define transactions as logical units of work that can be committed or rolled back as a single change, even if they operate on multiple tables. All queries to a database are executed within the context of a transaction, even if the database driver manages them implicitly. This is called *auto-commit* mode, in which the database driver creates a transaction for every statement you execute, and commits that transaction after your SQL statement has been executed. By default, all Zend_Db Adapter classes operate in auto-commit mode.

Alternatively, you can specify the beginning and resolution of a transaction, and thus control how many SQL queries are included in a single group that is committed (or rolled back) as a single operation. Use the `beginTransaction()` method to initiate a transaction. Subsequent SQL statements are executed in the context of the same transaction until you resolve it explicitly.

To resolve the transaction, use either the `commit()` or `rollBack()` methods. The `commit()` method marks changes made during your transaction as committed, which means the effects of these changes are shown in queries run in other transactions.

The `rollBack()` method does the opposite: it discards the changes made during your transaction. The changes are effectively undone, and the state of the data returns to how it was before you began your transaction. However, rolling back your transaction has no effect on changes made by other transactions running concurrently.

After you resolve this transaction, `Zend_Db_Adapter` returns to auto-commit mode until you call `beginTransaction()` again.

Example 11.29. Managing a transaction to ensure consistency

```
// Start a transaction explicitly.
$db->beginTransaction();

try {
    // Attempt to execute one or more queries:
    $db->query(...);
    $db->query(...);
    $db->query(...);

    // If all succeed, commit the transaction and all changes
    // are committed at once.
    $db->commit();

} catch (Exception $e) {
    // If any of the queries failed and threw an exception,
    // we want to roll back the whole transaction, reversing
    // changes made in the transaction, even those that succeeded.
    // Thus all changes are committed together, or none are.
    $db->rollBack();
    echo $e->getMessage();
}
```

Listing and Describing Tables

The `listTables()` method returns an array of strings, naming all tables in the current database.

The `describeTable()` method returns an associative array of metadata about a table. Specify the name of the table as a string in the first argument to this method. The second argument is optional, and names the schema in which the table exists.

The keys of the associative array returned are the column names of the table. The value corresponding to each column is also an associative array, with the following keys and values:

Table 11.1. Metadata fields returned by describeTable()

Key	**Type**	**Description**
SCHEMA_NAME	(string)	Name of the database schema in which this table exists.
TABLE_NAME	(string)	Name of the table to which this column belongs.
COLUMN_NAME	(string)	Name of the column.
COLUMN_POSITION	(integer)	Ordinal position of the column in the table.
DATA_TYPE	(string)	RDBMS name of the datatype of the column.
DEFAULT	(string)	Default value for the column, if any.
NULLABLE	(boolean)	True if the column accepts SQL NULLs, false if the column has a NOT NULL constraint.
LENGTH	(integer)	Length or size of the column as reported by the RDBMS.
SCALE	(integer)	Scale of SQL NUMERIC or DECIMAL type.
PRECISION	(integer)	Precision of SQL NUMERIC or DECIMAL type.
UNSIGNED	(boolean)	True if an integer-based type is reported as UNSIGNED.
PRIMARY	(boolean)	True if the column is part of the primary key of this table.
PRIMARY_POSITION	(integer)	Ordinal position (1-based) of the column in the primary key.
IDENTITY	(boolean)	True if the column uses an auto-generated value.

How the IDENTITY metadata field relates to specific RDBMS

The IDENTITY metadata field was chosen as an 'idiomatic' term to represent a relation to surrogate keys. This field can be commonly known by the following values:-

- `IDENTITY` - DB2, MSSQL
- `AUTO_INCREMENT` - MySQL
- `SERIAL` - PostgreSQL
- `SEQUENCE` - Oracle

If no table exists matching the table name and optional schema name specified, then `describeTable()` returns an empty array.

Closing a Connection

Normally it is not necessary to close a database connection. PHP automatically cleans up all resources and the end of a request. Database extensions are designed to close the connection as the reference to the resource object is cleaned up.

However, if you have a long-duration PHP script that initiates many database connections, you might need to close the connection, to avoid exhausting the capacity of your RDBMS server. You can use the Adapter's `closeConnection()` method to explicitly close the underlying database connection.

Example 11.30. Closing a database connection

```
$db->closeConnection();
```

Does Zend_Db support persistent connections?

The usage of persistent connections is not supported or encouraged in Zend_Db.

Using persistent connections can cause an excess of idle connections on the RDBMS server, which causes more problems than any performance gain you might achieve by reducing the overhead of making connections.

Database connections have state. That is, some objects in the RDBMS server exist in session scope. Examples are locks, user variables, temporary tables, and information about the most recently executed query, such as rows affected, and last generated id value. If you use persistent connections, your application could access invalid or privileged data that were created in a previous PHP request.

Running Other Database Statements

There might be cases in which you need to access the connection object directly, as provided by the PHP database extension. Some of these extensions may offer features that are not surfaced by methods of Zend_Db_Adapter_Abstract.

For example, all SQL statements run by Zend_Db are prepared, then executed. However, some database features are incompatible with prepared statements. DDL statements like CREATE and ALTER cannot be prepared in MySQL. Also, SQL statements don't benefit from the MySQL Query Cache [http://dev.mysql.com/doc/refman/5.1/en/query-cache-how.html], prior to MySQL 5.1.17.

Most PHP database extensions provide a method to execute SQL statements without preparing them. For example, in PDO, this method is `exec()`. You can access the connection object in the PHP extension directly using getConnection().

Example 11.31. Running a non-prepared statement in a PDO adapter

```
$result = $db->getConnection()->exec('DROP TABLE bugs');
```

Similarly, you can access other methods or properties that are specific to PHP database extensions. Be aware, though, that by doing this you might constrain your application to the interface provided by the extension for a specific brand of RDBMS.

In future versions of Zend_Db, there will be opportunities to add method entry points for functionality that is common to the supported PHP database extensions. This will not affect backward compatibility.

Notes on Specific Adapters

This section lists differences between the Adapter classes of which you should be aware.

IBM DB2

- Specify this Adapter to the factory() method with the name 'Db2'.
- This Adapter uses the PHP extension ibm_db2.
- IBM DB2 supports both sequences and auto-incrementing keys. Therefore the arguments to `lastInsertId()` are optional. If you give no arguments, the Adapter returns the last value generated for an auto-increment key. If you give arguments, the Adapter returns the last value generated by the sequence named according to the convention '*table_column*_seq'.

MySQLi

- Specify this Adapter to the `factory()` method with the name 'Mysqli'.
- This Adapter utilizes the PHP extension mysqli.
- MySQL does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `null`.

Oracle

- Specify this Adapter to the `factory()` method with the name 'Oracle'.
- This Adapter uses the PHP extension oci8.
- Oracle does not support auto-incrementing keys, so you should specify the name of a sequence to `lastInsertId()` or `lastSequenceId()`.
- The Oracle extension does not support positional parameters. You must use named parameters.
- Currently the `Zend_Db::CASE_FOLDING` option is not supported by the Oracle adapter. To use this option with Oracle, you must use the PDO OCI adapter.

PDO for IBM DB2 and Informix Dynamic Server (IDS)

- Specify this Adapter to the `factory()` method with the name 'Pdo_Ibm'.
- This Adapter uses the PHP extensions pdo and pdo_ibm.
- You must use at least PDO_IBM extension version 1.2.2. If you have an earlier version of this extension, you must upgrade the PDO_IBM extension from PECL.

PDO Microsoft SQL Server

- Specify this Adapter to the `factory()` method with the name 'Pdo_Mssql'.
- This Adapter uses the PHP extensions pdo and pdo_mssql.
- Microsoft SQL Server does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `null`.

- Zend_Db_Adapter_Pdo_Mssql sets `QUOTED_IDENTIFIER ON` immediately after connecting to a SQL Server database. This makes the driver use the standard SQL identifier delimiter symbol (`"`) instead of the proprietary square-brackets syntax SQL Server uses for delimiting identifiers.
- You can specify `pdoType` as a key in the options array. The value can be "mssql" (the default), "dblib", "freetds", or "sybase". This option affects the DSN prefix the adapter uses when constructing the DSN string. Both "freetds" and "sybase" imply a prefix of "sybase:", which is used for the FreeTDS [http://www.freetds.org/] set of libraries. See also http://www.php.net/manual/en/ref.pdo-dblib.connection.php [http://www.php.net/manual/en/ref.pdo-dblib.connection.php] for more information on the DSN prefixes used in this driver.

PDO MySQL

- Specify this Adapter to the `factory()` method with the name 'Pdo_Mysql'.
- This Adapter uses the PHP extensions pdo and pdo_mysql.
- MySQL does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `null`.

PDO Oracle

- Specify this Adapter to the `factory()` method with the name 'Pdo_Oci'.
- This Adapter uses the PHP extensions pdo and pdo_oci.
- Oracle does not support auto-incrementing keys, so you should specify the name of a sequence to `lastInsertId()` or `lastSequenceId()`.

PDO PostgreSQL

- Specify this Adapter to the `factory()` method with the name 'Pdo_Pgsql'.
- This Adapter uses the PHP extensions pdo and pdo_pgsql.
- PostgreSQL supports both sequences and auto-incrementing keys. Therefore the arguments to `lastInsertId()` are optional. If you give no arguments, the Adapter returns the last value generated for an auto-increment key. If you give arguments, the Adapter returns the last value generated by the sequence named according to the convention '*table_column*_seq'.

PDO SQLite

- Specify this Adapter to the `factory()` method with the name 'Pdo_Sqlite'.
- This Adapter uses the PHP extensions pdo and pdo_sqlite.
- SQLite does not support sequences, so `lastInsertId()` ignores its arguments and always returns the last value generated for an auto-increment key. The `lastSequenceId()` method returns `null`.
- To connect to an SQLite2 database, specify `'sqlite2'=>true` in the array of parameters when creating an instance of the Pdo_Sqlite Adapter.
- To connect to an in-memory SQLite database, specify `'dbname'=>':memory:'` in the array of parameters when creating an instance of the Pdo_Sqlite Adapter.

- Older versions of the SQLite driver for PHP do not seem to support the PRAGMA commands necessary to ensure that short column names are used in result sets. If you have problems that your result sets are returned with keys of the form "tablename.columnname" when you do a join query, then you should upgrade to the current version of PHP.

Firebird/Interbase

- This Adapter uses the PHP extension php_interbase.
- Firebird/interbase does not support auto-incrementing keys, so you should specify the name of a sequence to `lastInsertId()` or `lastSequenceId()`.
- Currently the `Zend_Db::CASE_FOLDING` option is not supported by the Firebird/interbase adapter. Unquoted identifiers are automatically returned in upper case.

Zend_Db_Statement

In addition to convenient methods such as `fetchAll()` and `insert()` documented in the section called “Zend_Db_Adapter”, you can use a statement object to gain more options for running queries and fetching result sets. This section describes how to get an instance of a statement object, and how to use its methods.

Zend_Db_Statement is based on the PDOStatement object in the PHP Data Objects [http://www.php.net/pdo] extension.

Creating a Statement

Typically, a statement object is returned by the `query()` method of the database Adapter class. This method is a general way to prepare any SQL statement. The first argument is a string containing an SQL statement. The optional second argument is an array of values to bind to parameter placeholders in the SQL string.

Example 11.32. Creating a SQL statement object with query()

```
$stmt = $db->query(
            'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?',
            array('goofy', 'FIXED')
        );
```

The statement object corresponds to a SQL statement that has been prepared, and executed once with the bind-values specified. If the statement was a SELECT query or other type of statement that returns a result set, it is now ready to fetch results.

You can create a statement with its constructor, but this is less typical usage. There is no factory method to create this object, so you need to load the specific statement class and call its constructor. Pass the Adapter object as the first argument, and a string containing an SQL statement as the second argument. The statement is prepared, but not executed.

Example 11.33. Using a SQL statement constructor

```
$sql = 'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?';

$stmt = new Zend_Db_Statement_Mysqli($db, $sql);
```

Executing a Statement

You need to execute a statement object if you create it using its constructor, or if you want to execute the same statement multiple times. Use the `execute()` method of the statement object. The single argument is an array of value to bind to parameter placeholders in the statement.

If you use *positional parameters*, or those that are marked with a question mark symbol (`?`), pass the bind values in a plain array.

Example 11.34. Executing a statement with positional parameters

```
$sql = 'SELECT * FROM bugs WHERE reported_by = ? AND bug_status = ?';

$stmt = new Zend_Db_Statement_Mysqli($db, $sql);

$stmt->execute(array('goofy', 'FIXED'));
```

If you use *named parameters*, or those that are indicated by a string identifier preceded by a colon character (`:`), pass the bind values in an associative array. The keys of this array should match the parameter names.

Example 11.35. Executing a statement with named parameters

```
$sql = 'SELECT * FROM bugs WHERE ' .
       'reported_by = :reporter AND bug_status = :status';

$stmt = new Zend_Db_Statement_Mysqli($db, $sql);

$stmt->execute(array(':reporter' => 'goofy', ':status' => 'FIXED'));
```

PDO statements support both positional parameters and named parameters, but not both types in a single SQL statement. Some of the Zend_Db_Statement classes for non-PDO extensions may support only one type of parameter or the other.

Fetching Results from a SELECT Statement

You can call methods on the statement object to retrieve rows from SQL statements that produce result set. SELECT, SHOW, DESCRIBE and EXPLAIN are examples of statements that produce a result set. INSERT, UPDATE, and DELETE are examples of statements that don't produce a result set. You can execute the latter SQL statements using Zend_Db_Statement, but you cannot call methods to fetch rows of results from them.

Fetching a Single Row from a Result Set

To retrieve one row from the result set, use the `fetch()` method of the statement object. All three arguments of this method are optional:

- **Fetch style** is the first argument. This controls the structure in which the row is returned. See the section called "Changing the Fetch Mode" for a description of the valid values and the corresponding data formats.
- **Cursor orientation** is the second argument. The default is Zend_Db::FETCH_ORI_NEXT, which simply means that each call to `fetch()` returns the next row in the result set, in the order returned by the RDBMS.
- **Offset** is the third argument. If the cursor orientation is Zend_Db::FETCH_ORI_ABS, then the offset number is the ordinal number of the row to return. If the cursor orientation is Zend_Db::FETCH_ORI_REL, then the offset number is relative to the cursor position before `fetch()` was called.

`fetch()` returns `false` if all rows of the result set have been fetched.

Example 11.36. Using fetch() in a loop

```
$stmt = $db->query('SELECT * FROM bugs');

while ($row = $stmt->fetch()) {
    echo $row['bug_description'];
}
```

See also PDOStatement::fetch() [http://www.php.net/PDOStatement-fetch].

Fetching a Complete Result Set

To retrieve all the rows of the result set in one step, use the `fetchAll()` method. This is equivalent to calling the `fetch()` method in a loop and returning all the rows in an array. The `fetchAll()` method accepts two arguments. The first is the fetch style, as described above, and the second indicates the number of the column to return, when the fetch style is Zend_Db::FETCH_COLUMN.

Example 11.37. Using fetchAll()

```
$stmt = $db->query('SELECT * FROM bugs');

$rows = $stmt->fetchAll();

echo $rows[0]['bug_description'];
```

See also PDOStatement::fetchAll() [http://www.php.net/PDOStatement-fetchAll].

Changing the Fetch Mode

By default, the statement object returns rows of the result set as associative arrays, mapping column names to column values. You can specify a different format for the statement class to return rows, just as you can in the Adapter class. You can use the `setFetchMode()` method of the statement object to specify the fetch mode. Specify the fetch mode using Zend_Db class constants FETCH_ASSOC, FETCH_NUM, FETCH_BOTH, FETCH_COLUMN, and FETCH_OBJ. See the section called "Changing the Fetch Mode" for more information on these modes. Subsequent calls to the statement methods `fetch()` or `fetchAll()` use the fetch mode that you specify.

Example 11.38. Setting the fetch mode

```
$stmt = $db->query('SELECT * FROM bugs');

$stmt->setFetchMode(Zend_Db::FETCH_NUM);

$rows = $stmt->fetchAll();

echo $rows[0][0];
```

See also PDOStatement::setFetchMode() [http://www.php.net/PDOStatement-setFetchMode].

Fetching a Single Column from a Result Set

To return a single column from the next row of the result set, use `fetchColumn()`. The optional argument is the integer index of the column, and it defaults to 0. This method returns a scalar value, or `false` if all rows of the result set have been fetched.

Note this method operates differently than the `fetchCol()` method of the Adapter class. The `fetch-Column()` method of a statement returns a single value from one row. The `fetchCol()` method of an adapter returns an array of values, taken from the first column of all rows of the result set.

Example 11.39. Using fetchColumn()

```
$stmt = $db->query('SELECT bug_id, bug_description, bug_status FROM bugs');

$bug_status = $stmt->fetchColumn(2);
```

See also PDOStatement::fetchColumn() [http://www.php.net/PDOStatement-fetchColumn].

Fetching a Row as an Object

To retrieve a row from the result set structured as an object, use the `fetchObject()`. This method takes two optional arguments. The first argument is a string that names the class name of the object to return; the default is 'stdClass'. The second argument is an array of values that will be passed to the constructor of that class.

Example 11.40. Using fetchObject()

```
$stmt = $db->query('SELECT bug_id, bug_description, bug_status FROM bugs');

$obj = $stmt->fetchObject();

echo $obj->bug_description;
```

See also PDOStatement::fetchObject() [http://www.php.net/PDOStatement-fetchObject].

Zend_Db_Profiler

Introduction

`Zend_Db_Profiler` can be enabled to allow profiling of queries. Profiles include the queries processed by the adapter as well as elapsed time to run the queries, allowing inspection of the queries that have been performed without needing to add extra debugging code to classes. Advanced usage also allows the developer to filter which queries are profiled.

Enable the profiler by either passing a directive to the adapter constructor, or by asking the adapter to enable it later.

```
$params = array(
    'host'     => '127.0.0.1',
    'username' => 'webuser',
    'password' => 'xxxxxxxx',
    'dbname'   => 'test'
    'profiler' => true  // turn on profiler
                        // set to false to disable (disabled by default)
);

$db = Zend_Db::factory('PDO_MYSQL', $params);

// turn off profiler:
$db->getProfiler()->setEnabled(false);

// turn on profiler:
$db->getProfiler()->setEnabled(true);
```

The value of the '`profiler`' option is flexible. It is interpreted differently depending on its type. Most often, you should use a simple boolean value, but other types enable you to customize the profiler behavior.

A boolean argument sets the profiler to enabled if it is a `true` value, or disabled if `false`. The profiler class is the adapter's default profiler class, `Zend_Db_Profiler`.

```
$params['profiler'] = true;
```

```
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

An instance of a profiler object makes the adapter use that object. The object type must be Zend_Db_Profiler or a subclass thereof. Enabling the profiler is done separately.

```
$profiler = MyProject_Db_Profiler();
$profiler->setEnabled(true);
$params['profiler'] = $profiler;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

The argument can be an associative array containing any or all of the keys 'enabled', 'instance', and 'class'. The 'enabled' and 'instance' keys correspond to the boolean and instance types documented above. The 'class' key is used to name a class to use for a custom profiler. The class must be Zend_Db_Profiler or a subclass. The class is instantiated with no constructor arguments. The 'class' option is ignored when the 'instance' option is supplied.

```
$params['profiler'] = array(
    'enabled' => true,
    'class'   => 'MyProject_Db_Profiler'
);
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

Finally, the argument can be an object of type Zend_Config containing properties, which are treated as the array keys described above. For example, a file "config.ini" might contain the following data:

```
[main]
db.profiler.class   = "MyProject_Db_Profiler"
db.profiler.enabled = true
```

This configuration can be applied by the following PHP code:

```
$config = new Zend_Config_Ini('config.ini', 'main');
$params['profiler'] = $config->db->profiler;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

The 'instance' property may be used as in the following:

```
$profiler = new MyProject_Db_Profiler();
$profiler->setEnabled(true);
$configData = array(
    'instance' => $profiler
    );
$config = new Zend_Config($configData);
$params['profiler'] = $config;
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

Using the Profiler

At any point, grab the profiler using the adapter's `getProfiler()` method:

```
$profiler = $db->getProfiler();
```

This returns a `Zend_Db_Profiler` object instance. With that instance, the developer can examine your queries using a variety of methods:

- `getTotalNumQueries()` returns the total number of queries that have been profiled.
- `getTotalElapsedSecs()` returns the total number of seconds elapsed for all profiled queries.
- `getQueryProfiles()` returns an array of all query profiles.
- `getLastQueryProfile()` returns the last (most recent) query profile, regardless of whether or not the query has finished (if it hasn't, the end time will be null)
- `clear()` clears any past query profiles from the stack.

The return value of `getLastQueryProfile()` and the individual elements of `getQueryProfiles()` are `Zend_Db_Profiler_Query` objects, which provide the ability to inspect the individual queries themselves:

- `getQuery()` returns the SQL text of the query. The SQL text of a prepared statement with parameters is the text at the time the query was prepared, so it contains parameter placeholders, not the values used when the statement is executed.
- `getQueryParams()` returns an array of parameter values used when executing a prepared query. This includes both bound parameters and arguments to the statement's `execute()` method. The keys of the array are the positional (1-based) or named (string) parameter indices.
- `getElapsedSecs()` returns the number of seconds the query ran.

The information `Zend_Db_Profiler` provides is useful for profiling bottlenecks in applications, and for debugging queries that have been run. For instance, to see the exact query that was last run:

```
$query = $profiler->getLastQueryProfile();

echo $query->getQuery();
```

Perhaps a page is generating slowly; use the profiler to determine first the total number of seconds of all queries, and then step through the queries to find the one that ran longest:

```
$totalTime    = $profiler->getTotalElapsedSecs();
$queryCount   = $profiler->getTotalNumQueries();
$longestTime  = 0;
$longestQuery = null;

foreach ($profiler->getQueryProfiles() as $query) {
    if ($query->getElapsedSecs() > $longestTime) {
        $longestTime  = $query->getElapsedSecs();
        $longestQuery = $query->getQuery();
    }
}

echo 'Executed ' . $queryCount . ' queries in ' . $totalTime .
     ' seconds' . "\n";
echo 'Average query length: ' . $totalTime / $queryCount .
     ' seconds' . "\n";
echo 'Queries per second: ' . $queryCount / $totalTime . "\n";
echo 'Longest query length: ' . $longestTime . "\n";
echo "Longest query: \n" . $longestQuery . "\n";
```

Advanced Profiler Usage

In addition to query inspection, the profiler also allows the developer to filter which queries get profiled. The following methods operate on a `Zend_Db_Profiler` instance:

Filter by query elapsed time

`setFilterElapsedSecs()` allows the developer to set a minimum query time before a query is profiled. To remove the filter, pass the method a null value.

```
// Only profile queries that take at least 5 seconds:
$profiler->setFilterElapsedSecs(5);

// Profile all queries regardless of length:
$profiler->setFilterElapsedSecs(null);
```

Filter by query type

`setFilterQueryType()` allows the developer to set which types of queries should be profiled; to profile multiple types, logical OR them. Query types are defined as the following `Zend_Db_Profiler` constants:

- `Zend_Db_Profiler::CONNECT`: connection operations, or selecting a database.
- `Zend_Db_Profiler::QUERY`: general database queries that do not match other types.
- `Zend_Db_Profiler::INSERT`: any query that adds new data to the database, generally SQL INSERT.
- `Zend_Db_Profiler::UPDATE`: any query that updates existing data, usually SQL UPDATE.
- `Zend_Db_Profiler::DELETE`: any query that deletes existing data, usually SQL DELETE.
- `Zend_Db_Profiler::SELECT`: any query that retrieves existing data, usually SQL SELECT.
- `Zend_Db_Profiler::TRANSACTION`: any transactional operation, such as start transaction, commit, or rollback.

As with `setFilterElapsedSecs()`, you can remove any existing filters by passing `null` as the sole argument.

```
// profile only SELECT queries
$profiler->setFilterQueryType(Zend_Db_Profiler::SELECT);

// profile SELECT, INSERT, and UPDATE queries
$profiler->setFilterQueryType(Zend_Db_Profiler::SELECT |
                              Zend_Db_Profiler::INSERT |
                              Zend_Db_Profiler::UPDATE);

// profile DELETE queries
$profiler->setFilterQueryType(Zend_Db_Profiler::DELETE);

// Remove all filters
$profiler->setFilterQueryType(null);
```

Retrieve profiles by query type

Using `setFilterQueryType()` can cut down on the profiles generated. However, sometimes it can be more useful to keep all profiles, but view only those you need at a given moment. Another feature of `getQueryProfiles()` is that it can do this filtering on-the-fly, by passing a query type (or logical combination of query types) as its first argument; see the section called “Filter by query type” for a list of the query type constants.

```
// Retrieve only SELECT query profiles
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::SELECT);

// Retrieve only SELECT, INSERT, and UPDATE query profiles
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::SELECT |
                                        Zend_Db_Profiler::INSERT |
                                        Zend_Db_Profiler::UPDATE);

// Retrieve DELETE query profiles
```

```
$profiles = $profiler->getQueryProfiles(Zend_Db_Profiler::DELETE);
```

Specialized Profilers

A Specialized Profiler is an object that inherits from `Zend_Db_Profiler`. Specialized Profilers treat profiling information in specific ways.

Profiling with Firebug

`Zend_Db_Profiler_Firebug` sends profiling infomation to the Firebug [http://www.getfirebug.com/] Console [http://getfirebug.com/logging.html].

All data is sent via the `Zend_Wildfire_Channel_HttpHeaders` component which uses HTTP headers to ensure the page content is not disturbed. Debugging AJAX requests that require clean JSON and XML responses is possible with this approach.

Requirements:

- Firefox Browser ideally version 3 but version 2 is also supported.
- Firebug Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/1843.
- FirePHP Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/6149.

Example 11.41. DB Profiling with `Zend_Controller_Front`

```
// In your bootstrap file

// Instantiate the profiler
$profiler = new Zend_Db_Profiler_Firebug('All DB Queries');

// Attach the profiler to your db adapter
$db->setProfiler($profiler)

// Dispatch your front controller

// All DB queries in your model, view and controller
// files will now be profiled and sent to Firebug
```

Example 11.42. DB Profiling without `Zend_Controller_Front`

```
$profiler = new Zend_Db_Profiler_Firebug('All DB Queries');
$db->setProfiler($profiler)

$request  = new Zend_Controller_Request_Http();
$response = new Zend_Controller_Response_Http();
$channel  = Zend_Wildfire_Channel_HttpHeaders::getInstance();
$channel->setRequest($request);
$channel->setResponse($response);

// Now you can run your DB queries to be profiled

// Flush profiling data to browser
$channel->flush();
$response->sendHeaders();
```

Zend_Db_Select

Overview of the Select Object

The Zend_Db_Select object represents a SQL `SELECT` query statement. The class has methods for adding individual parts to the query. You can specify some parts of the query using PHP methods and data structures, and the class forms the correct SQL syntax for you. After you build a query, you can execute the query as if you had written it as a string.

The value offered by Zend_Db_Select includes:

- Object-oriented methods for specifying SQL queries in a piece-by-piece manner;
- Database-independent abstraction of some parts of the SQL query;
- Automatic quoting of metadata identifiers in most cases, to support identifiers containing SQL reserved words and special characters;
- Quoting identifiers and values, to help reduce risk of SQL injection attacks.

Using Zend_Db_Select is not mandatory. For very simple SELECT queries, it is usually simpler to specify the entire SQL query as a string and execute it using Adapter methods like `query()` or `fetchAll()`. Using Zend_Db_Select is helpful if you need to assemble a SELECT query procedurally, or based on conditional logic in your application.

Creating a Select Object

You can create an instance of a Zend_Db_Select object using the `select()` method of a Zend_Db_Adapter_Abstract object.

Example 11.43. Example of the database adapter's select() method

```
$db = Zend_Db::factory( ...options... );
$select = $db->select();
```

Another way to create a Zend_Db_Select object is with its constructor, specifying the database adapter as an argument.

Example 11.44. Example of creating a new Select object

```
$db = Zend_Db::factory( ...options... );
$select = new Zend_Db_Select($db);
```

Building Select queries

When building the query, you can add clauses of the query one by one. There is a separate method to add each clause to the Zend_Db_Select object.

Example 11.45. Example of the using methods to add clauses

```
// Create the Zend_Db_Select object
$select = $db->select();

// Add a FROM clause
$select->from( ...specify table and columns... )

// Add a WHERE clause
$select->where( ...specify search criteria... )

// Add an ORDER BY clause
$select->order( ...specify sorting criteria... );
```

You also can use most methods of the Zend_Db_Select object with a convenient fluent interface. A fluent interface means that each method returns a reference to the object on which it was called, so you can immediately call another method.

Example 11.46. Example of the using the fluent interface

```
$select = $db->select()
    ->from( ...specify table and columns... )
    ->where( ...specify search criteria... )
    ->order( ...specify sorting criteria... );
```

The examples in this section show usage of the fluent interface, but you can use the non-fluent interface in all cases. It is often necessary to use the non-fluent interface, for example, if your application needs to perform some logic before adding a clause to a query.

Adding a FROM clause

Specify the table for this query using the `from()` method. You can specify the table name as a simple string. Zend_Db_Select applies identifier quoting around the table name, so you can use special characters.

Example 11.47. Example of the from() method

```
// Build this query:
//   SELECT *
//   FROM "products"

$select = $db->select()
             ->from( 'products' );
```

You can also specify the correlation name (sometimes called the "table alias") for a table. Instead of a simple string, use an associative array mapping the correlation name to the table name. In other clauses of the SQL query, use this correlation name. If your query joins more than one table, Zend_Db_Select generates unique correlation names based on the table names, for any tables for which you don't specify the correlation name.

Example 11.48. Example of specifying a table correlation name

```
// Build this query:
//   SELECT p.*
//   FROM "products" AS p

$select = $db->select()
             ->from( array('p' => 'products') );
```

Some RDBMS brands support a leading schema specifier for a table. You can specify the table name as "`schemaName.tableName`", where Zend_Db_Select quotes each part individually, or you may specify the schema name separately. A schema name specified in the table name takes precedence over a schema provided separately in the event that both are provided.

Example 11.49. Example of specifying a schema name

```
// Build this query:
//   SELECT *
//   FROM "myschema"."products"

$select = $db->select()
             ->from( 'myschema.products' );

// or

$select = $db->select()
             ->from('products', '*', 'myschema');
```

Adding Columns

In the second argument of the `from()` method, you can specify the columns to select from the respective table. If you specify no columns, the default is "`*`", the SQL wildcard for "all columns".

You can list the columns in a simple array of strings, or as an associative mapping of column alias to column name. If you only have one column to query, and you don't need to specify a column alias, you can list it as a plain string instead of an array.

If you give an empty array as the columns argument, no columns from the respective table are included in the result set. See a code example under the section on the `join()` method.

You can specify the column name as "`correlationName.columnName`". Zend_Db_Select quotes each part individually. If you don't specify a correlation name for a column, it uses the correlation name for the table named in the current `from()` method.

Example 11.50. Examples of specifying columns

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'));

// Build the same query, specifying correlation names:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('p.product_id', 'p.product_name'));

// Build this query with an alias for one column:
//   SELECT p."product_id" AS prodno, p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('prodno' => 'product_id', 'product_name'));
```

Adding Expression Columns

Columns in SQL queries are sometimes expressions, not simply column names from a table. Expressions should not have correlation names or quoting applied. If your column string contains parentheses, Zend_Db_Select recognizes it as an expression.

You also can create an object of type Zend_Db_Expr explicitly, to prevent a string from being treated as a column name. Zend_Db_Expr is a minimal class that contains a single string. Zend_Db_Select recognizes objects of type Zend_Db_Expr and converts them back to string, but does not apply any alterations, such as quoting or correlation names.

Note

Using Zend_Db_Expr for column names is not necessary if your column expression contains parentheses; Zend_Db_Select recognizes parentheses and treats the string as an expression, skipping quoting and correlation names.

Example 11.51. Examples of specifying columns containing expressions

```
// Build this query:
//   SELECT p."product_id", LOWER(product_name)
//   FROM "products" AS p
// An expression with parentheses implicitly becomes
// a Zend_Db_Expr.

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'LOWER(product_name)'));

// Build this query:
//   SELECT p."product_id", (p.cost * 1.08) AS cost_plus_tax
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id',
                          'cost_plus_tax' => '(p.cost * 1.08)')
                   );

// Build this query using Zend_Db_Expr explicitly:
//   SELECT p."product_id", p.cost * 1.08 AS cost_plus_tax
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id',
                          'cost_plus_tax' =>
                              new Zend_Db_Expr('p.cost * 1.08'))
                    );
```

In the cases above, Zend_Db_Select does not alter the string to apply correlation names or identifier quoting. If those changes are necessary to resolve ambiguity, you must make the changes manually in the string.

If your column names are SQL keywords or contain special characters, you should use the Adapter's `quoteIdentifier()` method and interpolate the result into the string. The `quoteIdentifier()` method uses SQL quoting to delimit the identifier, which makes it clear that it is an identifier for a table or a column, and not any other part of SQL syntax.

Your code is more database-independent if you use the `quoteIdentifier()` method instead of typing quotes literally in your string, because some RDBMS brands use nonstandard symbols for quoting identifiers. The `quoteIdentifier()` method is designed to use the appropriate quoting symbols based on the adapter type. The `quoteIdentifier()` method also escapes any quote characters that appear within the identifier name itself.

Example 11.52. Examples of quoting columns in an expression

```
// Build this query,
// quoting the special column name "from" in the expression:
//   SELECT p."from" + 10 AS origin
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('origin' =>
                              '(p.' . $db->quoteIdentifier('from') . ' + 10)')
                   );
```

Adding columns to an existing FROM or JOIN table

There may be cases where you wish to add columns to an existing FROM or JOIN table after those methods have been called. The `columns()` method allows you to add specific columns at any point before the query is executed. You can supply the columns as either a string or `Zend_Db_Expr` or as an array of these elements. The second argument to this method can be omitted, implying that the columns are to be added to the FROM table, otherwise an existing correlation name must be used.

Example 11.53. Examples of adding columns with the `columns()` method

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'), 'product_id')
             ->columns('product_name');

// Build the same query, specifying correlation names:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->from(array('p' => 'products'), 'p.product_id')
             ->columns('product_name', 'p');
             // Alternatively use columns('p.product_name')
```

Adding Another Table to the Query with JOIN

Many useful queries involve using a `JOIN` to combine rows from multiple tables. You can add tables to a Zend_Db_Select query using the `join()` method. Using this method is similar to the `from()` method, except you can also specify a join condition in most cases.

Example 11.54. Example of the join() method

```
// Build this query:
//   SELECT p."product_id", p."product_name", l.*
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id');
```

The second argument to `join()` is a string that is the join condition. This is an expression that declares the criteria by which rows in one table match rows in the the other table. You can use correlation names in this expression.

> **Note**
>
> No quoting is applied to the expression you specify for the join condition; if you have column names that need to be quoted, you must use `quoteIdentifier()` as you form the string for the join condition.

The third argument to `join()` is an array of column names, like that used in the `from()` method. It defaults to "`*`", supports correlation names, expressions, and Zend_Db_Expr in the same way as the array of column names in the `from()` method.

To select no columns from a table, use an empty array for the list of columns. This usage works in the `from()` method too, but typically you want some columns from the primary table in your queries, whereas you might want no columns from a joined table.

Example 11.55. Example of specifying no columns

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array() ); // empty list of columns
```

Note the empty `array()` in the above example in place of a list of columns from the joined table.

SQL has several types of joins. See the list below for the methods to support different join types in Zend_Db_Select.

- **INNER JOIN** with the `join(table, join, [columns])` or `joinInner(table, join, [columns])` methods.

  This may be the most common type of join. Rows from each table are compared using the join condition you specify. The result set includes only the rows that satisfy the join condition. The result set can be empty if no rows satisfy this condition.

  All RDBMS brands support this join type.

- **LEFT JOIN** with the `joinLeft(table, condition, [columns])` method.

  All rows from the left operand table are included, matching rows from the right operand table included, and the columns from the right operand table are filled with NULLs if no row exists matching the left table.

  All RDBMS brands support this join type.

- **RIGHT JOIN** with the `joinRight(table, condition, [columns])` method.

  Right outer join is the complement of left outer join. All rows from the right operand table are included, matching rows from the left operand table included, and the columns from the left operand table are filled with NULLs if no row exists matching the right table.

  Some RDBMS brands don't support this join type, but in general any right join can be represented as a left join by reversing the order of the tables.

- **FULL JOIN** with the `joinFull(table, condition, [columns])` method.

  A full outer join is like combining a left outer join and a right outer join. All rows from both tables are included, paired with each other on the same row of the result set if they satisfy the join condition, and otherwise paired with NULLs in place of columns from the other table.

  Some RDBMS brands don't support this join type.

- **CROSS JOIN** with the `joinCross(table, [columns])` method.

  A cross join is a Cartesian product. Every row in the first table is matched to every row in the second table. Therefore the number of rows in the result set is equal to the product of the number of rows in each table. You can filter the result set using conditions in a WHERE clause; in this way a cross join is similar to the old SQL-89 join syntax.

  The `joinCross()` method has no parameter to specify the join condition. Some RDBMS brands don't support this join type.

- **NATURAL JOIN** with the `joinNatural(table, [columns])` method.

  A natural join compares any column(s) that appear with the same name in both tables. The comparison is equality of all the column(s); comparing the columns using inequality is not a natural join. Only natural inner joins are supported by this API, even though SQL permits natural outer joins as well.

  The `joinNatural()` method has no parameter to specify the join condition.

In addition to these join methods, you can simplify your queries by using the JoinUsing methods. Instead of supplying a full condition to your join, you simply pass the column name on which to join and the Zend_Db_Select object completes the condition for you.

Example 11.56. Example of the joinUsing() method

```
// Build this query:
//   SELECT *
//   FROM "table1"
//   JOIN "table2"
//   ON "table1".column1 = "table2".column1
//   WHERE column2 = 'foo'

$select = $db->select()
             ->from('table1')
             ->joinUsing('table2', 'column1')
             ->where('column2 = ?', 'foo');
```

Each of the applicable join methods in the Zend_Db_Select component has a corresponding 'using' method.

- `joinUsing(table, join, [columns])` and `joinInnerUsing(table, join, [columns])`
- `joinLeftUsing(table, join, [columns])`
- `joinRightUsing(table, join, [columns])`
- `joinFullUsing(table, join, [columns])`

Adding a WHERE Clause

You can specify criteria for restricting rows of the result set using the `where()` method. The first argument of this method is a SQL expression, and this expression is used in a SQL `WHERE` clause in the query.

Example 11.57. Example of the where() method

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE price > 100.00

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price > 100.00');
```

Note

No quoting is applied to expressions given to the `where()` or `orWhere()` methods. If you have column names that need to be quoted, you must use `quoteIdentifier()` as you form the string for the condition.

The second argument to the `where()` method is optional. It is a value to substitute into the expression. Zend_Db_Select quotes the value and substitutes it for a question-mark ("`?`") symbol in the expression.

This method accepts only one parameter. If you have an expression into which you need to substitute multiple variables, you must format the string manually, interpolating variables and performing quoting yourself.

Example 11.58. Example of a parameter in the where() method

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price > 100.00)

$minimumPrice = 100;

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price > ?', $minimumPrice);
```

You can invoke the `where()` method multiple times on the same Zend_Db_Select object. The resulting query combines the multiple terms together using `AND` between them.

Example 11.59. Example of multiple where() methods

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price > 100.00)
//     AND (price < 500.00)

$minimumPrice = 100;
$maximumPrice = 500;

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price > ?', $minimumPrice)
             ->where('price < ?', $maximumPrice);
```

If you need to combine terms together using `OR`, use the `orWhere()` method. This method is used in the same way as the `where()` method, except that the term specified is preceded by `OR`, instead of `AND`.

Example 11.60. Example of the orWhere() method

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price < 100.00)
//     OR (price > 500.00)

$minimumPrice = 100;
$maximumPrice = 500;

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where('price < ?', $minimumPrice)
             ->orWhere('price > ?', $maximumPrice);
```

Zend_Db_Select automatically puts parentheses around each expression you specify using the `where()` or `orWhere()` methods. This helps to ensure that Boolean operator precedence does not cause unexpected results.

Example 11.61. Example of parenthesizing Boolean expressions

```
// Build this query:
//   SELECT product_id, product_name, price
//   FROM "products"
//   WHERE (price < 100.00 OR price > 500.00)
//     AND (product_name = 'Apple')

$minimumPrice = 100;
$maximumPrice = 500;
$prod = 'Apple';

$select = $db->select()
             ->from('products',
                    array('product_id', 'product_name', 'price'))
             ->where("price < $minimumPrice OR price > $maximumPrice")
             ->where('product_name = ?', $prod);
```

In the example above, the results would be quite different without the parentheses, because `AND` has higher precedence than `OR`. Zend_Db_Select applies the parentheses so the effect is that each expression in successive calls to the `where()` bind more tightly than the `AND` that combines the expressions.

Adding a GROUP BY Clause

In SQL, the `GROUP BY` clause allows you to reduce the rows of a query result set to one row per unique value found in the column(s) named in the `GROUP BY` clause.

In Zend_Db_Select, you can specify the column(s) to use for calculating the groups of rows using the `group()` method. The argument to this method is a column or an array of columns to use in the `GROUP BY` clause.

Example 11.62. Example of the group() method

```
// Build this query:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array('line_items_per_product' => 'COUNT(*)'))
             ->group('p.product_id');
```

Like the columns array in the `from()` method, you can use correlation names in the column name strings, and the column is quoted as an identifier unless the string contains parentheses or is an object of type Zend_Db_Expr.

Adding a HAVING Clause

In SQL, the `HAVING` clause applies a restriction condition on groups of rows. This is similar to how a `WHERE` clause applies a restriction condition on rows. But the two clauses are different because `WHERE` conditions are applied before groups are defined, whereas `HAVING` conditions are applied after groups are defined.

In Zend_Db_Select, you can specify conditions for restricting groups using the `having()` method. Its usage is similar to that of the `where()` method. The first argument is a string containing a SQL expression. The optional second argument is a value that is used to replace a positional parameter placeholder in the SQL expression. Expressions given in multiple invocations of the `having()` method are combined using the Boolean `AND` operator, or the `OR` operator if you use the `orHaving()` method.

Example 11.63. Example of the having() method

```
// Build this query:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
//   HAVING line_items_per_product > 10

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array('line_items_per_product' => 'COUNT(*)'))
             ->group('p.product_id')
             ->having('line_items_per_product > 10');
```

Note

No quoting is applied to expressions given to the `having()` or `orHaving()` methods. If you have column names that need to be quoted, you must use `quoteIdentifier()` as you form the string for the condition.

Adding an ORDER BY Clause

In SQL, the `ORDER BY` clause specifies one or more columns or expressions by which the result set of a query is sorted. If multiple columns are listed, the secondary columns are used to resolve ties; the sort order is determined by the secondary columns if the preceding columns contain identical values. The default sorting is from least value to greatest value. You can also sort by greatest value to least value for a given column in the list by specifying the keyword `DESC` after that column.

In Zend_Db_Select, you can use the `order()` method to specify a column or an array of columns by which to sort. Each element of the array is a string naming a column. Optionally with the `ASC` `DESC` keyword following it, separated by a space.

Like in the `from()` and `group()` methods, column names are quoted as identifiers, unless they contain contain parentheses or are an object of type Zend_Db_Expr.

Example 11.64. Example of the order() method

```
// Build this query:
//   SELECT p."product_id", COUNT(*) AS line_items_per_product
//   FROM "products" AS p JOIN "line_items" AS l
//     ON p.product_id = l.product_id
//   GROUP BY p.product_id
//   ORDER BY "line_items_per_product" DESC, "product_id"

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id'))
             ->join(array('l' => 'line_items'),
                    'p.product_id = l.product_id',
                    array('line_items_per_product' => 'COUNT(*)'))
             ->group('p.product_id')
             ->order(array('line_items_per_product DESC',
                           'product_id'));
```

Adding a LIMIT Clause

Some RDBMS brands extend SQL with a query clause known as the `LIMIT` clause. This clause reduces the number of rows in the result set to at most a number you specify. You can also specify to skip a number of rows before starting to output. This feature makes it easy to take a subset of a result set, for example when displaying query results on progressive pages of output.

In Zend_Db_Select, you can use the `limit()` method to specify the count of rows and the number of rows to skip. The first argument to this method is the desired count of rows. The second argument is the number of rows to skip.

Example 11.65. Example of the limit() method

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 10, 20

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'))
             ->limit(10, 20);
```

Note

The `LIMIT` syntax is not supported by all RDBMS brands. Some RDBMS require different syntax to support similar functionality. Each Zend_Db_Adapter_Abstract class includes a method to produce SQL appropriate for that RDBMS.

Use the `limitPage()` method for an alternative way to specify row count and offset. This method allows you to limit the result set to one of a series of fixed-length subsets of rows from the query's total result set. In other words, you specify the length of a "page" of results, and the ordinal number of the single page of results you want the query to return. The page number is the first argument of the `limitPage()` method, and the page length is the second argument. Both arguments are required; they have no default values.

Example 11.66. Example of the limitPage() method

```
// Build this query:
//   SELECT p."product_id", p."product_name"
//   FROM "products" AS p
//   LIMIT 10, 20

$select = $db->select()
             ->from(array('p' => 'products'),
                    array('product_id', 'product_name'))
             ->limitPage(2, 10);
```

Adding the DISTINCT Query Modifier

The `distinct()` method enables you to add the `DISTINCT` keyword to your SQL query.

Example 11.67. Example of the distinct() method

```
// Build this query:
//   SELECT DISTINCT p."product_name"
//   FROM "products" AS p

$select = $db->select()
             ->distinct()
             ->from(array('p' => 'products'), 'product_name');
```

Adding the FOR UPDATE Query Modifier

The `forUpdate()` method enables you to add the `FOR UPDATE` modifier to your SQL query.

Example 11.68. Example of forUpdate() method

```
// Build this query:
//   SELECT FOR UPDATE p.*
//   FROM "products" AS p

$select = $db->select()
             ->forUpdate()
             ->from(array('p' => 'products'));
```

Executing Select Queries

This section describes how to execute the query represented by a Zend_Db_Select object.

Executing Select Queries from the Db Adapter

You can execute the query represented by the Zend_Db_Select object by passing it as the first argument to the `query()` method of a Zend_Db_Adapter_Abstract object. Use the Zend_Db_Select objects instead of a string query.

The `query()` method returns an object of type Zend_Db_Statement or PDOStatement, depending on the adapter type.

Example 11.69. Example using the Db adapter's query() method

```
$select = $db->select()
             ->from('products');

$stmt = $db->query($select);
$result = $stmt->fetchAll();
```

Executing Select Queries from the Object

As an alternative to using the `query()` method of the adapter object, you can use the `query()` method of the Zend_Db_Select object. Both methods return an object of type Zend_Db_Statement or PDOStatement, depending on the adapter type.

Example 11.70. Example using the Select object's query method

```
$select = $db->select()
             ->from('products');

$stmt = $select->query();
$result = $stmt->fetchAll();
```

Converting a Select Object to a SQL String

If you need access to a string representation of the SQL query corresponding to the Zend_Db_Select object, use the `__toString()` method.

Example 11.71. Example of the __toString() method

```
$select = $db->select()
             ->from('products');

$sql = $select->__toString();
echo "$sql\n";

// The output is the string:
//   SELECT * FROM "products"
```

Other methods

This section describes other methods of the Zend_Db_Select class that are not covered above: `getPart()` and `reset()`.

Retrieving Parts of the Select Object

The `getPart()` method returns a representation of one part of your SQL query. For example, you can use this method to return the array of expressions for the `WHERE` clause, or the array of columns (or column expressions) that are in the `SELECT` list, or the values of the count and offset for the `LIMIT` clause.

The return value is not a string containing a fragment of SQL syntax. The return value is an internal representation, which is typically an array structure containing values and expressions. Each part of the query has a different structure.

The single argument to the `getPart()` method is a string that identifies which part of the Select query to return. For example, the string `'from'` identifies the part of the Select object that stores information about the tables in the `FROM` clause, including joined tables.

The Zend_Db_Select class defines constants you can use for parts of the SQL query. You can use these constant definitions, or you can the literal strings.

Table 11.2. Constants used by getPart() and reset()

Constant	String value
`Zend_Db_Select::DISTINCT`	`'distinct'`
`Zend_Db_Select::FOR_UPDATE`	`'forupdate'`
`Zend_Db_Select::COLUMNS`	`'columns'`
`Zend_Db_Select::FROM`	`'from'`
`Zend_Db_Select::WHERE`	`'where'`
`Zend_Db_Select::GROUP`	`'group'`
`Zend_Db_Select::HAVING`	`'having'`
`Zend_Db_Select::ORDER`	`'order'`
`Zend_Db_Select::LIMIT_COUNT`	`'limitcount'`
`Zend_Db_Select::LIMIT_OFFSET`	`'limitoffset'`

Example 11.72. Example of the getPart() method

```
$select = $db->select()
             ->from('products')
             ->order('product_id');

// You can use a string literal to specify the part
$orderData = $select->getPart( 'order' );

// You can use a constant to specify the same part
$orderData = $select->getPart( Zend_Db_Select::ORDER );

// The return value may be an array structure, not a string.
// Each part has a different structure.
print_r( $orderData );
```

Resetting Parts of the Select Object

The `reset()` method enables you to clear one specified part of the SQL query, or else clear all parts of the SQL query if you omit the argument.

The single argument is optional. You can specify the part of the query to clear, using the same strings you used in the argument to the `getPart()` method. The part of the query you specify is reset to a default state.

If you omit the parameter, `reset()` changes all parts of the query to their default state. This makes the Zend_Db_Select object equivalent to a new object, as though you had just instantiated it.

Example 11.73. Example of the reset() method

```
// Build this query:
//   SELECT p.*
//   FROM "products" AS p
//   ORDER BY "product_name"

$select = $db->select()
             ->from(array('p' => 'products')
             ->order('product_name');

// Changed requirement, instead order by a different columns:
//   SELECT p.*
//   FROM "products" AS p
//   ORDER BY "product_id"

// Clear one part so we can redefine it
$select->reset( Zend_Db_Select::ORDER );

// And specify a different column
$select->order('product_id');

// Clear all parts of the query
$select->reset();
```

Zend_Db_Table

Introduction to Table Class

The Zend_Db_Table class is an object-oriented interface to database tables. It provides methods for many common operations on tables. The base class is extensible, so you can add custom logic.

The Zend_Db_Table solution is an implementation of the Table Data Gateway [http://www.martinfowler.com/eaaCatalog/tableDataGateway.html] pattern. The solution also includes a class that implements the Row Data Gateway [http://www.martinfowler.com/eaaCatalog/rowDataGateway.html] pattern.

Defining a Table Class

For each table in your database that you want to access, define a class that extends Zend_Db_Table_Abstract.

Defining the Table Name and Schema

Declare the database table for which this class is defined, using the protected variable `$_name`. This is a string, and must contain the name of the table spelled as it appears in the database.

Example 11.74. Declaring a table class with explicit table name

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
}
```

If you don't specify the table name, it defaults to the name of the class. If you rely on this default, the class name must match the spelling of the table name as it appears in the database.

Example 11.75. Declaring a table class with implicit table name

```
class bugs extends Zend_Db_Table_Abstract
{
    // table name matches class name
}
```

You can also declare the schema for the table, either with the protected variable `$_schema`, or with the schema prepended to the table name in the `$_name` property. Any schema specified with the `$_name` property takes precedence over a schema specified with the `$_schema` property. In some RDBMS brands, the term for schema is "database" or "tablespace," but it is used similarly.

Example 11.76. Declaring a table class with schema

```
// First alternative:
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_schema = 'bug_db';
    protected $_name   = 'bugs';
}

// Second alternative:
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bug_db.bugs';
}

// If schemas are specified in both $_name and $_schema, the one
// specified in $_name takes precedence:

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name   = 'bug_db.bugs';
    protected $_schema = 'ignored';
}
```

The schema and table names may also be specified via constructor configuration directives, which override any default values specified with the `$_name` and `$_schema` properties. A schema specification given with the `name` directive overrides any value provided with the `schema` option.

Example 11.77. Declaring table and schema names upon instantiation

```
class Bugs extends Zend_Db_Table_Abstract
{
}

// First alternative:

$tableBugs = new Bugs(array('name' => 'bugs', 'schema' => 'bug_db'));

// Second alternative:

$tableBugs = new Bugs(array('name' => 'bug_db.bugs');

// If schemas are specified in both 'name' and 'schema', the one
// specified in 'name' takes precedence:

$tableBugs = new Bugs(array('name' => 'bug_db.bugs',
                            'schema' => 'ignored');
```

If you don't specify the schema name, it defaults to the schema to which your database adapter instance is connected.

Defining the Table Primary Key

Every table must have a primary key. You can declare the column for the primary key using the protected variable `$_primary`. This is either a string that names the single column for the primary key, or else it is an array of column names if your primary key is a compound key.

Example 11.78. Example of specifying the primary key

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_primary = 'bug_id';
}
```

If you don't specify the primary key, Zend_Db_Table_Abstract tries to discover the primary key based on the information provided by the `describeTable()´` method.

Note

Every table class must know which column(s) can be used to address rows uniquely. If no primary key column(s) are specified in the table class definition or the table constructor arguments, or discovered in the table metadata provided by `describeTable()`, then the table cannot be used with Zend_Db_Table.

Overriding Table Setup Methods

When you create an instance of a Table class, the constructor calls a set of protected methods that initialize metadata for the table. You can extend any of these methods to define metadata explicitly. Remember to call the method of the same name in the parent class at the end of your method.

Example 11.79. Example of overriding the _setupTableName() method

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected function _setupTableName()
    {
        $this->_name = 'bugs';
        parent::_setupTableName();
    }
}
```

The setup methods you can override are the following:

- `_setupDatabaseAdapter()` checks that an adapter has been provided; gets a default adapter from the registry if needed. By overriding this method, you can set a database adapter from some other source.
- `_setupTableName()` defaults the table name to the name of the class. By overriding this method, you can set the table name before this default behavior runs.
- `_setupMetadata()` sets the schema if the table name contains the pattern "schema.table"; calls `describeTable()` to get metadata information; defaults the `$_cols` array to the columns reported by `describeTable()`. By overriding this method, you can specify the columns.
- `_setupPrimaryKey()` defaults the primary key columns to those reported by `describeTable()`; checks that the primary key columns are included in the `$_cols` array. By overriding this method, you can specify the primary key columns.

Table initialization

If application-specific logic needs to be initialized when a Table class is constructed, you can select to move your tasks to the `init()` method, which is called after all Table metadata has been processed. This is recommended over the `__construct` method if you do not need to alter the metadata in any programmatic way.

Example 11.80. Example usage of init() method

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_observer;

    protected function init()
    {
        $this->_observer = new MyObserverClass();
    }
}
```

Creating an Instance of a Table

Before you use a Table class, create an instance using its constructor. The constructor's argument is an array of options. The most important option to a Table constructor is the database adapter instance, representing a live connection to an RDBMS. There are three ways of specifying the database adapter to a Table class, and these three ways are described below:

Specifying a Database Adapter

The first way to provide a database adapter to a Table class is by passing it as an object of type Zend_Db_Adapter_Abstract in the options array, identified by the key `'db'`.

Example 11.81. Example of constructing a Table using an Adapter object

```
$db = Zend_Db::factory('PDO_MYSQL', $options);

$table = new Bugs(array('db' => $db));
```

Setting a Default Database Adapter

The second way to provide a database adapter to a Table class is by declaring an object of type Zend_Db_Adapter_Abstract to be a default database adapter for all subsequent instances of Tables in your application. You can do this with the static method `Zend_Db_Table_Abstract::setDefaultAdapter()`. The argument is an object of type Zend_Db_Adapter_Abstract.

Example 11.82. Example of constructing a Table using a the Default Adapter

```
$db = Zend_Db::factory('PDO_MYSQL', $options);
Zend_Db_Table_Abstract::setDefaultAdapter($db);

// Later...

$table = new Bugs();
```

It can be convenient to create the database adapter object in a central place of your application, such as the bootstrap, and then store it as the default adapter. This gives you a means to ensure that the adapter instance is the same throughout your application. However, setting a default adapter is limited to a single adapter instance.

Storing a Database Adapter in the Registry

The third way to provide a database adapter to a Table class is by passing a string in the options array, also identified by the `'db'` key. The string is used as a key to the static Zend_Registry instance, where the entry at that key is an object of type Zend_Db_Adapter_Abstract.

Example 11.83. Example of constructing a Table using a Registry key

```
$db = Zend_Db::factory('PDO_MYSQL', $options);
Zend_Registry::set('my_db', $db);

// Later...

$table = new Bugs(array('db' => 'my_db'));
```

Like setting the default adapter, this gives you the means to ensure that the same adapter instance is used throughout your application. Using the registry is more flexible, because you can store more than one adapter instance. A given adapter instance is specific to a certain RDBMS brand and database instance. If your application needs access to multiple databases or even multiple database brands, then you need to use multiple adapters.

Inserting Rows to a Table

You can use the Table object to insert rows into the database table on which the Table object is based. Use the `insert()` method of your Table object. The argument is an associative array, mapping column names to values.

Example 11.84. Example of inserting to a Table

```
$table = new Bugs();

$data = array(
    'created_on'      => '2007-03-22',
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);

$table->insert($data);
```

By default, the values in your data array are inserted as literal values, using parameters. If you need them to be treated as SQL expressions, you must make sure they are distinct from plain strings. Use an object of type Zend_Db_Expr to do this.

Example 11.85. Example of inserting expressions to a Table

```
$table = new Bugs();

$data = array(
    'created_on'      => new Zend_Db_Expr('CURDATE()'),
    'bug_description' => 'Something wrong',
    'bug_status'      => 'NEW'
);
```

In the examples of inserting rows above, it is assumed that the table has an auto-incrementing primary key. This is the default behavior of Zend_Db_Table_Abstract, but there are other types of primary keys as well. The following sections describe how to support different types of primary keys.

Using a Table with an Auto-incrementing Key

An auto-incrementing primary key generates a unique integer value for you if you omit the primary key column from your SQL `INSERT` statement.

In Zend_Db_Table_Abstract, if you define the protected variable `$_sequence` to be the Boolean value `true`, then the class assumes that the table has an auto-incrementing primary key.

Example 11.86. Example of declaring a Table with auto-incrementing primary key

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    // This is the default in the Zend_Db_Table_Abstract class;
    // you do not need to define this.
    protected $_sequence = true;
}
```

MySQL, Microsoft SQL Server, and SQLite are examples of RDBMS brands that support auto-incrementing primary keys.

PostgreSQL has a `SERIAL` notation that implicitly defines a sequence based on the table and column name, and uses the sequence to generate key values for new rows. IBM DB2 has an `IDENTITY` notation that works similarly. If you use either of these notations, treat your Zend_Db_Table class as having an auto-incrementing column with respect to declaring the `$_sequence` member as `true`.

Using a Table with a Sequence

A sequence is a database object that generates a unique value, which can be used as a primary key value in one or more tables of the database.

If you define `$_sequence` to be a string, then Zend_Db_Table_Abstract assumes the string to name a sequence object in the database. The sequence is invoked to generate a new value, and this value is used in the `INSERT` operation.

Example 11.87. Example of declaring a Table with a sequence

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    protected $_sequence = 'bug_sequence';
}
```

Oracle, PostgreSQL, and IBM DB2 are examples of RDBMS brands that support sequence objects in the database.

PostgreSQL and IBM DB2 also have syntax that defines sequences implicitly and associated with columns. If you use this notation, treat the table as having an auto-incrementing key column. Define the sequence name as a string only in cases where you would invoke the sequence explicitly to get the next key value.

Using a Table with a Natural Key

Some tables have a natural key. This means that the key is not automatically generated by the table or by a sequence. You must specify the value for the primary key in this case.

If you define the `$_sequence` to be the Boolean value `false`, then Zend_Db_Table_Abstract assumes that the table has a natural primary key. You must provide values for the primary key columns in the array of data to the `insert()` method, or else this method throws a Zend_Db_Table_Exception.

Example 11.88. Example of declaring a Table with a natural key

```
class BugStatus extends Zend_Db_Table_Abstract
{
    protected $_name = 'bug_status';

    protected $_sequence = false;
}
```

Note

All RDBMS brands support tables with natural keys. Examples of tables that are often declared as having natural keys are lookup tables, intersection tables in many-to-many relationships, or most tables with compound primary keys.

Updating Rows in a Table

You can update rows in a database table using the `update` method of a Table class. This method takes two arguments: an associative array of columns to change and new values to assign to these columns; and an SQL expression that is used in a `WHERE` clause, as criteria for the rows to change in the `UPDATE` operation.

Example 11.89. Example of updating rows in a Table

```
$table = new Bugs();

$data = array(
    'updated_on'      => '2007-03-23',
    'bug_status'      => 'FIXED'
);

$where = $table->getAdapter()->quoteInto('bug_id = ?', 1234);

$table->update($data, $where);
```

Since the table `update()` method proxies to the database adapter `update()` method, the second argument can be an array of SQL expressions. The expressions are combined as Boolean terms using an `AND` operator.

Note

The values and identifiers in the SQL expression are not quoted for you. If you have values or identifiers that require quoting, you are responsible for doing this. Use the `quote()`, `quoteInto()`, and `quoteIdentifier()` methods of the database adapter.

Deleting Rows from a Table

You can delete rows from a database table using the `delete()` method. This method takes one argument, which is an SQL expression that is used in a `WHERE` clause, as criteria for the rows to delete.

Example 11.90. Example of deleting rows from a Table

```
$table = new Bugs();

$where = $table->getAdapter()->quoteInto('bug_id = ?', 1235);

$table->delete($where);
```

The second argument can be an array of SQL expressions. The expressions are combined as Boolean terms using an `AND` operator.

Since the table `delete()` method proxies to the database adapter `delete()` method, the second argument can be an array of SQL expressions. The expressions are combined as Boolean terms using an `AND` operator.

Note

The values and identifiers in the SQL expression are not quoted for you. If you have values or identifiers that require quoting, you are responsible for doing this. Use the `quote()`, `quoteInto()`, and `quoteIdentifier()` methods of the database adapter.

Finding Rows by Primary Key

You can query the database table for rows matching specific values in the primary key, using the `find()` method. The first argument of this method is either a single value or an array of values to match against the primary key of the table.

Example 11.91. Example of finding rows by primary key values

```
$table = new Bugs();

// Find a single row
// Returns a Rowset
$rows = $table->find(1234);

// Find multiple rows
// Also returns a Rowset
$rows = $table->find(array(1234, 5678));
```

If you specify a single value, the method returns at most one row, because a primary key cannot have duplicate values and there is at most one row in the database table matching the value you specify. If you specify multiple values in an array, the method returns at most as many rows as the number of distinct values you specify.

The `find()` method might return fewer rows than the number of values you specify for the primary key, if some of the values don't match any rows in the database table. The method even may return zero rows. Because the number of rows returned is variable, the `find()` method returns an object of type `Zend_Db_Table_Rowset_Abstract`.

If the primary key is a compound key, that is, it consists of multiple columns, you can specify the additional columns as additional arguments to the `find()` method. You must provide as many arguments as the number of columns in the table's primary key.

To find multiple rows from a table with a compound primary key, provide an array for each of the arguments. All of these arrays must have the same number of elements. The values in each array are formed into tuples in order; for example, the first element in all the array arguments define the first compound primary key value, then the second elements of all the arrays define the second compound primary key value, and so on.

Example 11.92. Example of finding rows by compound primary key values

The call to `find()` below to match multiple rows can match two rows in the database. The first row must have primary key value (1234, 'ABC'), and the second row must have primary key value (5678, 'DEF').

```
class BugsProducts extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs_products';
    protected $_primary = array('bug_id', 'product_id');
}

$table = new BugsProducts();

// Find a single row with a compound primary key
// Returns a Rowset
$rows = $table->find(1234, 'ABC');

// Find multiple rows with compound primary keys
// Also returns a Rowset
$rows = $table->find(array(1234, 5678), array('ABC', 'DEF'));
```

Querying for a Set of Rows

Select API

Warning

The API for fetch operations has been superceded to allow a `Zend_Db_Table_Select` object to modify the query. However, the deprecated usage of the `fetchRow()` and `fetchAll()` methods will continue to work without modification.

The following statements are all legal and functionally identical, however it is recommended to update your code to take advantage of the new usage where possible.

```
// Fetching a rowset
$rows = $table->fetchAll('bug_status = "NEW"', 'bug_id ASC', 10, 0);
$rows = $table->fetchAll($table->select()->where('bug_status = ?', 'NEW')
                                         ->order('bug_id ASC')
                                         ->limit(10, 0));

// Fetching a single row
$row = $table->fetchRow('bug_status = "NEW"', 'bug_id ASC');
$row = $table->fetchRow($table->select()->where('bug_status = ?', 'NEW')
                                        ->order('bug_id ASC'));
```

The `Zend_Db_Table_Select` object is an extension of the `Zend_Db_Select` object that applies specific restrictions to a query. The enhancements and restrictions are:

- You *can* elect to return a subset of columns within a fetchRow or fetchAll query. This can provide optimization benefits where returning a large set of results for all columns is not desirable.
- You *can* specify columns that evaluate expressions from within the selected table. However this will mean that the returned row or rowset will be readOnly and cannot be used for save() operations. A `Zend_Db_Table_Row` with readOnly status will throw an exception if a `save()` operation is attempted.
- You *can* allow JOIN clauses on a select to allow multi-table lookups.
- You *can not* specify columns from a JOINed tabled to be returned in a row/rowset. Doing so will trigger a PHP error. This was done to ensure the integrity of the `Zend_Db_Table is retained`. i.e. A `Zend_Db_Table_Row` should only reference columns derived from its parent table.

Example 11.93. Simple usage

```
$table = new Bugs();

$select = $table->select();
$select->where('bug_status = ?', 'NEW');

$rows = $table->fetchAll($select);
```

Fluent interfaces are implemented across the component, so this can be rewritten this in a more abbreviated form.

Example 11.94. Example of fluent interface

```
$table = new Bugs();

$rows =
    $table->fetchAll($table->select()->where('bug_status = ?', 'NEW'));
```

Fetching a rowset

You can query for a set of rows using any criteria other than the primary key values, using the `fetchAll()` method of the Table class. This method returns an object of type `Zend_Db_Table_Rowset_Abstract`.

Example 11.95. Example of finding rows by an expression

```
$table = new Bugs();

$select = $table->select()->where('bug_status = ?', 'NEW');

$rows = $table->fetchAll($select);
```

You may also pass sorting criteria in an `ORDER BY` clause, as well as count and offset integer values, used to make the query return a specific subset of rows. These values are used in a `LIMIT` clause, or in equivalent logic for RDBMS brands that do not support the `LIMIT` syntax.

Example 11.96. Example of finding rows by an expression

```
$table = new Bugs();

$order  = 'bug_id';

// Return the 21st through 30th rows
$count  = 10;
$offset = 20;

$select = $table->select()->where(array('bug_status = ?' => 'NEW'))
                          ->order($order)
                          ->limit($count, $offset);

$rows = $table->fetchAll($select);
```

All of the arguments above are optional. If you omit the ORDER clause, the result set includes rows from the table in an unpredictable order. If no LIMIT clause is set, you retrieve every row in the table that matches the WHERE clause.

Advanced usage

For more specific and optimized requests, you may wish to limit the number of columns returned in a row/rowset. This can be achieved by passing a FROM clause to the select object. The first argument in the FROM clause is identical to that of a Zend_Db_Select object with the addition of being able to pass an instance of Zend_Db_Table_Abstract and have it automatically determine the table name.

Example 11.97. Retrieving specific columns

```
$table = new Bugs();

$select = $table->select();
$select->from($table, array('bug_id', 'bug_description'))
       ->where('bug_status = ?', 'NEW');

$rows = $table->fetchAll($select);
```

Important

The rowset contains rows that are still 'valid' - they simply contain a subset of the columns of a table. If a save() method is called on a partial row then only the fields available will be modified.

You can also specify expressions within a FROM clause and have these returned as a readOnly row/rowset. In this example we will return a rows from the bugs table that show an aggregate of the number of new bugs reported by individuals. Note the GROUP clause. The 'count' column will be made available to the row for evaluation and can be accessed as if it were part of the schema.

Example 11.98. Retrieving expressions as columns

```
$table = new Bugs();

$select = $table->select();
$select->from($table,
              array('COUNT(reported_by) as `count`', 'reported_by'))
       ->where('bug_status = ?', 'NEW')
       ->group('reported_by');

$rows = $table->fetchAll($select);
```

You can also use a lookup as part of your query to further refine your fetch operations. In this example the accounts table is queried as part of a search for all new bugs reported by 'Bob'.

Example 11.99. Using a lookup table to refine the results of fetchAll()

```
$table = new Bugs();

$select = $table->select();
$select->where('bug_status = ?', 'NEW')
       ->join('accounts', 'accounts.account_name = bugs.reported_by')
       ->where('accounts.account_name = ?', 'Bob');

$rows = $table->fetchAll($select);
```

The `Zend_Db_Table_Select` is primarily used to constrain and validate so that it may enforce the criteria for a legal SELECT query. However there may be certain cases where you require the flexibility of the Zend_Db_Table_Row component and do not require a writable or deletable row. For this specific user case, it is possible to retrieve a row/rowset by passing a false value to setIntegrityCheck. The resulting row/rowset will be returned as a 'locked' row (meaning the save(), delete() and any field-setting methods will throw an exception).

Example 11.100. Removing the integrity check on Zend_Db_Table_Select to allow JOINed rows

```
$table = new Bugs();

$select = $table->select()->setIntegrityCheck(false);
$select->where('bug_status = ?', 'NEW')
       ->join('accounts',
              'accounts.account_name = bugs.reported_by',
              'account_name')
       ->where('accounts.account_name = ?', 'Bob');

$rows = $table->fetchAll($select);
```

Querying for a Single Row

You can query for a single row using criteria similar to that of the `fetchAll()` method.

Example 11.101. Example of finding a single row by an expression

```
$table = new Bugs();

$select  = $table->select()->where('bug_status = ?', 'NEW')
                           ->order('bug_id');

$row = $table->fetchRow($select);
```

This method returns an object of type Zend_Db_Table_Row_Abstract. If the search criteria you specified match no rows in the database table, then `fetchRow()` returns PHP's `null` value.

Retrieving Table Metadata Information

The Zend_Db_Table_Abstract class provides some information about its metadata. The `info()` method returns an array structure with information about the table, its columns and primary key, and other metadata.

Example 11.102. Example of getting the table name

```
$table = new Bugs();

$info = $table->info();

echo "The table name is " . $info['name'] . "\n";
```

The keys of the array returned by the `info()` method are described below:

- **name** => the name of the table.
- **cols** => an array, naming the column(s) of the table.
- **primary** => an array, naming the column(s) in the primary key.
- **metadata** => an associative array, mapping column names to information about the columns. This is the information returned by the `describeTable()` method.
- **rowClass** => the name of the concrete class used for Row objects returned by methods of this table instance. This defaults to Zend_Db_Table_Row.
- **rowsetClass** => the name of the concrete class used for Rowset objects returned by methods of this table instance. This defaults to Zend_Db_Table_Rowset.
- **referenceMap** => an associative array, with information about references from this table to any parent tables. See the section called “Defining Relationships”.
- **dependentTables** => an array of class names of tables that reference this table. See the section called “Defining Relationships”.
- **schema** => the name of the schema (or database or tablespace) for this table.

Caching Table Metadata

By default, `Zend_Db_Table_Abstract` queries the underlying database for table metadata upon instantiation of a table object. That is, when a new table object is created, the object's default behavior is to fetch the table metadata from the database using the adapter's `describeTable()` method.

In some circumstances, particularly when many table objects are instantiated against the same database table, querying the database for the table metadata for each instance may be undesirable from a performance standpoint. In such cases, users may benefit by caching the table metadata retrieved from the database.

There are two primary ways in which a user may take advantage of table metadata caching:

- **Call Zend_Db_Table_Abstract::setDefaultMetadataCache()** - This allows a developer to once set the default cache object to be used for all table classes.
- **Configure Zend_Db_Table_Abstract::__construct()** - This allows a developer to set the cache object to be used for a particular table class instance.

In both cases, the cache specification must be either `null` (i.e., no cache used) or an instance of `Zend_Cache_Core`. The methods may be used in conjunction when it is desirable to have both a default metadata cache and the ability to change the cache for individual table objects.

Example 11.103. Using a Default Metadata Cache for all Table Objects

The following code demonstrates how to set a default metadata cache to be used for all table objects:

```
<
// First, set up the Cache
$frontendOptions = array(
    'automatic_serialization' => true
    );

$backendOptions  = array(
    'cache_dir'                => 'cacheDir'
    );

$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);


// Next, set the cache to be used with all table objects
Zend_Db_Table_Abstract::setDefaultMetadataCache($cache);


// A table class is also needed
class Bugs extends Zend_Db_Table_Abstract
{
    // ...
}


// Each instance of Bugs now uses the default metadata cache
$bugs = new Bugs();
```

Example 11.104. Using a Metadata Cache for a Specific Table Object

The following code demonstrates how to set a metadata cache for a specific table object instance:

```
// First, set up the Cache
$frontendOptions = array(
    'automatic_serialization' => true
    );

$backendOptions  = array(
    'cache_dir'                => 'cacheDir'
    );

$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);

// A table class is also needed
class Bugs extends Zend_Db_Table_Abstract
{
    // ...
}

// Configure an instance upon instantiation
$bugs = new Bugs(array('metadataCache' => $cache));
```

Automatic Serialization with the Cache Frontend

Since the information returned from the adapter's describeTable() method is an array, ensure that the `automatic_serialization` option is set to `true` for the `Zend_Cache_Core` frontend.

Though the above examples use `Zend_Cache_Backend_File`, developers may use whatever cache backend is appropriate for the situation. Please see Zend_Cache for more information.

Customizing and Extending a Table Class

Using Custom Row or Rowset Classes

By default, methods of the Table class return a Rowset in instances of the concrete class Zend_Db_Table_Rowset, and Rowsets contain a collection of instances of the concrete class Zend_Db_Table_Row. You can specify an alternative class to use for either of these, but they must be classes that extend Zend_Db_Table_Rowset_Abstract and Zend_Db_Table_Row_Abstract, respectively.

You can specify Row and Rowset classes using the Table constructor's options array, in keys `'rowClass'` and `'rowsetClass'` respectively. Specify the names of the classes using strings.

Example 11.105. Example of specifying the Row and Rowset classes

```
class My_Row extends Zend_Db_Table_Row_Abstract
{
    ...
}

class My_Rowset extends Zend_Db_Table_Rowset_Abstract
{
    ...
}

$table = new Bugs(
    array(
        'rowClass'    => 'My_Row',
        'rowsetClass' => 'My_Rowset'
    )
);

$where = $table->getAdapter()->quoteInto('bug_status = ?', 'NEW')

// Returns an object of type My_Rowset,
// containing an array of objects of type My_Row.
$rows = $table->fetchAll($where);
```

You can change the classes by specifying them with the `setRowClass()` and `setRowsetClass()` methods. This applies to rows and rowsets created subsequently; it does not change the class of any row or rowset objects you have created previously.

Example 11.106. Example of changing the Row and Rowset classes

```
$table = new Bugs();

$where = $table->getAdapter()->quoteInto('bug_status = ?', 'NEW')

// Returns an object of type Zend_Db_Table_Rowset
// containing an array of objects of type Zend_Db_Table_Row.
$rowsStandard = $table->fetchAll($where);

$table->setRowClass('My_Row');
$table->setRowsetClass('My_Rowset');

// Returns an object of type My_Rowset,
// containing an array of objects of type My_Row.
$rowsCustom = $table->fetchAll($where);

// The $rowsStandard object still exists, and it is unchanged.
```

For more information on the Row and Rowset classes, see the section called "Zend_Db_Table_Row" and the section called "Zend_Db_Table_Rowset".

Defining Custom Logic for Insert, Update, and Delete

You can override the `insert()` and `update()` methods in your Table class. This gives you the opportunity to implement custom code that is executed before performing the database operation. Be sure to call the parent class method when you are done.

Example 11.107. Custom logic to manage timestamps

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    public function insert(array $data)
    {
        // add a timestamp
        if (empty($data['created_on'])) {
            $data['created_on'] = time();
        }
        return parent::insert($data);
    }

    public function update(array $data, $where)
    {
        // add a timestamp
        if (empty($data['updated_on'])) {
            $data['updated_on'] = time();
        }
        return parent::update($data, $where);
    }
}
```

You can also override the `delete()` method.

Define Custom Search Methods in Zend_Db_Table

You can implement custom query methods in your Table class, if you have frequent need to do queries against this table with specific criteria. Most queries can be written using `fetchAll()`, but this requires that you duplicate code to form the query conditions if you need to run the query in several places in your application. Therefore it can be convenient to implement a method in the Table class to perform frequently-used queries against this table.

Example 11.108. Custom method to find bugs by status

```
class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';

    public function findByStatus($status)
    {
        $where = $this->getAdapter()->quoteInto('bug_status = ?', $status);
        return $this->fetchAll($where, 'bug_id');
    }
}
```

Define Inflection in Zend_Db_Table

Some people prefer that the table class name match a table name in the RDBMS by using a string transformation called *inflection*.

For example, if your table class name is "`BugsProducts`", it would match the physical table in the database called "`bugs_products`," if you omit the explicit declaration of the `$_name` class property. In this inflection mapping, the class name spelled in "CamelCase" format would be transformed to lower case, and words are separated with an underscore.

You can specify the database table name independently from the class name by declaring the table name with the `$_name` class property in each of your table classes.

Zend_Db_Table_Abstract performs no inflection to map the class name to the table name. If you omit the declaration of `$_name` in your table class, the class maps to a database table that matches the spelling of the class name exactly.

It is inappropriate to transform identifiers from the database, because this can lead to ambiguity or make some identifiers inaccessible. Using the SQL identifiers exactly as they appear in the database makes Zend_Db_Table_Abstract both simpler and more flexible.

If you prefer to use inflection, then you must implement the transformation yourself, by overriding the `_setupTableName()` method in your Table classes. One way to do this is to define an abstract class that extends Zend_Db_Table_Abstract, and then the rest of your tables extend your new abstract class.

Example 11.109. Example of an abstract table class that implements inflection

```
abstract class MyAbstractTable extends Zend_Db_Table_Abstract
{
    protected function _setupTableName()
    {
        if (!$this->_name) {
            $this->_name = myCustomInflector(get_class($this));
        }
        parent::_setupTableName();
    }
}

class BugsProducts extends MyAbstractTable
{
}
```

You are responsible for writing the functions to perform inflection transformation. Zend Framework does not provide such a function.

Zend_Db_Table_Row

Introduction

Zend_Db_Table_Row is a class that contains an individual row of a Zend_Db_Table object. When you run a query against a Table class, the result is returned in a set of Zend_Db_Table_Row objects. You can also use this object to create new rows and add them to the database table.

Zend_Db_Table_Row is an implementation of the Row Data Gateway [http://www.martinfowler.com/eaaCatalog/rowDataGateway.html] pattern.

Fetching a Row

Zend_Db_Table_Abstract provides methods `find()` and `fetchAll()`, which each return an object of type Zend_Db_Table_Rowset, and the method `fetchRow()`, which returns an object of type Zend_Db_Table_Row.

Example 11.110. Example of fetching a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));
```

A Zend_Db_Table_Rowset object contains a collection of Zend_Db_Table_Row objects. See the section called “Zend_Db_Table_Rowset”.

Example 11.111. Example of reading a row in a rowset

```
$bugs = new Bugs();
$rowset = $bugs->fetchAll($bugs->select()->where('bug_status = ?', 1));
$row = $rowset->current();
```

Reading column values from a row

Zend_Db_Table_Row_Abstract provides accessor methods so you can reference columns in the row as object properties.

Example 11.112. Example of reading a column in a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));

// Echo the value of the bug_description column
echo $row->bug_description;
```

Note

Earlier versions of Zend_Db_Table_Row mapped these column accessors to the database column names using a string transformation called *inflection*.

Currently, Zend_Db_Table_Row does not implement inflection. Accessed property names need to match the spelling of the column names as they appear in your database.

Retrieving Row Data as an Array

You can access the row's data as an array using the `toArray()` method of the Row object. This returns an associative array of the column names to the column values.

Example 11.113. Example of using the toArray() method

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));

// Get the column/value associative array from the Row object
$rowArray = $row->toArray();

// Now use it as a normal array
foreach ($rowArray as $column => $value) {
    echo "Column: $column\n";
    echo "Value:  $value\n";
}
```

The array returned from `toArray()` is not updateable. You can modify values in the array as you can with any array, but you cannot save changes to this array to the database directly.

Fetching data from related tables

The Zend_Db_Table_Row_Abstract class provides methods for fetching rows and rowsets from related tables. See the section called “Zend_Db_Table Relationships” for more information on table relationships.

Writing rows to the database

Changing column values in a row

You can set individual column values using column accessors, similar to how the columns are read as object properties in the example above.

Using a column accessor to set a value changes the column value of the row object in your application, but it does not commit the change to the database yet. You can do that with the `save()` method.

Example 11.114. Example of changing a column in a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow($bugs->select()->where('bug_id = ?', 1));

// Change the value of one or more columns
$row->bug_status = 'FIXED';

// UPDATE the row in the database with new values
$row->save();
```

Inserting a new row

You can create a new row for a given table with the `createRow()` method of the table class. You can access fields of this row with the object-oriented interface, but the row is not stored in the database until you call the `save()` method.

Example 11.115. Example of creating a new row for a table

```
$bugs = new Bugs();
$newRow = $bugs->createRow();

// Set column values as appropriate for your application
$newRow->bug_description = '...description...';
$newRow->bug_status = 'NEW';

// INSERT the new row to the database
$newRow->save();
```

The optional argument to the createRow() method is an associative array, with which you can populate fields of the new row.

Example 11.116. Example of populating a new row for a table

```
$data = array(
    'bug_description' => '...description...',
    'bug_status'      => 'NEW'
);

$bugs = new Bugs();
$newRow = $bugs->createRow($data);

// INSERT the new row to the database
$newRow->save();
```

Note

The `createRow()` method was called `fetchNew()` in earlier releases of Zend_Db_Table. You are encouraged to use the new method name, even though the old name continues to work for the sake of backward compatibility.

Changing values in multiple columns

Zend_Db_Table_Row_Abstract provides the `setFromArray()` method to enable you to set several columns in a single row at once, specified in an associative array that maps the column names to values. You may find this method convenient for setting values both for new rows and for rows you need to update.

Example 11.117. Example of using setFromArray() to set values in a new Row

```
$bugs = new Bugs();
$newRow = $bugs->createRow();

// Data are arranged in an associative array
$data = array(
    'bug_description' => '...description...',
    'bug_status'      => 'NEW'
);

// Set all the column values at once
$newRow->setFromArray($data);

// INSERT the new row to the database
$newRow->save();
```

Deleting a row

You can call the `delete()` method on a Row object. This deletes rows in the database matching the primary key in the Row object.

Example 11.118. Example of deleting a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow('bug_id = 1');

// DELETE this row
$row->delete();
```

You do not have to call `save()` to apply the delete; it is executed against the database immediately.

Serializing and unserializing rows

It is often convenient to save the contents of a database row to be used later. *Serialization* is the name for the operation that converts an object into a form that is easy to save in offline storage (for example, a file). Objects of type Zend_Db_Table_Row_Abstract are serializable.

Serializing a Row

Simply use PHP's `serialize()` function to create a string containing a byte-stream representation of the Row object argument.

Example 11.119. Example of serializing a row

```
$bugs = new Bugs();
$row = $bugs->fetchRow('bug_id = 1');

// Convert object to serialized form
$serializedRow = serialize($row);

// Now you can write $serializedRow to a file, etc.
```

Unserializing Row Data

Use PHP's `unserialize()` function to restore a string containing a byte-stream representation of an object. The function returns the original object.

Note that the Row object returned is in a *disconnected* state. You can read the Row object and its properties, but you cannot change values in the Row or execute other methods that require a database connection (for example, queries against related tables).

Example 11.120. Example of unserializing a serialized row

```
$rowClone = unserialize($serializedRow);

// Now you can use object properties, but read-only
echo $rowClone->bug_description;
```

Why do Rows unserialize in a disconnected state?

A serialized object is a string that is readable to anyone who possesses it. It could be a security risk to store parameters such as database account and password in plain, unencrypted text in the serialized string. You would not want to store such data to a text file that is not protected, or send it in an email or other medium that is easily read by potential attackers. The reader of the serialized object should not be able to use it to gain access to your database without knowing valid credentials.

Reactivating a Row as Live Data

You can reactivate a disconnected Row, using the `setTable()` method. The argument to this method is a valid object of type Zend_Db_Table_Abstract, which you create. Creating a Table object requires a live connection to the database, so by reassociating the Table with the Row, the Row gains access to the database. Subsequently, you can change values in the Row object and save the changes to the database.

Example 11.121. Example of reactivating a row

```
$rowClone = unserialize($serializedRow);

$bugs = new Bugs();

// Reconnect the row to a table, and
// thus to a live database connection
$rowClone->setTable($bugs);

// Now you can make changes to the row and save them
$rowClone->bug_status = 'FIXED';
$rowClone->save();
```

Extending the Row class

Zend_Db_Table_Row is the default concrete class that extends Zend_Db_Table_Row_Abstract. You can define your own concrete class for instances of Row by extending Zend_Db_Table_Row_Abstract. To use your new Row class to store results of Table queries, specify the custom Row class by name either in the `$_rowClass` protected member of a Table class, or in the array argument of the constructor of a Table object.

Example 11.122. Specifying a custom Row class

```
class MyRow extends Zend_Db_Table_Row_Abstract
{
    // ...customizations
}

// Specify a custom Row to be used by default
// in all instances of a Table class.
class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowClass = 'MyRow';
}

// Or specify a custom Row to be used in one
// instance of a Table class.
$bugs = new Bugs(array('rowClass' => 'MyRow'));
```

Row initialization

If application-specific logic needs to be initialized when a row is constructed, you can select to move your tasks to the `init()` method, which is called after all row metadata has been processed. This is recommended over the `__construct` method if you do not need to alter the metadata in any programmatic way.

Example 11.123. Example usage of init() method

```
class MyApplicationRow extends Zend_Db_Table_Row_Abstract
{
    protected $_role;

    public function init()
    {
        $this->_role = new MyRoleClass();
    }
}
```

Defining Custom Logic for Insert, Update, and Delete in Zend_Db_Table_Row

The Row class calls protected methods `_insert()`, `_update()`, and `_delete()` before performing the corresponding operations `INSERT`, `UPDATE`, and `DELETE`. You can add logic to these methods in your custom Row subclass.

If you need to do custom logic in a specific table, and the custom logic must occur for every operation on that table, it may make more sense to implement your custom code in the `insert()`, `update()` and `delete()` methods of your Table class. However, sometimes it may be necessary to do custom logic in the Row class.

Below are some example cases where it might make sense to implement custom logic in a Row class instead of in the Table class:

Example 11.124. Example of custom logic in a Row class

The custom logic may not apply in all cases of operations on the respective Table. You can provide custom logic on demand by implementing it in a Row class and creating an instance of the Table class with that custom Row class specified. Otherwise, the Table uses the default Row class.

You need data operations on this table to record the operation to a Zend_Log object, but only if the application configuration has enabled this behavior.

```
class MyLoggingRow extends Zend_Db_Table_Row_Abstract
{
    protected function _insert()
    {
        $log = Zend_Registry::get('database_log');
        $log->info(Zend_Debug::dump($this->_data,
                                    "INSERT: $this->_tableClass",
                                    false)
                  );
    }
}

// $loggingEnabled is an example property that depends
// on your application configuration
if ($loggingEnabled) {
    $bugs = new Bugs(array('rowClass' => 'MyLoggingRow'));
} else {
    $bugs = new Bugs();
}
```

Example 11.125. Example of a Row class that logs insert data for multiple tables

The custom logic may be common to multiple tables. Instead of implementing the same custom logic in every one of your Table classes, you can implement the code for such actions in the definition of a Row class, and use this Row in each of your Table classes.

In this example, the logging code is identical in all table classes.

```
class MyLoggingRow extends Zend_Db_Table_Row_Abstract
{
    protected function _insert()
    {
        $log = Zend_Registry::get('database_log');
        $log->info(Zend_Debug::dump($this->_data,
                                    "INSERT: $this->_tableClass",
                                    false)
                  );
    }
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowClass = 'MyLoggingRow';
}

class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowClass = 'MyLoggingRow';
}
```

Define Inflection in Zend_Db_Table_Row

Some people prefer that the table class name match a table name in the RDBMS by using a string transformation called *inflection*.

Zend_Db classes do not implement inflection by default. See the section called “Define Inflection in Zend_Db_Table” for an explanation of this policy.

If you prefer to use inflection, then you must implement the transformation yourself, by overriding the `_transformColumn()` method in a custom Row class, and using that custom Row class when you perform queries against your Table class.

Example 11.126. Example of defining an inflection transformation

This allows you to use an inflected version of the column name in the accessors. The Row class uses the `_transformColumn()` method to change the name you use to the native column name in the database table.

```
class MyInflectedRow extends Zend_Db_Table_Row_Abstract
{
    protected function _transformColumn($columnName)
    {
        $nativeColumnName = myCustomInflector($columnName);
        return $nativeColumnName;
    }
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowClass = 'MyInflectedRow';
}

$bugs = new Bugs();
$row = $bugs->fetchNew();

// Use camelcase column names, and rely on the
// transformation function to change it into the
// native representation.
$row->bugDescription = 'New description';
```

You are responsible for writing the functions to perform inflection transformation. Zend Framework does not provide such a function.

Zend_Db_Table_Rowset

Introduction

When you run a query against a Table class using the `find()` or `fetchAll()` methods, the result is returned in an object of type `Zend_Db_Table_Rowset_Abstract`. A Rowset contains a collection of objects descending from `Zend_Db_Table_Row_Abstract`. You can iterate through the Rowset and access individual Row objects, reading or modifying data in the Rows.

Fetching a Rowset

`Zend_Db_Table_Abstract` provides methods `find()` and `fetchAll()`, each of which returns an object of type `Zend_Db_Table_Rowset_Abstract`.

Example 11.127. Example of fetching a rowset

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll("bug_status = 'NEW'");
```

Retrieving Rows from a Rowset

The Rowset itself is usually less interesting than the Rows that it contains. This section illustrates how to get the Rows that comprise the Rowset.

A legitimate query returns zero rows when no rows in the database match the query conditions. Therefore, a Rowset object might contain zero Row objects. Since `Zend_Db_Table_Rowset_Abstract` implements the `Countable` interface, you can use `count()` to determine the number of Rows in the Rowset.

Example 11.128. Counting the Rows in a Rowset

```
$rowset   = $bugs->fetchAll("bug_status = 'FIXED'");

$rowCount = count($rowset);

if ($rowCount > 0) {
    echo "found $rowCount rows";
} else {
    echo 'no rows matched the query';
}
```

Example 11.129. Reading a Single Row from a Rowset

The simplest way to access a Row from a Rowset is to use the `current()` method. This is particularly appropriate when the Rowset contains exactly one Row.

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll("bug_id = 1");
$row    = $rowset->current();
```

If the Rowset contains zero rows, `current()` returns PHP's `null` value.

Example 11.130. Iterating through a Rowset

Objects descending from `Zend_Db_Table_Rowset_Abstract` implement the `SeekableIterator` interface, which means you can loop through them using the `foreach` construct. Each value you retrieve this way is a `Zend_Db_Table_Row_Abstract` object that corresponds to one record from the table.

```
$bugs = new Bugs();

// fetch all records from the table
$rowset = $bugs->fetchAll();

foreach ($rowset as $row) {

    // output 'Zend_Db_Table_Row' or similar
    echo get_class($row) . "\n";

    // read a column in the row
    $status = $row->bug_status;

    // modify a column in the current row
    $row->assigned_to = 'mmouse';

    // write the change to the database
    $row->save();
}
```

Example 11.131. Seeking to a known position into a Rowset

`SeekableIterator` allows you to seek to a position that you would like the iterator to jump to. Simply use the `seek()` method for that. Pass it an integer representing the number of the Row you would like your Rowset to point to next, don't forget that it starts with index 0. If the index is wrong, ie doesn't exist, an exception will be thrown. You should use `count()` to check the number of results before seeking to a position.

```
$bugs = new Bugs();

// fetch all records from the table
$rowset = $bugs->fetchAll();

// takes the iterator to the 9th element (zero is one element) :
$rowset->seek(8);

// retrive it
$row9 = $rowset->current();

// and use it
$row9->assigned_to = 'mmouse';
$row9->save();
```

`getRow()` allows you to get a specific row in the Rowset, knowing its position; don't forget however that positions start with index zero. The first parameter for `getRow()` is an integer for the position asked. The second optional parameter is a boolean; it tells the Rowset iterator if it must seek to that position in the same time, or not (default is false). This method returns a Zend_Db_Table_Row object by default. If the position requested does not exist, an exception will be thrown. Here is an example :

```
$bugs = new Bugs();

// fetch all records from the table
$rowset = $bugs->fetchAll();

// retrieve the 9th element immediately:
$row9->getRow(8);

// and use it:
$row9->assigned_to = 'mmouse';
$row9->save();
```

After you have access to an individual Row object, you can manipulate the Row using methods described in the section called “Zend_Db_Table_Row”.

Retrieving a Rowset as an Array

You can access all the data in the Rowset as an array using the `toArray()` method of the Rowset object. This returns an array containing one entry per Row. Each entry is an associative array having keys that correspond to column names and elements that correspond to the respective column values.

Example 11.132. Using toArray()

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll();

$rowsetArray = $rowset->toArray();

$rowCount = 1;
foreach ($rowsetArray as $rowArray) {
    echo "row #$rowCount:\n";
    foreach ($rowArray as $column => $value) {
        echo "\t$column => $value\n";
    }
    ++$rowCount;
    echo "\n";
}
```

The array returned from `toArray()` is not updateable. That is, you can modify values in the array as you can with any array, but changes to the array data are not propagated to the database.

Serializing and Unserializing a Rowset

Objects of type `Zend_Db_Table_Rowset_Abstract` are serializable. In a similar fashion to serializing an individual Row object, you can serialize a Rowset and unserialize it later.

Example 11.133. Serializing a Rowset

Simply use PHP's `serialize()` function to create a string containing a byte-stream representation of the Rowset object argument.

```
$bugs   = new Bugs();
$rowset = $bugs->fetchAll();

// Convert object to serialized form
$serializedRowset = serialize($rowset);

// Now you can write $serializedRowset to a file, etc.
```

Example 11.134. Unserializing a Serialized Rowset

Use PHP's `unserialize()` function to restore a string containing a byte-stream representation of an object. The function returns the original object.

Note that the Rowset object returned is in a *disconnected* state. You can iterate through the Rowset and read the Row objects and their properties, but you cannot change values in the Rows or execute other methods that require a database connection (for example, queries against related tables).

```
$rowsetDisconnected = unserialize($serializedRowset);

// Now you can use object methods and properties, but read-only
$row = $rowsetDisconnected->current();
echo $row->bug_description;
```

Why do Rowsets unserialize in a disconnected state?

A serialized object is a string that is readable to anyone who possesses it. It could be a security risk to store parameters such as database account and password in plain, unencrypted text in the serialized string. You would not want to store such data to a text file that is not protected, or send it in an email or other medium that is easily read by potential attackers. The reader of the serialized object should not be able to use it to gain access to your database without knowing valid credentials.

You can reactivate a disconnected Rowset using the `setTable()` method. The argument to this method is a valid object of type `Zend_Db_Table_Abstract`, which you create. Creating a Table object requires a live connection to the database, so by reassociating the Table with the Rowset, the Rowset gains access to the database. Subsequently, you can change values in the Row objects contained in the Rowset and save the changes to the database.

Example 11.135. Reactivating a Rowset as Live Data

```
$rowset = unserialize($serializedRowset);

$bugs = new Bugs();

// Reconnect the rowset to a table, and
// thus to a live database connection
$rowset->setTable($bugs);

$row = $rowset->current();

// Now you can make changes to the row and save them
$row->bug_status = 'FIXED';
$row->save();
```

Reactivating a Rowset with `setTable()` also reactivates all the Row objects contained in that Rowset.

Extending the Rowset class

You can use an alternative concrete class for instances of Rowsets by extending Zend_Db_Table_Rowset_Abstract. Specify the custom Rowset class by name either in the `$_rowsetClass` protected member of a Table class, or in the array argument of the constructor of a Table object.

Example 11.136. Specifying a custom Rowset class

```
class MyRowset extends Zend_Db_Table_Rowset_Abstract
{
    // ...customizations
}

// Specify a custom Rowset to be used by default
// in all instances of a Table class.
class Products extends Zend_Db_Table_Abstract
{
    protected $_name = 'products';
    protected $_rowsetClass = 'MyRowset';
}

// Or specify a custom Rowset to be used in one
// instance of a Table class.
$bugs = new Bugs(array('rowsetClass' => 'MyRowset'));
```

Typically, the standard Zend_Db_Rowset concrete class is sufficient for most usage. However, you might find it useful to add new logic to a Rowset, specific to a given Table. For example, a new method could calculate an aggregate over all the Rows in the Rowset.

Example 11.137. Example of Rowset class with a new method

```
class MyBugsRowset extends Zend_Db_Table_Rowset_Abstract
{
    /**
     * Find the Row in the current Rowset with the
     * greatest value in its 'updated_at' column.
     */
    public function getLatestUpdatedRow()
    {
        $max_updated_at = 0;
        $latestRow = null;
        foreach ($this as $row) {
            if ($row->updated_at > $max_updated_at) {
                $latestRow = $row;
            }
        }
        return $latestRow;
    }
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs';
    protected $_rowsetClass = 'MyBugsRowset';
}
```

Zend_Db_Table Relationships

Introduction

Tables have relationships to each other in a relational database. An entity in one table can be linked to one or more entities in another table by using referential integrity constraints defined in the database schema.

The Zend_Db_Table_Row class has methods for querying related rows in other tables.

Defining Relationships

Define classes for each of your tables, extending the abstract class Zend_Db_Table_Abstract, as described in the section called “Defining a Table Class”. Also see the section called “The example database” for a description of the example database for which the following example code is designed.

Below are the PHP class definitions for these tables:

```
class Accounts extends Zend_Db_Table_Abstract
{
    protected $_name            = 'accounts';
    protected $_dependentTables = array('Bugs');
}

class Products extends Zend_Db_Table_Abstract
{
    protected $_name            = 'products';
    protected $_dependentTables = array('BugsProducts');
}

class Bugs extends Zend_Db_Table_Abstract
{
    protected $_name            = 'bugs';

    protected $_dependentTables = array('BugsProducts');

    protected $_referenceMap    = array(
        'Reporter' => array(
            'columns'           => 'reported_by',
            'refTableClass'     => 'Accounts',
            'refColumns'        => 'account_name'
        ),
        'Engineer' => array(
            'columns'           => 'assigned_to',
            'refTableClass'     => 'Accounts',
            'refColumns'        => 'account_name'
        ),
        'Verifier' => array(
            'columns'           => array('verified_by'),
            'refTableClass'     => 'Accounts',
            'refColumns'        => array('account_name')
        )
    );
}

class BugsProducts extends Zend_Db_Table_Abstract
{
    protected $_name = 'bugs_products';

    protected $_referenceMap    = array(
        'Bug' => array(
            'columns'           => array('bug_id'),
            'refTableClass'     => 'Bugs',
            'refColumns'        => array('bug_id')
        ),
        'Product' => array(
            'columns'           => array('product_id'),
            'refTableClass'     => 'Products',
            'refColumns'        => array('product_id')
        )
    );

}
```

If you use Zend_Db_Table to emulate cascading UPDATE and DELETE operations, declare the `$_dependentTables` array in the class for the parent table. List the class name for each dependent table. Use the class name, not the physical name of the SQL table.

Note

Skip declaration of `$_dependentTables` if you use referential integrity constraints in the RDBMS server to implement cascading operations. See the section called “Cascading Write Operations” for more information.

Declare the `$_referenceMap` array in the class for each dependent table. This is an associative array of reference "rules". A reference rule identifies which table is the parent table in the relationship, and also lists which columns in the dependent table reference which columns in the parent table.

The rule key is a string used as an index to the `$_referenceMap` array. This rule key is used to identify each reference relationship. Choose a descriptive name for this rule key. It's best to use a string that can be part of a PHP method name, as you will see later.

In the example PHP code above, the rule keys in the Bugs table class are: `'Reporter'`, `'Engineer'`, `'Verifier'`, and `'Product'`.

The value of each rule entry in the `$_referenceMap` array is also an associative array. The elements of this rule entry are described below:

- **columns** => A string or an array of strings naming the foreign key column name(s) in the dependent table.

  It's common for this to be a single column, but some tables have multi-column keys.

- **refTableClass** => The class name of the parent table. Use the class name, not the physical name of the SQL table.

  It's common for a dependent table to have only one reference to its parent table, but some tables have multiple references to the same parent table. In the example database, there is one reference from the `bugs` table to the `products` table, but three references from the `bugs` table to the `accounts` table. Put each reference in a separate entry in the `$_referenceMap` array.

- **refColumns** => A string or an array of strings naming the primary key column name(s) in the parent table.

  It's common for this to be a single column, but some tables have multi-column keys. If the reference uses a multi-column key, the order of columns in the `'columns'` entry must match the order of columns in the `'refColumns'` entry.

  It is optional to specify this element. If you don't specify the `refColumns`, the column(s) reported as the primary key columns of the parent table are used by default.

- **onDelete** => The rule for an action to execute if a row is deleted in the parent table. See the section called “Cascading Write Operations” for more information.

- **onUpdate** => The rule for an action to execute if values in primary key columns are updated in the parent table. See the section called “Cascading Write Operations” for more information.

Fetching a Dependent Rowset

If you have a Row object as the result of a query on a parent table, you can fetch rows from dependent tables that reference the current row. Use the method:

```
$row->findDependentRowset($table, [$rule]);
```

This method returns a Zend_Db_Table_Rowset_Abstract object, containing a set of rows from the dependent table `$table` that refer to the row identified by the `$row` object.

The first argument `$table` can be a string that specifies the dependent table by its class name. You can also specify the dependent table by using an object of that table class.

Example 11.138. Fetching a Dependent Rowset

This example shows getting a Row object from the table `Accounts`, and finding the `Bugs` reported by that account.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();

$bugsReportedByUser = $user1234->findDependentRowset('Bugs');
```

The second argument `$rule` is optional. It is a string that names the rule key in the `$_referenceMap` array of the dependent table class. If you don't specify a rule, the first rule in the array that references the parent table is used. If you need to use a rule other than the first, you need to specify the key.

In the example code above, the rule key is not specified, so the rule used by default is the first one that matches the parent table. This is the rule `'Reporter'`.

Example 11.139. Fetching a Dependent Rowset By a Specific Rule

This example shows getting a Row object from the table `Accounts`, and finding the `Bugs` assigned to be fixed by the user of that account. The rule key string that corresponds to this reference relationship in this example is `'Engineer'`.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();

$bugsAssignedToUser = $user1234->findDependentRowset('Bugs', 'Engineer');
```

You can also add criteria, ordering and limits to your relationships using the parent row's select object.

Example 11.140. Fetching a Dependent Rowset using a Zend_Db_Table_Select

This example shows getting a Row object from the table `Accounts`, and finding the `Bugs` assigned to be fixed by the user of that account, limited only to 3 rows and ordered by name.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();
$select = $accountsTable->select()->order('name ASC')
                                  ->limit(3);

$bugsAssignedToUser = $user1234->findDependentRowset('Bugs',
                                                     'Engineer',
                                                     $select);
```

Alternatively, you can query rows from a dependent table using a special mechanism called a "magic method". Zend_Db_Table_Row_Abstract invokes the method: `findDependentRowset('<TableClass>', '<Rule>')` if you invoke a method on the Row object matching either of the following patterns:

- `$row->find<TableClass>()`
- `$row->find<TableClass>By<Rule>()`

In the patterns above, `<TableClass>` and `<Rule>` are strings that correspond to the class name of the dependent table, and the dependent table's rule key that references the parent table.

Note

Some application frameworks, such as Ruby on Rails, use a mechanism called "inflection" to allow the spelling of identifiers to change depending on usage. For simplicity, Zend_Db_Table_Row does not provide any inflection mechanism. The table identity and the rule key named in the method call must match the spelling of the class and rule key exactly.

Example 11.141. Fetching Dependent Rowsets using the Magic Method

This example shows finding dependent Rowsets equivalent to those in the previous examples. In this case, the application uses the magic method invocation instead of specifying the table and rule as strings.

```
$accountsTable = new Accounts();
$accountsRowset = $accountsTable->find(1234);
$user1234 = $accountsRowset->current();

// Use the default reference rule
$bugsReportedBy = $user1234->findBugs();

// Specify the reference rule
$bugsAssignedTo = $user1234->findBugsByEngineer();
```

Fetching a Parent Row

If you have a Row object as the result of a query on a dependent table, you can fetch the row in the parent to which the dependent row refers. Use the method:

```
$row->findParentRow($table, [$rule]);
```

There always should be exactly one row in the parent table referenced by a dependent row, therefore this method returns a Row object, not a Rowset object.

The first argument `$table` can be a string that specifies the parent table by its class name. You can also specify the parent table by using an object of that table class.

Example 11.142. Fetching the Parent Row

This example shows getting a Row object from the table `Bugs` (for example one of those bugs with status 'NEW'), and finding the row in the `Accounts` table for the user who reported the bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->fetchAll(array('bug_status = ?' => 'NEW'));
$bug1 = $bugsRowset->current();

$reporter = $bug1->findParentRow('Accounts');
```

The second argument `$rule` is optional. It is a string that names the rule key in the `$_referenceMap` array of the dependent table class. If you don't specify a rule, the first rule in the array that references the parent table is used. If you need to use a rule other than the first, you need to specify the key.

In the example above, the rule key is not specified, so the rule used by default is the first one that matches the parent table. This is the rule `'Reporter'`.

Example 11.143. Fetching a Parent Row By a Specific Rule

This example shows getting a Row object from the table `Bugs`, and finding the account for the engineer assigned to fix that bug. The rule key string that corresponds to this reference relationship in this example is `'Engineer'`.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->fetchAll(array('bug_status = ?', 'NEW'));
$bug1 = $bugsRowset->current();

$engineer = $bug1->findParentRow('Accounts', 'Engineer');
```

Alternatively, you can query rows from a parent table using a "magic method". Zend_Db_Table_Row_Abstract invokes the method: `findParentRow('<TableClass>', '<Rule>')` if you invoke a method on the Row object matching either of the following patterns:

- `$row->findParent<TableClass>([Zend_Db_Table_Select $select])`
- `$row->findParent<TableClass>By<Rule>([Zend_Db_Table_Select $select])`

In the patterns above, `<TableClass>` and `<Rule>` are strings that correspond to the class name of the parent table, and the dependent table's rule key that references the parent table.

Note

The table identity and the rule key named in the method call must match the spelling of the class and rule key exactly.

Example 11.144. Fetching the Parent Row using the Magic Method

This example shows finding parent Rows equivalent to those in the previous examples. In this case, the application uses the magic method invocation instead of specifying the table and rule as strings.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->fetchAll(array('bug_status = ?', 'NEW'));
$bug1 = $bugsRowset->current();

// Use the default reference rule
$reporter = $bug1->findParentAccounts();

// Specify the reference rule
$engineer = $bug1->findParentAccountsByEngineer();
```

Fetching a Rowset via a Many-to-many Relationship

If you have a Row object as the result of a query on one table in a many-to-many relationship (for purposes of the example, call this the "origin" table), you can fetch corresponding rows in the other table (call this the "destination" table) via an intersection table. Use the method:

```
$row->findManyToManyRowset($table,
                           $intersectionTable,
                           [$rule1,
                               [$rule2,
                                   [Zend_Db_Table_Select $select]
                               ]
                           ]);
```

This method returns a Zend_Db_Table_Rowset_Abstract containing rows from the table `$table`, satisfying the many-to-many relationship. The current Row object `$row` from the origin table is used to find rows in the intersection table, and that is joined to the destination table.

The first argument `$table` can be a string that specifies the destination table in the many-to-many relationship by its class name. You can also specify the destination table by using an object of that table class.

The second argument `$intersectionTable` can be a string that specifies the intersection table between the two tables in the the many-to-many relationship by its class name. You can also specify the intersection table by using an object of that table class.

Example 11.145. Fetching a Rowset with the Many-to-many Method

This example shows getting a Row object from from the origin table `Bugs`, and finding rows from the destination table `Products`, representing products related to that bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->find(1234);
$bug1234 = $bugsRowset->current();

$productsRowset = $bug1234->findManyToManyRowset('Products',
                                                 'BugsProducts');
```

The third and fourth arguments `$rule1` and `$rule2` are optional. These are strings that name the rule keys in the `$_referenceMap` array of the intersection table.

The `$rule1` key names the rule for the relationship from the intersection table to the origin table. In this example, this is the relationship from `BugsProducts` to `Bugs`.

The `$rule2` key names the rule for the relationship from the intersection table to the destination table. In this example, this is the relationship from `Bugs` to `Products`.

Similarly to the methods for finding parent and dependent rows, if you don't specify a rule, the method uses the first rule in the `$_referenceMap` array that matches the tables in the relationship. If you need to use a rule other than the first, you need to specify the key.

In the example code above, the rule key is not specified, so the rules used by default are the first ones that match. In this case, `$rule1` is `'Reporter'` and `$rule2` is `'Product'`.

Example 11.146. Fetching a Rowset with the Many-to-many Method By a Specific Rule

This example shows geting a Row object from from the origin table `Bugs`, and finding rows from the destination table `Products`, representing products related to that bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->find(1234);
$bug1234 = $bugsRowset->current();

$productsRowset = $bug1234->findManyToManyRowset('Products',
                                                 'BugsProducts',
                                                 'Bug');
```

Alternatively, you can query rows from the destination table in a many-to-many relationship using a "magic method." Zend_Db_Table_Row_Abstract invokes the method: `findManyToManyRowset('<TableClass>', '<IntersectionTableClass>', '<Rule1>', '<Rule2>')` if you invoke a method matching any of the following patterns:

- `$row->find<TableClass>Via<IntersectionTableClass> ([Zend_Db_Table_Select $select])`
- `$row->find<TableClass>Via<IntersectionTableClass>By<Rule1> ([Zend_Db_Table_Select $select])`
- `$row->find<TableClass>Via<IntersectionTableClass>By<Rule1>And<Rule2> ([Zend_Db_Table_Select $select])`

In the patterns above, `<TableClass>` and `<IntersectionTableClass>` are strings that correspond to the class names of the destination table and the intersection table, respectively. `<Rule1>` and `<Rule2>` are strings that correspond to the rule keys in the intersection table that reference the origin table and the destination table, respectively.

Note

The table identities and the rule keys named in the method call must match the spelling of the class and rule key exactly.

Example 11.147. Fetching Rowsets using the Magic Many-to-many Method

This example shows finding rows in the destination table of a many-to-many relationship representing products related to a given bug.

```
$bugsTable = new Bugs();
$bugsRowset = $bugsTable->find(1234);
$bug1234 = $bugsRowset->current();

// Use the default reference rule
$products = $bug1234->findProductsViaBugsProducts();

// Specify the reference rule
$products = $bug1234->findProductsViaBugsProductsByBug();
```

Cascading Write Operations

Declare DRI in the database:

Declaring cascading operations in Zend_Db_Table is intended **only** for RDBMS brands that do not support declarative referential integrity (DRI).

For example, if you use MySQL's MyISAM storage engine, or SQLite, these solutions do not support DRI. You may find it helpful to declare the cascading operations with Zend_Db_Table.

If your RDBMS implements DRI and the `ON DELETE` and `ON UPDATE` clauses, you should declare these clauses in your database schema, instead of using the cascading feature in Zend_Db_Table. Declaring cascading DRI rules in the RDBMS is better for database performance, consistency, and integrity.

Most importantly, do not declare cascading operations both in the RDBMS and in your Zend_Db_Table class.

You can declare cascading operations to execute against a dependent table when you apply an `UPDATE` or a `DELETE` to a row in a parent table.

Example 11.148. Example of a Cascading Delete

This example shows deleting a row in the `Products` table, which is configured to automatically delete dependent rows in the `Bugs` table.

```
$productsTable = new Products();
$productsRowset = $productsTable->find(1234);
$product1234 = $productsRowset->current();

$product1234->delete();
// Automatically cascades to Bugs table
// and deletes dependent rows.
```

Similarly, if you use `UPDATE` to change the value of a primary key in a parent table, you may want the value in foreign keys of dependent tables to be updated automatically to match the new value, so that such references are kept up to date.

It's usually not necessary to update the value of a primary key that was generated by a sequence or other mechanism. But if you use a *natural key* that may change value occasionally, it is more likely that you need to apply cascading updates to dependent tables.

To declare a cascading relationship in the Zend_Db_Table, edit the rules in the `$_referenceMap`. Set the associative array keys `'onDelete'` and `'onUpdate'` to the string 'cascade' (or the constant `self::CASCADE`). Before a row is deleted from the parent table, or its primary key values updated, any rows in the dependent table that refer to the parent's row are deleted or updated first.

Example 11.149. Example Declaration of Cascading Operations

In the example below, rows in the `Bugs` table are automatically deleted if the row in the `Products` table to which they refer is deleted. The `'onDelete'` element of the reference map entry is set to `self::CASCADE`.

No cascading update is done in the example below if the primary key value in the parent class is changed. The `'onUpdate'` element of the reference map entry is `self::RESTRICT`. You can get the same result using the value `self::NO_ACTION`, or by omitting the `'onUpdate'` entry.

```
class BugsProducts extends Zend_Db_Table_Abstract
{
    ...
    protected $_referenceMap = array(
        'Product' => array(
            'columns'           => array('product_id'),
            'refTableClass'     => 'Products',
            'refColumns'        => array('product_id'),
            'onDelete'          => self::CASCADE,
            'onUpdate'          => self::RESTRICT
        ),
        ...
    );
}
```

Notes Regarding Cascading Operations

Cascading operations invoked by Zend_Db_Table are not atomic.

This means that if your database implements and enforces referential integrity constraints, a cascading `UPDATE` executed by a Zend_Db_Table class conflicts with the constraint, and results in a referential integrity violation. You can use cascading `UPDATE` in Zend_Db_Table *only* if your database does not enforce that referential integrity constraint.

Cascading `DELETE` suffers less from the problem of referential integrity violations. You can delete dependent rows as a non-atomic action before deleting the parent row that they reference.

However, for both `UPDATE` and `DELETE`, changing the database in a non-atomic way also creates the risk that another database user can see the data in an inconsistent state. For example, if you delete a row and all its dependent rows, there is a small chance that another database client program can query the database after you have deleted the dependent rows, but before you delete the parent row. That client program may see the parent row with no dependent rows, and assume this is the intended state of the data. There is no way for that client to know that its query read the database in the middle of a change.

The issue of non-atomic change can be mitigated by using transactions to isolate your change. But some RDBMS brands don't support transactions, or allow clients to read "dirty" changes that have not been committed yet.

Cascading operations in Zend_Db_Table are invoked only by Zend_Db_Table.

Cascading deletes and updates defined in your Zend_Db_Table classes are applied if you execute the `save()` or `delete()` methods on the Row class. However, if you update or delete data using another interface, such as a query tool or another application, the cascading operations are not applied. Even when using `update()` and `delete()` methods in the Zend_Db_Adapter class, cascading operations defined in your Zend_Db_Table classes are not executed.

No Cascading `INSERT`.

There is no support for a cascading `INSERT`. You must insert a row to a parent table in one operation, and insert row(s) to a dependent table in a separate operation.

Chapter 12. Zend_Debug

Dumping Variables

The static method `Zend_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 12.1. Example of dump() method

```
Zend_Debug::dump($var, $label=null, $echo=true);
```

The `$var` argument specifies the expression or variable about which the `Zend_Debug::dump()` method outputs information.

The `$label` argument is a string to be prepended to the output of `Zend_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 `$echo` argument specifies whether the output of `Zend_Debug::dump()` is echoed or not. If `true`, the output is echoed. Regardless of the value of the `$echo` argument, the return value of this method contains the output.

It may be helpful to understand that internally, `Zend_Debug::dump()` method wraps the PHP function `var_dump()` [http://php.net/var_dump]. If the output stream is detected as a web presentation, the output of `var_dump()` is escaped using `htmlspecialchars()` [http://php.net/htmlspecialchars] and wrapped with (X)HTML `<pre>` tags.

Debugging with Zend_Log

Using `Zend_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 Zend_Log 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::dump()`.

Chapter 13. Zend_Dojo

Introduction

As of version 1.6.0, Zend Framework ships Dojo Toolkit [http://dojotoolkit.org] to support out-of-the-box rich internet application development. Integration points with Dojo include:

- JSON-RPC support
- dojo.data compatibility
- View helper to help setup the Dojo environment
- Dijit-specific Zend_View helpers
- Dijit-specific Zend_Form elements and decorators

The Dojo distribution itself may be found in the `externals/dojo/` directory of the Zend Framework distribution. This is a source distribution, which includes Dojo's full javascript source, unit tests, and build tools. You can symlink this into your javascript directory, copy it, or use the build tool to create your own custom build to include in your project. Alternately, you can use one of the Content Delivery Networks that offer Dojo (ZF supports both the official AOL CDN as well as the Google CDN).

Zend_Dojo_Data: dojo.data Envelopes

Dojo provides data abstraction for data-enabled widgets via its dojo.data component. This component provides the ability to attach a datastore, provide some metadata regarding the identity field and optionally a label field, and an API for querying, sorting, and retrieving records and sets of records from the datastore.

dojo.data is often used with XmlHttpRequest to pull dynamic data from the server. The primary mechanism for this is to extend the QueryReadStore to point at a URL and specify the query information; the server side then returns data in the following JSON format:

```
{
    identifier: '<name>',
    <label: '<label>',>
    items: [
        { name: '...', label: '...', someKey: '...' },
        ...
    ]
}
```

`Zend_Dojo_Data` provides a simple interface for building such structures programmatically, interacting with them, and serializing them to an array or JSON.

Zend_Dojo_Data Usage

At its simplest, dojo.data requires that you provide the name of the identifier field in each item, and a set of items (data). You can either pass these in via the constructor, or via mutators:

Example 13.1. Zend_Dojo_Data initialization via constructor

```
$data = new Zend_Dojo_Data('id', $items);
```

Example 13.2. Zend_Dojo_Data initialization via mutators

```
$data = new Zend_Dojo_Data();
$data->setIdentifier('id')
     ->addItems($items);
```

You can also add a single item at a time, or append items, using `addItem()` and `addItems()`.

Example 13.3. Appending data to Zend_Dojo_Data

```
$data = new Zend_Dojo_Data($identifier, $items);
$data->addItem($someItem);

$data->addItems($someMoreItems);
```

Always use an identifier!

Every dojo.data data store requires that the identifier column be provided as metadata, and `Zend_Dojo_Data` is no different. In fact, if you attempt to add items without an identifier, it will raise an exception.

Individual items may be one of the following:

- Associative arrays
- Objects implementing a `toArray()` method
- Any other objects (will serialize via get_object_vars())

You can attach collections of the above items via `addItems()` or `setItems()` (overwrites all previously set items); when doing so, you may pass a single argument:

- Arrays
- Objects implementing the `Traversable` interface (which includes the interfaces `Iterator` and `ArrayAccess`).

If you want to specify a field that will act as a label for the item, call `setLabel()`:

Example 13.4. Specifying a label field in Zend_Dojo_Data

```
$data->setLabel('name');
```

Finally, you can also load a `Zend_Dojo_Data` item from a dojo.data JSON array, using the `fromJson()` method.

Example 13.5. Populating Zend_Dojo_Data from JSON

```
$data->fromJson($json);
```

Advanced Use Cases

Besides acting as a serializable data container, `Zend_Dojo_Data` also provides the ability to manipulate and traverse the data in a variety of ways.

`Zend_Dojo_Data` implements the interfaces `ArrayAccess`, `Iterator`, and `Countable`. This means that you can use the data collection almost as if it were an array.

All items are referenced by the identifier field. Since identifiers must be unique, you can then use the values of this field to pull individual records. There are two ways to do this: via the `getItem()` method, or via array notation.

```
// Using getItem():
$item = $data->getItem('foo');

// Or use array notation:
$item = $data['foo'];
```

If you know the identifier, you can use it to retrieve an item, update it, delete it, create it, or test for it:

```
// Update or create an item:
$data['foo'] = array('title' => 'Foo', 'email' => 'foo@foo.com');

// Delete an item:
unset($data['foo']);

// Test for an item:
if (isset($data[foo])) {
}
```

You can loop over all items as well. Internally, all items are stored as arrays.

```
foreach ($data as $item) {
    echo $item['title'] . ': ' . $item['description'] . "\n";
}
```

Or even count to see how many items you have:

```
echo count($data), " items found!";
```

Finally, as the class implements `__toString()`, you can also cast it to JSON simply by echoing it or casting to string:

```
echo $data; // echo as JSON string

$json = (string) $data; // cast to string == cast to JSON
```

Available Methods

Besides the methods necessary for implementing the interfaces listed above, the following methods are available.

- `setItems($items)`: set multiple items at once, overwriting any items that were previously set in the object. `$items` should be an array or a `Traversable` object.
- `setItem($item, $id = null)`: set an individual item, optionally passing an explicit identifier. Overwrites the item if previously in the collection. Valid items include associative arrays, objects implementing `toArray()`, or any object with public properties.
- `addItem($item, $id = null)`: add an individual item, optionally passing an explicit identifier. Will raise an exception if the item already exists in the collection. Valid items include associative arrays, objects implementing `toArray()`, or any object with public properties.
- `addItems($items)`: add multiple items at once, appending them to any current items. Will raise an exception if any of the new items have an identifier matching an identifier already in the collection. `$items` should be an array or a `Traversable` object.
- `getItems()`: Retrieve all items as an array of arrays.
- `hasItem($id)`: determine whether an item with the given identifier exists in the collection.
- `getItem($id)`: retrieve an item with the given identifier from the collection; the item returned will be an associative array. If no item matches, a null value is returned.
- `removeItem($id)`: remove an item with the given identifier from the collection.
- `clearItems()`: remove all items from the collection.

- `setIdentifier($identifier)`: set the name of the field that represents the unique identifier for each item in the collection.
- `getIdentifier()`: retrieve the name of the identifier field.
- `setLabel($label)`: set the name of a field that should be used as a display label for an item.
- `getLabel()`: retrieve the label field name.
- `toArray()`: cast the object to an array. The array will contain minimally the keys 'identifier' and 'items', and the key 'label' if a label field has been set in the object.
- `toJson()`: cast the object to a JSON representation.

Dojo View Helpers

Zend Framework provides the following Dojo-specific view helpers:

- *dojo():* setup the Dojo environment for your page, including dojo configuration values, custom module paths, module require statements, theme stylesheets, whether or not to use the CDN, and more.

Example 13.6. Using Dojo View Helpers

To use Dojo view helpers, you will need to tell your view object where to find them. You can do this by calling `addHelperPath()`:

```
$view->addHelperPath('Zend/Dojo/View/Helper/', 'Zend_Dojo_View_Helper');
```

Alternately, you can use `Zend_Dojo`'s `enableView()` method to do the work for you:

```
Zend_Dojo::enableView($view);
```

dojo() View Helper

The `dojo()` view helper is intended to simplify setting up the Dojo environment, including the following responsibilities:

- Specifying either a CDN or a local path to a Dojo install.
- Specifying paths to custom Dojo modules.
- Specifying dojo.require statements.
- Specifying dijit stylesheet themes to use.
- Specifying dojo.addOnLoad() events.

The `dojo()` view helper implementation is an example of a placeholder implementation; the data set in it persists between view objects, and may be directly echo'd from your layout script.

Example 13.7. dojo() View Helper Usage Example

For this example, let's assume the developer will be using Dojo from a local path; will need to require several dijits; and will be utilizing the Tundra dijit theme.

On many pages, the developer may not utilize Dojo at all. So, we will focus on a view script where Dojo is needed, and then on the layout script, where we will setup some of the Dojo environment and then render it.

First, we need to tell our view object to use the Dojo view helper paths. This can be done in your bootstrap or an early-running plugin; simply grab your view object and execute the following:

```
$view->addHelperPath('Zend/Dojo/View/Helper/', 'Zend_Dojo_View_Helper');
```

Next, the view script. In this case, we're going to specify that we will be using a FilteringSelect -- which will consume a custom store based on QueryReadStore, which we'll call 'PairedStore' and store in our 'custom' module.

```
<? // setup data store for FilteringSelect ?>
<div dojoType="custom.PairedStore" jsId="stateStore"
    url="/data/autocomplete/type/state/format/ajax"
    requestMethod="get"></div>

<? // Input element: ?>
State: <input id="state" dojoType="dijit.form.FilteringSelect"
    store="stateStore" pageSize="5" />

<? // setup required dojo elements:
$this->dojo()->enable()
             ->setDjConfigOption('parseOnLoad', true)
             ->registerModulePath('../custom/')
             ->requireModule('dijit.form.FilteringSelect')
             ->requireModule('custom.PairedStore'); ?>
```

In our layout script, we'll then check to see if Dojo is enabled, and, if so, we'll do some more general configuration and assemble it:

```
<?= $this->doctype() ?>
<html>
<head>
    <?= $this->headTitle() ?>
    <?= $this->headMeta() ?>
    <?= $this->headLink() ?>
    <?= $this->headStyle() ?>
<? if ($this->dojo()->isEnabled()):
    $this->dojo()->setLocalPath('/js/dojo/dojo.js')
                 ->addStyleSheetModule('dijit.themes.tundra');
    echo $this->dojo();
```

```
?>
    <?= $this->headScript() ?>
</head>
<body class="tundra">
    <?= $this->layout()->content ?>
    <?= $this->inlineScript() ?>
</body>
</html>
```

At this point, you only need to ensure that your files are in the correct locations and that you've created the end point action for your FilteringSelect!

Programmatic and Declarative Usage of Dojo

Dojo allows both *declarative* and *programmatic* usage of many of its features. *Declarative* usage uses standard HTML elements with non-standard attributes that are parsed when the page is loaded. While this is a powerful and simple syntax to utilize, for many developers this can cause issues with page validation.

Programmatic usage allows the developer to decorate existing elements by pulling them by ID or CSS selectors and passing them to the appropriate object constructors in Dojo. Because no non-standard HTML attributes are used, pages continue to validate.

In practice, both use cases allow for graceful degradation when javascript is disabled or the various Dojo script resources are unreachable. To promote standards and document validation, Zend Framework uses programmatic usage by default; the various view helpers will generate javascript and push it to the `dojo()` view helper for inclusion when rendered.

Developers using this technique may also wish to explore the option of writing their own programmatic decoration of the page. One benefit would be the ability to specify handlers for dijit events.

To allow this, as well as the ability to use declarative syntax, there are a number of static methods available to set this behavior globally.

Example 13.8. Specifying Declarative and Programmatic Dojo Usage

To specify declarative usage, simply call the static `setUseDeclarative()` method:

```
Zend_Dojo_View_Helper_Dojo::setUseDeclarative();
```

If you decide instead to use programmatic usage, call the static `setUseProgrammatic()` method:

```
Zend_Dojo_View_Helper_Dojo::setUseProgrammatic();
```

Finally, if you want to create your own programmatic rules, you should specify programmatic usage, but pass in the value '-1'; in this situation, no javascript for decorating any dijits used will be created.

```
Zend_Dojo_View_Helper_Dojo::setUseProgrammatic(-1);
```

Themes

Dojo allows the creation of themes for its dijits (widgets). You may select one by passing in a module path:

```
$view->dojo()->addStylesheetModule('dijit.themes.tundra');
```

The module path is discovered by using the character '.' as a directory separator and using the last value in the list as the name of the CSS file in that theme directory to use; in the example above, Dojo will look for the theme in 'dijit/themes/tundra/tundra.css'.

When using a theme, it is important to remember to pass the theme class to, at the least, a container surrounding any dijits you are using; the most common use case is to pass it in the body:

```
<body class="tundra">
```

Using Layers (Custom Builds)

By default, when you use a dojo.require statement, dojo will make a request back to the server to grab the appropriate javascript file. If you have many dijits in place, this results in many requests to the server -- which is not optimal.

Dojo's answer to this is to provide the ability to create *custom builds*. Builds do several things:

- Groups required files into *layers*; a layer lumps all required files into a single JS file. (Hence the name of this section.)

- "Interns" non-javascript files used by dijits (typically, template files). These are also grouped in the same JS file as the layer.
- Passes the file through ShrinkSafe, which strips whitespace and comments, as well as shortens variable names.

Some files can not be layered, but the build process will create a special release directory with the layer file and all other files. This allows you to have a slimmed-down distribution customized for your site or application needs.

To use a layer, the `dojo()` view helper has a `addLayer()` method for adding paths to required layers:

```
$view->dojo()->addLayer('/js/foo/foo.js');
```

For more information on creating custom builds, please refer to the Dojo build documentation [http://dojotoolkit.org/book/dojo-book-0-9/part-4-meta-dojo/package-system-and-custom-builds].

Methods Available

The `dojo()` view helper always returns an instance of the dojo placeholder container. That container object has the following methods available:

- `setView(Zend_View_Interface $view)`: set a view instance in the container.
- `enable()`: explicitly enable Dojo integration.
- `disable()`: disable Dojo integration.
- `isEnabled()`: determine whether or not Dojo integration is enabled.
- `requireModule($module)`: setup a `dojo.require` statement.
- `getModules()`: determine what modules have been required.
- `registerModulePath($module, $path)`: register a custom Dojo module path.
- `getModulePaths()`: get list of registered module paths.
- `addLayer($path)`: add a layer (custom build) path to use.
- `getLayers()`: get a list of all registered layer paths (custom builds).
- `removeLayer($path)`: remove the layer that matches `$path` from the list of registered layers (custom builds).
- `setCdnBase($url)`: set the base URL for a CDN; typically, one of the `Zend_Dojo::CDN_BASE_AOL` or `Zend_Dojo::CDN_BASE_GOOGLE`, but it only needs to be the URL string prior to the version number.
- `getCdnBase()`: retrieve the base CDN url to utilize.
- `setCdnVersion($version = null)`: set which version of Dojo to utilize from the CDN.
- `getCdnVersion()`: retrieve what version of Dojo from the CDN will be used.

- `setCdnDojoPath($path)`: set the relative path to the dojo.js or dojo.xd.js file on a CDN; typically, one of the `Zend_Dojo::CDN_DOJO_PATH_AOL` or `Zend_Dojo::CDN_DOJO_PATH_GOOGLE`, but it only needs to be the path string following the version number.
- `getCdnDojoPath()`: retrieve the last path segment of the CDN url pointing to the dojo.js file.
- `useCdn()`: tell the container to utilize the CDN; implicitly enables integration.
- `setLocalPath($path)`: tell the container the path to a local Dojo install (should be a path relative to the server, and contain the dojo.js file itself); implicitly enables integration.
- `getLocalPath()`: determine what local path to Dojo is being used.
- `useLocalPath()`: is the integration utilizing a Dojo local path?
- `setDjConfig(array $config)`: set dojo/dijit configuration values (expects assoc array).
- `setDjConfigOption($option, $value)`: set a single dojo/dijit configuration value.
- `getDjConfig()`: get all dojo/dijit configuration values.
- `getDjConfigOption($option, $default = null)`: get a single dojo/dijit configuration value.
- `addStylesheetModule($module)`: add a stylesheet based on a module theme.
- `getStylesheetModules()`: get stylesheets registered as module themes.
- `addStylesheet($path)`: add a local stylesheet for use with Dojo.
- `getStylesheets()`: get local Dojo stylesheets.
- `addOnLoad($spec, $function = null)`: add a lambda for dojo.onLoad to call. If one argument is passed, it is assumed to be either a function name or a javascript closure. If two arguments are passed, the first is assumed to be the name of an object instance variable and the second either a method name in that object or a closure to utilize with that object.
- `getOnLoadActions()`: retrieve all dojo.onLoad actions registered with the container. This will be an array of arrays.
- `onLoadCaptureStart($obj = null)`: capture data to be used as a lambda for dojo.onLoad(). If $obj is provided, the captured JS code will be considered a closure to use with that Javascript object.
- `onLoadCaptureEnd($obj = null)`: finish capturing data for use with dojo.onLoad().
- `javascriptCaptureStart()`: capture arbitrary javascript to be included with Dojo JS (onLoad, require, etc. statements).
- `javascriptCaptureEnd()`: finish capturing javascript.
- `__toString()`: cast the container to a string; renders all HTML style and script elements.

Dijit-Specific View Helpers

To quote the Dojo manual, "Dijit is a widget system layered on top of dojo." Dijit includes a variety of layout and form widgets designed to provide accessibility features, localization, and standardized (and themeable) look-and-feel.

Zend Framework ships a variety of view helpers that allow you to render and utilize dijits within your view scripts. There are three basic types:

- *Layout Containers*: these are designed to be used within your view scripts or consumed by form decorators for forms, sub forms, and display groups. They wrap the various classes offerred in dijit.layout. Each dijit layout view helper expects the following arguments:

  - `$id`: the container name or DOM ID.
  - `$content`: the content to wrap in the layout container.
  - `$params` (optional): dijit-specific parameters. Basically, any non-HTML attribute that can be used to configure the dijit layout container.
  - `$attribs` (optional): any additional HTML attributes that should be used to render the container div. If the key 'id' is passed in this array, it will be used for the form element DOM id, and `$id` will be used for its name.

  If you pass no arguments to a dijit layout view helper, the helper itself will be returned. This allows you to capture content, which is often an easier way to pass content to the layout container. Examples of this functionality will be shown later in this section.

- *Form Dijit*: the dijit.form.Form dijit, while not completely necessary for use with dijit form elements, will ensure that if an attempt is made to submit a form that does not validate against client-side validations, submission will be halted and validation error messages raised. The form dijit view helper expects the following arguments:

  - `$id`: the container name or DOM ID.
  - `$attribs` (optional): any additional HTML attributes that should be used to render the container div.
  - `$content` (optional): the content to wrap in the form. If none is passed, an empty string will be used.

  The argument order varies from the other dijits in order to keep compatibility with the standard `form()` view helper.

- *Form Elements*: these are designed to be consumed with `Zend_Form`, but can be used standalone within view scripts as well. Each dijit element view helper expects the following arguments:

  - `$id`: the element name or DOM ID.
  - `$value` (optional): the current value of the element.
  - `$params` (optional): dijit-specific parameters. Basically, any non-HTML attribute that can be used to configure a dijit.

- `$attribs` (optional): any additional HTML attributes that should be used to render the dijit. If the key 'id' is passed in this array, it will be used for the form element DOM id, and `$id` will be used for its name.

Some elements require more arguments; these will be noted with the individual element helper descriptions.

In order to utilize these view helpers, you need to register the path to the dojo view helpers with your view object.

Example 13.9. Registering the Dojo View Helper Prefix Path

```
$view->addHelperPath('Zend/Dojo/View/Helper', 'Zend_Dojo_View_Helper');
```

Dijit Layout Elements

The dijit.layout family of elements are for creating custom, predictable layouts for your site. For any questions on general usage, read more about them in the Dojo manual [http://dojotoolkit.org/book/dojo-book-0-9/part-2-dijit/layout].

All dijit layout elements have the signature `string ($id = null, $content = '', array $params = array(), array $attribs = array())`. In all caess, if you pass no arguments, the helper object itself will be returned. This gives you access to the `captureStart()` and `captureEnd()` methods, which allow you to capture content instead of passing it to the layout container.

- *AccordionContainer*: dijit.layout.AccordionContainer. Stack all panes together vertically; clicking on a pane titlebar will expand and display that particular pane.

```
<?= $view->accordionContainer(
    'foo',
    $content,
    array(
        'duration' => 200,
    ),
    array(
        'style' => 'width: 200px; height: 300px;',
    ),
); ?>
```

- *AccordionPane*: dijit.layout.AccordionPane. For use within AccordionContainer.

```
<?= $view->accordionPane(
    'foo',
    $content,
    array(
        'title' => 'Pane Title',
    ),
    array(
        'style' => 'background-color: lightgray;',
    ),
); ?>
```

- *BorderContainer*: dijit.layout.BorderContainer. Achieve layouts with optionally resizable panes such as you might see in a traditional application.

```
<?= $view->borderContainer(
    'foo',
    $content,
    array(
        'design' => 'headline',
    ),
    array(
        'style' => 'width: 100%; height: 100%',
    ),
); ?>
```

- *ContentPane*: dijit.layout.ContentPane. Use inside any container except AccordionContainer.

```
<?= $view->contentPane(
    'foo',
    $content,
    array(
        'title'  => 'Pane Title',
        'region' => 'left',
    ),
    array(
        'style' => 'width: 120px; background-color: lightgray;',
    ),
); ?>
```

- *SplitContainer*: dijit.layout.SplitContainer. Allows resizable content panes; deprecated in Dojo in favor of BorderContainer.

```
<?= $view->splitContainer(
    'foo',
    $content,
    array(
        'orientation'  => 'horizontal',
        'sizerWidth'   => 7,
        'activeSizing' => true,
    ),
    array(
        'style' => 'width: 400px; height: 500px;',
    ),
); ?>
```

- *StackContainer*: dijit.layout.StackContainer. All panes within a StackContainer are placed in a stack; build buttons or functionality to reveal one at a time.

  ```
  <?= $view->stackContainer(
      'foo',
      $content,
      array(),
      array(
          'style' => 'width: 400px; height: 500px; border: 1px;',
      ),
  ); ?>
  ```

- *TabContainer*: dijit.layout.TabContainer. All panes within a TabContainer are placed in a stack, with tabs positioned on one side for switching between them.

  ```
  <?= $view->stackContainer(
      'foo',
      $content,
      array(),
      array(
          'style' => 'width: 400px; height: 500px; border: 1px;',
      ),
  ); ?>
  ```

The following capture methods are available for all layout containers:

- `captureStart($id, array $params = array(), array $attribs = array())`: begin capturing content to include in a container. `$params` refers to the dijit params to use with the container, while `$attribs` refer to any general HTML attributes to use.

  Containers may be nested when capturing, *so long as no ids are duplicated*.

- `captureEnd($id)`: finish capturing content to include in a container. `$id` should refer to an id previously used with a `captureStart()` call. Returns a string representing the container and its contents, just as if you'd simply passed content to the helper itself.

Example 13.10. BorderContainer layout dijit example

BorderContainers, particularly when coupled with the ability to capture content, are especially useful for achieving complex layout effects.

```
$view->borderContainer()->captureStart('masterLayout',
                                       array('design' => 'headline'));

echo $view->contentPane(
    'menuPane',
    'This is the menu pane',
    array('region' => 'top'),
    array('style' => 'background-color: darkblue;')
);

echo  $view->contentPane(
    'navPane',
    'This is the navigation pane',
    array('region' => 'left'),
    array('style' => 'width: 200px; background-color: lightblue;')
);

echo $view->contentPane(
    'mainPane',
    'This is the main content pane area',
    array('region' => 'center'),
    array('style' => 'background-color: white;')
);

echo $view->contentPane(
    'statusPane',
    'Status area',
    array('region' => 'bottom'),
    array('style' => 'background-color: lightgray;')
);

echo $view->borderContainer()->captureEnd('masterLayout');
```

Dijit Form Elements

Dojo's form validation and input dijits are in the dijit.form tree. For more information on general usage of these elements, as well as accepted parameters, please visit the dijit.form documentation [http://dojotoolkit.org/book/dojo-book-0-9/part-2-dijit/form-validation-specialized-input].

The following dijit form elements are available in Zend Framework. Except where noted, all have the signature `string ($id, $value = '', array $params = array(), array $attribs = array())`.

- *Button*: dijit.form.Button. Display a form button.

```
<?= $view->button(
    'foo',
    'Show Me!',
    array('iconClass' => 'myButtons'),
); ?>
```

- *CheckBox*: dijit.form.CheckBox. Display a checkbox. Accepts an optional fifth argument, the array `$checkedOptions`, which may contain either:
  - an indexed array with two values, a checked value and unchecked value, in that order; or
  - an associative array with the keys 'checkedValue' and 'unCheckedValue'.

  If `$checkedOptions` is not provided, 1 and 0 are assumed.

  ```
  <?= $view->checkBox(
      'foo',
      'bar',
      array(),
      array(),
      array('checkedValue' => 'foo', 'unCheckedValue' => 'bar')
  ); ?>
  ```

- *ComboBox*: dijit.layout.ComboBox. ComboBoxes are a hybrid between a select and a text box with autocompletion. The key difference is that you may type an option that is not in the list of available options, and it will still consider it valid input. It accepts an optional fifth argument, an associative array `$options`; if provided, ComboBox will be rendered as a `select`. Note also that the *label values* of the `$options` array will be returned in the form -- not the values themselves.

  Alternately, you may pass information regarding a dojo.data datastore to use with the element. If provided, the ComboBox will be rendered as a text `input`, and will pull its options via that datastore.

  To specify a datastore, provide one of the following `$params` key combinations:
  - The key 'store', with an array value; the array should contain the keys:
    - *store*: the name of the javascript variable representing the datastore (this could be the name you would like for it to use).
    - *type*: the datastore type to use; e.g., 'dojo.data.ItemFileReadStore'.
    - *params* (optional): an associative array of key/value pairs to use to configure the datastore. The 'url' param is a typical example.
  - The keys:
    - *store*: a string indicating the datastore name to use.

  - *storeType*: a string indicating the datastore dojo.data type to use (e.g., 'dojo.data.ItemFileReadStore').
  - *storeParams*: an associative array of key/value pairs with which to configure the datastore.

  ```
  // As a select element:
  echo $view->comboBox(
      'foo',
      'bar',
      array(
          'autocomplete' => false,
      ),
      array(),
      array(
          'foo' => 'Foo',
          'bar' => 'Bar',
          'baz' => 'Baz',
      )
  );

  // As a dojo.data-enabled element:
  echo $view->comboBox(
      'foo',
      'bar',
      array(
          'autocomplete' => false,
          'store'        => 'stateStore',
          'storeType'    => 'dojo.data.ItemFileReadStore',
          'storeParams'  => array('url' => '/js/states.json'),
      ),
  );
  ```

- *CurrencyTextBox*: dijit.form.CurrencyTextBox. Inherits from ValidationTextBox, and provides client-side validation of currency. It expects that the dijit parameter 'currency' will be provided with an appropriate 3-character currency code. You may also specify any dijit parameters valid for ValidationTextBox and TextBox.

  ```
  echo $view->currencyTextBox(
      'foo',
      '$25.00',
      array('currency' => 'USD'),
      array('maxlength' => 20)
  );
  ```

Issues with Builds

There are currently known issues with using CurrencyTextBox with build layers [http://trac.dojotoolkit.org/ticket/7183]. A known work-around is to ensure that your document's Content-Type http-equiv meta tag sets the character set to utf-8, which you can do by calling:

```
$view->headMeta()->appendHttpEquiv('Content-Type',
                                   'text/html; charset=utf-8');
```

This will mean, of course, that you will need to ensure that the `headMeta()` placeholder is echoed in your layout script.

- *DateTextBox*: dijit.form.DateTextBox. Inherits from ValidationTextBox, and provides both client-side validation of dates, as well as a dropdown calendar from which to select a date. You may specify any dijit parameters available to ValidationTextBox or TextBox.

  ```
  echo $view->dateTextBox(
      'foo',
      '2008-07-11',
      array('required' => true)
  );
  ```

- *FilteringSelect*: dijit.form.FilteringSelect. Similar to ComboBox, this is a select/text hybrid that can either render a provided list of options or those fetched via a dojo.data datastore. Unlike ComboBox, however, FilteringSelect does not allow typing in an option not in its list. Additionally, it operates like a standard select in that the option values, not the labels, are returned when the form is submitted.

  Please see the information above on ComboBox for examples and available options for defining datastores.

- *HorizontalSlider* and *VerticalSlider*: dijit.form.HorizontalSlider and dijit.form.VerticalSlider. Sliders allow are UI widgets for selecting numbers in a given range; these are horizontal and vertical variants.

  At their most basic, they require the dijit parameters 'minimum', 'maximum', and 'discreteValues'. These define the range of values. Other common options are:

  - 'intermediateChanges' can be set to indicate whether or not to fire onChange events while the handle is being dragged.
  - 'clickSelect' can be set to allow clicking a location on the slider to set the value.
  - 'pageIncrement' can specify the value by which to increase/decrease when pageUp and pageDown are used.
  - 'showButtons' can be set to allow displaying buttons on either end of the slider for manipulating the value.

  The Zend Framework implementation creates a hidden element to store the value of the slider.

You may optionally desire to show a rule or labels for the slider. To do so, you will assign one or more of the dijit params 'topDecoration' and/or 'bottomDecoration' (HorizontalSlider) or 'leftDecoration' and/or 'rightDecoration' (VerticalSlider). Each of these expects the following options:

- *container*: name of the container.
- *labels* (optional): an array of labels to utilize. Use empty strings on either end to provide labels for inner values only. Required when specifying one of the 'Labels' dijit variants.
- *dijit* (optional): one of HorizontalRule, HorizontalRuleLabels, VerticalRule, or VerticalRuleLabels, Defaults to one of the Rule dijits.
- *params* (optional): dijit params for configuring the Rule dijit in use. Parameters specific to these dijits include:
  - *container* (optional): array of parameters and attributes for the rule container.
  - *labels* (optional): array of parameters and attributes for the labels list container.
- *attribs* (optional): HTML attributes to use with the rules/labels. This should follow the `params` option format and be an associative array with the keys 'container' and 'labels'.

```
echo $view->horizontalSlider(
    'foo',
    1,
    array(
        'minimum'             => -10,
        'maximum'             => 10,
        'discreteValues'      => 11,
        'intermediateChanges' => true,
        'showButtons'         => true,
        'topDecoration'       => array(
            'container' => 'topContainer'
            'dijit'     => 'HorizontalRuleLabels',
            'labels'    => array(
                ' ',
                '20%',
                '40%',
                '60%',
                '80%',
                ' ',
            ),
            'params' => array(
                'container' => array(
                    'style' => 'height:1.2em; font-size=75%;color:gray;',
                ),
                'labels' => array(
                    'style' => 'height:1em; font-size=75%;color:gray;',
                ),
            ),
        ),
        'bottomDecoration'    => array(
            'container' => 'bottomContainer'
            'labels'    => array(
                '0%',
                '50%',
                '100%',
            ),
            'params' => array(
                'container' => array(
                    'style' => 'height:1.2em; font-size=75%;color:gray;',
                ),
                'labels' => array(
                    'style' => 'height:1em; font-size=75%;color:gray;',
                ),
            ),
        ),
    )
);
```

- *NumberSpinner*: dijit.form.NumberSpinner. Text box for numeric entry, with buttons for incrementing and decrementing.

  Expects either an associative array for the dijit parameter 'constraints', or simply the keys 'min', 'max', and 'places' (these would be the expected entries of the constraints parameter as well). 'places' can be used to indicate how much the number spinner will increment and decrement.

```
echo $view->numberSpinner(
    'foo',
    5,
    array(
        'min'    => -10,
        'max'    => 10,
        'places' => 2,
    ),
    array(
        'maxlenth' => 3,
    )
);
```

- *NumberTextBox*: dijit.form.NumberTextBox. NumberTextBox provides the ability to format and display number entries in a localized fashion, as well as validate numerical entries, optionally against given constraints.

```
echo $view->numberTextBox(
    'foo',
    5,
    array(
        'places' => 4,
        'type'   => 'percent',
    ),
    array(
        'maxlength' => 20,
    )
);
```

- *PasswordTextBox*: dijit.form.ValidationTextBox tied to a password input. PasswordTextBox provides the ability to create password input that adheres to the current dijit theme, as well as allow for client-side validation.

```
echo $view->passwordTextBox(
    'foo',
    '',
    array(
        'required' => true,
    ),
    array(
        'maxlength' => 20,
    )
);
```

- *RadioButton*: dijit.form.RadioButton. A set of options from which only one may be selected. This behaves in every way like a regular radio, but has a look-and-feel consistent with other dijits.

  RadioButton accepts an option fourth argument, `$options`, an associative array of value/label pairs used as the radio options. You may also pass these as the `$attribs` key `options`.

```
echo $view->radioButton(
    'foo',
    'bar',
    array(),
    array(),
    array(
        'foo' => 'Foo',
        'bar' => 'Bar',
        'baz' => 'Baz',
    )
);
```

- *SubmitButton*: a dijit.form.Button tied to a submit input element. See the Button view helper for more details; the key difference is that this button can submit a form.
- *Textarea*: dijit.form.Textarea. These act like normal textareas, except that instead of having a set number of rows, they expand as the user types. The width should be specified via a style setting.

```
echo $view->textarea(
    'foo',
    'Start writing here...',
    array(),
    array('style' => 'width: 300px;')
);
```

- *TextBox*: dijit.form.TextBox. This element is primarily present to provide a common look-and-feel between various dijit elements, and to provide base functionality for the other TextBox-derived classes (ValidationTextBox, NumberTextBox, CurrencyTextBox, DateTextBox, and TimeTextBox).

  Common dijit parameter flags include 'lowercase' (cast to lowercase), 'uppercase' (cast to UPPERCASE), 'propercase' (cast to Proper Case), and trim (trim leading and trailing whitespace); all accept boolean values. Additionally, you may specifiy the parameters 'size' and 'maxLength'.

```
echo $view->textBox(
    'foo',
    'some text',
    array(
        'trim'       => true,
        'propercase' => true,
        'maxLength'  => 20,
    ),
    array(
        'size' => 20,
    )
);
```

- *TimeTextBox*: dijit.form.TimeTextBox. Also in the TextBox family, TimeTextBox provides a scrollable drop down selection of times from which a user may select. Dijit parameters allow you to specify the time increments available in the select as well as the visible range of times available.

```
echo $view->timeTextBox(
    'foo',
    '',
    array(
        'am.pm'            => true,
        'visibleIncrement' => 'T00:05:00', // 5-minute increments
        'visibleRange'     => 'T02:00:00', // show 2 hours of increments
    ),
    array(
        'size' => 20,
    )
);
```

- *ValidationTextBox*: dijit.form.ValidateTextBox. Provide client-side validations for a text element. Inherits from TextBox.

  Common dijit parameters include:

  - *invalidMessage*: a message to display when an invalid entry is detected.
  - *promptMessage*: a tooltip help message to use.
  - *regExp*: a regular expression to use to validate the text. Regular expression does not require boundary markers.
  - *required*: whether or not the element is required. If so, and the element is embedded in a dijit.form.Form, it will be flagged as invalid and prevent submission.

```
echo $view->validationTextBox(
    'foo',
    '',
    array(
        'required' => true,
        'regExp'   => '[\w]+',
        'invalidMessage' => 'No spaces or non-word characters allowed',
        'promptMessage'  => 'Single word consisting of alphanumeric ' .
                            'characters and underscores only',
    ),
    array(
        'maxlength' => 20,
    )
);
```

Dojo Form Elements and Decorators

Building on the dijit view helpers, the `Zend_Dojo_Form` family of classes provides the ability to utilize Dijits natively within your forms.

There are three options for utilizing the Dojo form elements with your forms:

- Use `Zend_Dojo::enableForm()`. This will add plugin paths for decorators and elements to all attached form items, recursively. Additionally, it will dojo-enable the view object. Note, however, that any sub forms you attach *after* this call will also need to be passed through `Zend_Dojo::enableForm()`.
- Use the Dojo-specific form and subform implementations, `Zend_Dojo_Form` and `Zend_Dojo_Form_SubForm` respectively. These can be used as drop-in replacements for `Zend_Form` and `Zend_Form_SubForm`, contain all the appropriate decorator and element paths, set a Dojo-specific default DisplayGroup class, and dojo-enable the view.
- Last, and most tedious, you can set the appropriate decorator and element paths yourself, set the default DisplayGroup class, and dojo-enable the view. Since `Zend_Dojo::enableForm()` does this already, there's little reason to go this route.

Example 13.11. Enabling Dojo in your existing forms

"But wait," you say; "I'm already extending Zend_Form with my own custom form class! How can I Dojo-enable it?'"

First, and easiest, simply change from extending `Zend_Form` to extending `Zend_Dojo_Form`, and update any places where you instantiate `Zend_Form_SubForm` to instantiate `Zend_Dojo_Form_SubForm`.

A second approach is to call `Zend_Dojo::enableForm()` within your custom form's `init()` method; when the form definition is complete, loop through all SubForms to dojo-enable them:

```
class My_Form_Custom extends Zend_Form
{
    public function init()
    {
        // Dojo-enable the form:
        Zend_Dojo::enableForm($this);

        // ... continue form definition from here

        // Dojo-enable all sub forms:
        foreach ($this->getSubForms() as $subForm) {
            Zend_Dojo::enableForm($subForm);
        }
    }
}
```

Usage of the dijit-specific form decorators and elements is just like using any other form decorator or element.

Dijit-Specific Form Decorators

Most form elements can use the DijitElement decorator, which will grab the dijit parameters from the elements, and pass these and other metadata to the view helper specified by the element. For decorating forms, sub forms, and display groups, however, there are a set of decorators corresponding to the various layout dijits.

All dijit decorators look for the `dijitParams` property of the given element being decorated, and push them as the `$params` array to the dijit view helper being used; these are then separated from any other properties so that no duplication of information occurs.

DijitElement Decorator

Just like the ViewHelper decorator, DijitElement expects a `helper` property in the element which it will then use as the view helper when rendering. Dijit parameters will typically be pulled directly from the element, but may also be passed in as options via the `dijitParams` key (the value of that key should be an associative array of options).

It is important that each element have a unique ID (as fetched from the element's `getId()` method). If duplicates are detected within the `dojo()` view helper, the decorator will trigger a notice, but then create a unique ID by appending the return of `uniqid()` to the identifier.

Standard usage is to simply associate this decorator as the first in your decorator chain, with no additional options.

Example 13.12. DijitElement Decorator Usage

```
$element->setDecorators(array(
    'DijitElement',
    'Errors',
    'Label',
    'ContentPane',
));
```

DijitForm Decorator

The DijitForm decorator is very similar to the Form decorator; in fact, it can be used basically interchangeably with it, as it utilizes the same view helper name ('form').

Since dijit.form.Form does not require any dijit parameters for configuration, the main difference is that the dijit form view helper require that a DOM ID is passed to ensure that programmatic dijit creation can work. The decorator ensures this, by passing the form name as the identifier.

DijitContainer-based Decorators

The `DijitContainer` decorator is actually an abstract class from which a variety of other decorators derive. It offers the same functionality of DijitElement, with the addition of title support. Many layout dijits require or can utilize a title; DijitContainer will utilize the element's legend property, if available, and can also utilize either the 'legend' or 'title' decorator option, if passed. The title will be translated if a translation adapter with a corresponding translation is present.

The following is a list of decorators that inherit from `DijitContainer`:

- AccordionContainer
- AccordionPane
- BorderContainer
- ContentPane
- SplitContainer
- StackContainer
- TabContainer

Example 13.13. DijitContainer Decorator Usage

```
// Use a TabContainer for your form:
$form->setDecorators(array(
    'FormElements',
    array('TabContainer', array(
        'id'          => 'tabContainer',
        'style'       => 'width: 600px; height: 500px;',
        'dijitParams' => array(
            'tabPosition' => 'top'
        ),
    )),
    'DijitForm',
));

// Use a ContentPane in your sub form (which can be used with all but
// AccordionContainer):
$subForm->setDecorators(array(
    'FormElements',
    array('HtmlTag', array('tag' => 'dl')),
    'ContentPane',
));
```

Dijit-Specific Form Elements

Each form dijit for which a view helper is provided has a corresponding `Zend_Form` element. All of them have the following methods available for manipulating dijit parameters:

- `setDijitParam($key, $value)`: set a single dijit parameter. If the dijit parameter already exists, it will be overwritten.
- `setDijitParams(array $params)`: set several dijit parameters at once. Any passed parameters matching those already present will overwrite.
- `hasDijitParam($key)`: whether or not a given dijit parameter is defined and present.
- `getDijitParam($key)`: retrieve the given dijit parameter. If not available, a null value is returned.
- `getDijitParams()`: retrieve all dijit parameters.
- `removeDijitParam($key)`: remove the given dijit parameter.
- `clearDijitParams()`: clear all currently defined dijit parameters.

Dijit parameters are stored in the `dijitParams` public property. Thus, you can dijit-enable an existing form element simply by setting this property on the element; you simply will not have the above accessors to facilitate manipulating the parameters.

Additionally, dijit-specific elements implement a different list of decorators, corresponding to the following:

```
$element->addDecorator('DijitElement')
```

```
        ->addDecorator('Errors')
        ->addDecorator('HtmlTag', array('tag' => 'dd'))
        ->addDecorator('Label', array('tag' => 'dt'));
```

In effect, the DijitElement decorator is used in place of the standard ViewHelper decorator.

Finally, the base Dijit element ensures that the Dojo view helper path is set on the view.

A variant on DijitElement, DijitMulti, provides the functionality of the `Multi` abstract form element, allowing the developer to specify 'multiOptions' -- typically select options or radio options.

The following dijit elements are shipped in the standard Zend Framework distribution.

Button

While not deriving from the standard Button element, it does implement the same functionality, and can be used as a drop-in replacement for it. The following functionality is exposed:

- `getLabel()` will utilize the element name as the button label if no name is provided. Additionally, it will translate the name if a translation adapter with a matching translation message is available.
- `isChecked()` determines if the value submitted matches the label; if so, it returns true. This is useful for determining which button was used when a form was submitted.

Additionally, only the decorators `DijitElement` and `DtDdWrapper` are utilized for Button elements.

Example 13.14. Example Button dijit element usage

```
$form->addElement(
    'Button',
    'foo',
    array(
        'label' => 'Button Label',
    )
);
```

CheckBox

While not deriving from the standard Checkbox element, it does implement the same functionality. This means that the following methods are exposed:

- `setCheckedValue($value)`: set the value to use when the element is checked.
- `getCheckedValue()`: get the value of the item to use when checked.
- `setUncheckedValue($value)`: set the value of the item to use when it is unchecked.
- `getUncheckedValue()`: get the value of the item to use when it is unchecked.
- `setChecked($flag)`: mark the element as checked or unchecked.

- `isChecked()`: determine if the element is currently checked.

Example 13.15. Example CheckBox dijit element usage

```
$form->addElement(
    'CheckBox',
    'foo',
    array(
        'label'          => 'A check box',
        'checkedValue'   => 'foo',
        'uncheckedValue' => 'bar',
        'checked'        => true,
    )
);
```

ComboBox and FilteringSelect

As noted in the ComboBox dijit view helper documentation, ComboBoxes are a hybrid between select and text input, allowing for autocompletion and the ability to specify an alternate to the options provided. FilteringSelects are the same, but do not allow arbitrary input.

ComboBoxes return the label values

ComboBoxes return the label values, and not the option values, which can lead to a disconnect in expectations. For this reason, ComboBoxes do not auto-register an `InArray` validator (though FilteringSelects do).

The ComboBox and FilteringSelect form elements provide accessors and mutators for examining and setting the select options as well as specifying a dojo.data datastore (if used). They extend from DijitMulti, which allows you to specify select options via the `setMultiOptions()` and `setMultiOption()` methods. In addition, the following methods are available:

- `getStoreInfo()`: get all datastore information currently set. Returns an empty array if no data is currently set.
- `setStoreId($identifier)`: set the store identifier variable (usually referred to by the attribute 'jsId' in Dojo). This should be a valid javascript variable name.
- `getStoreId()`: retrieve the store identifier variable name.
- `setStoreType($dojoType)`: set the datastore class to use; e.g., "dojo.data.ItemFileReadStore".
- `getStoreType()`: get the dojo datastore class to use.
- `setStoreParams(array $params)`: set any parameters used to configure the datastore object. As an example, dojo.data.ItemFileReadStore datastore would expect a 'url' parameter pointing to a location that would return the dojo.data object.
- `getStoreParams()`: get any datastore parameters currently set; if none, an empty array is returned.
- `setAutocomplete($flag)`: indicate whether or not the selected item will be used when the user leaves the element.

- `getAutocomplete()`: get the value of the autocomplete flag.

By default, if no dojo.data store is registered with the element, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a false value to the `registerInArrayValidator` configuration key.

Example 13.16. ComboBox dijit element usage as select input

```
$form->addElement(
    'ComboBox',
    'foo',
    array(
        'label'        => 'ComboBox (select)',
        'value'        => 'blue',
        'autocomplete' => false,
        'multiOptions' => array(
            'red'    => 'Rouge',
            'blue'   => 'Bleu',
            'white'  => 'Blanc',
            'orange' => 'Orange',
            'black'  => 'Noir',
            'green'  => 'Vert',
        ),
    )
);
```

Example 13.17. ComboBox dijit element usage with datastore

```
$form->addElement(
    'ComboBox',
    'foo',
    array(
        'label'       => 'ComboBox (datastore)',
        'storeId'     => 'stateStore',
        'storeType'   => 'dojo.data.ItemFileReadStore',
        'storeParams' => array(
            'url' => '/js/states.txt',
        ),
        'dijitParams' => array(
            'searchAttr' => 'name',
        ),
    )
);
```

The above examples could also utilize `FilteringSelect` instead of `ComboBox`.

CurrencyTextBox

The CurrencyTextBox is primarily for supporting currency input. The currency may be localized, and can support both fractional and non-fractional values.

Internally, CurrencyTextBox derives from NumberTextBox, ValidationTextBox, and TextBox; all methods available to those classes are available. In addition, the following constraint methods can be used:

- `setCurrency($currency)`: set the currency type to use; should follow the ISO-4217 [http://en.wikipedia.org/wiki/ISO_4217] specification.
- `getCurrency()`: retrieve the current currency type.
- `setSymbol($symbol)`: set the 3-letter ISO-4217 [http://en.wikipedia.org/wiki/ISO_4217] currency symbol to use.
- `getSymbol()`: get the current currency symbol.
- `setFractional($flag)`: set whether or not the currency should allow for fractional values.
- `getFractional()`: retrieve the status of the fractional flag.

Example 13.18. Example CurrencyTextBox dijit element usage

```
$form->addElement(
    'CurrencyTextBox',
    'foo',
    array(
        'label'          => 'Currency:',
        'required'       => true,
        'currency'       => 'USD',
        'invalidMessage' => 'Invalid amount. ' .
                            'Include dollar sign, commas, and cents.',
        'fractional'     => false,
    )
);
```

DateTextBox

DateTextBox provides a calendar drop-down for selecting a date, as well as client-side date validation and formatting.

Internally, DateTextBox derives from ValidationTextBox and TextBox; all methods available to those classes are available. In addition, the following methods can be used to set individual constraints:

- `setAmPm($flag)` and `getAmPm()`: Whether or not to use AM/PM strings in times.
- `setStrict($flag)` and `getStrict()`: whether or not to use strict regular expression matching when validating input. If false, which is the default, it will be lenient about whitespace and some abbreviations.

- `setLocale($locale)` and `getLocale()`: Set and retrieve the locale to use with this specific element.
- `setDatePattern($pattern)` and `getDatePattern()`: provide and retrieve the unicode date format pattern [http://www.unicode.org/reports/tr35/#Date_Format_Patterns] for formatting the date.
- `setFormatLength($formatLength)` and `getFormatLength()`: provide and retrieve the format length type to use; should be one of "long", "short", "medium" or "full".
- `setSelector($selector)` and `getSelector()`: provide and retrieve the style of selector; should be either "date" or "time".

Example 13.19. Example DateTextBox dijit element usage

```
$form->addElement(
    'DateTextBox',
    'foo',
    array(
        'label'          => 'Date:',
        'required'       => true,
        'invalidMessage' => 'Invalid date specified.',
        'formatLength'   => 'long',
    )
);
```

HorizontalSlider

HorizontalSlider provides a slider UI widget for selecting a numeric value in a range. Internally, it sets the value of a hidden element which is submitted by the form.

HorizontalSlider derives from the abstract Slider dijit element. Additionally, it has a variety of methods for setting and configuring slider rules and rule labels.

- `setTopDecorationDijit($dijit)` and `setBottomDecorationDijit($dijit)`: set the name of the dijit to use for either the top or bottom of the slider. This should not include the "dijit.form." prefix, but rather only the final name -- one of "HorizontalRule" or "HorizontalRuleLabels".
- `setTopDecorationContainer($container)` and `setBottomDecorationContainer($container)`: specify the name to use for the container element of the rules; e.g. 'topRule', 'topContainer', etc.
- `setTopDecorationLabels(array $labels)` and `setBottomDecorationLabels(array $labels)`: set the labels to use for one of the RuleLabels dijit types. These should be an indexed array; specify a single empty space to skip a given label position (such as the beginning or end).
- `setTopDecorationParams(array $params)` and `setBottomDecorationParams(array $params)`: dijit parameters to use when configuring the given Rule or RuleLabels dijit.
- `setTopDecorationAttribs(array $attribs)` and `setBottomDecorationAttribs(array $attribs)`: HTML attributes to specify for the given Rule or RuleLabels HTML element container.

- `getTopDecoration()` and `getBottomDecoration()`: retrieve all metadata for a given Rule or RuleLabels definition, as provided by the above mutators.

Example 13.20. Example HorizontalSlider dijit element usage

The following will create a horizontal slider selection with integer values ranging from -10 to 10. The top will have labels at the 20%, 40%, 60%, and 80% marks. The botton will have rules at 0, 50%, and 100%. Each time the value is changed, the hidden element storing the value will be updated.

```
$form->addElement(
    'HorizontalSlider',
    'horizontal',
    array(
        'label'                     => 'HorizontalSlider',
        'value'                     => 5,
        'minimum'                   => -10,
        'maximum'                   => 10,
        'discreteValues'            => 11,
        'intermediateChanges'       => true,
        'showButtons'               => true,
        'topDecorationDijit'        => 'HorizontalRuleLabels',
        'topDecorationContainer'    => 'topContainer',
        'topDecorationLabels'       => array(
                ' ',
                '20%',
                '40%',
                '60%',
                '80%',
                ' ',
        ),
        'topDecorationParams'      => array(
            'container' => array(
                'style' => 'height:1.2em; font-size=75%;color:gray;',
            ),
            'list' => array(
                'style' => 'height:1em; font-size=75%;color:gray;',
            ),
        ),
        'bottomDecorationDijit'     => 'HorizontalRule',
        'bottomDecorationContainer' => 'bottomContainer',
        'bottomDecorationLabels'    => array(
                '0%',
                '50%',
                '100%',
        ),
        'bottomDecorationParams'   => array(
            'list' => array(
                'style' => 'height:1em; font-size=75%;color:gray;',
            ),
        ),
    )
);
```

NumberSpinner

A number spinner is a text element for entering numeric values; it also includes elements for incrementing and decrementing the value by a set amount.

The following methods are available:

- `setDefaultTimeout($timeout)` and `getDefaultTimeout()`: set and retrieve the default timeout, in milliseconds, between when the button is held pressed and the value is changed.
- `setTimeoutChangeRate($rate)` and `getTimeoutChangeRate()`: set and retrieve the rate, in milliseconds, at which changes will be made when a button is held pressed.
- `setLargeDelta($delta)` and `getLargeDelta()`: set and retrieve the amount by which the numeric value should change when a button is held pressed.
- `setSmallDelta($delta)` and `getSmallDelta()`: set and retrieve the delta by which the number should change when a button is pressed once.
- `setIntermediateChanges($flag)` and `getIntermediateChanges()`: set and retrieve the flag indicating whether or not each value change should be shown when a button is held pressed.
- `setRangeMessage($message)` and `getRangeMessage()`: set and retrieve the message indicating the range of values available.
- `setMin($value)` and `getMin()`: set and retrieve the minimum value possible.
- `setMax($value)` and `getMax()`: set and retrieve the maximum value possible.

Example 13.21. Example NumberSpinner dijit element usage

```
$form->addElement(
    'NumberSpinner',
    'foo',
    array(
        'value'             => '7',
        'label'             => 'NumberSpinner',
        'smallDelta'        => 5,
        'largeDelta'        => 25,
        'defaultTimeout'    => 500,
        'timeoutChangeRate' => 100,
        'min'               => 9,
        'max'               => 1550,
        'places'            => 0,
        'maxlength'         => 20,
    )
);
```

NumberTextBox

A number text box is a text element for entering numeric values; unlike NumberSpinner, numbers are entered manually. Validations and constraints can be provided to ensure the number stays in a particular range or format.

Internally, NumberTextBox derives from ValidationTextBox and TextBox; all methods available to those classes are available. In addition, the following methods can be used to set individual constraints:

- `setLocale($locale)` and `getLocale()`: specify and retrieve a specific or alternate locale to use with this dijit.
- `setPattern($pattern)` and `getPattern()`: set and retrieve a number pattern format [http://www.unicode.org/reports/tr35/#Number_Format_Patterns] to use to format the number.
- `setType($type)` and `getType()`: set and retrieve the numeric format type to use (should be one of 'decimal', 'percent', or 'currency').
- `setPlaces($places)` and `getPlaces()`: set and retrieve the number of decimal places to support.
- `setStrict($flag)` and `getStrict()`: set and retrieve the value of the strict flag, which indicates how much leniency is allowed in relation to whitespace and non-numeric characters.

Example 13.22. Example NumberTextBox dijit element usage

```
$form->addElement(
    'NumberTextBox',
    'elevation',
    array(
        'label'          => 'NumberTextBox',
        'required'       => true,
        'invalidMessage' => 'Invalid elevation.',
        'places'         => 0,
        'constraints'    => array(
            'min'    => -20000,
            'max'    => 20000,
        ),
    )
);
```

PasswordTextBox

PasswordTextBox is simply a ValidationTextBox that is tied to a password input; its sole purpose is to allow for a dijit-themed text entry for passwords that also provides client-side validation.

Internally, NumberTextBox derives from ValidationTextBox and TextBox; all methods available to those classes are available.

Example 13.23. Example PasswordTextBox dijit element usage

```
$form->addElement(
    'PasswordTextBox',
    'password',
    array(
        'label'          => 'Password',
        'required'       => true,
        'trim'           => true,
        'lowercase'      => true,
        'regExp'         => '^[a-z0-9]{6,}$',
        'invalidMessage' => 'Invalid password; ' .
                            'must be at least 6 alphanumeric characters',
    )
);
```

RadioButton

RadioButton wraps standard radio input elements to provide a consistent look and feel with other dojo dijits.

RadioButton extends from DijitMulti, which allows you to specify select options via the `setMultiOptions()` and `setMultiOption()` methods.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a false value to the `registerInArrayValidator` configuration key.

Example 13.24. Example RadioButton dijit element usage

```
$form->addElement(
    'RadioButton',
    'foo',
    array(
        'label' => 'RadioButton',
        'multiOptions'  => array(
            'foo' => 'Foo',
            'bar' => 'Bar',
            'baz' => 'Baz',
        ),
        'value' => 'bar',
    )
);
```

Slider abstract element

Slider is an abstract element from which HorizontalSlider and VerticalSlider both derive. It exposes a number of common methods for configuring your sliders, including:

- `setClickSelect($flag)` and `getClickSelect()`: set and retrieve the flag indicating whether or not clicking the slider changes the value.
- `setIntermediateChanges($flag)` and `getIntermediateChanges()`: set and retrieve the flag indicating whether or not the dijit will send a notification on each slider change event.
- `setShowButtons($flag)` and `getShowButtons()`: set and retrieve the flag indicating whether or not buttons on either end will be displayed; if so, the user can click on these to change the value of the slider.
- `setDiscreteValues($value)` and `getDiscreteValues()`: set and retrieve the number of discrete values represented by the slider.
- `setMaximum($value)` and `getMaximum()`: set the maximum value of the slider.
- `setMinimum($value)` and `getMinimum()`: set the minimum value of the slider.
- `setPageIncrement($value)` and `getPageIncrement()`: set the amount by which the slider will change on keyboard events.

Example usage is provided with each concrete extending class.

SubmitButton

While there is no Dijit named SubmitButton, we include one here to provide a button dijit capable of submitting a form without requiring any additional javascript bindings. It works exactly like the Button dijit.

Example 13.25. Example SubmitButton dijit element usage

```
$form->addElement(
    'SubmitButton',
    'foo',
    array(
        'required'   => false,
        'ignore'     => true,
        'label'      => 'Submit Button!',
    )
);
```

TextBox

TextBox is included primarily to provide a text input with consistent look-and-feel to the other dijits. However, it also includes some minor filtering and validation capabilities, represented in the following methods:

- `setLowercase($flag)` and `getLowercase()`: set and retrieve the flag indicating whether or not input should be cast to lowercase.
- `setPropercase($flag)` and `getPropercase()`: set and retrieve the flag indicating whether or not the input should be cast to Proper Case.

- `setUppercase($flag)` and `getUppercase()`: set and retrieve the flag indicating whether or not the input should be cast to UPPERCASE.
- `setTrim($flag)` and `getTrim()`: set and retrieve the flag indicating whether or not leading or trailing whitespace should be stripped.
- `setMaxLength($length)` and `getMaxLength()`: set and retrieve the maximum length of input.

Example 13.26. Example TextBox dijit element usage

```
$form->addElement(
    'TextBox',
    'foo',
    array(
        'value'      => 'some text',
        'label'      => 'TextBox',
        'trim'       => true,
        'propercase' => true,
    )
);
```

Textarea

Textarea acts primarily like a standard HTML textarea. However, it does not support either the rows or cols settings. Instead, the textarea width should be specified using standard CSS measurements; rows should be omitted entirely. The textarea will then grow vertically as text is added to it.

Example 13.27. Example Textarea dijit element usage

```
$form->addElement(
    'Textarea',
    'textarea',
    array(
        'label'    => 'Textarea',
        'required' => true,
        'style'    => 'width: 200px;',
    )
);
```

TimeTextBox

TimeTextBox is a text input that provides a drop-down for selecting a time. The drop-down may be configured to show a certain window of time, with specified increments.

Internally, TimeTextBox derives from DateTextBox, ValidationTextBox and TextBox; all methods available to those classes are available. In addition, the following methods can be used to set individual constraints:

- `setTimePattern($pattern)` and `getTimePattern()`: set and retrieve the unicode time format pattern [http://www.unicode.org/reports/tr35/#Date_Format_Patterns] for formatting the time.
- `setClickableIncrement($format)` and `getClickableIncrement()`: set the ISO-8601 [http://en.wikipedia.org/wiki/ISO_8601] string representing the amount by which every clickable element in the time picker increases.
- `setVisibleIncrement($format)` and `getVisibleIncrement()`: set the increment visible in the time chooser; must follow ISO-8601 formats.
- `setVisibleRange($format)` and `getVisibleRange()`: set and retrieve the range of time visible in the time chooser at any given moment; must follow ISO-8601 formats.

Example 13.28. Example TimeTextBox dijit element usage

The following will create a TimeTextBox that displays 2 hours at a time, with increments of 10 minutes.

```
$form->addElement(
    'TimeTextBox',
    'foo',
    array(
        'label'              => 'TimeTextBox',
        'required'           => true,
        'visibleRange'       => 'T04:00:00',
        'visibleIncrement'   => 'T00:10:00',
        'clickableIncrement' => 'T00:10:00',
    )
);
```

ValidationTextBox

ValidationTextBox provides the ability to add validations and constraints to a text input. Internally, it derives from TextBox, and adds the following accessors and mutators for manipulating dijit parameters:

- `setInvalidMessage($message)` and `getInvalidMessage()`: set and retrieve the tooltip message to display when the value does not validate.
- `setPromptMessage($message)` and `getPromptMessage()`: set and retrieve the tooltip message to display for element usage.
- `setRegExp($regexp)` and `getRegExp()`: set and retrieve the regular expression to use for validating the element. The regular expression does not need boundaries (unlike PHP's preg* family of functions).
- `setConstraint($key, $value)` and `getConstraint($key)`: set and retrieve additional constraints to use when validating the element; used primarily with subclasses. Constraints are stored in the 'constraints' key of the dijit parameters.
- `setConstraints(array $constraints)` and `getConstraints()`: set and retrieve individual constraints to use when validating the element; used primarily with subclasses.
- `hasConstraint($key)`: test whether a given constraint exists.

- `removeConstraint($key)` and `clearConstraints()`: remove an individual or all constraints for the element.

Example 13.29. Example ValidationTextBox dijit element usage

The following will create a ValidationTextBox that requires a single string consisting solely of word characters (i.e., no spaces, most punctuation is invalid).

```
$form->addElement(
    'ValidationTextBox',
    'foo',
    array(
        'label'          => 'ValidationTextBox',
        'required'       => true,
        'regExp'         => '[\w]+',
        'invalidMessage' => 'Invalid non-space text.',
    )
);
```

VerticalSlider

VerticalSlider is the sibling of HorizontalSlider, and operates in every way like that element. The only real difference is that the 'top*' and 'bottom*' methods are replaced by 'left*' and 'right*', and instead of using HorizontalRule and HorizontalRuleLabels, VerticalRule and VerticalRuleLabels should be used.

Example 13.30. Example VerticalSlider dijit element usage

The following will create a vertical slider selection with integer values ranging from -10 to 10. The left will have labels at the 20%, 40%, 60%, and 80% marks. The right will have rules at 0, 50%, and 100%. Each time the value is changed, the hidden element storing the value will be updated.

```
$form->addElement(
    'VerticalSlider',
    'foo',
    array(
        'label'                     => 'VerticalSlider',
        'value'                     => 5,
        'style'                     => 'height: 200px; width: 3em;',
        'minimum'                   => -10,
        'maximum'                   => 10,
        'discreteValues'            => 11,
        'intermediateChanges'       => true,
        'showButtons'               => true,
        'leftDecorationDijit'       => 'VerticalRuleLabels',
        'leftDecorationContainer'  => 'leftContainer',
        'leftDecorationLabels'      => array(
                ' ',
                '20%',
                '40%',
                '60%',
                '80%',
                ' ',
        ),
        'rightDecorationDijit' => 'VerticalRule',
        'rightDecorationContainer' => 'rightContainer',
        'rightDecorationLabels' => array(
                '0%',
                '50%',
                '100%',
        ),
    )
);
```

Dojo Form Examples

Example 13.31. Using Zend_Dojo_Form

The easiest way to utilize Dojo with `Zend_Form` is to utilize `Zend_Dojo_Form`, either through direct usage or by extending it. This example shows extending Zend_Dojo_Form, and shows usage of all dijit elements. It creates four sub forms, and decorates the form to utilize a TabContainer, showing each sub form in its own tab.

```
class My_Form_Test extends Zend_Dojo_Form
{
    /**
     * Options to use with select elements
     */
    protected $_selectOptions = array(
        'red'    => 'Rouge',
        'blue'   => 'Bleu',
        'white'  => 'Blanc',
        'orange' => 'Orange',
        'black'  => 'Noir',
        'green'  => 'Vert',
    );

    /**
     * Form initialization
     *
     * @return void
     */
    public function init()
    {
        $this->setMethod('post');
        $this->setAttribs(array(
            'name'  => 'masterForm',
        ));
        $this->setDecorators(array(
            'FormElements',
            array('TabContainer', array(
                'id' => 'tabContainer',
                'style' => 'width: 600px; height: 500px;',
                'dijitParams' => array(
                    'tabPosition' => 'top'
                ),
            )),
            'DijitForm',
        ));
        $textForm = new Zend_Dojo_Form_SubForm();
        $textForm->setAttribs(array(
            'name'   => 'textboxtab',
            'legend' => 'Text Elements',
            'dijitParams' => array(
                'title' => 'Text Elements',
            ),
        ));
        $textForm->addElement(
                'TextBox',
                'textbox',
                array(
                    'value'      => 'some text',
                    'label'      => 'TextBox',
                    'trim'       => true,
                    'propercase' => true,
                )
            )
            ->addElement(
                'DateTextBox',
                'datebox',
                array(
                    'value' => '2008-07-05',
                    'label' => 'DateTextBox',
                    'required'  => true,
                )
            )
            ->addElement(
                'TimeTextBox',
                'timebox',
                array(
                    'label' => 'TimeTextBox',
                    'required'  => true,
                )
            )
            ->addElement(
                'CurrencyTextBox',
                'currencybox',
                array(
                    'label' => 'CurrencyTextBox',
                    'required'  => true,
                    // 'currency' => 'USD',
                    'invalidMessage' => 'Invalid amount. ' .
                                        'Include dollar sign, commas, ' .
                                        'and cents.',
                    // 'fractional' => true,
                    // 'symbol' => 'USD',
                    // 'type' => 'currency',
                )
            )
            ->addElement(
                'NumberTextBox',
                'numberbox',
                array(
                    'label' => 'NumberTextBox',
                    'required'  => true,
                    'invalidMessage' => 'Invalid elevation.',
                    'constraints' => array(
                        'min' => -20000,
                        'max' => 20000,
                        'places' => 0,
                    )
                )
            )
```

```
                ->addElement(
                    'ValidationTextBox',
                    'validationbox',
                    array(
                        'label' => 'ValidationTextBox',
                        'required'  => true,
                        'regExp' => '[\w]+',
                        'invalidMessage' => 'Invalid non-space text.',
                    )
                )
                ->addElement(
                    'Textarea',
                    'textarea',
                    array(
                        'label'    => 'Textarea',
                        'required' => true,
                        'style'    => 'width: 200px;',
                    )
                );
            $toggleForm = new Zend_Dojo_Form_SubForm();
            $toggleForm->setAttribs(array(
                'name'   => 'toggletab',
                'legend' => 'Toggle Elements',
            ));
            $toggleForm->addElement(
                    'NumberSpinner',
                    'ns',
                    array(
                        'value'             => '7',
                        'label'             => 'NumberSpinner',
                        'smallDelta'        => 5,
                        'largeDelta'        => 25,
                        'defaultTimeout'    => 1000,
                        'timeoutChangeRate' => 100,
                        'min'               => 9,
                        'max'               => 1550,
                        'places'            => 0,
                        'maxlength'         => 20,
                    )
                )
                ->addElement(
                    'Button',
                    'dijitButton',
                    array(
                        'label' => 'Button',
                    )
                )
                ->addElement(
                    'CheckBox',
                    'checkbox',
                    array(
                        'label' => 'CheckBox',
                        'checkedValue'  => 'foo',
                        'uncheckedValue'  => 'bar',
                        'checked' => true,
                    )
                )
                ->addElement(
                    'RadioButton',
                    'radiobutton',
                    array(
                        'label' => 'RadioButton',
                        'multiOptions'  => array(
                            'foo' => 'Foo',
                            'bar' => 'Bar',
                            'baz' => 'Baz',
                        ),
                        'value' => 'bar',
                    )
                );
            $selectForm = new Zend_Dojo_Form_SubForm();
            $selectForm->setAttribs(array(
                'name'   => 'selecttab',
                'legend' => 'Select Elements',
            ));
            $selectForm->addElement(
                    'ComboBox',
                    'comboboxselect',
                    array(
                        'label' => 'ComboBox (select)',
                        'value' => 'blue',
                        'autocomplete' => false,
                        'multiOptions' => $this->_selectOptions,
                    )
                )
                ->addElement(
                    'ComboBox',
                    'comboboxremote',
                    array(
                        'label' => 'ComboBox (remoter)',
                        'storeId' => 'stateStore',
                        'storeType' => 'dojo.data.ItemFileReadStore',
                        'storeParams' => array(
                            'url' => '/js/states.txt',
                        ),
                        'dijitParams' => array(
                            'searchAttr' => 'name',
                        ),
                    )
                )
                ->addElement(
                    'FilteringSelect',
                    'filterselect',
                    array(
                        'label' => 'FilteringSelect (select)',
                        'value' => 'blue',
                        'autocomplete' => false,
                        'multiOptions' => $this->_selectOptions,
                    )
                )
                ->addElement(
                    'FilteringSelect',
                    'filterselectremote',
                    array(
                        'label' => 'FilteringSelect (remoter)',
                        'storeId' => 'stateStore',
                        'storeType' => 'dojo.data.ItemFileReadStore',
                        'storeParams' => array(
                            'url' => '/js/states.txt',
                        ),
                        'dijitParams' => array(
                            'searchAttr' => 'name',
                        ),
                    )
                );
            $sliderForm = new Zend_Dojo_Form_SubForm();
            $sliderForm->setAttribs(array(
                'name'   => 'slidertab',
                'legend' => 'Slider Elements',
            ));
            $sliderForm->addElement(
                    'HorizontalSlider',
                    'horizontal',
                    array(
                        'label' => 'HorizontalSlider',
                        'value' => 5,
                        'minimum' => -10,
                        'maximum' => 10,
                        'discreteValues' => 11,
                        'intermediateChanges' => true,
                        'showButtons' => true,
                        'topDecorationDijit' => 'HorizontalRuleLabels',
                        'topDecorationContainer' => 'topContainer',
                        'topDecorationLabels' => array(
                                ' ',
                                '20%',
                                '40%',
                                '60%',
                                '80%',
                                ' ',
                        ),
                        'topDecorationParams' => array(
                            'container' => array(
                                'style' => 'height:1.2em; ' .
                                           'font-size=75%;color:gray;',
                            ),
                            'list' => array(
                                'style' => 'height:1em; ' .
                                           'font-size=75%;color:gray;',
                            ),
                        ),
                        'bottomDecorationDijit' => 'HorizontalRule',
                        'bottomDecorationContainer' => 'bottomContainer',
                        'bottomDecorationLabels' => array(
                                '0%',
                                '50%',
                                '100%',
                        ),
                        'bottomDecorationParams' => array(
                            'list' => array(
                                'style' => 'height:1em; ' .
                                           'font-size=75%;color:gray;',
                            ),
                        ),
                    )
                )
                ->addElement(
                    'VerticalSlider',
                    'vertical',
                    array(
                        'label' => 'VerticalSlider',
                        'value' => 5,
                        'style' => 'height: 200px; width: 3em;',
                        'minimum' => -10,
                        'maximum' => 10,
                        'discreteValues' => 11,
                        'intermediateChanges' => true,
                        'showButtons' => true,
                        'leftDecorationDijit' => 'VerticalRuleLabels',
                        'leftDecorationContainer' => 'leftContainer',
                        'leftDecorationLabels' => array(
                                ' ',
                                '20%',
                                '40%',
                                '60%',
                                '80%',
                                ' ',
                        ),
                        'rightDecorationDijit' => 'VerticalRule',
                        'rightDecorationContainer' => 'rightContainer',
                        'rightDecorationLabels' => array(
                                '0%',
                                '50%',
                                '100%',
                        ),
                    )
                );

            $this->addSubForm($textForm, 'textboxtab')
                 ->addSubForm($toggleForm, 'toggletab')
                 ->addSubForm($selectForm, 'selecttab')
                 ->addSubForm($sliderForm, 'slidertab');
        }
    }
```

Example 13.32. Modifying an existing form to utilize Dojo

Existing forms can be modified to utilize Dojo as well, by use of the `Zend_Dojo::enableForm()` static method.

This first example shows decorating an existing form instance:

```
$form = new My_Custom_Form();
Zend_Dojo::enableForm($form);
$form->addElement(
'ComboBox',
'query',
array(
    'label'        => 'Color:',
    'value'        => 'blue',
    'autocomplete' => false,
    'multiOptions' => array(
        'red'    => 'Rouge',
        'blue'   => 'Bleu',
        'white'  => 'Blanc',
        'orange' => 'Orange',
        'black'  => 'Noir',
        'green'  => 'Vert',
    ),
)
);
```

Alternately, you can make a slight tweak to your form initialization:

```
class My_Custom_Form extends Zend_Form
{
    public function init()
    {
        Zend_Dojo::enableForm($this);

        // ...
    }
}
```

Of course, if you can do that... you could and should simply alter the class to inherit from Zend_Dojo_Form, which is a drop-in replacement of Zend_Form that's already Dojo-enabled...

Chapter 14. Zend_Dom

Introduction

`Zend_Dom` 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

`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 $$() [http://prototypejs.org/api/utility/dollar-dollar] and Dojo's dojo.query [http://api.dojotoolkit.org/jsdoc/dojo/HEAD/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 `query()` or `queryXpath()` methods; each method will return a `Zend_Dom_Query_Result` 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 descendents*: utilize '>' between selectors to denote direct descendents. 'div > span' would select only 'span' elements that are direct descendents of a 'div'. Can also be used with any of the selectors above.

- *descendents*: string together multiple selectors to indicate a hierarchy along which to search. 'div .foo span #one' would select an element of id 'one' that is a descendent of arbitrary depth beneath a 'span' element, which is in turn a descendent of arbitrary depth beneath an element with a class of 'foo', that is an descendent of arbitrary depth beneath a 'div' element. For example, it would match the link to the word 'One' in the listing below:

```
<div>
<table>
    <tr>
        <td class="foo">
            <div>
                Lorem ipsum <span class="bar">
                    <a href="/foo/bar" id="one">One</a>
                    <a href="/foo/baz" id="two">Two</a>
                    <a href="/foo/bat" id="three">Three</a>
                    <a href="/foo/bla" id="four">Four</a>
                </span>
            </div>
        </td>
    </tr>
</table>
</div>
```

Once you've performed your query, you can then work with the result object to determine information about the nodes, as well as to pull them and/or their content directly for examination and manipulation. `Zend_Dom_Query_Result` implements `Countable` and `Iterator`, and store the results internally as DOMNodes/DOMElements. As an example, consider the following call, that selects against the HTML above:

```
$dom = new Zend_Dom_Query($html);
$results = $dom->query('.foo .bar a');

$count = count($results); // get number of matches: 4
foreach ($results as $result) {
    // $result is a DOMElement
}
```

`Zend_Dom_Query` also allows straight XPath queries utilizing the `queryXpath()` method; you can pass any valid XPath query to this method, and it will return a `Zend_Dom_Query_Result` 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($document)`: specify an XML string to query against.
- `setDocumentXhtml($document)`: specify an XHTML string to query against.
- `setDocumentHtml($document)`: specify an HTML string to query against.
- `setDocument($document)`: specify a string to query against; `Zend_Dom_Query` will then attempt to autodetect the document type.
- `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.
- `query($query)`: query the document using CSS selector notation.
- `queryXpath($xPathQuery)`: query the document using XPath notation.

Zend_Dom_Query_Result

As mentioned previously, `Zend_Dom_Query_Result` implements both `Iterator` and `Countable`, and as such can be used in a `foreach` loop as well as with the `count()` function. Additionally, it exposes the following methods:

- `getCssQuery()`: return the CSS selector query used to produce the result (if any).
- `getXpathQuery()`: return the XPath query used to produce the result. Internally, `Zend_Dom_Query` converts CSS selector queries to XPath, so this value will always be populated.
- `getDocument()`: retrieve the DOMDocument the selection was made against.

Chapter 15. Zend_Exception

Using Exceptions

All exceptions thrown by Zend Framework classes should throw an exception that derives from the base class Zend_Exception.

Example 15.1. Example of catching an exception

```
try {
    Zend_Loader::loadClass('nonexistantclass');
} catch (Zend_Exception $e) {
    echo "Caught exception: " . get_class($e) . "\n";
    echo "Message: " . $e->getMessage() . "\n";
    // other code to recover from the failure.
}
```

See the documentation for each respective Zend Framework component for more specific information on which methods throw exceptions, the circumstances for the exceptions, and which exception classes derive from Zend_Exception.

Chapter 16. Zend_Feed

Introduction

`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.

Programmatically, `Zend_Feed` consists of a base `Zend_Feed` class, abstract `Zend_Feed_Abstract` and `Zend_Feed_Entry_Abstract` base classes for representing Feeds and Entries, specific implementations of feeds and entries for RSS and Atom, and a behind-the-scenes helper for making the natural syntax magic work.

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.

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.

Example 16.1. Putting Zend_Feed to Work on RSS Feed Data

```
// Fetch the latest Slashdot headlines
try {
    $slashdotRss =
        Zend_Feed::import('http://rss.slashdot.org/Slashdot/slashdot');
} catch (Zend_Feed_Exception $e) {
    // feed import failed
    echo "Exception caught importing feed: {$e->getMessage()}\n";
    exit;
}

// Initialize the channel data array
$channel = array(
    'title'       => $slashdotRss->title(),
    'link'        => $slashdotRss->link(),
    'description' => $slashdotRss->description(),
    'items'       => array()
    );

// Loop over each channel item and store relevant data
foreach ($slashdotRss as $item) {
    $channel['items'][] = array(
        'title'       => $item->title(),
        'link'        => $item->link(),
        'description' => $item->description()
        );
}
```

Importing Feeds

`Zend_Feed` enables developers to retrieve feeds very easily. If you know the URI of a feed, simply use the `Zend_Feed::import()` method:

```
$feed = Zend_Feed::import('http://feeds.example.com/feedName');
```

You can also use `Zend_Feed` to fetch the contents of a feed from a file or the contents of a PHP string variable:

```
// importing a feed from a text file
$feedFromFile = Zend_Feed::importFile('feed.xml');

// importing a feed from a PHP string variable
$feedFromPHP = Zend_Feed::importString($feedString);
```

In each of the examples above, an object of a class that extends `Zend_Feed_Abstract` 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_Rss` object would be returned. On the other hand, if an Atom feed were imported, then a `Zend_Feed_Atom` object is returned. The import methods will also throw a `Zend_Feed_Exception` object upon failure, such as an unreadable or malformed feed.

Custom feeds

`Zend_Feed` enables developers to create custom feeds very easily. You just have to create an array and to import it with Zend_Feed. This array can be imported with `Zend_Feed::importArray()` or with `Zend_Feed::importBuilder()`. In this last case the array will be computed on the fly by a custom data source implementing `Zend_Feed_Builder_Interface`.

Importing a custom array

```
// importing a feed from an array
$atomFeedFromArray = Zend_Feed::importArray($array);

// the following line is equivalent to the above;
// by default a Zend_Feed_Atom instance is returned
$atomFeedFromArray = Zend_Feed::importArray($array, 'atom');

// importing a rss feed from an array
$rssFeedFromArray = Zend_Feed::importArray($array, 'rss');
```

The format of the array must conform to this structure:

```
array(
      'title'       => 'title of the feed', //required
      'link'        => 'canonical url to the feed', //required
      'lastUpdate'  => 'timestamp of the update date', // optional
      'published'   => 'timestamp of the publication date', //optional
      'charset'     => 'charset of the textual data', // required
      'description' => 'short description of the feed', //optional
      'author'      => 'author/publisher of the feed', //optional
      'email'       => 'email of the author', //optional
      'webmaster'   =>     // optional, ignored if atom is used
                       'email address for person responsible ' .
                       'for technical issues'
      'copyright'   => 'copyright notice', //optional
      'image'       => 'url to image', //optional
      'generator'   => 'generator', // optional
      'language'    => 'language the feed is written in', // optional
      'ttl'         =>     // optional, ignored if atom is used
                       'how long in minutes a feed can be cached ' .
                       'before refreshing',
      'rating'      =>     // optional, ignored if atom is used
                       'The PICS rating for the channel.',
      'cloud'       => array(
                             'domain'            => 'domain of the cloud, e.g. rpc
                             'port'              => 'port to connect to' // option
                             'path'              => 'path of the cloud, e.g. /RPC2
                             'registerProcedure' => 'procedure to call, e.g. myClo
                             'protocol'          => 'protocol to use, e.g. soap or
                             ), // a cloud to be notified of updates // optional,
      'textInput'   => array(
                             'title'       => 'the label of the Submit button in t
                             'description' => 'explains the text input area' // re
                             'name'        => 'the name of the text object in the
                             'link'        => 'the URL of the CGI script that proc
                             ) // a text input box that can be displayed with the
      'skipHours'   => array(
                             'hour in 24 format', // e.g 13 (1pm)
                             // up to 24 rows whose value is a number between 0 an
                             ) // Hint telling aggregators which hours they can sk
      'skipDays '   => array(
                             'a day to skip', // e.g Monday
                             // up to 7 rows whose value is a Monday, Tuesday, Wed
                             ) // Hint telling aggregators which days they can ski
      'itunes'      => array(
                             'author'       => 'Artist column' // optional, defaul
                             'owner'        => array(
                                                     'name' => 'name of the owner'
                                                     'email' => 'email of the owne
                                                     ) // Owner of the podcast //
                             'image'        => 'album/podcast art' // optional, de
                             'subtitle'     => 'short description' // optional, de
                             'summary'      => 'longer description' // optional, d
                             'block'        => 'Prevent an episode from appearing
                             'category'     => array(
                                                     array('main' => 'main categor
                                                           'sub'  => 'sub category
                                                           ),
                                                     // up to 3 rows
                                                     ) // 'Category column and in
                             'explicit'     => 'parental advisory graphic (yes|no|
                             'keywords'     => 'a comma separated list of 12 keywo
                             'new-feed-url' => 'used to inform iTunes of new feed
                             ) // Itunes extension data // optional, ignored if at
      'entries'     => array(
                             array(
                                   'title'        => 'title of the feed entry', //
                                   'link'         => 'url to a feed entry', //requ
                                   'description'  => 'short version of a feed entr
                                   'guid'         => 'id of the article, if not gi
                                   'content'      => 'long version', // can contai
                                   'lastUpdate'   => 'timestamp of the publication
                                   'comments'     => 'comments page of the feed en
                                   'commentRss'   => 'the feed url of the associat
                                   'source'       => array(
                                                           'title' => 'title of th
                                                           'url' => 'url of the or
                                                           ) // original source of
```

```
                            'category'     => array(
                                                    array(
                                                          'term' => 'first
                                                          'scheme' => 'url
                                                          ),
                                                    array(
                                                          //data for the se
                                                          )
                                                    ) // list of the attach
                            'enclosure'    => array(
                                                    array(
                                                          'url' => 'url of
                                                          'type' => 'mime t
                                                          'length' => 'leng
                                                          ),
                                                    array(
                                                          //data for the se
                                                          )
                                                    ) // list of the enclos
                            ),
                      array(
                            //data for the second entry and so on
                            )
                      )
);
```

References:

- RSS 2.0 specification: RSS 2.0 [http://blogs.law.harvard.edu/tech/rss]
- Atom specification: RFC 4287 [http://tools.ietf.org/html/rfc4287]
- WFW specification: Well Formed Web [http://wellformedweb.org/news/wfw_namespace_elements]
- iTunes specification: iTunes Technical Specifications [http://www.apple.com/itunes/store/podcaststechspecs.html]

Importing a custom data source

You can create a Zeed_Feed instance from any data source implementing `Zend_Feed_Builder_Interface`. You just have to implement the `getHeader()` and `getEntries()` methods to be able to use your object with `Zend_Feed::importBuilder()`. As a simple reference implementation, you can use `Zend_Feed_Builder`, which takes an array in its constructor, performs some minor validation, and then can be used in the `importBuilder()` method. The `getHeader()` method must return an instance of `Zend_Feed_Builder_Header`, and `getEntries()` must return an array of `Zend_Feed_Builder_Entry` instances.

Note

`Zend_Feed_Builder` serves as a concrete implementation to demonstrate the usage. Users are encouraged to make their own classes to implement `Zend_Feed_Builder_Interface`.

Here is an example of `Zend_Feed::importBuilder()` usage:

```
// importing a feed from a custom builder source
$atomFeedFromArray =
    Zend_Feed::importBuilder(new Zend_Feed_Builder($array));

// the following line is equivalent to the above;
// by default a Zend_Feed_Atom instance is returned
$atomFeedFromArray =
    Zend_Feed::importArray(new Zend_Feed_Builder($array), 'atom');

// importing a rss feed from a custom builder array
$rssFeedFromArray =
    Zend_Feed::importArray(new Zend_Feed_Builder($array), 'rss');
```

Dumping the contents of a feed

To dump the contents of a `Zend_Feed_Abstract` instance, you may use `send()` or `saveXml()` methods.

```
assert($feed instanceof Zend_Feed_Abstract);

// dump the feed to standard output
print $feed->saveXML();

// send http headers and dump the feed
$feed->send();
```

Retrieving Feeds from Web Pages

Web pages often contain `<link>` tags that refer to feeds with content relevant to the particular page. `Zend_Feed` enables you to retrieve all feeds referenced by a web page with one simple method call:

```
$feedArray = Zend_Feed::findFeeds('http://www.example.com/news.html');
```

Here the `findFeeds()` method returns an array of `Zend_Feed_Abstract` objects that are referenced by `<link>` tags on the news.html web page. Depending on the type of each feed, each respective entry in the `$feedArray` array may be a `Zend_Feed_Rss` or `Zend_Feed_Atom` instance. `Zend_Feed` will throw a `Zend_Feed_Exception` upon failure, such as an HTTP 404 response code or a malformed feed.

Consuming an RSS Feed

Reading an RSS feed is as simple as instantiating a `Zend_Feed_Rss` object with the URL of the feed:

```
$channel = new Zend_Feed_Rss('http://rss.example.com/channelName');
```

If any errors occur fetching the feed, a `Zend_Feed_Exception` will be thrown.

Once you have a feed object, you can access any of the standard RSS "channel" properties directly on the object:

```
echo $channel->title();
```

Note the function syntax. `Zend_Feed` uses a convention of treating properties as XML object if they are requested with variable "getter" syntax (`$obj->property`) and as strings if they are access with method syntax (`$obj->property()`). This enables access to the full text of any individual node while still allowing full access to all children.

If channel properties have attributes, they are accessible using PHP's array syntax:

```
echo $channel->category['domain'];
```

Since XML attributes cannot have children, method syntax is not necessary for accessing attribute values.

Most commonly you'll want to loop through the feed and do something with its entries. `Zend_Feed_Abstract` implements PHP's `Iterator` interface, so printing all titles of articles in a channel is just a matter of:

```
foreach ($channel as $item) {
    echo $item->title() . "\n";
}
```

If you are not familiar with RSS, here are the standard elements you can expect to be available in an RSS channel and in individual RSS items (entries).

Required channel elements:

- `title` - The name of the channel
- `link` - The URL of the web site corresponding to the channel
- `description` - A sentence or several describing the channel

Common optional channel elements:

- `pubDate` - The publication date of this set of content, in RFC 822 date format
- `language` - The language the channel is written in
- `category` - One or more (specified by multiple tags) categories the channel belongs to

RSS `<item>` elements do not have any strictly required elements. However, either `title` or `description` must be present.

Common item elements:

- `title` - The title of the item
- `link` - The URL of the item
- `description` - A synopsis of the item
- `author` - The author's email address
- `category` - One more more categories that the item belongs to
- `comments` - URL of comments relating to this item
- `pubDate` - The date the item was published, in RFC 822 date format

In your code you can always test to see if an element is non-empty with:

```
if ($item->propname()) {
    // ... proceed.
}
```

If you use `$item->propname` instead, you will always get an empty object which will evaluate to `TRUE`, so your check will fail.

For further information, the official RSS 2.0 specification is available at: http://blogs.law.harvard.edu/tech/rss

Consuming an Atom Feed

`Zend_Feed_Atom` is used in much the same way as `Zend_Feed_Rss`. It provides the same access to feed-level properties and iteration over entries in the feed. The main difference is in the structure of the Atom protocol itself. Atom is a successor to RSS; it is 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.

Example 16.2. Basic Use of an Atom Feed

Read an Atom feed and print the `title` and `summary` of each entry:

```
$feed = new Zend_Feed_Atom('http://atom.example.com/feed/');
echo 'The feed contains ' . $feed->count() . ' entries.' . "\n\n";
foreach ($feed as $entry) {
    echo 'Title: ' . $entry->title() . "\n";
    echo 'Summary: ' . $entry->summary() . "\n\n";
}
```

In an Atom feed you can expect to find the following feed properties:

- `title` - The feed's title, same as RSS's channel title
- `id` - Every feed and entry in Atom has a unique identifier
- `link` - Feeds can have multiple links, which are distinguished by a `type` attribute

  The equivalent to RSS's channel link would be `type="text/html"`. If the link is to an alternate version of the same content that's in the feed, it would have a `rel="alternate"` attribute.
- `subtitle` - The feed's description, equivalent to RSS' channel description

  `author->name()` - The feed author's name

  `author->email()` - The feed author's email address

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
- `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

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.

If you read a single entry, you will still have a `Zend_Feed_Atom` object, but it will automatically create an "anonymous" feed to contain the entry.

Example 16.3. Reading a Single-Entry Atom Feed

```
$feed = new Zend_Feed_Atom('http://atom.example.com/feed/1');
echo 'The feed has: ' . $feed->count() . ' entry.';

$entry = $feed->current();
```

Alternatively, you could instantiate the entry object directly if you know you are accessing an `<entry>`-only document:

Example 16.4. Using the Entry Object Directly for a Single-Entry Atom Feed

```
$entry = new Zend_Feed_Entry_Atom('http://atom.example.com/feed/1');
echo $entry->title();
```

Modifying Feed and Entry structures

`Zend_Feed`'s natural syntax extends to constructing and modifying feeds and entries as well as reading them. You can easily turn your new or modified objects back into well-formed XML for saving to a file or sending to a server.

Example 16.5. Modifying an Existing Feed Entry

```
$feed = new Zend_Feed_Atom('http://atom.example.com/feed/1');
$entry = $feed->current();

$entry->title = 'This is a new title';
$entry->author->email = 'my_email@example.com';

echo $entry->saveXML();
```

This will output a full (includes `<?xml ... >` prologue) XML representation of the new entry, including any necessary XML namespaces.

Note that the above will work even if the existing entry does not already have an author tag. You can use as many levels of `->` access as you like before getting to an assignment; all of the intervening levels will be created for you automatically if necessary.

If you want to use a namespace other than `atom:`, `rss:`, or `osrss:` in your entry, you need to register the namespace with `Zend_Feed` using `Zend_Feed::registerNamespace()`. When you are modifying an existing element, it will always maintain its original namespace. When adding a new element, it will go into the default namespace if you do not explicitly specify another namespace.

Example 16.6. Creating an Atom Entry with Elements of Custom Namespaces

```
$entry = new Zend_Feed_Entry_Atom();
// id is always assigned by the server in Atom
$entry->title = 'my custom entry';
$entry->author->name = 'Example Author';
$entry->author->email = 'me@example.com';

// Now do the custom part.
Zend_Feed::registerNamespace('myns', 'http://www.example.com/myns/1.0');

$entry->{'myns:myelement_one'} = 'my first custom value';
$entry->{'myns:container_elt'}->part1 = 'first nested custom part';
$entry->{'myns:container_elt'}->part2 = 'second nested custom part';

echo $entry->saveXML();
```

Custom Feed and Entry Classes

Finally, you can extend the `Zend_Feed` classes if you'd like to provide your own format or niceties like automatic handling of elements that should go into a custom namespace.

Here is an example of a custom Atom entry class that handles its own `myns:` namespace entries. Note that it also makes the `registerNamespace()` call for you, so the end user doesn't need to worry about namespaces at all.

Example 16.7. Extending the Atom Entry Class with Custom Namespaces

```
/**
 * The custom entry class automatically knows the feed URI (optional) and
 * can automatically add extra namespaces.
 */
class MyEntry extends Zend_Feed_Entry_Atom
{

    public function __construct($uri = 'http://www.example.com/myfeed/',
                                $xml = null)
    {
        parent::__construct($uri, $xml);

        Zend_Feed::registerNamespace('myns',
                                     'http://www.example.com/myns/1.0');
    }

    public function __get($var)
    {
        switch ($var) {
            case 'myUpdated':
                // Translate myUpdated to myns:updated.
                return parent::__get('myns:updated');

            default:
                return parent::__get($var);
            }
    }

    public function __set($var, $value)
    {
        switch ($var) {
            case 'myUpdated':
                // Translate myUpdated to myns:updated.
                parent::__set('myns:updated', $value);
                break;

            default:
                parent::__set($var, $value);
        }
    }

    public function __call($var, $unused)
    {
        switch ($var) {
            case 'myUpdated':
                // Translate myUpdated to myns:updated.
                return parent::__call('myns:updated', $unused);

            default:
                return parent::__call($var, $unused);
        }
    }
}
```

Then to use this class, you'd just instantiate it directly and set the `myUpdated` property:

```
$entry = new MyEntry();
$entry->myUpdated = '2005-04-19T15:30';

// method-style call is handled by __call function
$entry->myUpdated();
// property-style call is handled by __get function
$entry->myUpdated;
```

Chapter 17. Zend_File

Zend_File_Transfer

`Zend_File_Transfer` enables developers to take control over file uploads and also over file downloads. It allows you to use built in validators for file purposes and gives you the ability even to change files with filters. `Zend_File_Transfer` works with adapters which allow to use the same API for different transport protocols like HTTP, FTP, WEBDAV and more.

Limitation

The current implementation of `Zend_File_Transfer` shipped in 1.6.0 is limited to HTTP Post Uploads. Download of files and other Adapters will be added in the next releases. Also there is actually no filter available and the functionality for it is not implemented. Not implemented methods will throw an exception. So actually you should use an instance of `Zend_File_Transfer_Adapter_Http` directly. This will change in future, as soon as there are multiple adapters available.

The usage of `Zend_File_Transfer` is quite simple. It consist of two parts. The HTTP Form which does the upload, and the handling of the uploaded files with `Zend_File_Transfer`. See the following example:

Example 17.1. Simple File-Upload Form

This example illustrates a basic file upload which uses `Zend_File_Transfer`. The first part is the file form. In our example there is one file which we want to upload.

```
<form enctype="multipart/form-data" action="/file/upload" method="POST">
    <input type="hidden" name="MAX_FILE_SIZE" value="100000" />
        Choose a file to upload: <input name="uploadedfile" type="file" />
    <br />
    <input type="submit" value="Upload File" />
</form>
```

Note that you should use Zend_Form_Element_File for your convenience instead of creating the HTML manually.

The next step is to create the receiver of the upload. In our example the receiver is `/file/upload`. So next we will create the controller `file` with the action `upload`.

```
$adapter = new Zend_File_Transfer_Adapter_Http();

$adapter->setDestination('C:\temp');

if (!$adapter->receive()) {
    $messages = $adapter->getMessages();
    echo implode("\n", $messages);
}
```

As you see the simplest usage is to define a destination with the `setDestination` method and to call the `receive()` method. If there are any upload errors then you will get them within an exception returned.

Attention

Keep in mind that this is just the simplest usage. You should **never** just this example is an living environment as it causes severe security issues. You should always use validators to increase security.

Validators for Zend_File_Transfer

`Zend_File_Transfer` is delivered with several file related validators which should be used to increase security and prevent possible attacks. Note that the validators are only as good as you are using them. All validators which are provided with `Zend_File_Transfer` can be found in the `Zend_Validator` component and are named `Zend_Validate_File_*`. The following validators are actually available:

- `Count`: This validator checks for the amount of files. It provides a minimum and a maximum and will throw an error when any of these are crossed.
- `Exists`: This validator checks for the existence of files. It will throw an error when an given file does not exist.

- `Extension`: This validator checks the extension of files. It will throw an error when an given file has an undefined extension.
- `FilesSize`: This validator checks the complete size of all validated files. It remembers internally the size of all checked files and throws an error when the sum of all files exceed the defined size. It does also provide a minimum and a maximum size.
- `ImageSize`: This validator checks the size of image. It validates the width and height and provides for both a minimum and a maximum size.
- `MimeType`: This validator can validate the mimetype of files. It is also able to validate types of mimetypes and will throw error when the mimetype of a given file does not match.
- `NotExists`: This validator checks for the existence of files. It will throw an error when an given file does exist.
- `Size`: This validator is able to check files for it's filesize. It provides a minimum and a maximum size and will throw an error when any of these are crossed.
- `Upload`: This validator is an internal one, which checks if a upload has produced a problem. You must not set it, as it's automatically set by `Zend_File_Transfer` itself. So you can forget this validator. You should only know that it exists.

Using validators with `Zend_File_Transfer`

The usage of validators is quite simple. There are several methods for adding and manipulating validators.

- `addValidator($validator, $options = null, $files = null)`: Adds the given validator to the validator stack (optionally only to the file(s) specified). `$validator` may be either an actual validator instance, or a short name specifying the validator type (e.g., 'Count').
- `addValidators(array $validators, $files = null)`: Adds the given validators to the stack of validators. Each entry may be either a validator type/options pair, or an array with the key 'validator' specifying the validator (all other options will be considered validator options for instantiation).
- `setValidators(array $validators, $files = null)`: Overwrites any existing validators with the validators specified. The validators should follow the syntax for `addValidators()`.
- `hasValidator($name)`: Indicates if a validator has been registered.
- `getValidator($name)`: Returns a previously registered validator.
- `getValidators($files = null)`: Returns registered validators; if `$files` is passed, returns validators for that particular file or set of files.
- `removeValidator($name)`: Removes a previously registered validator.
- `clearValidators()`: Clears all registered validators.

Example 17.2. Add validators to a file transfer

```
$upload = new Zend_File_Transfer();

// Set a filesize with 20000 bytes
$upload->addValidator('Size', 20000);

// Set a filesize with 20 bytes minimum and 20000 bytes maximum
$upload->addValidator('Size', array(20, 20000));

// Set a filesize with 20 bytes minimum and 20000 bytes maximum and
// a file count in one step
$upload->setValidators(array(
    'Size'  => array(20, 20000),
    'Count' => array(1, 3),
));
```

Example 17.3. Limit validators to single files

`addValidator()`, `addValidators()`, and `setValidators()` each accept a final `$files` argument. This argument can be used to specify a particular file or array of files on which to set the given validator.

```
$upload = new Zend_File_Transfer();

// Set a filesize with 20000 bytes and limits it only to 'file2'
$upload->addValidator('Size', 20000, 'file2');
```

Generally you should simply use the `addValidators()` method, which can be called multiple times.

Example 17.4. Add multiple validators

Often it's simpler just to call `addValidator()` multiple times. One call for each validator. This also increases the readability and makes your code more maintainable. As all methods provide a fluent interface you can couple the calls as shown below:

```
$upload = new Zend_File_Transfer();

// Set a filesize with 20000 bytes
$upload->addValidator('Size', 20000)
       ->addValidator('Count', 2)
       ->addValidator('Filessize', 25000);
```

Note

Note that even though setting the same validator multiple times is allowed, doing so can lead to issues when using different options for the same validator.

Count validator

The `Count` validator checks for the number of files which are provided. It supports the following options:

- `min`: Sets the minimum number of files to transfer.

  ### Note

  Beware: When using this option you must give the minimum number of files when calling this validator the first time; otherwise you will get an error in return.

  With this option you can define the minimum number of files you expect to receive.

- `max`: Set the maximum number of files to transfer.

  With this option you can limit the number of files which are accepted but also detect a possible attack when more files are given than defined in your form.

You can initiate this validator with both options. The first option is `min`, the second option is `max`. When only one option is given it is used as `max`. But you can also use the methods `setMin()` and `setMax()` to set both options afterwards and `getMin()` and `getMax()` to retrieve the actual set values.

Example 17.5. Using the Count validator

```
$upload = new Zend_File_Transfer();

// Limit the amount of files to maximum 2
$upload->addValidator('Count', 2);

// Limit the amount of files to maximum 5 and expects minimum 1 file
// to be returned
$upload->addValidator('Count', array(1, 5);
```

Note

Note that this validator stores the number of checked files internally. The file which exceeds the maximum will be returned as error.

Exists validator

The `Exists` validator checks for the existence of files which are provided. It supports the following options:

- `directory`: Checks if the file exists in the given directory.

This validator accepts multiple directories either as a comma-delimited string, or as an array. You may also use the methods `setDirectory()`, `addDirectory()`, and `getDirectory()` to set and retrieve directories.

Example 17.6. Using the Exists validator

```
$upload = new Zend_File_Transfer();

// Add the temp directory to check for
$upload->addValidator('Exists', '\temp');

// Add two directories using the array notation
$upload->addValidator('Exists', array('\home\images', '\home\uploads'));
```

Note

Note that this validator checks if the file exists in all set directories. The validation will fail if the file does not exist in any of the given directories.

Extension validator

The `Extension` validator checks the file extension of files which are provided. It supports the following options:

- `extension`: Checks if the given file uses this file extension.
- `case`: Sets if validation should be done case sensitive. Default is not case sensitive. Note the this option is used for all used extensions.

This validator accepts multiple extensions either as a comma-delimited string, or as an array. You may also use the methods `setExtension()`, `addExtension()`, and `getExtension()` to set and retrieve extensions.

In some cases it is usefull to test case sensitive. Therefor the constructor allows a second parameter `$case` which, if set to true, will validate the extension case sensitive.

Example 17.7. Using the Extension validator

```
$upload = new Zend_File_Transfer();

// Limit the extensions to jpg and png files
$upload->addValidator('Extension', 'jpg,png');

// Limit the extensions to jpg and png files but use array notation
$upload->addValidator('Extension', array('jpg', 'png'));

// Check case sensitive
$upload = new Zend_File_Transfer('mo,png', true);
if (!$upload->isValid('C:\temp\myfile.MO')) {
    print 'Not valid due to MO instead of mo';
}
```

Note

Note that this validator just checks the file extension. It does not check the actual file MIME type.

FilesSize validator

The `FilesSize` validator checks for the aggregate size of all transferred files. It supports the following options:

- `min`: Sets the minimum aggregate filesize.

  With this option you can define the minimum aggregate filesize of files you expect to transfer.

- `max`: Sets the maximum aggregate filesize.

  With this option you can limit the aggregate filesize of all files which are transferred, but not the filesize of individual files.

You can initiate this validator with both options. The first option is `min`, the second option is `max`. When only one option is given it is used as `max`. But you can also use the methods `setMin()` and `setMax()` to set both options afterwards and `getMin()` and `getMax()` to receive the actual set values.

The size itself is also accepted in SI notation as done by most operating systems. Instead of 20000 bytes you can just give **20kB**. All units are converted by using 1024 as base value. The following Units are accepted: `kB`, `MB`, `GB`, `TB`, `PB` and `EB`. As mentioned you have to note that 1kB is equal to 1024 bytes.

Example 17.8. Using the FilesSize validator

```
$upload = new Zend_File_Transfer();

// Limit the size of all given files to 40000 bytes
$upload->addValidator('FilesSize', 40000);

// Limit the size of all given files to maximum 4MB and mimimum 10kB
$upload->setValidator('FilesSize', array('10kB', '4MB');
```

Note

Note that this validator stores the filesize of checked files internally. The file which exceeds the size will be returned as error.

ImageSize validator

The `ImageSize` validator checks for the size of image files. It supports the following options:

- `minheight`: Sets the minimum image height.

  With this option you can define the minimum height of the image you want to validate.

- `maxheight`: Sets the maximum image height.

  With this option you can limit the maximum height of the image you want to validate.

- `minwidth`: Sets the minimum image width.

  With this option you can define the minimum width of the image you want to validate.

- `maxwidth`: Sets the maximum image width.

  With this option you can limit the maximum width of the image you want to validate.

You can initiate this validator with all four options set. When `minheight` or `minwidth` are not given, they will be set to 0. And when `maxwidth` or `maxheight` are not given, they will be set to null. But you can also use the methods `setImageMin()` and `setImageMax()` to set both minimum and maximum values to set the options afterwards and `getMin()` and `getMax()` to receive the actual set values.

For your convinience there is also a `setImageWidth` and `setImageHeight` method which will set the minimum and maximum height and width. Of course also the related `getImageWidth` and `getImageHeight` methods are available.

To suppress the validation of a dimension just set the related value to `null`.

Example 17.9. Using the ImageSize validator

```
$upload = new Zend_File_Transfer();

// Limit the size of a image to a height of 100-200 and a width of
// 40-80 pixel
$upload->addValidator('ImageSize', 40, 100, 80, 200);

// Use the array notation
$upload->setValidator('ImageSize', array(40, 100, 80, 200);

// Use the named array notation
$upload->setValidator('ImageSize',
                      array('minwidth' => 40,
                            'maxwidth' => 80,
                            'minheight' => 100,
                            'maxheight' => 200)
                      );

// Set other image dimensions
$upload->setImageWidth(20, 200);
```

MimeType validator

The `MimeType` validator checks for the mimetype of transferred files. It supports the following options:

- `MimeType`: Set the mimetype type to validate against.

  With this option you can define the mimetype of files which will be accepted.

This validator accepts multiple mimetype either as a comma-delimited string, or as an array. You may also use the methods `setMimeType()`, `addMimeType()`, and `getMimeType()` to set and retrieve mimetype.

Example 17.10. Using the MimeType validator

```
$upload = new Zend_File_Transfer();

// Limit the mimetype of all given files to gif images
$upload->addValidator('MimeType', 'image/gif');

// Limit the mimetype of all given files to gif and jpeg images
$upload->setValidator('MimeType', array('image/gif', 'image/jpeg');

// Limit the mimetype of all given files to the group images
$upload->setValidator('MimeType', 'image');
```

The above example shows that it is also possible to limit the accepted mimetype to a group of mimetypes. To allow all images just use 'image' as mimetype. This can be used for all groups of mimetypes like 'image', 'audio', 'video', 'text, and so on.

Note

Note that allowing groups of mimetypes will accept all members of this group even if your application does not support them. When you allow 'image' you will also get 'image/xpixmap' or 'image/vasa' which could be problematic. When you are not sure if your application supports all types you should better allow only defined mimetypes instead of the complete group.

NotExists validator

The `NotExists` validator checks for the existence of files which are provided. It supports the following options:

- `directory`: Checks if the file does not exist in the given directory.

This validator accepts multiple directories either as a comma-delimited string, or as an array. You may also use the methods `setDirectory()`, `addDirectory()`, and `getDirectory()` to set and retrieve directories.

Example 17.11. Using the NotExists validator

```
$upload = new Zend_File_Transfer();

// Add the temp directory to check for
$upload->addValidator('NotExists', '\temp');

// Add two directories using the array notation
$upload->addValidator('NotExists',
                      array('\home\images',
                            '\home\uploads')
                     );
```

Note

Note that this validator checks if the file does not exist in all of the set directories. The validation will fail if the file does exist in any of the given directories.

Size validator

The `Size` validator checks for the size of a single file. It supports the following options:

- `Min`: Set the minimum filesize.

  With this option you can define the minimum filesize for an individual file you expect to transfer.

- `Max`: Set the maximum filesize.

  With this option you can limit the filesize of a single file you transfer.

You can initiate this validator with both options. The first option is `min`, the second option is `max`. When only one option is given it is used as `max`. But you can also use the methods `setMin()` and `setMax()` to set both options afterwards and `getMin()` and `getMax()` to receive the actual set values.

The size itself is also accepted in SI notation as done by most operating systems. Instead of 20000 bytes you can just give **20kB**. All units are converted by using 1024 as base value. The following Units are accepted: `kB`, `MB`, `GB`, `TB`, `PB` and `EB`. As mentioned you have to note that 1kB is equal to 1024 bytes.

Example 17.12. Using the Size validator

```
$upload = new Zend_File_Transfer();

// Limit the size of a file to 40000 bytes
$upload->addValidator('Size', 40000);

// Limit the size a given file to maximum 4MB and mimimum 10kB and
// limits this validator to the file "uploadfile"
$upload->addValidator('Size', array('10kB', '4MB', 'uploadfile');
```

Chapter 18. Zend_Filter

Introduction

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 "`&amp;`"). 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_Interface`, which requires a single method named `filter()` to be implemented by a filter class.

Following is a basic example of using a filter upon two input data, the ampersand (&) and double quote (") characters:

```
$htmlEntities = new Zend_Filter_HtmlEntities();

echo $htmlEntities->filter('&'); // &amp;
echo $htmlEntities->filter('"'); // &quot;
```

Using the static `get()` method

If it is inconvenient to load a given filter class and create an instance of the filter, you can use the static method `Zend_Filter::get()` 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 `get()` method automatically loads the class, creates an instance, and applies the `filter()` method to the data input.

```
echo Zend_Filter::get('&', 'HtmlEntities');
```

You can also pass an array of constructor arguments, if they are needed for the filter class.

```
echo Zend_Filter::get('"', 'HtmlEntities', array(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 Zend_Filter_Input class allows you to instantiate and run multiple filter and validator classes on demand to process sets of input data. See the section called “Zend_Filter_Input”.

Standard Filter Classes

The Zend Framework comes with a standard set of filters, which are ready for you to use.

Alnum

Returns the string `$value`, removing all but alphabetic and digit characters. This filter includes an option to also allow white space characters.

Alpha

Returns the string `$value`, removing all but alphabetic characters. This filter includes an option to also allow white space characters.

BaseName

Given a string containing a path to a file, this filter will return the base name of the file

Digits

Returns the string `$value`, removing all but digit characters.

Dir

Returns directory name component of path.

HtmlEntities

Returns the string `$value`, converting characters to their corresponding HTML entity equivalents where they exist.

Int

Returns (int) `$value`

StripNewlines

Returns the string `$value` without any newline control characters.

RealPath

Expands all symbolic links and resolves references to '/./', '/../' and extra '/' characters in the input path and return the canonicalized absolute pathname. The resulting path will have no symbolic link, '/./' or '/../' components.

`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`.

StringToLower

Returns the string `$value`, converting alphabetic characters to lowercase as necessary.

StringToUpper

Returns the string `$value`, converting alphabetic characters to uppercase as necessary.

StringTrim

Returns the string `$value` with characters stripped from the beginning and end.

StripTags

This filter returns the input string, with all HTML and PHP tags stripped from it, except those that have been explicitly allowed. In addition to the ability to specify which tags are allowed, developers can specify which attributes are allowed across all allowed tags and for specific tags only. Finally, this filter offers control over whether comments (e.g., `<!-- ... -->`) are removed or allowed.

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` provides a simple method by which filters may be chained together. The following code illustrates how to chain together two filters for the submitted username:

```
<// Create a filter chain and add filters to the chain
$filterChain = new Zend_Filter();
$filterChain->addFilter(new Zend_Filter_Alpha())
            ->addFilter(new Zend_Filter_StringToLower());

// Filter the username
$username = $filterChain->filter($_POST['username']);
```

Filters are run in the order they were added to `Zend_Filter`. 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_Interface` may be used in a filter chain.

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_Interface`.

`Zend_Filter_Interface` defines a single method, `filter()`, that may be implemented by user classes. An object that implements this interface may be added to a filter chain with `Zend_Filter::addFilter()`.

The following example demonstrates how to write a custom filter:

```
class MyFilter implements Zend_Filter_Interface
{
    public function filter($value)
    {
        // perform some transformation upon $value to arrive on $valueFiltered

        return $valueFiltered;
    }
}
```

To add an instance of the filter defined above to a filter chain:

```
$filterChain = new Zend_Filter();
$filterChain->addFilter(new MyFilter());
```

Zend_Filter_Input

Zend_Filter_Input provides a declarative interface to associate multiple filters and validators, apply them to collections of data, and to retrieve input values after they have been processed by the filters and validators. Values are returned in escaped format by default for safe HTML output.

Consider the metaphor that this class is a cage for external data. Data enter the application from external sources, such as HTTP request parameters, HTTP headers, a web service, or even read from a database or another file. Data are first put into the cage, and subsequently the application can access data only by telling the cage what the data should be and how they plan to use it. The cage inspects the data for validity. It might apply escaping to the data values for the appropriate context. The cage releases data only if it can fulfill these responsibilities. With a simple and convenient interface, it encourages good programming habits and makes developers think about how data are used.

- **Filters** transform input values, by removing or changing characters within the value. The goal is to "normalize" input values until they match an expected format. For example, if a string of numeric digits is needed, and the input value is "abc123", then it might be a reasonable transformation to change the value to the string "123".
- **Validators** check input values against criteria and report whether they passed the test or not. The value is not changed, but the check may fail. For example, if a string must look like an email address, and the input value is "abc123", then the value is not considered valid.
- **Escapers** transform a value by removing magic behavior of certain characters. In some output contexts, special characters have meaning. For example, the characters '<' and '>' delimit HTML tags, and if a string containing those characters is output in an HTML context, the content between them might affect the output or functionality of the HTML presentation. Escaping the characters removes the special meaning, so they are output as literal characters.

To use Zend_Filter_Input, perform the following steps:

1. Declare filter and validator rules
2. Create the filter and validator processor
3. Provide input data
4. Retrieve validated fields and other reports

The following sections describe the steps for using this class.

Declaring Filter and Validator Rules

Before creating an instance of Zend_Filter_Input, declare an array of filter rules and and an array of validator rules. This associative array maps a rule name to a filter or validator or a chain of filters or validators.

The following example filter rule set that declares the field 'month' is filtered by Zend_Filter_Digits, and the field 'account' is filtered by Zend_Filter_StringTrim. Then a validation rule set declares that the field 'account' is valid only if it contains only alphabetical characters.

```
$filters = array(
    'month'   => 'Digits',
    'account' => 'StringTrim'
);

$validators = array(
    'account' => 'Alpha'
);
```

Each key in the `$filters` array above is the name of a rule for applying a filter to a specific data field. By default, the name of the rule is also the name of the input data field to which to apply the rule.

You can declare a rule in several formats:

- A single string scalar, which is mapped to a class name.

```
$validators = array(
    'month'   => 'Digits',
);
```

- An object instance of one of the classes that implement Zend_Filter_Interface or Zend_Validate_Interface.

```
$digits = new Zend_Validate_Digits();

$validators = array(
    'month'   => $digits
);
```

- An array, to declare a chain of filters or validators. The elements of this array can be strings mapping to class names or filter/validator objects, as in the cases described above. In addition, you can use a third choice: an array containing a string mapping to the class name followed by arguments to pass to its constructor.

```
$validators = array(
    'month'   => array(
        'Digits',                // string
        new Zend_Validate_Int(), // object instance
        array('Between', 1, 12)  // string with constructor arguments
    )
);
```

Note

If you declare a filter or validator with constructor arguments in an array, then you must make an array for the rule, even if the rule has only one filter or validator.

You can use a special "wildcard" rule key `'*'` in either the filters array or the validators array. This means that the filters or validators declared in this rule will be applied to all input data fields. Note that the order of entries in the filters array or validators array is significant; the rules are applied in the same order in which you declare them.

```
$filters = array(
    '*'     => 'StringTrim',
    'month' => 'Digits'
);
```

Creating the Filter and Validator Processor

After declaring the filters and validators arrays, use them as arguments in the constructor of Zend_Filter_Input. This returns an object that knows all your filtering and validating rules, and you can use this object to process one or more sets of input data.

```
$input = new Zend_Filter_Input($filters, $validators);
```

You can specify input data as the third constructor argument. The data structure is an associative array. The keys are field names, and the values are data values. The standard `$_GET` and `$_POST` superglobal variables in PHP are examples of this format. You can use either of these variables as input data for Zend_Filter_Input.

```
$data = $_GET;

$input = new Zend_Filter_Input($filters, $validators, $data);
```

Alternatively, use the `setData()` method, passing an associative array of key/value pairs the same format as described above.

```
$input = new Zend_Filter_Input($filters, $validators);
$input->setData($newData);
```

The `setData()` method redefines data in an existing Zend_Filter_Input object without changing the filtering and validation rules. Using this method, you can run the same rules against different sets of input data.

Retrieving Validated Fields and other Reports

After you have declared filters and validators and created the input processor, you can retrieve reports of missing, unknown, and invalid fields. You also can get the values of fields after filters have been applied.

Querying if the input is valid

If all input data pass the validation rules, the `isValid()` method returns `true`. If any field is invalid or any required field is missing, `isValid()` returns `false`.

```
if ($input->isValid()) {
  echo "OK\n";
}
```

This method accepts an optional string argument, naming an individual field. If the specified field passed validation and is ready for fetching, `isValid('fieldName')` returns `true`.

```
if ($input->isValid('month')) {
  echo "Field 'month' is OK\n";
}
```

Getting Invalid, Missing, or Unknown Fields

- **Invalid** fields are those that don't pass one or more of their validation checks.
- **Missing** fields are those that are not present in the input data, but were declared with the metacommand `'presence'=>'required'` (see the later section on metacommands).
- **Unknown** fields are those that are not declared in any rule in the array of validators, but appear in the input data.

```
if ($input->hasInvalid() || $input->hasMissing()) {
  $messages = $input->getMessages();
}

// getMessages() simply returns the merge of getInvalid() and
// getMissing()

if ($input->hasInvalid()) {
  $invalidFields = $input->getInvalid();
}

if ($input->hasMissing()) {
  $missingFields = $input->getMissing();
}

if ($input->hasUnknown()) {
  $unknownFields = $input->getUnknown();
}
```

The results of the `getMessages()` method is an associative array, mapping a rule name to an array of error messages related to that rule. Note that the index of this array is the rule name used in the rule declaration, which may be different from the names of fields checked by the rule.

The `getMessages()` method returns the merge of the arrays returned by the `getInvalid()` and `getMissing()`. These methods return subsets of the messages, related to validation failures, or fields that were declared as required but missing from the input.

The `getErrors()` method returns an associative array, mapping a rule name to an array of error identifiers. Error identifiers are fixed strings, to identify the reason for a validation failure, while messages can be customized. See the section called “Basic usage of validators” for more information.

You can specify the message returned by `getMissing()` using the 'missingMessage' option, as an argument to the Zend_Filter_Input constructor or using the `setOptions()` method.

```
$options = array(
    'missingMessage' => "Field '%field%' is required"
);

$input = new Zend_Filter_Input($filters, $validators, $data, $options);

// alternative method:

$input = new Zend_Filter_Input($filters, $validators, $data);
$input->setOptions($options);
```

The results of the `getUnknown()` method is an associative array, mapping field names to field values. Field names are used as the array keys in this case, instead of rule names, because no rule mentions the fields considered to be unknown fields.

Getting Valid Fields

All fields that are neither invalid, missing, nor unknown are considered valid. You can get values for valid fields using a magic accessor. There are also non-magic accessor methods `getEscaped()` and `getUnescaped()`.

```
$m = $input->month;                 // escaped output from magic accessor
$m = $input->getEscaped('month');   // escaped output
$m = $input->getUnescaped('month'); // not escaped
```

By default, when retrieving a value, it is filtered with the Zend_Filter_HtmlEntities. This is the default because it is considered the most common usage to output the value of a field in HTML. The HtmlEntities filter helps prevent unintentional output of code, which can result in security problems.

Note

As shown above, you can retrieve the unescaped value using the `getUnescaped()` method, but you must write code to use the value safely, and avoid security issues such as vulnerability to cross-site scripting attacks.

You can specify a different filter for escaping values, by specifying it in the constructor options array:

```
$options = array('escapeFilter' => 'StringTrim');
$input = new Zend_Filter_Input($filters, $validators, $data, $options);
```

Alternatively, you can use the `setDefaultEscapeFilter()` method:

```
$input = new Zend_Filter_Input($filters, $validators, $data);
$input->setDefaultEscapeFilter(new Zend_Filter_StringTrim());
```

In either usage, you can specify the escape filter as a string base name of the filter class, or as an object instance of a filter class. The escape filter can be an instance of a filter chain, an object of the class Zend_Filter.

Filters to escape output should be run in this way, to make sure they run after validation. Other filters you declare in the array of filter rules are applied to input data before data are validated. If escaping filters were run before validation, the process of validation would be more complex, and it would be harder to provide both escaped and unescaped versions of the data. So it is recommended to declare filters to escape output using `setDefaultEscapeFilter()`, not in the `$filters` array.

There is only one method `getEscaped()`, and therefore you can specify only one filter for escaping (although this filter can be a filter chain). If you need a single instance of Zend_Filter_Input to return escaped output using more than one filtering method, you should extend Zend_Filter_Input and implement new methods in your subclass to get values in different ways.

Using Metacommands to Control Filter or Validator Rules

In addition to declaring the mapping from fields to filters or validators, you can specify some "metacommands" in the array declarations, to control some optional behavior of Zend_Filter_Input. Metacommands appear as string-indexed entries in a given filter or validator array value.

The `FIELDS` metacommand

If the rule name for a filter or validator is different than the field to which it should apply, you can specify the field name with the 'fields' metacommand.

You can specify this metacommand using the class constant `Zend_Filter_Input::FIELDS` instead of the string.

```
$filters = array(
    'month' => array(
        'Digits',        // filter name at integer index [0]
        'fields' => 'mo' // field name at string index ['fields']
    )
);
```

In the example above, the filter rule applies the 'digits' filter to the input field named 'mo'. The string 'month' simply becomes a mnemonic key for this filtering rule; it is not used as the field name if the field is specified with the 'fields' metacommand, but it is used as the rule name.

The default value of the 'fields' metacommand is the index of the current rule. In the example above, if the 'fields' metacommand is not specified, the rule would apply to the input field named 'month'.

Another use of the 'fields' metacommand is to specify fields for filters or validators that require multiple fields as input. If the 'fields' metacommand is an array, the argument to the corresponding filter or validator is an array of the values of those fields. For example, it is common for users to specify a password string in two fields, and they must type the same string in both fields. Suppose you implement a validator class that takes an array argument, and returns `true` if all the values in the array are equal to each other.

```
$validators = array(
    'password' => array(
        'StringEquals',
        'fields' => array('password1', 'password2')
    )
);
// Invokes hypothetical class Zend_Validate_StringEquals,
// passing an array argument containing the values of the two input
// data fields named 'password1' and 'password2'.
```

If the validation of this rule fails, the rule key (`'password'`) is used in the return value of `getInvalid()`, not any of the fields named in the 'fields' metacommand.

The PRESENCE metacommand

Each entry in the validator array may have a metacommand called 'presence'. If the value of this metacommand is 'required' then the field must exist in the input data, or else it is reported as a missing field.

You can specify this metacommand using the class constant `Zend_Filter_Input::PRESENCE` instead of the string.

```
$validators = array(
    'month' => array(
        'digits',
        'presence' => 'required'
    )
);
```

The default value of this metacommand is 'optional'.

The DEFAULT_VALUE metacommand

If a field is not present in the input data, and you specify a value for the 'default' metacommand for that rule, the field takes the value of the metacommand.

You can specify this metacommand using the class constant `Zend_Filter_Input::DEFAULT_VALUE` instead of the string.

This default value is assigned to the field before any of the validators are invoked. The default value is applied to the field only for the current rule; if the same field is referenced in a subsequent rule, the field has no value when evaluating that rule. Thus different rules can declare different default values for a given field.

```
$validators = array(
    'month' => array(
        'digits',
        'default' => '1'
    )
);

// no value for 'month' field
$data = array();

$input = new Zend_Filter_Input(null, $validators, $data);
echo $input->month; // echoes 1
```

If your rule uses the `FIELDS` metacommand to define an array of multiple fields, you can define an array for the `DEFAULT_VALUE` metacommand and the defaults of corresponding keys are used for any missing fields. If `FIELDS` defines multiple fields but `DEFAULT_VALUE` is a scalar, then that default value is used as the value for any missing fields in the array.

There is no default value for this metacommand.

The `ALLOW_EMPTY` metacommand

By default, if a field exists in the input data, then validators are applied to it, even if the value of the field is an empty string (`''`). This is likely to result in a failure to validate. For example, if the validator checks for digit characters, and there are none because a zero-length string has no characters, then the validator reports the data as invalid.

If in your case an empty string should be considered valid, you can set the metacommand 'allowEmpty' to `true`. Then the input data passes validation if it is present in the input data, but has the value of an empty string.

You can specify this metacommand using the class constant `Zend_Filter_Input::ALLOW_EMPTY` instead of the string.

```
$validators = array(
    'address2' => array(
        'Alnum',
        'allowEmpty' => true
    )
);
```

The default value of this metacommand is `false`.

In the uncommon case that you declare a validation rule with no validators, but the 'allowEmpty' metacommand is `false` (that is, the field is considered invalid if it is empty), Zend_Filter_Input returns a default error message that you can retrieve with `getMessages()`. You can specify this message using the 'notEmptyMessage' option, as an argument to the Zend_Filter_Input constructor or using the `setOptions()` method.

```
$options = array(
    'notEmptyMessage' => "A non-empty value is required for field '%field%'"
);

$input = new Zend_Filter_Input($filters, $validators, $data, $options);

// alternative method:

$input = new Zend_Filter_Input($filters, $validators, $data);
$input->setOptions($options);
```

The `BREAK_CHAIN` metacommand

By default if a rule has more than one validator, all validators are applied to the input, and the resulting messages contain all error messages caused by the input.

Alternatively, if the value of the 'breakChainOnFailure' metacommand is `true`, the validator chain terminates after the first validator fails. The input data is not checked against subsequent validators in the chain, so it might cause more violations even if you correct the one reported.

You can specify this metacommand using the class constant `Zend_Filter_Input::BREAK_CHAIN` instead of the string.

```
$validators = array(
    'month' => array(
        'Digits',
        new Zend_Validate_Between(1,12),
        new Zend_Validate_GreaterThan(0),
        'breakChainOnFailure' => true
    )
);
$input = new Zend_Filter_Input(null, $validators);
```

The default value of this metacommand is `false`.

The validator chain class, Zend_Validate, is more flexible with respect to breaking chain execution than Zend_Filter_Input. With the former class, you can set the option to break the chain on failure independently for each validator in the chain. With the latter class, the defined value of the 'breakChainOnFailure' metacommand for a rule applies uniformly for all validators in the rule. If you require the more flexible usage, you should create the validator chain yourself, and use it as an object in the validator rule definition:

```
// Create validator chain with non-uniform breakChainOnFailure
// attributes
$chain = new Zend_Validate();
$chain->addValidator(new Zend_Validate_Digits(), true);
$chain->addValidator(new Zend_Validate_Between(1,12), false);
$chain->addValidator(new Zend_Validate_GreaterThan(0), true);

// Declare validator rule using the chain defined above
```

```
$validators = array(
    'month' => $chain
);
$input = new Zend_Filter_Input(null, $validators);
```

The `MESSAGES` metacommand

You can specify error messages for each validator in a rule using the metacommand 'messages'. The value of this metacommand varies based on whether you have multiple validators in the rule, or if you want to set the message for a specific error condition in a given validator.

You can specify this metacommand using the class constant `Zend_Filter_Input::MESSAGES` instead of the string.

Below is a simple example of setting the default error message for a single validator.

```
$validators = array(
    'month' => array(
        'digits',
        'messages' => 'A month must consist only of digits'
    )
);
```

If you have multiple validators for which you want to set the error message, you should use an array for the value of the 'messages' metacommand.

Each element of this array is applied to the validator at the same index position. You can specify a message for the validator at position *n* by using the value *n* as the array index. Thus you can allow some validators to use their default message, while setting the message for a subsequent validator in the chain.

```
$validators = array(
    'month' => array(
        'digits',
        new Zend_Validate_Between(1, 12),
        'messages' => array(
            // use default message for validator [0]
            // set new message for validator [1]
            1 => 'A month value must be between 1 and 12'
        )
    )
);
```

If one of your validators has multiple error messages, they are identified by a message key. There are different keys in each validator class, serving as identifiers for error messages that the respective validator class might generate. Each validate class defines constants for its message keys. You can use these keys in the 'messages' metacommand by passing an associative array instead of a string.

```
$validators = array(
    'month' => array(
        'digits', new Zend_Validate_Between(1, 12),
        'messages' => array(
            'A month must consist only of digits',
            array(
                Zend_Validate_Between::NOT_BETWEEN =>
                    'Month value %value% must be between ' .
                    '%min% and %max%',
                Zend_Validate_Between::NOT_BETWEEN_STRICT =>
                    'Month value %value% must be strictly between ' .
                    '%min% and %max%'
            )
        )
    )
);
```

You should refer to documentation for each validator class to know if it has multiple error messages, the keys of these messages, and the tokens you can use in the message templates.

Using options to set metacommands for all rules

The default value for 'allowEmpty', 'breakChainOnFailure', and 'presence' metacommands can be set for all rules using the `$options` argument to the constructor of Zend_Filter_Input. This allows you to set the default value for all rules, without requiring you to set the metacommand for every rule.

```
// The default is set so all fields allow an empty string.
$options = array('allowEmpty' => true);

// You can override this in a rule definition,
// if a field should not accept an empty string.
$validators = array(
    'month' => array(
        'Digits',
        'allowEmpty' => false
    )
);

$input = new Zend_Filter_Input($filters, $validators, $data, $options);
```

The 'fields', 'messages', and 'default' metacommands cannot be set using this technique.

Adding Filter Class Namespaces

By default, when you declare a filter or validator as a string, Zend_Filter_Input searches for the corresponding classes under the Zend_Filter or Zend_Validate namespaces. For example, a filter named by the string 'digits' is found in the class Zend_Filter_Digits.

If you write your own filter or validator classes, or use filters or validators provided by a third-party, the classes may exist in different namespaces than Zend_Filter or Zend_Validate. You can tell Zend_Filter_Input to search more namespaces. You can specify namespaces in the constructor options:

```
$options = array('inputNamespace' => 'My_Namespace');
$input = new Zend_Filter_Input($filters, $validators, $data, $options);
```

Alternatively, you can use the addNamespace() method:

```
$input->addNamespace('Other_Namespace');

// Now the search order is:
// 1. My_Namespace
// 2. Other_Namespace
// 3. Zend_Filter
// 4. Zend_Validate
```

You cannot remove Zend_Filter and Zend_Validate as namespaces, you only can add namespaces. User-defined namespaces are searched first, Zend namespaces are searched last.

Note

As of version 1.0.4, `Zend_Filter_Input::NAMESPACE`, having value `namespace`, was changed to `Zend_Filter_Input::INPUT_NAMESPACE`, having value `inputNamespace`, in order to comply with the PHP 5.3 reservation of the keyword `namespace`.

Zend_Filter_Inflector

`Zend_Filter_Inflector` is a general purpose tool for rules-based inflection of strings to a given target.

As an example, you may find you need to transform MixedCase or camelCasedWords into a path; for readability, OS policies, or other reasons, you also need to lower case this, and you want to separate the words using a dash ('-'). An inflector can do this for you.

`Zend_Filter_Inflector` implements `Zend_Filter_Interface`; you perform inflection by calling `filter()` on the object instance.

Example 18.1. Transforming MixedCase and camelCaseText to another format

```
$inflector = new Zend_Filter_Inflector('pages/:page.:suffix');
$inflector->setRules(array(
    ':page'  => array('Word_CamelCaseToDash', 'StringToLower'),
    'suffix' => 'html'
));

$string   = 'camelCasedWords';
$filtered = $inflector->filter(array('page' => $string));
// pages/camel-cased-words.html

$string   = 'this_is_not_camel_cased';
$filtered = $inflector->filter(array('page' => $string));
// pages/this_is_not_camel_cased.html
```

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/value pairs corresponding to the variables in the target.

Each variable in the target can have zero or more rules associated with them. Rules may be either *static* or refer to a `Zend_Filter` class. Static rules will replace with the text provided. Otherwise, a class matching the rule provided will be used to inflect the text. Classes are typically specified using a short name indicating the filter name stripped of any common prefix.

As an example, you can use any `Zend_Filter` concrete implementations; however, instead of referring to them as 'Zend_Filter_Alpha' or 'Zend_Filter_StringToLower', you'd specify only 'Alpha' or 'StringToLower'.

Setting Paths To Alternate Filters

`Zend_Filter_Inflector` uses `Zend_Loader_PluginLoader` to manage loading filters to use with inflection. By default, any filter prefixed with `Zend_Filter` will be available. To access filters with that prefix but which occur deeper in the hierarchy, such as the various Word filters, simply strip off the Zend_Filter prefix:

```
// use Zend_Filter_Word_CamelCaseToDash as a rule
$inflector->addRules(array('script' => 'Word_CamelCaseToDash'));
```

To set alternate paths, `Zend_Filter_Inflector` has a utility method that proxies to the plugin loader, `addFilterPrefixPath()`:

```
$inflector->addFilterPrefixPath('My_Filter', 'My/Filter/');
```

Alternatively, you can retrieve the plugin loader from the inflector, and interact with it directly:

```
$loader = $inflector->getPluginLoader();
$loader->addPrefixPath('My_Filter', 'My/Filter/');
```

For more options on modifying the paths to filters, please see the PluginLoader documentation.

Setting the Inflector Target

The inflector target is a string with some placeholders for variables. Placeholders take the form of an identifier, a colon (':') by default, followed by a variable name: ':script', ':path', etc. The `filter()` method looks for the identifier followed by the variable name being replaced.

You can change the identifier using the `setTargetReplacementIdentifier()` method, or passing it as the third argument to the constructor:

```
// Via constructor:
$inflector = new Zend_Filter_Inflector('#foo/#bar.#sfx', null, '#');

// Via accessor:
$inflector->setTargetReplacementIdentifier('#');
```

Typically, you will set the target via the constructor. However, you may want to re-set the target later (for instance, to modify the default inflector in core components, such as the `ViewRenderer` or `Zend_Layout`). `setTarget()` can be used for this purpose:

```
$inflector = $layout->getInflector();
$inflector->setTarget('layouts/:script.phtml');
```

Additionally, you may wish to have a class member for your class that you can use to keep the inflector target updated -- without needing to directly update the target each time (thus saving on method calls). `setTargetReference()` allows you to do this:

```
class Foo
{
    /**
     * @var string Inflector target
     */
    protected $_target = 'foo/:bar/:baz.:suffix';

    /**
     * Constructor
```

```
         * @return void
         */
        public function __construct()
        {
            $this->_inflector = new Zend_Filter_Inflector();
            $this->_inflector->setTargetReference($this->_target);
        }

        /**
         * Set target; updates target in inflector
         *
         * @param  string $target
         * @return Foo
         */
        public function setTarget($target)
        {
            $this->_target = $target;
            return $this;
        }
    }
```

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:

```
$inflector = new Zend_Filter_Inflector(':script.:suffix');
$inflector->setStaticRule('suffix', 'phtml');

// change it later:
$inflector->setStaticRule('suffix', 'php');
```

Much like the target itself, you can also bind a static rule to a reference, allowing you to update a single variable instead of require a method call; this is often useful when your class uses an inflector internally, and you don't want your users to need to fetch the inflector in order to update it. The `setStaticRuleReference()` method is used to accomplish this:

```
class Foo
{
    /**
     * @var string Suffix
     */
    protected $_suffix = 'phtml';

    /**
     * Constructor
     * @return void
     */
    public function __construct()
    {
        $this->_inflector = new Zend_Filter_Inflector(':script.:suffix');
        $this->_inflector->setStaticRuleReference('suffix', $this->_suffix);
    }

    /**
     * Set suffix; updates suffix static rule in inflector
     *
     * @param  string $suffix
     * @return Foo
     */
    public function setSuffix($suffix)
    {
        $this->_suffix = $suffix;
        return $this;
    }
}
```

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_Interface` may be passed as a filter.
- *Array*. An array of one or more strings or filter objects as defined above.

```
$inflector = new Zend_Filter_Inflector(':script.:suffix');

// Set rule to use Zend_Filter_Word_CamelCaseToDash filter
$inflector->setFilterRule('script', 'Word_CamelCaseToDash');

// Add rule to lowercase string
$inflector->addFilterRule('script', new Zend_Filter_StringToLower());

// Set rules en-masse
$inflector->setFilterRule('script', array(
    'Word_CamelCaseToDash',
    new Zend_Filter_StringToLower()
));
```

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/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.

Example 18.2. Setting Multiple Rules at Once

```
// Could also use setRules() with this notation:
$inflector->addRules(array(
    // filter rules:
    ':controller' => array('CamelCaseToUnderscore','StringToLower'),
    ':action'     => array('CamelCaseToUnderscore','StringToLower'),

    // Static rule:
    'suffix'      => 'phtml'
));
```

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.

- `setPluginLoader()` can be used when you have configured your own plugin loader and wish to use it with `Zend_Filter_Inflector`; `getPluginLoader()` 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 Zend_Config with Zend_Filter_Inflector

You can use `Zend_Config` to set rules, filter prefix paths, and other object state in your inflectors, either by passing a `Zend_Config` object to the constructor or `setConfig()`. The following settings may be specified:

- `target` specifies the inflection target.
- `filterPrefixPath` specifies one or more filter prefix/path pairs for use with the inflector.
- `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 identifiying 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 18.3. Using Zend_Config with Zend_Filter_Inflector

```
// With the constructor:
$config    = new Zend_Config($options);
$inflector = new Zend_Filter_Inflector($config);

// Or with setConfig():
$inflector = new Zend_Filter_Inflector();
$inflector->setConfig($config);
```

Chapter 19. Zend_Form

Zend_Form

Zend_Form simplifies form creation and handling in your web application. It accomplishes the following goals:

- Element input filtering and validation
- Element ordering
- Element and Form rendering, including escaping
- Element and form grouping
- Element and form-level configuration

It heavily leverages other Zend Framework components to accomplish its goals, including `Zend_Config`, `Zend_Validate`, `Zend_Filter`, `Zend_Loader_PluginLoader`, and optionally `Zend_View`.

Zend_Form Quick Start

This quick start guide is intended to cover the basics of creating, validating, and rendering forms using `Zend_Form`.

Create a form object

Creating a form object is very simple: simply instantiate `Zend_Form`:

```
$form = new Zend_Form;
```

For advanced use cases, you may want to create a `Zend_Form` subclass, but for simple forms, you can create a form programmatically using a `Zend_Form` object.

If you wish to specify the form action and method (always good ideas), you can do so with the `setAction()` and `setMethod()` accessors:

```
$form->setAction('/resource/process')
     ->setMethod('post');
```

The above code sets the form action to the partial URL "/resource/process" and the form method to HTTP POST. This will be reflected during final rendering.

You can set additional HTML attributes for the `<form>` tag by using the setAttrib() or setAttribs() methods. For instance, if you wish to set the id, set the "id" attribute:

```
$form->setAttrib('id', 'login');
```

Add elements to the form

A form is nothing without its elements. `Zend_Form` ships with some default elements that render XHTML via `Zend_View` helpers. These are as follows:

- button
- checkbox (or many checkboxes at once with multiCheckbox)
- hidden
- image
- password
- radio
- reset
- select (both regular and multi-select types)
- submit
- text
- textarea

You have two options for adding elements to a form: you can instantiate concrete elements and pass in these objects, or you can pass in simply the element type and have `Zend_Form` instantiate an object of the correct type for you.

As some examples:

```
// Instantiating an element and passing to the form object:
$form->addElement(new Zend_Form_Element_Text('username'));

// Passing a form element type to the form object:
$form->addElement('text', 'username');
```

By default, these do not have any validators or filters. This means you will need to configure your elements with minimally validators, and potentially filters. You can either do this (a) before you pass the element to the form, (b) via configuration options passed in when creating an element via `Zend_Form`, or (c) by pulling the element from the form object and configuring it after the fact.

Let's first look at creating validators for a concrete element instance. You can either pass in `Zend_Validate_*` objects, or the name of a validator to utilize:

```
$username = new Zend_Form_Element_Text('username');

// Passing a Zend_Validate_* object:
$username->addValidator(new Zend_Validate_Alnum());

// Passing a validator name:
$username->addValidator('alnum');
```

When using this second option, if the validator can accept constructor arguments, you can pass those in an array as the third parameter:

```
// Pass a pattern
$username->addValidator('regex', false, array('/^[a-z]/i'));
```

(The second parameter is used to indicate whether or not failure of this validator should prevent later validators from running; by default, this is false.)

You may also wish to specify an element as required. This can be done using either an accessor or by passing an option when creating the element. In the former case:

```
// Make this element required:
$username->setRequired(true);
```

When an element is required, a 'NotEmpty' validator is added to the top of the validator chain, ensuring that the element has a value when required.

Filters are registered in basically the same way as validators. For illustration purposes, let's add a filter to lowercase the final value:

```
$username->addFilter('StringtoLower');
```

So, our final element setup might look like this:

```
$username->addValidator('alnum')
         ->addValidator('regex', false, array('/^[a-z]/'))
         ->setRequired(true)
         ->addFilter('StringToLower');

// or, more compactly:
$username->addValidators(array('alnum',
        array('regex', false, '/^[a-z]/i')
    ))
    ->setRequired(true)
    ->addFilters(array('StringToLower'));
```

Simple as this is, doing this for every single element in a form can be a bit tedious. Let's try option (b) from above. When we create a new element using `Zend_Form::addElement()` as a factory, we can optionally pass in configuration options. These can include validators and filters to utilize. So, to do all of the above implicitly, try the following:

```
$form->addElement('text', 'username', array(
    'validators' => array(
        'alnum',
        array('regex', false, '/^[a-z]/i')
    ),
    'required' => true,
    'filters'  => array('StringToLower'),
));
```

Note

If you find you are setting up elements using the same options in many locations, you may want to consider creating your own `Zend_Form_Element` subclass and utilizing that class instead; this will save you typing in the long-run.

Render a form

Rendering a form is simple. Most elements use a `Zend_View` helper to render themselves, and thus need a view object in order to render. Other than that, you have two options: use the form's render() method, or simply echo it.

```
// Explicitly calling render(), and passing an optional view object:
echo $form->render($view);

// Assuming a view object has been previously set via setView():
echo $form;
```

By default, `Zend_Form` and `Zend_Form_Element` will attempt to use the view object initialized in the `ViewRenderer`, which means you won't need to set the view manually when using the Zend Framework MVC. Rendering a form in a view script is then as simple as:

```
<?= $this->form ?>
```

Under the hood, `Zend_Form` uses "decorators" to perform rendering. These decorators can replace content, append content, or prepend content, and have full introspection to the element passed to them. As a result, you can combine multiple decorators to achieve custom effects. By default, `Zend_Form_Element` actually combines four decorators to achieve its output; setup looks something like this:

```
$element->addDecorators(array(
    'ViewHelper',
    'Errors',
    array('HtmlTag', array('tag' => 'dd')),
    array('Label', array('tag' => 'dt')),
));
```

(Where <HELPERNAME> is the name of a view helper to use, and varies based on the element.)

The above creates output like the following:

```
<dt><label for="username" class="required">Username</dt>
<dd>
    <input type="text" name="username" value="123-abc" />
    <ul class="errors">
        <li>'123-abc' has not only alphabetic and digit characters</li>
        <li>'123-abc' does not match against pattern '/^[a-z]/i'</li>
    </ul>
</dd>
```

(Albeit not with the same formatting.)

You can change the decorators used by an element if you wish to have different output; see the section on decorators for more information.

The form itself simply loops through the elements, and dresses them in an HTML `<form>`. The action and method you provided when setting up the form are provided to the `<form>` tag, as are any attributes you set via `setAttribs()` and family.

Elements are looped either in the order in which they were registered, or, if your element contains an order attribute, that order will be used. You can set an element's order using:

```
$element->setOrder(10);
```

Or, when creating an element, by passing it as an option:

```
$form->addElement('text', 'username', array('order' => 10));
```

Check if a form is valid

After a form is submitted, you will need to check and see if it passes validations. Each element is checked against the data provided; if a key matching the element name is not present, and the item is marked as required, validations are run with a null value.

Where does the data come from? You can use `$_POST` or `$_GET`, or any other data source you might have at hand (web service requests, for instance):

```
if ($form->isValid($_POST)) {
    // success!
} else {
    // failure!
}
```

With AJAX requests, you sometimes can get away with validating single element, or groups of elements. `isValidPartial()` will validate a partial form. Unlike `isValid()`, however, if a particular key is not present, it will not run validations for that particular element:

```
if ($form->isValidPartial($_POST)) {
    // elements present all passed validations
} else {
    // one or more elements tested failed validations
}
```

An additional method, `processAjax()`, can also be used for validating partial forms. Unlike `isValidPartial()`, it returns a JSON-formatted string containing error messages on failure.

Assuming your validations have passed, you can now fetch the filtered values:

```
$values = $form->getValues();
```

If you need the unfiltered values at any point, use:

```
$unfiltered = $form->getUnfilteredValues();
```

Get error status

So, your form failed validations? In most cases, you can simply render the form again, and errors will be displayed when using the default decorators:

```
if (!$form->isValid($_POST)) {
    echo $form;

    // or assign to the view object and render a view...
    $this->view->form = $form;
    return $this->render('form');
}
```

If you want to inspect the errors, you have two methods. `getErrors()` returns an associative array of element names / codes (where codes is an array of error codes). `getMessages()` returns an associative array of element names / messages (where messages is an associative array of error code / error message pairs). If a given element does not have any errors, it will not be included in the array.

Putting it together

Let's build a simple login form. It will need elements representing:

- username
- password
- submit

For our purposes, let's assume that a valid username should be alphanumeric characters only, start with a letter, have a minimum length of 6, and maximum length of 20; they will be normalized to lowercase. Passwords must be a minimum of 6 characters. We'll simply toss the submit value when done, so it can remain unvalidated.

We'll use the power of `Zend_Form`'s configuration options to build the form:

```
$form = new Zend_Form();
$form->setAction('/user/login')
     ->setMethod('post');

// Create and configure username element:
$username = $form->createElement('text', 'username');
$username->addValidator('alnum')
         ->addValidator('regex', false, array('/^[a-z]+/'))
         ->addValidator('stringLength', false, array(6, 20))
         ->setRequired(true)
         ->addFilter('StringToLower');

// Create and configure password element:
$password = $form->createElement('password', 'password');
$password->addValidator('StringLength', false, array(6))
         ->setRequired(true);

// Add elements to form:
$form->addElement($username)
     ->addElement($password)
```

```
    // use addElement() as a factory to create 'Login' button:
    ->addElement('submit', 'login', array('label' => 'Login'));
```

Next, we'll create a controller for handling this:

```
class UserController extends Zend_Controller_Action
{
    public function getForm()
    {
        // create form as above
        return $form;
    }

    public function indexAction()
    {
        // render user/form.phtml
        $this->view->form = $this->getForm();
        $this->render('form');
    }

    public function loginAction()
    {
        if (!$this->getRequest()->isPost()) {
            return $this->_forward('index');
        }
        $form = $this->getForm();
        if (!$form->isValid($_POST)) {
            // Failed validation; redisplay form
            $this->view->form = $form;
            return $this->render('form');
        }

        $values = $form->getValues();
        // now try and authenticate....
    }
}
```

And a view script for displaying the form:

```
<h2>Please login:</h2>
<?= $this->form ?>
```

As you'll note from the controller code, there's more work to do: while the submission may be valid, you may still need to do some authentication using `Zend_Auth`, for instance.

Using a Zend_Config object

All `Zend_Form` classes are configurable using `Zend_Config`; you can either pass a `Zend_Config` object to the constructor or pass it in via `setConfig()`. Let's look at how we might create the above form using an INI file. First, let's follow the recommendations, and place our configurations into sections reflecting the release location, and focus on the 'development' section. Next, we'll setup a section for the given controller ('user'), and a key for the form ('login'):

```
[development]
; general form metainformation
user.login.action = "/user/login"
user.login.method = "post"

; username element
user.login.elements.username.type = "text"
user.login.elements.username.options.validators.alnum.validator = "alnum"
user.login.elements.username.options.validators.regex.validator = "regex"
user.login.elements.username.options.validators.regex.options.pattern = "/^[a-z]/i
user.login.elements.username.options.validators.strlen.validator = "StringLength"
user.login.elements.username.options.validators.strlen.options.min = "6"
user.login.elements.username.options.validators.strlen.options.max = "20"
user.login.elements.username.options.required = true
user.login.elements.username.options.filters.lower.filter = "StringToLower"

; password element
user.login.elements.password.type = "password"
user.login.elements.password.options.validators.strlen.validator = "StringLength"
user.login.elements.password.options.validators.strlen.options.min = "6"
user.login.elements.password.options.required = true

; submit element
user.login.elements.submit.type = "submit"
```

You could then pass this to the form constructor:

```
$config = new Zend_Config_Ini($configFile, 'development');
$form   = new Zend_Form($config->user->login);
```

and the entire form will be defined.

Conclusion

Hopefully with this little tutorial, you should now be well on your way to unlocking the power and flexibility of `Zend_Form`. Read on for more in-depth information!

Creating Form Elements Using Zend_Form_Element

A form is made of elements, which typically correspond to HTML form input. Zend_Form_Element encapsulates single form elements, with the following areas of responsibility:

- validation (is submitted data valid?)
  - capturing of validation error codes and messages
- filtering (how is the element escaped or normalized prior to validation and/or for output?)
- rendering (how is the element displayed?)
- metadata and attributes (what information further qualifies the element?)

The base class, `Zend_Form_Element`, has reasonable defaults for many cases, but it is best to extend the class for commonly used special purpose elements. Additionally, Zend Framework ships with a number of standard XHTML elements; you can read about them in the Standard Elements chapter.

Plugin Loaders

`Zend_Form_Element` makes use of Zend_Loader_PluginLoader to allow developers to specify locations of alternate validators, filters, and decorators. Each has its own plugin loader associated with it, and general accessors are used to retrieve and modify each.

The following loader types are used with the various plugin loader methods: 'validate', 'filter', and 'decorator'. The type names are case insensitive.

The methods used to interact with plugin loaders are as follows:

- `setPluginLoader($loader, $type)`: `$loader` is the plugin loader object itself, while `$type` is one of the types specified above. This sets the plugin loader for the given type to the newly specified loader object.
- `getPluginLoader($type)`: retrieves the plugin loader associated with `$type`.
- `addPrefixPath($prefix, $path, $type = null)`: adds a prefix/path association to the loader specified by `$type`. If `$type` is null, it will attempt to add the path to all loaders, by appending the prefix with each of "_Validate", "_Filter", and "_Decorator"; and appending the path with "Validate/", "Filter/", and "Decorator/". If you have all your extra form element classes under a common hierarchy, this is a convenience method for setting the base prefix for them.
- `addPrefixPaths(array $spec)`: allows you to add many paths at once to one or more plugin loaders. It expects each array item to be an array with the keys 'path', 'prefix', and 'type'.

Custom validators, filters, and decorators are an easy way to share functionality between forms and encapsulate custom functionality.

Example 19.1. Custom Label

One common use case for plugins is to provide replacements for standard classes. For instance, if you want to provide a different implementation of the 'Label' decorator -- for instance, to always append a colon -- you could create your own 'Label' decorator with your own class prefix, and then add it to your prefix path.

Let's start with a custom Label decorator. We'll give it the class prefix "My_Decorator", and the class itself will be in the file "My/Decorator/Label.php".

```
class My_Decorator_Label extends Zend_Form_Decorator_Abstract
{
    protected $_placement = 'PREPEND';

    public function render($content)
    {
        if (null === ($element = $this->getElement())) {
            return $content;
        }
        if (!method_exists($element, 'getLabel')) {
            return $content;
        }

        $label = $element->getLabel() . ':';

        if (null === ($view = $element->getView())) {
            return $this->renderLabel($content, $label);
        }

        $label = $view->formLabel($element->getName(), $label);

        return $this->renderLabel($content, $label);
    }

    public function renderLabel($content, $label)
    {
        $placement = $this->getPlacement();
        $separator = $this->getSeparator();

        switch ($placement) {
            case 'APPEND':
                return $content . $separator . $label;
            case 'PREPEND':
            default:
                return $label . $separator . $content;
        }
    }
}
```

Now we can tell the element to use this plugin path when looking for decorators:

```
$element->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
```

Alternately, we can do that at the form level to ensure all decorators use this path:

```
$form->addElementPrefixPath('My_Decorator', 'My/Decorator/', 'decorator');
```

With this path added, when you add a decorator, the 'My/Decorator/' path will be searched first to see if the decorator exists there. As a result, 'My_Decorator_Label' will now be used when the 'Label' decorator is requested.

Filters

It's often useful and/or necessary to perform some normalization on input prior to validation – for instance, you may want to strip out all HTML, but run your validations on what remains to ensure the submission is valid. Or you may want to trim empty space surrounding input so that a StringLength validator will not return a false positive. These operations may be performed using `Zend_Filter`, and `Zend_Form_Element` has support for filter chains, allowing you to specify multiple, sequential filters to utilize. Filtering happens both during validation and when you retrieve the element value via `getValue()`:

```
$filtered = $element->getValue();
```

Filters may be added to the chain in two ways:

- passing in a concrete filter instance
- providing a filter name – either a short name or fully qualified class name

Let's see some examples:

```
// Concrete filter instance:
$element->addFilter(new Zend_Filter_Alnum());

// Fully qualified class name:
$element->addFilter('Zend_Filter_Alnum');

// Short filter name:
$element->addFilter('Alnum');
$element->addFilter('alnum');
```

Short names are typically the filter name minus the prefix. In the default case, this will mean minus the 'Zend_Filter_' prefix. Additionally, the first letter need not be upper-cased.

Using Custom Filter Classes

If you have your own set of filter classes, you can tell `Zend_Form_Element` about these using `addPrefixPath()`. For instance, if you have filters under the 'My_Filter' prefix, you can tell `Zend_Form_Element` about this as follows:

```
$element->addPrefixPath('My_Filter', 'My/Filter/', 'filter');
```

(Recall that the third argument indicates which plugin loader on which to perform the action.)

If at any time you need the unfiltered value, use the `getUnfilteredValue()` method:

```
$unfiltered = $element->getUnfilteredValue();
```

For more information on filters, see the Zend_Filter documentation.

Methods associated with filters include:

- `addFilter($nameOfFilter, array $options = null)`
- `addFilters(array $filters)`
- `setFilters(array $filters)` (overwrites all filters)
- `getFilter($name)` (retrieve a filter object by name)
- `getFilters()` (retrieve all filters)
- `removeFilter($name)` (remove filter by name)
- `clearFilters()` (remove all filters)

Validators

If you subscribe to the security mantra of "filter input, escape output," you'll want to validate ("filter input") your form input. In `Zend_Form`, each element includes its own validator chain, consisting of `Zend_Validate_*` validators.

Validators may be added to the chain in two ways:

- passing in a concrete validator instance
- providing a validator name – either a short name or fully qualified class name

Let's see some examples:

```
// Concrete validator instance:
$element->addValidator(new Zend_Validate_Alnum());
```

```
// Fully qualified class name:
$element->addValidator('Zend_Validate_Alnum');

// Short validator name:
$element->addValidator('Alnum');
$element->addValidator('alnum');
```

Short names are typically the validator name minus the prefix. In the default case, this will mean minus the 'Zend_Validate_' prefix. Additionally, the first letter need not be upper-cased.

Using Custom Validator Classes

If you have your own set of validator classes, you can tell `Zend_Form_Element` about these using `addPrefixPath()`. For instance, if you have validators under the 'My_Validator' prefix, you can tell `Zend_Form_Element` about this as follows:

```
$element->addPrefixPath('My_Validator', 'My/Validator/', 'validate');
```

(Recall that the third argument indicates which plugin loader on which to perform the action.)

If failing a particular validation should prevent later validators from firing, pass boolean `true` as the second parameter:

```
$element->addValidator('alnum', true);
```

If you are using a string name to add a validator, and the validator class accepts arguments to the constructor, you may pass these to the third parameter of `addValidator()` as an array:

```
$element->addValidator('StringLength', false, array(6, 20));
```

Arguments passed in this way should be in the order in which they are defined in the constructor. The above example will instantiate the `Zend_Validate_StringLenth` class with its `$min` and `$max` parameters:

```
$validator = new Zend_Validate_StringLength(6, 20);
```

Providing Custom Validator Error Messages

Some developers may wish to provide custom error messages for a validator. `Zend_Form_Element::addValidator()`'s `$options` argument allows you to do so by providing the key 'messages' and setting it to an array of key/value pairs for setting the message templates. You will need to know the error codes of the various validation error types for the particular validator.

A better option is to use a `Zend_Translate_Adapter` with your form. Error codes are automatically passed to the adapter by the default Errors decorator; you can then specify your own error message strings by setting up translations for the various error codes of your validators.

You can also set many validators at once, using `addValidators()`. The basic usage is to pass an array of arrays, with each array containing 1 to 3 values, matching the constructor of `addValidator()`:

```
$element->addValidators(array(
    array('NotEmpty', true),
    array('alnum'),
    array('stringLength', false, array(6, 20)),
));
```

If you want to be more verbose or explicit, you can use the array keys 'validator', 'breakChainOnFailure', and 'options':

```
$element->addValidators(array(
    array(
        'validator'           => 'NotEmpty',
        'breakChainOnFailure' => true),
    array('validator' => 'alnum'),
    array(
        'validator' => 'stringLength',
        'options'   => array(6, 20)),
));
```

This usage is good for illustrating how you could then configure validators in a config file:

```
element.validators.notempty.validator = "NotEmpty"
element.validators.notempty.breakChainOnFailure = true
element.validators.alnum.validator = "Alnum"
element.validators.strlen.validator = "StringLength"
element.validators.strlen.options.min = 6
element.validators.strlen.options.max = 20
```

Notice that every item has a key, whether or not it needs one; this is a limitation of using configuration files -- but it also helps make explicit what the arguments are for. Just remember that any validator options must be specified in order.

To validate an element, pass the value to `isValid()`:

```
if ($element->isValid($value)) {
    // valid
} else {
    // invalid
}
```

Validation Operates On Filtered Values

`Zend_Form_Element::isValid()` filters values through the provided filter chain prior to validation. See the Filters section for more information.

Validation Context

`Zend_Form_Element::isValid()` supports an additional argument, `$context`. `Zend_Form::isValid()` passes the entire array of data being processed to `$context` when validating a form, and `Zend_Form_Element::isValid()`, in turn, passes it to each validator. This means you can write validators that are aware of data passed to other form elements. As an example, consider a standard registration form that has fields for both password and a password confirmation; one validation would be that the two fields match. Such a validator might look like the following:

```
class My_Validate_PasswordConfirmation extends Zend_Validate_Abstract
{
    const NOT_MATCH = 'notMatch';

    protected $_messageTemplates = array(
        self::NOT_MATCH => 'Password confirmation does not match'
    );

    public function isValid($value, $context = null)
    {
        $value = (string) $value;
        $this->_setValue($value);

        if (is_array($context)) {
            if (isset($context['password_confirm'])
                && ($value == $context['password_confirm']))
            {
                return true;
            }
        } elseif (is_string($context) && ($value == $context)) {
            return true;
        }

        $this->_error(self::NOT_MATCH);
        return false;
    }
}
```

Validators are processed in order. Each validator is processed, unless a validator created with a true `breakChainOnFailure` value fails its validation. Be sure to specify your validators in a reasonable order.

After a failed validation, you can retrieve the error codes and messages from the validator chain:

```
$errors   = $element->getErrors();
$messages = $element->getMessages();
```

(Note: error messages returned are an associative array of error code / error message pairs.)

In addition to validators, you can specify that an element is required, using `setRequired(true)`. By default, this flag is false, meaning that your validator chain will be skipped if no value is passed to `isValid()`. You can modify this behavior in a number of ways:

- By default, when an element is required, a flag, 'allowEmpty', is also true. This means that if a value evaluating to empty is passed to `isValid()`, the validators will be skipped. You can toggle this flag using the accessor `setAllowEmpty($flag)`; when the flag is false, then if a value is passed, the validators will still run.
- By default, if an element is required, but does not contain a 'NotEmpty' validator, `isValid()` will add one to the top of the stack, with the `breakChainOnFailure` flag set. This makes the required flag have semantic meaning: if no value is passed, we immediately invalidate the submission and notify the user, and prevent other validators from running on what we already know is invalid data.

  If you do not want this behavior, you can turn it off by passing a false value to `setAutoInsertNotEmptyValidator($flag)`; this will prevent `isValid()` from placing the 'NotEmpty' validator in the validator chain.

For more information on validators, see the Zend_Validate documentation.

Using Zend_Form_Elements as general-purpose validators

`Zend_Form_Element` implements `Zend_Validate_Interface`, meaning an element may also be used as a validator in other, non-form related validation chains.

Methods associated with validation include:

- `setRequired($flag)` and `isRequired()` allow you to set and retrieve the status of the 'required' flag. When set to boolean `true`, this flag requires that the element be in the data processed by `Zend_Form`.
- `setAllowEmpty($flag)` and `getAllowEmpty()` allow you to modify the behaviour of optional elements (i.e., elements where the required flag is false). When the 'allow empty' flag is true, empty values will not be passed to the validator chain.
- `setAutoInsertNotEmptyValidator($flag)` allows you to specify whether or not a 'NotEmpty' validator will be prepended to the validator chain when the element is required. By default, this flag is true.
- `addValidator($nameOrValidator, $breakChainOnFailure = false, array $options = null)`

- `addValidators(array $validators)`
- `setValidators(array $validators)` (overwrites all validators)
- `getValidator($name)` (retrieve a validator object by name)
- `getValidators()` (retrieve all validators)
- `removeValidator($name)` (remove validator by name)
- `clearValidators()` (remove all validators)

Custom Error Messages

At times, you may want to specify one or more specific error messages to use instead of the error messages generated by the validators attached to your element. Additionally, at times you may want to mark the element invalid yourself. As of 1.6.0, this functionality is possible via the following methods.

- `addErrorMessage($message)`: add an error message to display on form validation errors. You may call this more than once, and new messages are appended to the stack.
- `addErrorMessages(array $messages)`: add multiple error messages to display on form validation errors.
- `setErrorMessages(array $messages)`: add multiple error messages to display on form validation errors, overwriting all previously set error messages.
- `getErrorMessages()`: retrieve the list of custom error messages that have been defined.
- `clearErrorMessages()`: remove all custom error messages that have been defined.
- `markAsError()`: mark the element as having failed validation.
- `hasErrors()`: determine whether the element has either failed validation or been marked as invalid.
- `addError($message)`: add a message to the custom error messages stack and flag the element as invalid.
- `addErrors(array $messages)`: add several messages to the custom error messages stack and flag the element as invalid.
- `setErrors(array $messages)`: overwrite the custom error messages stack with the provided messages and flag the element as invalid.

All errors set in this fashion may be translated. Additionally, you may insert the placeholder "%value%" to represent the element value; this current element value will be substituted when the error messages are retrieved.

Decorators

One particular pain point for many web developers is the creation of the XHTML forms themselves. For each element, the developer needs to create markup for the element itself, typically a label, and, if they're being nice to their users, markup for displaying validation error messages. The more elements on the page, the less trivial this task becomes.

`Zend_Form_Element` tries to solve this issue through the use of "decorators". Decorators are simply classes that have access to the element and a method for rendering content. For more information on how decorators work, please see the section on Zend_Form_Decorator.

The default decorators used by `Zend_Form_Element` are:

- *ViewHelper*: specifies a view helper to use to render the element. The 'helper' element attribute can be used to specify which view helper to use. By default, `Zend_Form_Element` specifies the 'formText' view helper, but individual subclasses specify different helpers.
- *Errors*: appends error messages to the element using `Zend_View_Helper_FormErrors`. If none are present, nothing is appended.
- *HtmlTag*: wraps the element and errors in an HTML <dd> tag.
- *Label*: prepends a label to the element using `Zend_View_Helper_FormLabel`, and wraps it in a <dt> tag. If no label is provided, just the definition term tag is rendered.

Default Decorators Do Not Need to Be Loaded

By default, the default decorators are loaded during object initialization. You can disable this by passing the 'disableLoadDefaultDecorators' option to the constructor:

```
$element = new Zend_Form_Element('foo',
                                 array('disableLoadDefaultDecorators' =>
                                      true)
                                );
```

This option may be mixed with any other options you pass, both as array options or in a `Zend_Config` object.

Since the order in which decorators are registered matters -- first decorator registered is executed first -- you will need to make sure you register your decorators in an appropriate order, or ensure that you set the placement options in a sane fashion. To give an example, here is the code that registers the default decorators:

```
$this->addDecorators(array(
    array('ViewHelper'),
    array('Errors'),
    array('HtmlTag', array('tag' => 'dd')),
    array('Label', array('tag' => 'dt')),
));
```

The initial content is created by the 'ViewHelper' decorator, which creates the form element itself. Next, the 'Errors' decorator fetches error messages from the element, and, if any are present, passes them to the 'FormErrors' view helper to render. The next decorator, 'HtmlTag', wraps the element and errors in an HTML <dd> tag. Finally, the last decorator, 'label', retrieves the element's label and passes it to the 'FormLabel' view helper, wrapping it in an HTML <dt> tag; the value is prepended to the content by default. The resulting output looks basically like this:

```
<dt><label for="foo" class="optional">Foo</label></dt>
<dd>
    <input type="text" name="foo" id="foo" value="123" />
    <ul class="errors">
        <li>"123" is not an alphanumeric value</li>
    </ul>
</dd>
```

For more information on decorators, read the Zend_Form_Decorator section.

Using Multiple Decorators of the Same Type

Internally, `Zend_Form_Element` uses a decorator's class as the lookup mechanism when retrieving decorators. As a result, you cannot register multiple decorators of the same type; subsequent decorators will simply overwrite those that existed before.

To get around this, you can use *aliases*. Instead of passing a decorator or decorator name as the first argument to `addDecorator()`, pass an array with a single element, with the alias pointing to the decorator object or name:

```
// Alias to 'FooBar':
$element->addDecorator(array('FooBar' => 'HtmlTag'),
                       array('tag' => 'div'));

// And retrieve later:
$decorator = $element->getDecorator('FooBar');
```

In the `addDecorators()` and `setDecorators()` methods, you will need to pass the 'decorator' option in the array representing the decorator:

```
// Add two 'HtmlTag' decorators, aliasing one to 'FooBar':
$element->addDecorators(
    array('HtmlTag', array('tag' => 'div')),
    array(
        'decorator' => array('FooBar' => 'HtmlTag'),
        'options' => array('tag' => 'dd')
    ),
);

// And retrieve later:
$htmlTag = $element->getDecorator('HtmlTag');
$fooBar  = $element->getDecorator('FooBar');
```

Methods associated with decorators include:

- `addDecorator($nameOrDecorator, array $options = null)`

- `addDecorators(array $decorators)`
- `setDecorators(array $decorators)` (overwrites all decorators)
- `getDecorator($name)` (retrieve a decorator object by name)
- `getDecorators()` (retrieve all decorators)
- `removeDecorator($name)` (remove decorator by name)
- `clearDecorators()` (remove all decorators)

Metadata and Attributes

`Zend_Form_Element` handles a variety of attributes and element metadata. Basic attributes include:

- *name*: the element name. Uses the `setName()` and `getName()` accessors.
- *label*: the element label. Uses the `setLabel()` and `getLabel()` accessors.
- *order*: the index at which an element should appear in the form. Uses the `setOrder()` and `getOrder()` accessors.
- *value*: the current element value. Uses the `setValue()` and `getValue()` accessors.
- *description*: a description of the element; often used to provide tooltip or javascript contextual hinting describing the purpose of the element. Uses the `setDescription()` and `getDescription()` accessors.
- *required*: flag indicating whether or not the element is required when performing form validation. Uses the `setRequired()` and `getRequired()` accessors. This flag is false by default.
- *allowEmpty*: flag indicating whether or not a non-required (optional) element should attempt to validate empty values. When true, and the required flag is false, empty values are not passed to the validator chain, and presumed true. Uses the `setAllowEmpty()` and `getAllowEmpty()` accessors. This flag is true by default.
- *autoInsertNotEmptyValidator*: flag indicating whether or not to insert a 'NotEmpty' validator when the element is required. By default, this flag is true. Set the flag with `setAutoInsertNotEmptyValidator($flag)` and determine the value with `autoInsertNotEmptyValidator()`.

Form elements may require additional metadata. For XHTML form elements, for instance, you may want to specify attributes such as the class or id. To facilitate this are a set of accessors:

- *setAttrib($name, $value)*: add an attribute
- *setAttribs(array $attribs)*: like addAttribs(), but overwrites
- *getAttrib($name)*: retrieve a single attribute value
- *getAttribs()*: retrieve all attributes as key/value pairs

Most of the time, however, you can simply access them as object properties, as `Zend_Form_Element` utilizes overloading to facilitate access to them:

```
// Equivalent to $element->setAttrib('class', 'text'):
$element->class = 'text;
```

By default, all attributes are passed to the view helper used by the element during rendering, and rendered as HTML attributes of the element tag.

Standard Elements

`Zend_Form` ships with a number of standard elements; please read the Standard Elements chapter for full details.

Zend_Form_Element Methods

`Zend_Form_Element` has many, many methods. What follows is a quick summary of their signatures, grouped by type:

- Configuration:
  - `setOptions(array $options)`
  - `setConfig(Zend_Config $config)`
- I18n:
  - `setTranslator(Zend_Translate_Adapter $translator = null)`
  - `getTranslator()`
  - `setDisableTranslator($flag)`
  - `translatorIsDisabled()`
- Properties:
  - `setName($name)`
  - `getName()`
  - `setValue($value)`
  - `getValue()`
  - `getUnfilteredValue()`
  - `setLabel($label)`
  - `getLabel()`
  - `setDescription($description)`
  - `getDescription()`
  - `setOrder($order)`

  - `getOrder()`
  - `setRequired($flag)`
  - `getRequired()`
  - `setAllowEmpty($flag)`
  - `getAllowEmpty()`
  - `setAutoInsertNotEmptyValidator($flag)`
  - `autoInsertNotEmptyValidator()`
  - `setIgnore($flag)`
  - `getIgnore()`
  - `getType()`
  - `setAttrib($name, $value)`
  - `setAttribs(array $attribs)`
  - `getAttrib($name)`
  - `getAttribs()`
- Plugin loaders and paths:
  - `setPluginLoader(Zend_Loader_PluginLoader_Interface $loader, $type)`
  - `getPluginLoader($type)`
  - `addPrefixPath($prefix, $path, $type = null)`
  - `addPrefixPaths(array $spec)`
- Validation:
  - `addValidator($validator, $breakChainOnFailure = false, $options = array())`
  - `addValidators(array $validators)`
  - `setValidators(array $validators)`
  - `getValidator($name)`
  - `getValidators()`
  - `removeValidator($name)`
  - `clearValidators()`
  - `isValid($value, $context = null)`
  - `getErrors()`

  - `getMessages()`
- Filters:
  - `addFilter($filter, $options = array())`
  - `addFilters(array $filters)`
  - `setFilters(array $filters)`
  - `getFilter($name)`
  - `getFilters()`
  - `removeFilter($name)`
  - `clearFilters()`
- Rendering:
  - `setView(Zend_View_Interface $view = null)`
  - `getView()`
  - `addDecorator($decorator, $options = null)`
  - `addDecorators(array $decorators)`
  - `setDecorators(array $decorators)`
  - `getDecorator($name)`
  - `getDecorators()`
  - `removeDecorator($name)`
  - `clearDecorators()`
  - `render(Zend_View_Interface $view = null)`

Configuration

`Zend_Form_Element`'s constructor accepts either an array of options or a `Zend_Config` object containing options, and it can also be configured using either `setOptions()` or `setConfig()`. Generally speaking, keys are named as follows:

- If 'set' + key refers to a `Zend_Form_Element` method, then the value provided will be passed to that method.
- Otherwise, the value will be used to set an attribute.

Exceptions to the rule include the following:

- `prefixPath` will be passed to `addPrefixPaths()`
- The following setters cannot be set in this way:

- `setAttrib` (though `setAttribs` *will* work)
- `setConfig`
- `setOptions`
- `setPluginLoader`
- `setTranslator`
- `setView`

As an example, here is a config file that passes configuration for every type of configurable data:

```
[element]
name = "foo"
value = "foobar"
label = "Foo:"
order = 10
required = true
allowEmpty = false
autoInsertNotEmptyValidator = true
description = "Foo elements are for examples"
ignore = false
attribs.id = "foo"
attribs.class = "element"
; sets 'onclick' attribute
onclick = "autoComplete(this, '/form/autocomplete/element')"
prefixPaths.decorator.prefix = "My_Decorator"
prefixPaths.decorator.path = "My/Decorator/"
disableTranslator = 0
validators.required.validator = "NotEmpty"
validators.required.breakChainOnFailure = true
validators.alpha.validator = "alpha"
validators.regex.validator = "regex"
validators.regex.options.pattern = "/^[A-F].*/$"
filters.ucase.filter = "StringToUpper"
decorators.element.decorator = "ViewHelper"
decorators.element.options.helper = "FormText"
decorators.label.decorator = "Label"
```

Custom Elements

You can create your own custom elements by simply extending the `Zend_Form_Element` class. Common reasons to do so include:

- Elements that share common validators and/or filters
- Elements that have custom decorator functionality

There are two methods typically used to extend an element: `init()`, which can be used to add custom initialization logic to your element, and `loadDefaultDecorators()`, which can be used to set a list of default decorators used by your element.

As an example, let's say that all text elements in a form you are creating need to be filtered with `StringTrim`, validated with a common regular expression, and that you want to use a custom decorator you've created for displaying them, 'My_Decorator_TextItem'; additionally, you have a number of standard attributes, including 'size', 'maxLength', and 'class' you wish to specify. You could define such an element as follows:

```
class My_Element_Text extends Zend_Form_Element
{
    public function init()
    {
        $this->addPrefixPath('My_Decorator', 'My/Decorator/', 'decorator')
             ->addFilters('StringTrim')
             ->addValidator('Regex', false, array('/^[a-z0-9]{6,}$/i'))
             ->addDecorator('TextItem')
             ->setAttrib('size', 30)
             ->setAttrib('maxLength', 45)
             ->setAttrib('class', 'text');
    }
}
```

You could then inform your form object about the prefix path for such elements, and start creating elements:

```
$form->addPrefixPath('My_Element', 'My/Element/', 'element')
     ->addElement('foo', 'text');
```

The 'foo' element will now be of type `My_Element_Text`, and exhibit the behaviour you've outlined.

Another method you may want to override when extending `Zend_Form_Element` is the `loadDefault-Decorators()` method. This method conditionally loads a set of default decorators for your element; you may wish to substitute your own decorators in your extending class:

```
class My_Element_Text extends Zend_Form_Element
{
    public function loadDefaultDecorators()
    {
        $this->addDecorator('ViewHelper')
             ->addDecorator('DisplayError')
             ->addDecorator('Label')
             ->addDecorator('HtmlTag',
                            array('tag' => 'div', 'class' => 'element'));
    }
}
```

There are many ways to customize elements; be sure to read the API documentation of `Zend_Form_Element` to know all the methods available.

Creating Forms Using Zend_Form

The `Zend_Form` class is used to aggregate form elements, display groups, and subforms. It can then perform the following actions on those items:

- Validation, including retrieving error codes and messages
- Value aggregation, including populating items and retrieving both filtered and unfiltered values from all items
- Iteration over all items, in the order in which they are entered or based on the order hints retrieved from each item
- Rendering of the entire form, either via a single decorator that peforms custom rendering or by iterating over each item in the form

While forms created with `Zend_Form` may be complex, probably the best use case is for simple forms; it's best use is for Rapid Application Development and prototyping.

At its most basic, you simply instantiate a form object:

```
// Generic form object:
$form = new Zend_Form();

// Custom form object:
$form = new My_Form()
```

You can optionally pass in configuration, which will be used to set object state, as well as to potentially create new elements:

```
// Passing in configuration options:
$form = new Zend_Form($config);
```

`Zend_Form` is iterable, and will iterate over elements, display groups, and subforms, using the order they were registered and any order index each may have. This is useful in cases where you wish to render the elements manually in the appropriate order.

`Zend_Form`'s magic lies in its ability to serve as a factory for elements and display groups, as well as the ability to render itself through decorators.

Plugin Loaders

`Zend_Form` makes use of `Zend_Loader_PluginLoader` to allow developers to specify locations of alternate elements and decorators. Each has its own plugin loader associated with it, and general accessors are used to retrieve and modify each.

The following loader types are used with the various plugin loader methods: 'element' and 'decorator'. The type names are case insensitive.

The methods used to interact with plugin loaders are as follows:

- `setPluginLoader($loader, $type)`: $loader is the plugin loader object itself, while type is one of the types specified above. This sets the plugin loader for the given type to the newly specified loader object.
- `getPluginLoader($type)`: retrieves the plugin loader associated with $type.
- `addPrefixPath($prefix, $path, $type = null)`: adds a prefix/path association to the loader specified by $type. If $type is null, it will attempt to add the path to all loaders, by appending the prefix with each of "_Element" and "_Decorator"; and appending the path with "Element/" and "Decorator/". If you have all your extra form element classes under a common hierarchy, this is a convenience method for setting the base prefix for them.
- `addPrefixPaths(array $spec)`: allows you to add many paths at once to one or more plugin loaders. It expects each array item to be an array with the keys 'path', 'prefix', and 'type'.

Additionally, you can specify prefix paths for all elements and display groups created through a `Zend_Form` instance using the following methods:

- `addElementPrefixPath($prefix, $path, $type = null)`: Just like `addPrefixPath()`, you must specify a class prefix and a path. `$type`, when speified, must be one of the plugin loader types specified by `Zend_Form_Element`; see the element plugins section for more information on valid `$type` values. If no `$type` is specified, the method will assume it is a general prefix for all types.
- `addDisplayGroupPrefixPath($prefix, $path)`: Just like `addPrefixPath()`, you must specify a class prefix and a path; however, since display groups only support decorators as plugins, no `$type` is necessary.

Custom elements and decorators are an easy way to share functionality between forms and encapsulate custom functionality. See the Custom Label example in the elements documentation for an example of how custom elements can be used as replacements for standard classes.

Elements

`Zend_Form` provides several accessors for adding and removing elements from the form. These can take element object instances or serve as factories for instantiating the element objects themselves.

The most basic method for adding an element is `addElement()`. This method can take either an object of type `Zend_Form_Element` (or of a class extending `Zend_Form_Element`), or arguments for building a new element -- including the element type, name, and any configuration options.

As some examples:

```
// Using an element instance:
$element = new Zend_Form_Element_Text('foo');
$form->addElement($element);

// Using a factory
//
// Creates an element of type Zend_Form_Element_Text with the
// name of 'foo':
$form->addElement('text', 'foo');
```

```
// Pass label option to the element:
$form->addElement('text', 'foo', array('label' => 'Foo:'));
```

addElement() Implements Fluent Interface

`addElement()` implements a fluent interface; that is to say, it returns the `Zend_Form` object, and not the element. This is done to allow you to chain together multiple addElement() methods or other form methods that implement the fluent interface (all setters in Zend_Form implement the pattern).

If you wish to return the element instead, use `createElement()`, which is outlined below. Be aware, however, that `createElement()` does not attach the element to the form.

Internally, `addElement()` actually uses `createElement()` to create the element before attaching it to the form.

Once an element has been added to the form, you can retrieve it by name. This can be done either by using the `getElement()` method or by using overloading to access the element as an object property:

```
// getElement():
$foo = $form->getElement('foo');

// As object property:
$foo = $form->foo;
```

Occasionally, you may want to create an element without attaching it to the form (for instance, if you wish to make use of the various plugin paths registered with the form, but wish to later attach the object to a sub form). The `createElement()` method allows you to do so:

```
// $username becomes a Zend_Form_Element_Text object:
$username = $form->createElement('text', 'username');
```

Populating and Retrieving Values

After validating a form, you will typically need to retrieve the values so you can perform other operations, such as updating a database or notifying a web service. You can retrieve all values for all elements using `getValues()`; `getValue($name)` allows you to retrieve a single element's value by element name:

```
// Get all values:
$values = $form->getValues();

// Get only 'foo' element's value:
$value = $form->getValue('foo');
```

Sometimes you'll want to populate the form with specified values prior to rendering. This can be done with either the `setDefaults()` or `populate()` methods:

```
$form->setDefaults($data);
$form->populate($data);
```

On the flip side, you may want to clear a form after populating or validating it; this can be done using the `reset()` method:

```
$form->reset();
```

Global Operations

Occasionally you will want certain operations to affect all elements. Common scenarios include needing to set plugin prefix paths for all elements, setting decorators for all elements, and setting filters for all elements. As examples:

Example 19.2. Setting Prefix Paths for All Elements

You can set prefix paths for all elements by type, or using a global prefix. As some examples:

```
// Set global prefix path:
// Creates paths for prefixes My_Foo_Filter, My_Foo_Validate,
// and My_Foo_Decorator
$form->addElementPrefixPath('My_Foo', 'My/Foo/');

// Just filter paths:
$form->addElementPrefixPath('My_Foo_Filter',
                            'My/Foo/Filter',
                            'filter');

// Just validator paths:
$form->addElementPrefixPath('My_Foo_Validate',
                            'My/Foo/Validate',
                            'validate');

// Just decorator paths:
$form->addElementPrefixPath('My_Foo_Decorator',
                            'My/Foo/Decorator',
                            'decorator');
```

Example 19.3. Setting Decorators for All Elements

You can set decorators for all elements. `setElementDecorators()` accepts an array of decorators, just like `setDecorators()`, and will overwrite any previously set decorators in each element. In this example, we set the decorators to simply a ViewHelper and a Label:

```
$form->setElementDecorators(array(
    'ViewHelper',
    'Label'
));
```

Example 19.4. Setting Decorators for Some Elements

You can also set decorators for a subset of elements, either by inclusion or exclusion. The second argument to `setElementDecorators()` may be an array of element names; by default, specifying such an array will set the specified decorators on those elements only. You may also pass a third argument, a flag indicating whether this list of elements is for inclusion or exclusion purposes; if false, it will decorate all elements *except* those in the passed list. As with standard usage of the method, any decorators passed will overwrite any previously set decorators in each element.

In the following snippet, we indicate that we want only the ViewHelper and Label decorators for the 'foo' and 'bar' elements:

```
$form->setElementDecorators(
    array(
        'ViewHelper',
        'Label'
    ),
    array(
        'foo',
        'bar'
    )
);
```

On the flip side, with this snippet, we'll now indicate that we want to use only the ViewHelper and Label decorators for every element *except* the 'foo' and 'bar' elements:

```
$form->setElementDecorators(
    array(
        'ViewHelper',
        'Label'
    ),
    array(
        'foo',
        'bar'
    ),
    false
);
```

Some Decorators are Inappropriate for Some Elements

While `setElementDecorators()` may seem like a good solution, there are some cases where it may actually end up with unexpected results. For example, the various button elements (Submit, Button, Reset) currently use the label as the value of the button, and only use ViewHelper and DtDdWrapper decorators -- preventing an additional label, errors, and hint from being rendered; the example above would duplicate some content (the label).

You can use the inclusion/exclusion array to overcome this issue as noted in the previous example.

So, use this method wisely, and realize that you may need to exclude some elements or manually change some elements' decorators to prevent unwanted output.

Example 19.5. Setting Filters for All Elements

In many cases, you may want to apply the same filter to all elements; a common case is to `trim()` all values:

```
$form->setElementFilters(array('StringTrim'));
```

Methods For Interacting With Elements

The following methods may be used to interact with elements:

- `createElement($element, $name = null, $options = null)`
- `addElement($element, $name = null, $options = null)`
- `addElements(array $elements)`
- `setElements(array $elements)`
- `getElement($name)`
- `getElements()`
- `removeElement($name)`
- `clearElements()`
- `setDefaults(array $defaults)`
- `setDefault($name, $value)`
- `getValue($name)`
- `getValues()`
- `getUnfilteredValue($name)`
- `getUnfilteredValues()`
- `setElementFilters(array $filters)`
- `setElementDecorators(array $decorators)`
- `addElementPrefixPath($prefix, $path, $type = null)`
- `addElementPrefixPaths(array $spec)`

Display Groups

Display groups are a way to create virtual groupings of elements for display purposes. All elements remain accessible by name in the form, but when iterating over the form or rendering, any elements in a display group are rendered together. The most common use case for this is for grouping elements in fieldsets.

The base class for display groups is `Zend_Form_DisplayGroup`. While it can be instantiated directly, it is typically best to use `Zend_Form`'s `addDisplayGroup()` method to do so. This method takes an array of elements as its first argument, and a name for the display group as its second argument. You may optionally pass in an array of options or a `Zend_Config` object as the third argument.

Assuming that the elements 'username' and 'password' are already set in the form, the following code would group these elements in a 'login' display group:

```
$form->addDisplayGroup(array('username', 'password'), 'login');
```

You can access display groups using the `getDisplayGroup()` method, or via overloading using the display group's name:

```
// Using getDisplayGroup():
$login = $form->getDisplayGroup('login');

// Using overloading:
$login = $form->login;
```

Default Decorators Do Not Need to Be Loaded

By default, the default decorators are loaded during object initialization. You can disable this by passing the 'disableLoadDefaultDecorators' option when creating a display group:

```
$form->addDisplayGroup(
    array('foo', 'bar'),
    'foobar',
    array('disableLoadDefaultDecorators' => true)
);
```

This option may be mixed with any other options you pass, both as array options or in a `Zend_Config` object.

Global Operations

Just as with elements, there are some operations which might affect all display groups; these include setting decorators and setting the plugin path in which to look for decorators.

Example 19.6. Setting Decorator Prefix Path for All Display Groups

By default, display groups inherit whichever decorator paths the form uses; however, if they should look in alternate locations, you can use the `addDisplayGroupPrefixPath()` method.

```
$form->addDisplayGroupPrefixPath('My_Foo_Decorator', 'My/Foo/Decorator');
```

Example 19.7. Setting Decorators for All Display Groups

You can set decorators for all display groups. `setDisplayGroupDecorators()` accepts an array of decorators, just like `setDecorators()`, and will overwrite any previously set decorators in each display group. In this example, we set the decorators to simply a fieldset (the FormElements decorator is necessary to ensure that the elements are iterated):

```
$form->setDisplayGroupDecorators(array(
    'FormElements',
    'Fieldset'
));
```

Using Custom Display Group Classes

By default, `Zend_Form` uses the `Zend_Form_DisplayGroup` class for display groups. You may find you need to extend this class in order to provided custom functionality. `addDisplayGroup()` does not allow passing in a concrete instance, but does allow specifying the class to use as one of its options, using the 'displayGroupClass' key:

```
// Use the 'My_DisplayGroup' class
$form->addDisplayGroup(
    array('username', 'password'),
    'user',
    array('displayGroupClass' => 'My_DisplayGroup')
);
```

If the class has not yet been loaded, `Zend_Form` will attempt to do so using `Zend_Loader`.

You can also specify a default display group class to use with the form such that all display groups created with the form object will use that class:

```
// Use the 'My_DisplayGroup' class for all display groups:
$form->setDefaultDisplayGroupClass('My_DisplayGroup');
```

This setting may be specified in configurations as 'defaultDisplayGroupClass', and will be loaded early to ensure all display groups use that class.

Methods for Interacting With Display Groups

The following methods may be used to interact with display groups:

- `addDisplayGroup(array $elements, $name, $options = null)`
- `addDisplayGroups(array $groups)`
- `setDisplayGroups(array $groups)`
- `getDisplayGroup($name)`
- `getDisplayGroups()`
- `removeDisplayGroup($name)`
- `clearDisplayGroups()`
- `setDisplayGroupDecorators(array $decorators)`
- `addDisplayGroupPrefixPath($prefix, $path)`
- `setDefaultDisplayGroupClass($class)`
- `getDefaultDisplayGroupClass($class)`

Zend_Form_DisplayGroup Methods

`Zend_Form_DisplayGroup` has the following methods, grouped by type:

- Configuration:
  - `setOptions(array $options)`
  - `setConfig(Zend_Config $config)`
- Metadata:
  - `setAttrib($key, $value)`
  - `addAttribs(array $attribs)`
  - `setAttribs(array $attribs)`
  - `getAttrib($key)`
  - `getAttribs()`
  - `removeAttrib($key)`
  - `clearAttribs()`
  - `setName($name)`

  - `getName()`
  - `setDescription($value)`
  - `getDescription()`
  - `setLegend($legend)`
  - `getLegend()`
  - `setOrder($order)`
  - `getOrder()`
- Elements:
  - `createElement($type, $name, array $options = array())`
  - `addElement($typeOrElement, $name, array $options = array())`
  - `addElements(array $elements)`
  - `setElements(array $elements)`
  - `getElement($name)`
  - `getElements()`
  - `removeElement($name)`
  - `clearElements()`
- Plugin loaders:
  - `setPluginLoader(Zend_Loader_PluginLoader $loader)`
  - `getPluginLoader()`
  - `addPrefixPath($prefix, $path)`
  - `addPrefixPaths(array $spec)`
- Decorators:
  - `addDecorator($decorator, $options = null)`
  - `addDecorators(array $decorators)`
  - `setDecorators(array $decorators)`
  - `getDecorator($name)`
  - `getDecorators()`
  - `removeDecorator($name)`
  - `clearDecorators()`

- Rendering:
  - `setView(Zend_View_Interface $view = null)`
  - `getView()`
  - `render(Zend_View_Interface $view = null)`
- I18n:
  - `setTranslator(Zend_Translate_Adapter $translator = null)`
  - `getTranslator()`
  - `setDisableTranslator($flag)`
  - `translatorIsDisabled()`

Sub Forms

Sub forms serve several purposes:

- Creating logical element groups. Since sub forms are simply forms, you can validate subforms as individual entities.
- Creating multi-page forms. Since sub forms are simply forms, you can display a separate sub form per page, building up multi-page forms where each form has its own validation logic. Only once all sub forms validate would the form be considered complete.
- Display groupings. Like display groups, sub forms, when rendered as part of a larger form, can be used to group elements. Be aware, however, that the master form object will have no awareness of the elements in sub forms.

A sub form may be a `Zend_Form` object, or, more typically, a `Zend_Form_SubForm` object. The latter contains decorators suitable for inclusion in a larger form (i.e., it does not render additional HTML form tags, but does group elements). To attach a sub form, simply add it to the form and give it a name:

```
$form->addSubForm($subForm, 'subform');
```

You can retrieve a sub form using either `getSubForm($name)` or overloading using the sub form name:

```
// Using getSubForm():
$subForm = $form->getSubForm('subform');

// Using overloading:
$subForm = $form->subform;
```

Sub forms are included in form iteration, though the elements it contains are not.

Global Operations

Like elements and display groups, there are some operations that might need to affect all sub forms. Unlike display groups and elements, however, sub forms inherit most functionality from the master form object, and the only real operation that may need to be performed globally is setting decorators for sub forms. For this purpose, there is the `setSubFormDecorators()` method. In the next example, we'll set the decorator for all subforms to be simply a fieldset (the FormElements decorator is needed to ensure its elements are iterated):

```
$form->setSubFormDecorators(array(
    'FormElements',
    'Fieldset'
));
```

Methods for Interacting With Sub Forms

The following methods may be used to interact with sub forms:

- `addSubForm(Zend_Form $form, $name, $order = null)`
- `addSubForms(array $subForms)`
- `setSubForms(array $subForms)`
- `getSubForm($name)`
- `getSubForms()`
- `removeSubForm($name)`
- `clearSubForms()`
- `setSubFormDecorators(array $decorators)`

Metadata and Attributes

While a form's usefulness primarily derives from the elements it contains, it can also contain other metadata, such as a name (often used as a unique ID in the HTML markup); the form action and method; the number of elements, groups, and sub forms it contains; and arbitrary metadata (usually used to set HTML attributes for the form tag itself).

You can set and retrieve a form's name using the name accessors:

```
// Set the name:
$form->setName('registration');

// Retrieve the name:
$name = $form->getName();
```

To set the action (url to which the form submits) and method (method by which it should submit, e.g., 'POST' or 'GET'), use the action and method accessors:

```
// Set the action and method:
$form->setAction('/user/login')
     ->setMethod('post');
```

You may also specify the form encoding type specifically using the enctype accessors. Zend_Form defines two constants, `Zend_Form::ENCTYPE_URLENCODED` and `Zend_Form::ENCTYPE_MULTIPART`, corresponding to the values 'application/x-www-form-urlencoded' and 'multipart/form-data', respectively; however, you can set this to any arbitrary encoding type.

```
// Set the action, method, and enctype:
$form->setAction('/user/login')
     ->setMethod('post')
     ->setEnctype(Zend_Form::ENCTYPE_MULTIPART);
```

Note

The method, action, and enctype are only used internally for rendering, and not for any sort of validation.

`Zend_Form` implements the `Countable` interface, allowing you to pass it as an argument to count:

```
$numItems = count($form);
```

Setting arbitrary metadata is done through the attribs accessors. Since overloading in `Zend_Form` is used to access elements, display groups, and sub forms, this is the only method for accessing metadata.

```
// Setting attributes:
$form->setAttrib('class', 'zend-form')
     ->addAttribs(array(
         'id'       => 'registration',
         'onSubmit' => 'validate(this)',
     ));

// Retrieving attributes:
$class = $form->getAttrib('class');
$attribs = $form->getAttribs();

// Remove an attribute:
$form->removeAttrib('onSubmit');

// Clear all attributes:
$form->clearAttribs();
```

Decorators

Creating the markup for a form is often a time-consuming task, particularly if you plan on re-using the same markup to show things such as validation errors, submitted values, etc. `Zend_Form`'s answer to this issue is *decorators*.

Decorators for `Zend_Form` objects can be used to render a form. The FormElements decorator will iterate through all items in a form -- elements, display groups, and sub forms -- and render them, returning the result. Additional decorators may then be used to wrap this content, or append or prepend it.

The default decorators for `Zend_Form` are FormElements, HtmlTag (wraps in a definition list), and Form; the equivalent code for creating them is as follows:

```
$form->setDecorators(array(
    'FormElements',
    array('HtmlTag', array('tag' => 'dl')),
    'Form'
));
```

This creates output like the following:

```
<form action="/form/action" method="post">
<dl>
...
</dl>
</form>
```

Any attributes set on the form object will be used as HTML attributes of the `<form>` tag.

Default Decorators Do Not Need to Be Loaded

By default, the default decorators are loaded during object initialization. You can disable this by passing the 'disableLoadDefaultDecorators' option to the constructor:

```
$form = new Zend_Form(array('disableLoadDefaultDecorators' => true));
```

This option may be mixed with any other options you pass, both as array options or in a `Zend_Config` object.

Using Multiple Decorators of the Same Type

Internally, `Zend_Form` uses a decorator's class as the lookup mechanism when retrieving decorators. As a result, you cannot register multiple decorators of the same type; subsequent decorators will simply overwrite those that existed before.

To get around this, you can use aliases. Instead of passing a decorator or decorator name as the first argument to `addDecorator()`, pass an array with a single element, with the alias pointing to the decorator object or name:

```
// Alias to 'FooBar':
$form->addDecorator(array('FooBar' => 'HtmlTag'), array('tag' => 'div'));

// And retrieve later:
$form = $element->getDecorator('FooBar');
```

In the `addDecorators()` and `setDecorators()` methods, you will need to pass the 'decorator' option in the array representing the decorator:

```
// Add two 'HtmlTag' decorators, aliasing one to 'FooBar':
$form->addDecorators(
    array('HtmlTag', array('tag' => 'div')),
    array(
        'decorator' => array('FooBar' => 'HtmlTag'),
        'options' => array('tag' => 'dd')
    ),
);

// And retrieve later:
$htmlTag = $form->getDecorator('HtmlTag');
$fooBar  = $form->getDecorator('FooBar');
```

You may create your own decorators for generating the form. One common use case is if you know the exact HTML you wish to use; your decorator could create the exact HTML and simply return it, potentially using the decorators from individual elements or display groups.

The following methods may be used to interact with decorators:

- `addDecorator($decorator, $options = null)`
- `addDecorators(array $decorators)`
- `setDecorators(array $decorators)`
- `getDecorator($name)`
- `getDecorators()`
- `removeDecorator($name)`

- `clearDecorators()`

Validation

A primary use case for forms is validating submitted data. `Zend_Form` allows you to validate an entire form at once or a partial form, as well as to automate validation responses for XmlHttpRequests (AJAX). If the submitted data is not valid, it has methods for retrieving the various error codes and messages for elements and sub forms failing validations.

To validate a full form, use the `isValid()` method:

```
if (!$form->isValid($_POST)) {
    // failed validation
}
```

`isValid()` will validate every required element, and any unrequired element contained in the submitted data.

Sometimes you may need to validate only a subset of the data; for this, use `isValidPartial($data)`:

```
if (!$form->isValidPartial($data)) {
    // failed validation
}
```

`isValidPartial()` only attempts to validate those items in the data for which there are matching elements; if an element is not represented in the data, it is skipped.

When validating elements or groups of elements for an AJAX request, you will typically be validating a subset of the form, and want the response back in JSON. `processAjax()` does precisely that:

```
$json = $form->processAjax($data);
```

You can then simply send the JSON response to the client. If the form is valid, this will be a boolean true response. If not, it will be a javascript object containing key/message pairs, where each 'message' is an array of validation error messages.

For forms that fail validation, you can retrieve both error codes and error messages, using `getErrors()` and `getMessages()`, respectively:

```
$codes = $form->getErrors();
$messages = $form->getMessage();
```

Note

Since the messages returned by `getMessages()` are an array of error code/message pairs, `getErrors()` is typically not needed.

You can retrieve codes and error messages for individual elements by simply passing the element name to each:

```
$codes = $form->getErrors('username');
$messages = $form->getMessages('username');
```

Note

Note: When validating elements, `Zend_Form` sends a second argument to each element's `isValid()` method: the array of data being validated. This can then be used by individual validators to allow them to utilize other submitted values when determining the validity of the data. An example would be a registration form that requires both a password and password confirmation; the password element could use the password confirmation as part of its validation.

Custom Error Messages

At times, you may want to specify one or more specific error messages to use instead of the error messages generated by the validators attached to your elements. Additionally, at times you may want to mark the form invalid yourself. As of 1.6.0, this functionality is possible via the following methods.

- `addErrorMessage($message)`: add an error message to display on form validation errors. You may call this more than once, and new messages are appended to the stack.
- `addErrorMessages(array $messages)`: add multiple error messages to display on form validation errors.
- `setErrorMessages(array $messages)`: add multiple error messages to display on form validation errors, overwriting all previously set error messages.
- `getErrorMessages()`: retrieve the list of custom error messages that have been defined.
- `clearErrorMessages()`: remove all custom error messages that have been defined.
- `markAsError()`: mark the form as having failed validation.
- `addError($message)`: add a message to the custom error messages stack and flag the form as invalid.
- `addErrors(array $messages)`: add several messages to the custom error messages stack and flag the form as invalid.
- `setErrors(array $messages)`: overwrite the custom error messages stack with the provided messages and flag the form as invalid.

All errors set in this fashion may be translated.

Methods

The following is a full list of methods available to `Zend_Form`, grouped by type:

- Configuration and Options:
  - `setOptions(array $options)`
  - `setConfig(Zend_Config $config)`
- Plugin Loaders and paths:
  - `setPluginLoader(Zend_Loader_PluginLoader_Interface $loader, $type = null)`
  - `getPluginLoader($type = null)`
  - `addPrefixPath($prefix, $path, $type = null)`
  - `addPrefixPaths(array $spec)`
  - `addElementPrefixPath($prefix, $path, $type = null)`
  - `addElementPrefixPaths(array $spec)`
  - `addDisplayGroupPrefixPath($prefix, $path)`
- Metadata:
  - `setAttrib($key, $value)`
  - `addAttribs(array $attribs)`
  - `setAttribs(array $attribs)`
  - `getAttrib($key)`
  - `getAttribs()`
  - `removeAttrib($key)`
  - `clearAttribs()`
  - `setAction($action)`
  - `getAction()`
  - `setMethod($method)`
  - `getMethod()`
  - `setName($name)`
  - `getName()`
- Elements:

  - `addElement($element, $name = null, $options = null)`
  - `addElements(array $elements)`
  - `setElements(array $elements)`
  - `getElement($name)`
  - `getElements()`
  - `removeElement($name)`
  - `clearElements()`
  - `setDefaults(array $defaults)`
  - `setDefault($name, $value)`
  - `getValue($name)`
  - `getValues()`
  - `getUnfilteredValue($name)`
  - `getUnfilteredValues()`
  - `setElementFilters(array $filters)`
  - `setElementDecorators(array $decorators)`
- Sub forms:
  - `addSubForm(Zend_Form $form, $name, $order = null)`
  - `addSubForms(array $subForms)`
  - `setSubForms(array $subForms)`
  - `getSubForm($name)`
  - `getSubForms()`
  - `removeSubForm($name)`
  - `clearSubForms()`
  - `setSubFormDecorators(array $decorators)`
- Display groups:
  - `addDisplayGroup(array $elements, $name, $options = null)`
  - `addDisplayGroups(array $groups)`
  - `setDisplayGroups(array $groups)`
  - `getDisplayGroup($name)`

  - `getDisplayGroups()`
  - `removeDisplayGroup($name)`
  - `clearDisplayGroups()`
  - `setDisplayGroupDecorators(array $decorators)`
- Validation
  - `populate(array $values)`
  - `isValid(array $data)`
  - `isValidPartial(array $data)`
  - `processAjax(array $data)`
  - `persistData()`
  - `getErrors($name = null)`
  - `getMessages($name = null)`
- Rendering:
  - `setView(Zend_View_Interface $view = null)`
  - `getView()`
  - `addDecorator($decorator, $options = null)`
  - `addDecorators(array $decorators)`
  - `setDecorators(array $decorators)`
  - `getDecorator($name)`
  - `getDecorators()`
  - `removeDecorator($name)`
  - `clearDecorators()`
  - `render(Zend_View_Interface $view = null)`
- I18n:
  - `setTranslator(Zend_Translate_Adapter $translator = null)`
  - `getTranslator()`
  - `setDisableTranslator($flag)`
  - `translatorIsDisabled()`

Configuration

`Zend_Form` is fully configurable via `setOptions()` and `setConfig()` (or by passing options or a `Zend_Config` object to the constructor). Using these methods, you can specify form elements, display groups, decorators, and metadata.

As a general rule, if 'set' + the option key refers to a `Zend_Form` method, then the value provided will be passed to that method. If the accessor does not exist, the key is assumed to reference an attribute, and will be passed to `setAttrib()`.

Exceptions to the rule include the following:

- `prefixPaths` will be passed to `addPrefixPaths()`
- `elementPrefixPaths` will be passed to `addElementPrefixPaths()`
- `displayGroupPrefixPaths` will be passed to `addDisplayGroupPrefixPaths()`
- the following setters cannot be set in this way:
  - `setAttrib (though setAttribs *will* work)`
  - `setConfig`
  - `setDefault`
  - `setOptions`
  - `setPluginLoader`
  - `setSubForms`
  - `setTranslator`
  - `setView`

As an example, here is a config file that passes configuration for every type of configurable data:

```
[element]
name = "registration"
action = "/user/register"
method = "post"
attribs.class = "zend_form"
attribs.onclick = "validate(this)"

disableTranslator = 0

prefixPath.element.prefix = "My_Element"
prefixPath.element.path = "My/Element/"
elementPrefixPath.validate.prefix = "My_Validate"
elementPrefixPath.validate.path = "My/Validate/"
displayGroupPrefixPath.prefix = "My_Group"
displayGroupPrefixPath.path = "My/Group/"

elements.username.type = "text"
elements.username.options.label = "Username"
elements.username.options.validators.alpha.validator = "Alpha"
elements.username.options.filters.lcase = "StringToLower"
; more elements, of course...

elementFilters.trim = "StringTrim"
;elementDecorators.trim = "StringTrim"

displayGroups.login.elements.username = "username"
displayGroups.login.elements.password = "password"
displayGroupDecorators.elements.decorator = "FormElements"
displayGroupDecorators.fieldset.decorator = "Fieldset"

decorators.elements.decorator = "FormElements"
decorators.fieldset.decorator = "FieldSet"
decorators.fieldset.decorator.options.class = "zend_form"
decorators.form.decorator = "Form"
```

The above could easily be abstracted to an XML or PHP array-based configuration file.

Custom forms

An alternative to using configuration-based forms is to subclass `Zend_Form`. This has several benefits:

- You can unit test your form easily to ensure validations and rendering perform as expected.
- Fine-grained control over individual elements.
- Re-use of form objects, and greater portability (no need to track config files).
- Implementing custom functionality.

The most typical use case would be to use the `init()` method to setup specific form elements and configuration:

```
class My_Form_Login extends Zend_Form
{
    public function init()
    {
        $username = new Zend_Form_Element_Text('username');
        $username->class = 'formtext';
        $username->setLabel('Username:')
                 ->setDecorators(array(
                     array('ViewHelper',
                           array('helper' => 'formText')),
                     array('Label',
                           array('class' => 'label'))
                 ));

        $password = new Zend_Form_Element_Password('password');
        $password->class = 'formtext';
        $password->setLabel('Username:')
```

```
                  ->setDecorators(array(
                      array('ViewHelper',
                            array('helper' => 'formPassword')),
                      array('Label',
                            array('class' => 'label'))
                  ));

        $submit = new Zend_Form_Element_Submit('login');
        $submit->class = 'formsubmit';
        $submit->setValue('Login')
               ->setDecorators(array(
                   array('ViewHelper',
                   array('helper' => 'formSubmit'))
               ));

        $this->addElements(array(
            $username,
            $password,
            $submit
        ));

        $this->setDecorators(array(
            'FormElements',
            'Fieldset',
            'Form'
        ));
    }
}
```

This form can then be instantiated with simply:

```
$form = new My_Form_Login();
```

and all functionality is already setup and ready; no config files needed. (Note that this example is greatly simplified, as it contains no validators or filters for the elements.)

Another common reason for extension is to define a set of default decorators. You can do this by overriding the `loadDefaultDecorators()` method:

```
class My_Form_Login extends Zend_Form
{
    public function loadDefaultDecorators()
    {
        $this->setDecorators(array(
            'FormElements',
            'Fieldset',
            'Form'
        ));
        }
}
```

Creating Custom Form Markup Using Zend_Form_Decorator

Rendering a form object is completely optional -- you do not need to use `Zend_Form`'s render() methods at all. However, if you do, decorators are used to render the various form objects.

An arbitrary number of decorators may be attached to each item (elements, display groups, sub forms, or the form object itself); however, only one decorator of a given type may be attached to each item. Decorators are called in the order they are registered. Depending on the decorator, it may replace the content passed to it, or append or prepend the content.

Object state is set via configuration options passed to the constructor or the decorator's `setOptions()` method. When creating decorators via an item's `addDecorator()` or related methods, options may be passed as an argument to the method. These can be used to specify placement, a separator to use between passed in content and newly generated content, and whatever options the decorator supports.

Before each decorator's `render()` method is called, the current item is set in the decorator using `setElement()`, giving the decorator awareness of the item being rendered. This allows you to create decorators that only render specific portions of the item -- such as the label, the value, error messages, etc. By stringing together several decorators that render specific element segments, you can build complex markup representing the entire item.

Operation

To configure a decorator, pass an array of options or a `Zend_Config` object to its constructor, an array to `setOptions()`, or a `Zend_Config` object to `setConfig()`.

Standard options include:

- `placement`: Placement can be either 'append' or 'prepend' (case insensitive), and indicates whether content passed to `render()` will be appended or prepended, respectively. In the case that a decorator replaces the content, this setting is ignored. The default setting is to append.
- `separator`: The separator is used between the content passed to `render()` and new content generated by the decorator, or between items rendered by the decorator (e.g. FormElements uses the separator between each item rendered). In the case that a decorator replaces the content, this setting may be ignored. The default value is `PHP_EOL`.

The decorator interface specifies methods for interacting with options. These include:

- `setOption($key, $value)`: set a single option.
- `getOption($key)`: retrieve a single option value.
- `getOptions()`: retrieve all options.
- `removeOption($key)`: remove a single option.

- `clearOptions()`: remove all options.

Decorators are meant to interact with the various `Zend_Form` class types: `Zend_Form`, `Zend_Form_Element`, `Zend_Form_DisplayGroup`, and all classes deriving from them. The method `setElement()` allows you to set the object the decorator is currently working with, and `getElement()` is used to retrieve it.

Each decorator's `render()` method accepts a string, `$content`. When the first decorator is called, this string is typically empty, while on subsequent calls it will be populated. Based on the type of decorator and the options passed in, the decorator will either replace this string, prepend the string, or append the string; an optional separator will be used in the latter two situations.

Standard Decorators

`Zend_Form` ships with many standard decorators; see the chapter on Standard Decorators for details.

Custom Decorators

If you find your rendering needs are complex or need heavy customization, you should consider creating a custom decorator.

Decorators need only implement `Zend_Decorator_Interface`. The interface specifies the following:

```
interface Zend_Decorator_Interface
{
    public function __construct($options = null);
    public function setElement($element);
    public function getElement();
    public function setOptions(array $options);
    public function setConfig(Zend_Config $config);
    public function setOption($key, $value);
    public function getOption($key);
    public function getOptions();
    public function removeOption($key);
    public function clearOptions();
    public function render($content);
}
```

To make this simpler, you can simply extend `Zend_Decorator_Abstract`, which implements all methods except `render()`.

As an example, let's say you wanted to reduce the number of decorators you use, and build a "composite" decorator that took care of rendering the label, element, any error messages, and description in an HTML `div`. You might build such a 'Composite' decorator as follows:

```
class My_Decorator_Composite extends Zend_Form_Decorator_Abstract
{
    public function buildLabel()
    {
        $element = $this->getElement();
        $label = $element->getLabel();
        if ($translator = $element->getTranslator()) {
            $label = $translator->translate($label);
        }
        if ($element->isRequired()) {
            $label .= '*';
        }
        $label .= ':';
        return $element->getView()
                       ->formLabel($element->getName(), $label);
    }

    public function buildInput()
    {
        $element = $this->getElement();
        $helper  = $element->helper;
        return $element->getView()->$helper(
            $element->getName(),
            $element->getValue(),
            $element->getAttribs(),
            $element->options
        );
    }

    public function buildErrors()
    {
        $element  = $this->getElement();
        $messages = $element->getMessages();
        if (empty($messages)) {
            return '';
        }
        return '<div class="errors">' .
               $element->getView()->formErrors($messages) . '</div>';
    }

    public function buildDescription()
    {
        $element = $this->getElement();
        $desc    = $element->getDescription();
        if (empty($desc)) {
            return '';
        }
        return '<div class="description">' . $desc . '</div>';
    }

    public function render($content)
    {
        $element = $this->getElement();
        if (!$element instanceof Zend_Form_Element) {
            return $content;
        }
        if (null === $element->getView()) {
            return $content;
        }
        $separator = $this->getSeparator();
        $placement = $this->getPlacement();
        $label     = $this->buildLabel();
        $input     = $this->buildInput();
        $errors    = $this->buildErrors();
        $desc      = $this->buildDescription();

        $output = '<div class="form element">'
                . $label
                . $input
                . $errors
                . $desc
                . '</div>'

        switch ($placement) {
            case (self::PREPEND):
                return $output . $separator . $content;
            case (self::APPEND):
            default:
                return $content . $separator . $output;
        }
    }
}
```

You can then place this in the decorator path:

```
// for an element:
$element->addPrefixPath('My_Decorator',
                        'My/Decorator/',
                        'decorator');

// for all elements:
$form->addElementPrefixPath('My_Decorator',
                            'My/Decorator/',
                            'decorator');
```

You can then specify this decorator as 'Composite' and attach it to an element:

```
// Overwrite existing decorators with this single one:
$element->setDecorators(array('Composite'));
```

While this example showed how to create a decorator that renders complex output from several element properties, you can also create decorators that handle a single aspect of an element; the 'Decorator' and 'Label' decorators are excellent examples of this practice. Doing so allows you to mix and match decorators to achieve complex output -- and also override single aspects of decoration to customize for your needs.

For example, if you wanted to simply display that an error occurred when validating an element, but not display each of the individual validation error messages, you might create your own 'Errors' decorator:

```
class My_Decorator_Errors
{
    public function render($content = '')
    {
        $output = '<div class="errors">The value you provided was invalid;
            please try again</div>';

        $placement = $this->getPlacement();
        $separator = $this->getSeparator();

        switch ($placement) {
            case 'PREPEND':
                return $output . $separator . $content;
            case 'APPEND':
            default:
                return $content . $separator . $output;
        }
    }
}
```

In this particular example, because the decorator's final segment, 'Errors', matches the same as `Zend_Form_Decorator_Errors`, it will be rendered *in place of* that decorator -- meaning you would not need to change any decorators to modify the output. By naming your decorators after existing standard decorators, you can modify decoration without needing to modify your elements' decorators.

Standard Form Elements Shipped With Zend Framework

Zend Framework ships with concrete element classes covering most HTML form elements. Most simply specify a particular view helper for use when decorating the element, but several offer additional functionality. The following is a list of all such classes, as well as descriptions of the functionality they offer.

Zend_Form_Element_Button

Used for creating HTML button elements, `Zend_Form_Element_Button` extends Zend_Form_Element_Submit, deriving its custom functionality. It specifies the 'formButton' view helper for decoration.

Like the submit element, it uses the element's label as the element value for display purposes; in other words, to set the text of the button, set the value of the element. The label will be translated if a translation adapter is present.

Because the label is used as part of the element, the button element uses only the ViewHelper and DtDdWrapper decorators.

After populating or validating a form, you can check if the given button was clicked using the `isChecked()` method.

Zend_Form_Element_Captcha

CAPTCHAs are used to prevent automated submission of forms by bots and other automated processes.

The Captcha form element allows you to specify which Zend_Captcha adapter you wish to utilize as a form captcha. It then sets this adapter as a validator to the object, and uses a Captcha decorator for rendering (which proxies to the captcha adapter).

Adapters may be any adapters in `Zend_Captcha`, as well as any custom adapters you may have defined elsewhere. To allow this, you may pass an additional plugin loader type key, 'CAPTCHA' or 'captcha', when specifying a plugin loader prefix path:

```
$element->addPrefixPath('My_Captcha', 'My/Captcha/', 'captcha');
```

Captcha's may then be registered using the `setCaptcha()` method, which can take either a concrete captcha instance, or the short name of a captcha adapter:

```
// Concrete instance:
$element->setCaptcha(new Zend_Captcha_Figlet());

// Using shortnames:
$element->setCaptcha('Dumb');
```

If you wish to load your element via configuration, specify either the key 'captcha' with an array containing the key 'captcha', or both the keys 'captcha' and 'captchaOptions':

```
// Using single captcha key:
$element = new Zend_Form_Element_Captcha('foo', array(
    'label' => "Please verify you're a human",
    'captcha' => array(
        'captcha' => 'Figlet',
        'wordLen' => 6,
        'timeout' => 300,
    ),
));

// Using both captcha and captchaOptions:
$element = new Zend_Form_Element_Captcha('foo', array(
    'label' => "Please verify you're a human"
    'captcha' => 'Figlet',
    'captchaOptions' => array(
        'captcha' => 'Figlet',
        'wordLen' => 6,
        'timeout' => 300,
    ),
));
```

The decorator used is determined by querying the captcha adapter. By default, the Captcha decorator is used, but an adapter may specify a different one via its `getDecorator()` method.

As noted, the captcha adapter itself acts as a validator for the element. Additionally, the NotEmpty validator is not used, and the element is marked as required. In most cases, you should need to do nothing else to have a captcha present in your form.

Zend_Form_Element_Checkbox

HTML checkboxes allow you return a specific value, but basically operate as booleans: when it is checked, the value is submitted; when it's not checked, nothing is submitted. Internally, Zend_Form_Element_Checkbox enforces this state.

By default, the checked value is '1', and the unchecked value '0'. You can specify the values to use using the `setCheckedValue()` and `setUncheckedValue()` accessors, respectively. Internally, any time you set the value, if the provided value matches the checked value, then it is set, but any other value causes the unchecked value to be set.

Additionally, setting the value sets the `checked` property of the checkbox. You can query this using `isChecked()` or simply accessing the property. Using the `setChecked($flag)` method will both set the state of the flag as well as set the approriate checked or unchecked value in the element. Please use this method when setting the checked state of a checkbox element to ensure the value is set properly.

`Zend_Form_Element_Checkbox` uses the 'formCheckbox' view helper. The checked value is always used to populate it.

Zend_Form_Element_File

The File form element provides a mechanism for supplying file upload fields to your form. It utilizes Zend_File_Transfer internally to provide this functionality, and the `FormFile` view helper to display the form element.

By default, it uses the `Http` transfer adapter, which introspects the `$_FILES` array and allows you to attach validators and filters. Validators and filters attached to the form element will be attached to the transfer adapter.

Example 19.8. File form element usage

The above explanation of using the File form element may seem arcane, but actual usage is relatively trivial:

```
$element = new Zend_Form_Element_File('foo');
$element->setLabel('Upload an image:')
        ->setDestination('/var/www/upload')
        ->addValidator('Count', false, 1)     // ensure only 1 file
        ->addValidator('Size', false, 102400) // limit to 100K
        ->addValidator('Extension' false, 'jpg,png,gif'); // only JPEG, PNG, and G
$form->addElement($element, 'foo');
```

You also need to ensure the correct encoding type is provided to the form; you should use 'multipart/form-data'. You can do this by setting the 'enctype' attribute on the form:

```
$form->setAttrib('enctype', 'multipart/form-data');
```

When the element has successfully validated, you can determine the final location using getValue():

```
$location = $form->foo->getValue();
```

Default upload locations

By default, files are uploaded to the system temp directory.

There is one current limitation to the file upload element shipped in 1.6.0. At this time, you cannot specify filters to change the name of the final uploaded file; this functionality will be added in an upcoming release.

Zend_Form_Element_Hidden

Hidden elements merely inject data that should be submitted, but which the user should not manipulate. `Zend_Form_Element_Hidden` accomplishes this through use of the 'formHidden' view helper.

Zend_Form_Element_Hash

This element provides 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.

The name of the hash element should be unique. It is recommended to use the `salt` option for the element, two hashes with same names and different salts would not collide:

```
$form->addElement('hash', 'no_csrf_foo', array('salt' => 'unique'));
```

You can set the salt later using the `setSalt($salt)` method.

Internally, the element stores a unique identifier using `Zend_Session_Namespace`, and checks for it at submission (checking that the TTL has not expired). The 'Identical' validator is then used to ensure the submitted hash matches the stored hash.

The 'formHidden' view helper is used to render the element in the form.

Zend_Form_Element_Image

Images can be used as form elements, and allow you to specify graphical elements as form buttons.

Images need an image source. `Zend_Form_Element_Image` allows you to specify this by using the `setImage()` accessor (or 'image' configuration key). You can also optionally specify a value to use when submitting the image using the `setImageValue()` accessor (or 'imageValue' configuration key). When the value set for the element matches the `imageValue`, then the accessor `isChecked()` will return true.

The Image element uses the Image Decorator for rendering (as well as the standard Errors, HtmlTag, and Label decorators). You can optionally specify a tag to the `Image` decorator that will then wrap the image element.

Zend_Form_Element_MultiCheckbox

Often you have a set of related checkboxes, and you wish to group the results. This is much like a Multiselect, but instead of them being in a dropdown list, you need to show checkbox/value pairs.

`Zend_Form_Element_MultiCheckbox` makes this a snap. Like all other elements extending the base Multi element, you can specify a list of options, and easily validate against that same list. The 'formMultiCheckbox' view helper ensures that these are returned as an array in the form submission.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a false value to the `registerInArrayValidator` configuration key.

You may manipulate the various checkbox options using the following methods:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- getMultiOption($option)
- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

To mark checked items, you need to pass an array of values to `setValue()`. The following will check the values "bar" and "bat":

```
$element = new Zend_Form_Element_MultiCheckbox('foo', array(
    'multiOptions' => array(
        'foo' => 'Foo Option',
        'bar' => 'Bar Option',
        'baz' => 'Baz Option',
        'bat' => 'Bat Option',
    );
));

$element->setValue(array('bar', 'bat'));
```

Note that even when setting a single value, you must pass an array.

Zend_Form_Element_Multiselect

XHTML `select` elements allow a 'multiple' attribute, indicating multiple options may be selected for submission, instead of the usual one. `Zend_Form_Element_Multiselect` extends Zend_Form_Element_Select, and sets the `multiple` attribute to 'multiple'. Like other classes that inherit from the base `Zend_Form_Element_Multi` class, you can manipulate the options for the select using:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- getMultiOption($option)
- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

If a translation adapter is registered with the form and/or element, option values will be translated for display purposes.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a false value to the `registerInArrayValidator` configuration key.

Zend_Form_Element_Password

Password elements are basically normal text elements -- except that you typically do not want the submitted password displayed in error messages or the element itself when the form is re-displayed.

`Zend_Form_Element_Password` achieves this by calling `setObscureValue(true)` on each validator (ensuring that the password is obscured in validation error messages), and using the 'formPassword' view helper (which does not display the value passed to it).

Zend_Form_Element_Radio

Radio elements allow you to specify several options, of which you need a single value returned. `Zend_Form_Element_Radio` extends the base `Zend_Form_Element_Multi` class, allowing you to specify a number of options, and then uses the `formRadio` view helper to display these.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a false value to the `registerInArrayValidator` configuration key.

Like all elements extending the Multi element base class, the following methods may be used to manipulate the radio options displayed:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- getMultiOption($option)
- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

Zend_Form_Element_Reset

Reset buttons are typically used to clear a form, and are not part of submitted data. However, as they serve a purpose in the display, they are included in the standard elements.

`Zend_Form_Element_Reset` extends Zend_Form_Element_Submit. As such, the label is used for the button display, and will be translated if a translation adapter is present. It utilizes only the 'ViewHelper' and 'DtDdWrapper' decorators, as there should never be error messages for such elements, nor will a label be necessary.

Zend_Form_Element_Select

Select boxes are a common way of limiting to specific choices for a given form datum. `Zend_Form_Element_Select` allows you to generate these quickly and easily.

By default, this element registers an `InArray` validator which validates against the array keys of registered options. You can disable this behavior by either calling `setRegisterInArrayValidator(false)`, or by passing a false value to the `registerInArrayValidator` configuration key.

As it extends the base Multi element, the following methods may be used to manipulate the select options:

- `addMultiOption($option, $value)`
- `addMultiOptions(array $options)`
- `setMultiOptions(array $options)` (overwrites existing options)
- getMultiOption($option)

- getMultiOptions()
- `removeMultiOption($option)`
- `clearMultiOptions()`

`Zend_Form_Element_Select` uses the 'formSelect' view helper for decoration.

Zend_Form_Element_Submit

Submit buttons are used to submit a form. You may use multiple submit buttons; you can use the button used to submit the form to decide what action to take with the data submitted. `Zend_Form_Element_Submit` makes this decisioning easy, by adding a `isChecked()` method; as only one button element will be submitted by the form, after populating or validating the form, you can call this method on each submit button to determine which one was used.

`Zend_Form_Element_Submit` uses the label as the "value" of the submit button, translating it if a translation adapter is present. `isChecked()` checks the submitted value against the label in order to determine if the button was used.

The ViewHelper and DtDdWrapper decorators to render the element. No label decorator is used, as the button label is used when rendering the element; also, typically, you will not associate errors with a submit element.

Zend_Form_Element_Text

By far the most prevalent type of form element is the text element, allowing for limited text entry; it's an ideal element for most data entry. `Zend_Form_Element_Text` simply uses the 'formText' view helper to display the element.

Zend_Form_Element_Textarea

Textareas are used when large quantities of text are expected, and place no limits on the amount of text submitted (other than maximum size limits as dictated by your server or PHP). `Zend_Form_Element_Textarea` uses the 'textArea' view helper to display such elements, placing the value as the content of the element.

Standard Form Decorators Shipped With Zend Framework

`Zend_Form` ships with several standard decorators. For more information on general decorator usage, see the Decorators section.

Zend_Form_Decorator_Callback

The Callback decorator can execute an arbitrary callback to render content. Callbacks should be specified via the 'callback' option passed in the decorator configuration, and can be any valid PHP callback type. Callbacks should accept three arguments, `$content` (the original content passed to the decorator), `$element` (the item being decorated), and an array of `$options`. As an example callback:

```
class Util
{
    public static function label($content, $element, array $options)
    {
        return '<span class="label">' . $element->getLabel() . "</span>";
    }
}
```

This callback would be specified as `array('Util', 'label')`, and would generate some (bad) HTML markup for the label. The Callback decorator would then either replace, append, or prepend the original content with the return value of this.

The Callback decorator allows specifying a null value for the placement option, which will replace the original content with the callback return value; 'prepend' and 'append' are still valid as well.

Zend_Form_Decorator_Captcha

The Captcha decorator is for use with the Captcha form element. It utilizes the captcha adapter's `render()` method to generate the output.

A variant on the Captcha decorator, 'Captcha_Word', is also commonly used, and creates two elements, an id and input. The id indicates the session identifier to compare against, and the input is for the user verification of the captcha. These are validated as a single element.

Zend_Form_Decorator_Description

The Description decorator can be used to display a description set on a `Zend_Form`, `Zend_Form_Element`, or `Zend_Form_DisplayGroup` item; it pulls the description using the object's `getDescription()` method. Common use cases are for providing UI hints for your elements.

By default, if no description is present, no output is generated. If the description is present, then it is wrapped in an HTML `p` tag by default, though you may specify a tag by passing a `tag` option when creating the decorator, or calling `setTag()`. You may additionally specify a class for the tag using the `class` option or by calling `setClass()`; by default, the class 'hint' is used.

The description is escaped using the view object's escaping mechanisms by default. You can disable this by passing a `false` value to the decorator's 'escape' option or `setEscape()` method.

Zend_Form_Decorator_DtDdWrapper

The default decorators utilize definition lists (`<dl>`) to render form elements. Since form items can appear in any order, display groups and sub forms can be interspersed with other form items. To keep these particular item types within the definition list, the DtDdWrapper creates a new, empty definition term (`<dt>`) and wraps its content in a new definition datum (`<dd>`). The output looks something like this:

```
<dt></dt>
<dd><fieldset id="subform">
    <legend>User Information</legend>
    ...
```

```
</fieldset></dd>
```

This decorator replaces the content provided to it by wrapping it within the `<dd>` element.

Zend_Form_Decorator_Errors

Element errors get their own decorator with the Errors decorator. This decorator proxies to the FormErrors view helper, which renders error messages in an unordered list (`<ul>`) as list items. The `<ul>` element receives a class of "errors".

The Errors decorator can either prepend or append the content provided to it.

Zend_Form_Decorator_Fieldset

Display groups and sub forms render their content within fieldsets by default. The Fieldset decorator checks for either a 'legend' option or a `getLegend()` method in the registered element, and uses that as a legend if non-empty. Any content passed in is wrapped in the HTML fieldset, replacing the original content. Any attributes set in the decorated item are passed to the fieldset as HTML attributes.

Zend_Form_Decorator_Form

`Zend_Form` objects typically need to render an HTML form tag. The Form decorator proxies to the Form view helper. It wraps any provided content in an HTML form element, using the `Zend_Form` object's action and method, and any attributes as HTML attributes.

Zend_Form_Decorator_FormElements

Forms, display groups, and sub forms are collections of elements. In order to render these elements, they utilize the FormElements decorator, which iterates through all items, calling `render()` on each and joining them with the registered separator. It can either append or prepend content passed to it.

Zend_Form_Decorator_HtmlTag

The HtmlTag decorator allows you to utilize HTML tags to decorate content; the tag utilized is passed in the 'tag' option, and any other options are used as HTML attributes to that tag. The tag by default is assumed to be block level, and replaces the content by wrapping it in the given tag. However, you can specify a placement to append or prepend a tag as well.

Zend_Form_Decorator_Image

The Image decorator allows you to create an HTML image input (`<input type="image" ... />`), and optionally render it within another HTML tag.

By default, the decorator uses the element's src property, which can be set with the `setImage()` method, as the image source. Additionally, the element's label will be used as the alt tag, and the `imageValue` (manipulated with the Image element's `setImageValue()` and `getImageValue()` accessors) will be used for the value.

To specify an HTML tag with which to wrap the element, either pass a 'tag' option to the decorator, or explicitly call `setTag()`.

Zend_Form_Decorator_Label

Form elements typically have labels, and the Label decorator is used to render these labels. It proxies to the FormLabel view helper, and pulls the element label using the `getLabel()` method of the element. If no label is present, none is rendered. By default, labels are translated when a translation adapter exists and a translation for the label exists.

You may optionally specify a 'tag' option; if provided, it wraps the label in that block-level tag. If the 'tag' option is present, and no label present, the tag is rendered with no content. You can specify the class to use with the tag with the 'class' option or by calling `setClass()`.

Additionally, you can specify prefixes and suffixes to use when displaying the element, based on whether or not the label is for an optional or required element. Common use cases would be to append a ':' to the label, or a '*' indicating an item is required. You can do so with the following options and methods:

- `optionalPrefix`: set the text to prefix the label with when the element is optional. Use the `setOptionalPrefix()` and `getOptionalPrefix()` accessors to manipulate it.
- `optionalSuffix`: set the text to append the label with when the element is optional. Use the `setOptionalSuffix()` and `getOptionalSuffix()` accessors to manipulate it.
- `requiredPrefix`: set the text to prefix the label with when the element is required. Use the `setRequiredPrefix()` and `getRequiredPrefix()` accessors to manipulate it.
- `requiredSuffix`: set the text to append the label with when the element is required. Use the `setRequiredSuffix()` and `getRequiredSuffix()` accessors to manipulate it.

By default, the Label decorator prepends to the provided content; specify a 'placement' option of 'append' to place it after the content.

Zend_Form_Decorator_ViewHelper

Most elements utilize `Zend_View` helpers for rendering, and this is done with the ViewHelper decorator. With it, you may specify a 'helper' tag to explicitly set the view helper to utilize; if none is provided, it uses the last segment of the element's class name to determine the helper, prepending it with the string 'form': e.g., 'Zend_Form_Element_Text' would look for a view helper of 'formText'.

Any attributes of the provided element are passed to the view helper as element attributes.

By default, this decorator appends content; use the 'placement' option to specify alternate placement.

Zend_Form_Decorator_ViewScript

Sometimes you may wish to use a view script for creating your elements; this way you can have fine-grained control over your elements, turn the view script over to a designer, or simply create a way to easily override setting based on which module you're using (each module could optionally override element view scripts to suit their own needs). The ViewScript decorator solves this problem.

The ViewScript decorator requires a 'viewScript' option, either provided to the decorator, or as an attribute of the element. It then renders that view script as a partial script, meaning each call to it has its own variable scope; no variables from the view will be injected other than the element itself. Several variables are then populated:

- `element`: the element being decorated

- `content`: the content passed to the decorator
- `decorator`: the decorator object itself
- Additionally, all options passed to the decorator via `setOptions()` that are not used internally (such as placement, separator, etc.) are passed to the view script as view variables.

As an example, you might have the following element:

```
// Setting the decorator for the element to a single, ViewScript,
// decorator, specifying the viewScript as an option, and some extra
// options:
$element->setDecorators(array(array('ViewScript', array(
    'viewScript' => '_element.phtml',
    'class'      => 'form element'
))));

// OR specifying the viewScript as an element attribute:
$element->viewScript = '_element.phtml';
$element->setDecorators(array(array('ViewScript',
                                    array('class' => 'form element'))));
```

You could then have a view script something like this:

```
<div class="<?= $this->class ?>">
    <?= $this->formLabel($this->element->getName(),
                         $this->element->getLabel()) ?>
    <?= $this->{$this->element->helper}(
        $this->element->getName(),
        $this->element->getValue(),
        $this->element->getAttribs()
    ) ?>
    <?= $this->formErrors($this->element->getMessages()) ?>
    <div class="hint"><?= $this->element->getDescription() ?></div>
</div>
```

Replacing content with a view script

You may find it useful for the view script to replace the content provided to the decorator -- for instance, if you want to wrap it. You can do so by specifying a boolean false value for the decorator's 'placement' option:

```
// At decorator creation:
$element->addDecorator('ViewScript', array('placement' => false));

// Applying to an existing decorator instance:
$decorator->setOption('placement', false);
```

```
// Applying to a decorator already attached to an element:
$element->getDecorator('ViewScript')->setOption('placement', false);

// Within a view script used by a decorator:
$this->decorator->setOption('placement', false);
```

Using the ViewScript decorator is recommended for when you want to have very fine-grained control over how your elements are rendered.

Internationalization of Zend_Form

Increasingly, developers need to tailor their content for multiple languages and regions. Zend_Form aims to make such a task trivial, and leverages functionality in both Zend_Translate and Zend_Validate to do so.

By default, no internationalisation (I18n) is performed. To turn on I18n features in `Zend_Form`, you will need to instantiate a `Zend_Translate` object with an appropriate adapter, and attach it to `Zend_Form` and/or `Zend_Validate`. See the Zend_Translate documentation for more information on creating the translate object and translation files

Translation Can Be Turned Off Per Item

You can disable translation for any form, element, display group, or sub form by calling its `setDisableTranslator($flag)` method or passing a `disableTranslator` option to the object. This can be useful when you want to selectively disable translation for individual elements or sets of elements.

Initializing I18n in Forms

In order to initialize I18n in forms, you will need either a `Zend_Translate` object or a `Zend_Translate_Adapter` object, as detailed in the `Zend_Translate` documentation. Once you have a translation object, you have several options:

- *Easiest:* add it to the registry. All I18n aware components of Zend Framework will autodiscover a translate object that is in the registry under the 'Zend_Translate' key and use it to perform translation and/or localization:

  ```
  // use the 'Zend_Translate' key; $translate is a Zend_Translate object:
  Zend_Registry::set('Zend_Translate', $translate);
  ```

  This will be picked up by `Zend_Form`, `Zend_Validate`, and `Zend_View_Helper_Translate`.

- If all you are worried about is translating validation error messages, you can register the translation object with `Zend_Validate_Abstract`:

  ```
  // Tell all validation classes to use a specific translate adapter:
  Zend_Validate_Abstract::setDefaultTranslator($translate);
  ```

- Alternatively, you can attach to the `Zend_Form` object as a global translator. This has the side effect of also translating validation error messages:

  ```
  // Tell all form classes to use a specific translate adapter, as well
  // as use this adapter to translate validation error messages:
  Zend_Form::setDefaultTranslator($translate);
  ```

- Finally, you can attach a translator to a specific form instance or to specific elements using their `setTranslator()` methods:

  ```
  // Tell *this* form instance to use a specific translate adapter; it
  // will also be used to translate validation error messages for all
  // elements:
  $form->setTranslator($translate);

  // Tell *this* element to use a specific translate adapter; it will
  // also be used to translate validation error messages for this
  // particular element:
  $element->setTranslator($translate);
  ```

Standard I18n Targets

Now that you've attached a translation object to, what exactly can you translate by default?

- *Validation error messages.* Validation error messages may be translated. To do so, use the various error code constants from the `Zend_Validate` validation classes as the message IDs. For more information on these codes, see the Zend_Validate documentation.

  Alternately, as of 1.6.0, you may provide translation strings using the actual error messages as message identifiers. This is the preferred use case for 1.6.0 and up, as we will be deprecating translation of message keys in future releases.

- *Labels.* Element labels will be translated, if a translation exists.

- *Fieldset Legends.* Display groups and sub forms render in fieldsets by default. The Fieldset decorator attempts to translate the legend before rendering the fieldset.

- *Form and Element Descriptions.* All form types (element, form, display group, sub form) allow specifying an optional item description. The Description decorator can be used to render this, and by default will take the value and attempt to translate it.

- *Multi-option Values.* For the various items inheriting from `Zend_Form_Element_Multi` (including the MultiCheckbox, Multiselect, and Radio elements), the option values (not keys) will be translated if a translation is available; this means that the option labels presented to the user will be translated.
- *Submit and Button Labels.* The various Submit and Button elements (Button, Submit, and Reset) will translate the label displayed to the user.

Advanced Zend_Form Usage

`Zend_Form` has a wealth of functionality, much of it aimed at experienced developers. This chapter aims to document some of this functionality with examples and use cases.

Array Notation

Many experienced web developers like to group related form elements using array notation in the element names. For example, if you have two addresses you wish to capture, a shipping and a billing address, you may have identical elements; by grouping them in an array, you can ensure they are captured separately. Take the following form, for example:

```
<form>
    <fieldset>
        <legend>Shipping Address</legend>
        <dl>
            <dt><label for="recipient">Ship to:</label></dt>
            <dd><input name="recipient" type="text" value="" /></dd>

            <dt><label for="address">Address:</label></dt>
            <dd><input name="address" type="text" value="" /></dd>

            <dt><label for="municipality">City:</label></dt>
            <dd><input name="municipality" type="text" value="" /></dd>

            <dt><label for="province">State:</label></dt>
            <dd><input name="province" type="text" value="" /></dd>

            <dt><label for="postal">Postal Code:</label></dt>
            <dd><input name="postal" type="text" value="" /></dd>
        </dl>
    </fieldset>

    <fieldset>
        <legend>Billing Address</legend>
        <dl>
            <dt><label for="payer">Bill To:</label></dt>
            <dd><input name="payer" type="text" value="" /></dd>

            <dt><label for="address">Address:</label></dt>
            <dd><input name="address" type="text" value="" /></dd>

            <dt><label for="municipality">City:</label></dt>
            <dd><input name="municipality" type="text" value="" /></dd>
            <dt><label for="province">State:</label></dt>
            <dd><input name="province" type="text" value="" /></dd>

            <dt><label for="postal">Postal Code:</label></dt>
            <dd><input name="postal" type="text" value="" /></dd>
        </dl>
    </fieldset>

    <dl>
        <dt><label for="terms">I agree to the Terms of Service</label></dt>
        <dd><input name="terms" type="checkbox" value="" /></dd>

        <dt></dt>
        <dd><input name="save" type="submit" value="Save" /></dd>
    </dl>
</form>
```

In this example, the billing and shipping address contain some identical fields, which means one would overwrite the other. We can solve this solution using array notation:

```
<form>
    <fieldset>
        <legend>Shipping Address</legend>
        <dl>
            <dt><label for="shipping-recipient">Ship to:</label></dt>
            <dd><input name="shipping[recipient]" id="shipping-recipient"
                type="text" value="" /></dd>

            <dt><label for="shipping-address">Address:</label></dt>
            <dd><input name="shipping[address]" id="shipping-address"
                type="text" value="" /></dd>

            <dt><label for="shipping-municipality">City:</label></dt>
            <dd><input name="shipping[municipality]" id="shipping-municipality"
                type="text" value="" /></dd>

            <dt><label for="shipping-province">State:</label></dt>
            <dd><input name="shipping[province]" id="shipping-province"
                type="text" value="" /></dd>

            <dt><label for="shipping-postal">Postal Code:</label></dt>
            <dd><input name="shipping[postal]" id="shipping-postal"
                type="text" value="" /></dd>
        </dl>
    </fieldset>

    <fieldset>
        <legend>Billing Address</legend>
        <dl>
            <dt><label for="billing-payer">Bill To:</label></dt>
            <dd><input name="billing[payer]" id="billing-payer"
                type="text" value="" /></dd>
            <dt><label for="billing-address">Address:</label></dt>
            <dd><input name="billing[address]" id="billing-address"
                type="text" value="" /></dd>

            <dt><label for="billing-municipality">City:</label></dt>
            <dd><input name="billing[municipality]" id="billing-municipality"
                type="text" value="" /></dd>

            <dt><label for="billing-province">State:</label></dt>
            <dd><input name="billing[province]" id="billing-province"
                type="text" value="" /></dd>

            <dt><label for="billing-postal">Postal Code:</label></dt>
            <dd><input name="billing[postal]" id="billing-postal"
                type="text" value="" /></dd>
        </dl>
    </fieldset>

    <dl>
        <dt><label for="terms">I agree to the Terms of Service</label></dt>
        <dd><input name="terms" type="checkbox" value="" /></dd>

        <dt></dt>
        <dd><input name="save" type="submit" value="Save" /></dd>
    </dl>
</form>
```

In the above sample, we now get separate addresses. In the submitted form, we'll now have three elements, the 'save' element for the submit, and then two arrays, 'shipping' and 'billing', each with keys for their various elements.

`Zend_Form` attempts to automate this process with its sub forms. By default, sub forms render using the array notation as shown in the previous HTML form listing, complete with ids. The array name is based on the sub form name, with the keys based on the elements contained in the sub form. Sub forms may be nested arbitrarily deep, and this will create nested arrays to reflect the structure. Additionally, the various validation routines in `Zend_Form` honor the array structure, ensuring that your form validates correctly, no matter how arbitrarily deep you nest your sub forms. You need do nothing to benefit from this; this behaviour is enabled by default.

Additionally, there are facilities that allow you to turn on array notation conditionally, as well as specify the specific array to which an element or collection belongs:

- `Zend_Form::setIsArray($flag)`: By setting the flag true, you can indicate that an entire form should be treated as an array. By default, the form's name will be used as the name of the array, unless `setElementsBelongTo()` has been called. If the form has no specified name, or if `setElements-BelongTo()` has not been set, this flag will be ignored (as there is no array name to which the elements may belong).

  You may determine if a form is being treated as an array using the `isArray()` accessor.

- `Zend_Form::setElementsBelongTo($array)`: Using this method, you can specify the name of an array to which all elements of the form belong. You can determine the name using the `getElementsBelongTo()` accessor.

Additionally, on the element level, you can specify individual elements may belong to particular arrays using `Zend_Form_Element::setBelongsTo()` method. To discover what this value is -- whether set explicitly or implicitly via the form -- you may use the `getBelongsTo()` accessor.

Multi-Page Forms

Currently, Multi-Page forms are not officially supported in `Zend_Form`; however, most support for implementing them is available and can be utilized with a little extra tooling.

The key to creating a multi-page form is to utilize sub forms, but to display only one such sub form per page. This allows you to submit a single sub form at a time and validate it, but not process the form until all sub forms are complete.

Example 19.9. Registration Form Example

Let's use a registration form as an example. For our purposes, we want to capture the desired username and password on the first page, then the user's metadata -- given name, family name, and location -- and finally allow them to decide what mailing lists, if any, they wish to subscribe to.

First, let's create our own form, and define several sub forms within it:

```
class My_Form_Registration extends Zend_Form
{
    public function init()
    {
        // Create user sub form: username and password
        $user = new Zend_Form_SubForm();
        $user->addElements(array(
            new Zend_Form_Element_Text('username', array(
                'required'   => true,
                'label'      => 'Username:',
                'filters'    => array('StringTrim', 'StringToLower'),
                'validators' => array(
                    'Alnum',
                    array('Regex',
                          false,
                          array('/^[a-z][a-z0-9]{2,}$/'))
                )
            )),

            new Zend_Form_Element_Password('password', array(
                'required'   => true,
                'label'      => 'Password:',
                'filters'    => array('StringTrim'),
                'validators' => array(
                    'NotEmpty',
                    array('StringLength', false, array(6))
                )
            )),
        ));

        // Create demographics sub form: given name, family name, and
        // location
        $demog = new Zend_Form_SubForm();
        $demog->addElements(array(
            new Zend_Form_Element_Text('givenName', array(
                'required'   => true,
                'label'      => 'Given (First) Name:',
                'filters'    => array('StringTrim'),
                'validators' => array(
                    array('Regex',
                          false,
                          array('/^[a-z][a-z0-9., \'-]{2,}$/i'))
                )
            )),
            new Zend_Form_Element_Text('familyName', array(
                'required'   => true,
                'label'      => 'Family (Last) Name:',
                'filters'    => array('StringTrim'),
                'validators' => array(
                    array('Regex',
                          false,
                          array('/^[a-z][a-z0-9., \'-]{2,}$/i'))
                )
            )),

            new Zend_Form_Element_Text('location', array(
                'required'   => true,
                'label'      => 'Your Location:',
                'filters'    => array('StringTrim'),
                'validators' => array(
                    array('StringLength', false, array(2))
                )
            )),
        ));

        // Create mailing lists sub form
        $listOptions = array(
            'none'        => 'No lists, please',
            'fw-general'  => 'Zend Framework General List',
            'fw-mvc'      => 'Zend Framework MVC List',
            'fw-auth'     => 'Zend Framwork Authentication and ACL List',
            'fw-services' => 'Zend Framework Web Services List',
        );
        $lists = new Zend_Form_SubForm();
        $lists->addElements(array(
            new Zend_Form_Element_MultiCheckbox('subscriptions', array(
                'label'        =>
                    'Which lists would you like to subscribe to?',
                'multiOptions' => $listOptions,
                'required'     => true,
                'filters'      => array('StringTrim'),
                'validators'   => array(
                    array('InArray',
                          false,
                          array(array_keys($listOptions)))
                )
            )),
        ));

        // Attach sub forms to main form
        $this->addSubForms(array(
            'user'  => $user,
            'demog' => $demog,
            'lists' => $lists
        ));
    }
}
```

Note that there are no submit buttons, and that we have done nothing with the sub form decorators -- which means that by default they will be displayed as fieldsets. We will need to be able to override these as we display each individual sub form, and add in submit buttons so we can actually process them -- which will also require action and method properties. Let's add some scaffolding to our class to provide that information:

```
class My_Form_Registration extends Zend_Form
{
    // ...

    /**
     * Prepare a sub form for display
     *
     * @param  string|Zend_Form_SubForm $spec
     * @return Zend_Form_SubForm
     */
    public function prepareSubForm($spec)
    {
        if (is_string($spec)) {
            $subForm = $this->{$spec};
        } elseif ($spec instanceof Zend_Form_SubForm) {
            $subForm = $spec;
        } else {
            throw new Exception('Invalid argument passed to ' .
                                __FUNCTION__ . '()');
        }
        $this->setSubFormDecorators($subForm)
             ->addSubmitButton($subForm)
             ->addSubFormActions($subForm);
        return $subForm;
    }

    /**
     * Add form decorators to an individual sub form
     *
     * @param  Zend_Form_SubForm $subForm
     * @return My_Form_Registration
     */
    public function setSubFormDecorators(Zend_Form_SubForm $subForm)
    {
        $subForm->setDecorators(array(
            'FormElements',
            array('HtmlTag', array('tag' => 'dl',
                                   'class' => 'zend_form')),
            'Form',
        ));
        return $this;
    }

    /**
     * Add a submit button to an individual sub form
     *
     * @param  Zend_Form_SubForm $subForm
     * @return My_Form_Registration
     */
    public function addSubmitButton(Zend_Form_SubForm $subForm)
    {
        $subForm->addElement(new Zend_Form_Element_Submit(
            'save',
            array(
                'label'    => 'Save and continue',
                'required' => false,
                'ignore'   => true,
            )
        ));
        return $this;
    }

    /**
     * Add action and method to sub form
     *
     * @param  Zend_Form_SubForm $subForm
     * @return My_Form_Registration
     */
    public function addSubFormActions(Zend_Form_SubForm $subForm)
    {
        $subForm->setAction('/registration/process')
                ->setMethod('post');
        return $this;
    }
}
```

Next, we need to add some scaffolding in our action controller, and have several considerations. First, we need to make sure we persist form data between requests, so that we can determine when to quit. Second, we need some logic to determine what form segments have already been submitted, and what sub form to display based on that information. We'll use `Zend_Session_Namespace` to persist data, which will also help us answer the question of which form to submit.

Let's create our controller, and add a method for retrieving a form instance:

```
class RegistrationController extends Zend_Controller_Action
{
    protected $_form;

    public function getForm()
    {
        if (null === $this->_form) {
            $this->_form = new My_Form_Registration();
        }
        return $this->_form;
    }
}
```

Now, let's add some functionality for determining which form to display. Basically, until the entire form is considered valid, we need to continue displaying form segments. Additionally, we likely want to make sure they're in a particular order: user, demog, and then lists. We can determine what data has been submitted by checking our session namespace for particular keys representing each subform.

```
class RegistrationController extends Zend_Controller_Action
{
    // ...

    protected $_namespace = 'RegistrationController';
    protected $_session;

    /**
     * Get the session namespace we're using
     *
     * @return Zend_Session_Namespace
     */
    public function getSessionNamespace()
    {
        if (null === $this->_session) {
            $this->_session =
                new Zend_Session_Namespace($this->_namespace);
        }

        return $this->_session;
    }

    /**
     * Get a list of forms already stored in the session
     *
     * @return array
     */
    public function getStoredForms()
    {
        $stored = array();
        foreach ($this->getSessionNamespace() as $key => $value) {
            $stored[] = $key;
        }

        return $stored;
    }

    /**
     * Get list of all subforms available
     *
     * @return array
     */
    public function getPotentialForms()
    {
        return array_keys($this->getForm()->getSubForms());
    }

    /**
     * What sub form was submitted?
     *
     * @return false|Zend_Form_SubForm
     */
    public function getCurrentSubForm()
    {
        $request = $this->getRequest();
        if (!$request->isPost()) {
            return false;
        }

        foreach ($this->getPotentialForms() as $name) {
            if ($data = $request->getPost($name, false)) {
                if (is_array($data)) {
                    return $this->getForm()->getSubForm($name);
                    break;
                }
            }
        }

        return false;
    }

    /**
     * Get the next sub form to display
     *
     * @return Zend_Form_SubForm|false
     */
    public function getNextSubForm()
    {
        $storedForms    = $this->getStoredForms();
        $potentialForms = $this->getPotentialForms();

        foreach ($potentialForms as $name) {
            if (!in_array($name, $storedForms)) {
                return $this->getForm()->getSubForm($name);
            }
        }

        return false;
    }
}
```

The above methods allow us to use notations such as "`$subForm = $this->getCurrentSubForm();`" to retrieve the current sub form for validation, or "`$next = $this->getNextSubForm();`" to get the next one to display.

Now, let's figure out how to process and display the various sub forms. We can use `getCurrentSubForm()` to determine if any sub forms have been submitted (false return values indicate none have been displayed or submitted), and `getNextSubForm()` to retrieve a form to display. We can then use the form's `prepareSubForm()` method to ensure the form is ready for display.

When we have a form submission, we can validate the sub form, and then check to see if the entire form is now valid. To do these tasks, we'll need additional methods that ensure that submitted data is added to the session, and that when validating the form entire, we validate against all segments from the session:

```
class My_Form_Registration extends Zend_Form
{
    // ...

    /**
     * Is the sub form valid?
     *
     * @param  Zend_Form_SubForm $subForm
     * @param  array $data
     * @return bool
     */
    public function subFormIsValid(Zend_Form_SubForm $subForm,
                                   array $data)
    {
        $name = $subForm->getName();
        if ($subForm->isValid($data)) {
            $this->getSessionNamespace()->$name = $subForm->getValues();
            return true;
        }

        return false;
    }

    /**
     * Is the full form valid?
     *
     * @return bool
     */
    public function formIsValid()
    {
        $data = array();
        foreach ($this->getSessionNamespace() as $key => $info) {
            $data[$key] = $info;
        }

        return $this->getForm()->isValid($data);
    }
}
```

Now that we have the legwork out of the way, let's build the actions for this controller. We'll need a landing page for the form, and then a 'process' action for processing the form.

```
class RegistrationController extends Zend_Controller_Action
{
    // ...

    public function indexAction()
    {
        // Either re-display the current page, or grab the "next"
        // (first) sub form
        if (!$form = $this->getCurrentSubForm()) {
            $form = $this->getNextSubForm();
        }
        $this->view->form = $this->getForm()->prepareSubForm($form);
    }

    public function processAction()
    {
        if (!$form = $this->getCurrentSubForm()) {
            return $this->_forward('index');
        }

        if (!$this->subFormIsValid($form,
                                   $this->getRequest()->getPost())) {
            $this->view->form = $this->getForm()->prepareSubForm($form);
            return $this->render('index');
        }

        if (!$this->formIsValid()) {
            $form = $this->getNextSubForm();
            $this->view->form = $this->getForm()->prepareSubForm($form);
            return $this->render('index');
        }

        // Valid form!
        // Render information in a verification page
        $this->view->info = $this->getSessionNamespace();
        $this->render('verification');
    }
}
```

As you'll notice, the actual code for processing the form is relatively simple. We check to see if we have a current sub form submission, and if not, we go back to the landing page. If we do have a sub form, we attempt to validate it, redisplaying it if it fails. If the sub form is valid, we then check to see if the form is valid, which would indicate we're done; if not, we display the next form segment. Finally, we display a verification page with the contents of the session.

The view scripts are very simple:

```
<? // registration/index.phtml ?>
<h2>Registration</h2>
<?= $this->form ?>

<? // registration/verification.phtml ?>
<h2>Thank you for registering!</h2>
<p>
    Here is the information you provided:
</p>

<?
// Have to do this construct due to how items are stored in session
// namespaces
foreach ($this->info as $info):
    foreach ($info as $form => $data): ?>
<h4><?= ucfirst($form) ?>:</h4>
<dl>
    <? foreach ($data as $key => $value): ?>
    <dt><?= ucfirst($key) ?></dt>
    <? if (is_array($value)):
        foreach ($value as $label => $val): ?>
    <dd><?= $val ?></dd>
        <? endforeach;
       else: ?>
    <dd><?= $this->escape($value) ?></dd>
    <? endif;
    endforeach; ?>
</dl>
<? endforeach;
endforeach ?>
```

Upcoming releases of Zend Framework will include components to make multi page forms simpler by abstracting the session and ordering logic. In the meantime, the above example should serve as a reasonable guideline on how to accomplish this task for your site.

Chapter 20. Zend_Gdata

Introduction to Gdata

Google Data APIs provide programmatic interface to some of Google's online services. The Google data Protocol is based upon the Atom Publishing Protocol [http://ietfreport.isoc.org/idref/draft-ietf-atompub-protocol/] and allows client applications to retrieve data matching queries, post data, update data and delete data using standard HTTP and the Atom syndication formation. The Zend_Gdata component is a PHP 5 interface for accessing Google Data from PHP. The Zend_Gdata component also supports accessing other services implementing the Atom Publishing Protocol.

See http://code.google.com/apis/gdata/ for more information about Google Data API.

The services that are accessible by Zend_Gdata include the following:

- Google Calendar is a popular online calendar application.
- Google Spreadsheets provides an online collaborative spreadsheets tool which can be used as a simple data store for your applications.
- Google Documents List provides an online list of all spreadsheets, word processing documents, and presentations stored in a Google account.
- Google Provisioning provides the ability to create, retrieve, update, and delete user accounts, nicknames, and email lists on a Google Apps hosted domain.
- Google Base provides the ability to retrieve, post, update, and delete items in Google Base.
- YouTube provides the ability to search and retrieve videos, comments, favorites, subscriptions, user profiles and more.
- Picasa Web Albums provides an online photo sharing application.
- Google Blogger [http://code.google.com/apis/blogger/developers_guide_php.html] is a popular Internet provider of "push-button publishing" and syndication.
- Google CodeSearch allows you to search public source code from many projects.
- Google Notebook allows you to view public Notebook content.

Unsupported services

Zend_Gdata does not provide an interface to any other Google service, such as Search, Gmail, Translation, or Maps. Only services that support the Google Data API are supported.

Structure of Zend_Gdata

Zend_Gata is composed of several types of classes:

- Service classes - inheriting from Zend_Gdata_App. These also include other classes such as Zend_Gdata, Zend_Gdata_Spreadsheets, etc. These classes enable interacting with APP or GData services and provide the ability to retrieve feeds, retrieve entries, post entries, update entries and delete entries.

- Query classes - inheriting from Zend_Gdata_Query. These also include other classes for specific services, such as Zend_Gdata_Spreadsheets_ListQuery and Zend_Gdata_Spreadsheets_CellQuery. Query classes provide methods used to construct a query for data to be retrieved from GData services. Methods include getters and setters like `setUpdatedMin()`, `setStartIndex()`, and `getPublishedMin()`. The query classes also have a method to generate a URL representing the constructed query -- `getQueryUrl`. Alternatively, the query string component of the URL can be retrieved used the `getQueryString()` method.

- Feed classes - inheriting from Zend_Gdata_App_Feed. These also include other classes such as Zend_Gdata_Feed, Zend_Gdata_Spreadsheets_SpreadsheetFeed, and Zend_Gdata_Spreadsheets_ListFeed. These classes represent feeds of entries retrieved from services. They are primarily used to retrieve data returned from services.

- Entry classes - inheriting from Zend_Gdata_App_Entry. These also include other classes such as Zend_Gdata_Entry, and Zend_Gdata_Spreadsheets_ListEntry. These classes represent entries retrieved from services or used for constructing data to send to services. In addition to being able to set the properties of an entry (such as the spreadsheet cell value), you can use an entry object to send update or delete requests to a service. For example, you can call `$entry->save()` to save changes made to an entry back to service from which the entry initiated, or `$entry->delete()` to delete an entry from the server.

- Other Data model classes - inheriting from Zend_Gdata_App_Extension. These include classes such as Zend_Gdata_App_Extension_Title (representing the atom:title XML element), Zend_Gdata_Extension_When (representing the gd:when XML element used by the GData Event "Kind"), and Zend_Gdata_Extension_Cell (representing the gs:cell XML element used by Google Spreadsheets). These classes are used purely to store the data retrieved back from services and for constructing data to be sent to services. These include getters and setters such as `setText()` to set the child text node of an element, `getText()` to retrieve the text node of an element, `getStartTime()` to retrieve the start time attribute of a When element, and other similiar methods. The data model classes also include methods such as `getDOM()` to retrieve a DOM representation of the element and all children and `transferFromDOM()` to construct a data model representation of a DOM tree.

Interacting with Google Services

Google data services are based upon the Atom Publishing Protocol (APP) and the Atom syndication format. To interact with APP or Google services using the Zend_Gdata component, you need to use the service classes such as Zend_Gdata_App, Zend_Gdata, Zend_Gdata_Spreadsheets, etc. These service classes provide methods to retrieve data from services as feeds, insert new entries into feeds, update entries, and delete entries.

Note: A full example of working with Zend_Gdata is available in the `demos/Zend/Gdata` directory. This example is runnable from the command-line, but the methods contained within are easily portable to a web application.

Obtaining instances of Zend_Gdata classes

The Zend Framework naming standards require that all classes be named based upon the directory structure in which they are located. For instance, extensions related to Spreadsheets are stored in: `Zend/Gdata/Spreadsheets/Extension/...` and, as a result of this, are named `Zend_Gdata_Spreadsheets_Extension_...`. This causes a lot of typing if you're trying to construct a new instance of a spreadsheet cell element!

We've implemented a magic factory method in all service classes (such as Zend_Gdata_App, Zend_Gdata, Zend_Gdata_Spreadsheets) that should make constructing new instances of data model, query and other classes much easier. This magic factory is implemented by using the magic `__call` method to intercept all attempts to call `$service->newXXX(arg1, arg2, ...)`. Based off the value of XXX, a search is performed in all registered 'packages' for the desired class. Here's some examples:

```
$ss = new Zend_Gdata_Spreadsheets();

// creates a Zend_Gdata_App_Spreadsheets_CellEntry
$entry = $ss->newCellEntry();

// creates a Zend_Gdata_App_Spreadsheets_Extension_Cell
$cell = $ss->newCell();
$cell->setText('My cell value');
$cell->setRow('1');
$cell->setColumn('3');
$entry->cell = $cell;

// ... $entry can then be used to send an update to a Google Spreadsheet
```

Each service class in the inheritance tree is responsible for registering the appropriate 'packages' (directories) which are to be searched when calling the magic factory method.

Google Data Client Authentication

Most Google Data services require client applications to authenticate against the Google server before accessing private data, or saving or deleting data. There are two implementations of authentication for Google Data: AuthSub and ClientLogin. Zend_Gdata offers class interfaces for both of these methods.

Most other types of queries against Google Data services do not require authentication.

Dependencies

Zend_Gdata makes use of Zend_Http_Client to send requests to google.com and fetch results. The response to most Google Data requests is returned as a subclass of the Zend_Gdata_App_Feed or Zend_Gdata_App_Entry classes.

Zend_Gdata assumes your PHP application is running on a host that has a direct connection to the Internet. The Zend_Gdata client operates by contacting Google Data servers.

Creating a new Gdata client

Create a new object of class Zend_Gdata_App, Zend_Gdata, or one of the subclasses available that offer helper methods for service-specific behavior.

The single optional parameter to the Zend_Gdata_App constructor is an instance of Zend_Http_Client. If you don't pass this parameter, Zend_Gdata creates a default Zend_Http_Client object, which will not have associated credentials to access private feeds. Specifying the Zend_Http_Client object also allows you to pass configuration options to that client object.

```
$client = new Zend_Http_Client();
```

```
$client->setConfig( ...options... );

$gdata = new Zend_Gdata($client);
```

Also see the sections on authentication for methods to create an authenticated Zend_Http_Client object.

Common query parameters

You can specify parameters to customize queries with Zend_Gdata. Query parameters are specified using subclasses of Zend_Gdata_Query. The Zend_Gdata_Query class includes methods to set all query parameters used throughout GData services. Individual services, such as Spreadsheets, also provide query classes to defined parameters which are custom to the particular service and feeds. Spreadsheets includes a CellQuery class to query the Cell Feed and a ListQuery class to query the List Feed, as different query parameters are applicable to each of those feed types. The GData-wide parameters are described below.

- The `q` parameter specifies a full-text query. The value of the parameter is a string.

  Set this parameter with the `setQuery()` function.

- The `alt` parameter specifies the feed type. The value of the parameter can be `atom`, `rss`, `json`, or `json-in-script`. If you don't specify this parameter, the default feed type is `atom`. NOTE: Only the output of the atom feed format can be processed using `Zend_Gdata`. The `Zend_Http_Client` could be used to retrieve feeds in other formats, using query URLs generated by the `Zend_Gdata_Query` class and its subclasses.

  Set this parameter with the `setAlt()` function.

- The `maxResults` parameter limits the number of entries in the feed. The value of the parameter is an integer. The number of entries returned in the feed will not exceed this value.

  Set this parameter with the `setMaxResults()` function.

- The `startIndex` parameter specifies the ordinal number of the first entry returned in the feed. Entries before this number are skipped.

  Set this parameter with the `setStartIndex()` function.

- The `updatedMin` and `updatedMax` parameters specify bounds on the entry date. If you specify a value for `updatedMin`, no entries that were updated earlier than the date you specify are included in the feed. Likewise no entries updated after the date specified by `updatedMax` are included.

  You can use numeric timestamps, or a variety of date/time string representations as the value for these parameters.

  Set this parameter with the `setUpdatedMin()` and `setUpdatedMax()` functions.

There is a `get` function for each `set` function.

```
$query = new Zend_Gdata_Query();
$query->setMaxResults(10);
echo $query->getMaxResults();   // returns 10
```

The Zend_Gdata class also implements "magic" getter and setter methods, so you can use the name of the parameter as a virtual member of the class.

```
$query = new Zend_Gdata_Query();
$query->maxResults = 10;
echo $query->maxResults;        // returns 10
```

You can clear all parameters with the `resetParameters()` function. This is useful to do if you reuse a Zend_Gdata object for multiple queries.

```
$query = new Zend_Gdata_Query();
$query->maxResults = 10;
// ...get feed...

$query->resetParameters();      // clears all parameters
// ...get a different feed...
```

Fetching a feed

Use the `getFeed()` function to retrieve a feed from a specified URI. This function returns an instance of class specified as the second argument to getFeed, which defaults to Zend_Gdata_Feed.

```
$gdata = new Zend_Gdata();
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$query->setMaxResults(10);
$feed = $gdata->getFeed($query);
```

See later sections for special functions in each helper class for Google Data services. These functions help you to get feeds from the URI that is appropriate for the respective service.

Working with multi-page feeds

When retrieving a feed that contains a large number of entries, the feed may be broken up into many smaller "pages" of feeds. When this occurs, each page will contain a link to the next page in the series. This link can be accessed by calling `getLink('next')`. The following example shows how to retrieve the next page of a feed:

```
function getNextPage($feed) {
    $nextURL = $feed->getLink('next');
    if ($nextURL !== null) {
        return $gdata->getFeed($nextURL);
    } else {
        return null;
    }
}
```

If you would prefer not to work with pages in your application, pass the first page of the feed into `Zend_Gdata_App::retrieveAllEntriesForFeed()`, which will consolidate all entries from each page into a single feed. This example shows how to use this function:

```
$gdata = new Zend_Gdata();
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$feed = $gdata->retrieveAllEntriesForFeed($gdata->getFeed($query));
```

Keep in mind when calling this function that it may take a long time to complete on large feeds. You may need to increase PHP's execution time limit by calling `set_time_limit()`.

Working with data in feeds and entries

After retrieving a feed, you can read the data from the feed or the entries contained in the feed using either the accessors defined in each of the data model classes or the magic accessors. Here's an example:

```
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$gdata = new Zend_Gdata($client);
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$query->setMaxResults(10);
$feed = $gdata->getFeed($query);
foreach ($feed as $entry) {
    // using the magic accessor
    echo 'Title: ' . $entry->title->text;
    // using the defined accessors
    echo 'Content: ' . $entry->getContent()->getText();
}
```

Updating entries

After retrieving an entry, you can update that entry and save changes back to the server. Here's an example:

```
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$gdata = new Zend_Gdata($client);
$query = new Zend_Gdata_Query(
        'http://www.blogger.com/feeds/blogID/posts/default');
$query->setMaxResults(10);
```

```
$feed = $gdata->getFeed($query);
foreach ($feed as $entry) {
    // update the title to append 'NEW'
    echo 'Old Title: ' . $entry->title->text;
    $entry->title->text = $entry->title->text . ' NEW';

    // update the entry on the server
    $newEntry = $entry->save();
    echo 'New Title: ' . $newEntry->title->text;
}
```

Posting entries to Google servers

The Zend_Gdata object has a function `post()` with which you can upload data to save new entries to Google Data services.

You can use the data model classes for each service to construct the appropriate entry to post to Google's services. The `post()` function will accept a child of Zend_Gdata_App_Entry as data to post to the service. The method returns a child of Zend_Gdata_App_Entry which represents the state of the entry as it was returned from the server.

Alternatively, you could construct the XML structure for an entry as a string and pass the string to the `post()` function.

```
$gdata = new Zend_Gdata($authenticatedHttpClient);

$entry = $gdata->newEntry();
$entry->title = $gdata->newTitle('Playing football at the park');
$content =
    $gdata->newContent('We will visit the park and play football');
$content->setType('text');
$entry->content = $content;

$entryResult = $gdata->insertEntry($entry,
        'http://www.blogger.com/feeds/blogID/posts/default');

echo 'The <id> of the resulting entry is: ' . $entryResult->id->text;
```

To post entries, you must be using an authenticated Zend_Http_Client that you created using the Zend_Gdata_AuthSub or Zend_Gdata_ClientLogin classes.

Deleting entries on Google servers

Option 1: The Zend_Gdata object has a function `delete()` with which you can delete entries from Google Data services. Pass the edit URL value from a feed entry to the `delete()` method.

Option 2: Alternatively, you can call `$entry->delete()` on an entry retrieved from a Google service.

```
$gdata = new Zend_Gdata($authenticatedHttpClient);
// a Google Data feed
$feedUri = ...;
$feed = $gdata->getFeed($feedUri);
foreach ($feed as $feedEntry) {
    // Option 1 - delete the entry directly
    $feedEntry->delete();
    // Option 2 - delete the entry by passing the edit URL to
    // $gdata->delete()
    // $gdata->delete($feedEntry->getEditLink()->href);
}
```

To delete entries, you must be using an authenticated Zend_Http_Client that you created using the Zend_Gdata_AuthSub or Zend_Gdata_ClientLogin classes.

Authenticating with AuthSub

The AuthSub mechanism enables you to write web applications that acquire authenticated access Google Data services, without having to write code that handles user credentials.

See http://code.google.com/apis/accounts/AuthForWebApps.html for more information about Google Data AuthSub authentication.

The Google documentation says the ClientLogin mechanism is appropriate for "installed applications" whereas the AuthSub mechanism is for "web applications." The difference is that AuthSub requires interaction from the user, and a browser interface that can react to redirection requests. The ClientLogin solution uses PHP code to supply the account credentials; the user is not required to enter her credentials interactively.

The account credentials supplied via the AuthSub mechanism are entered by the user of the web application. Therefore they must be account credentials that are known to that user.

Registered applications

Zend_Gdata currently does not support use of secure tokens, because the AuthSub authentication does not support passing a digital certificate to acquire a secure token.

Creating an AuthSub authenticated Http Client

Your PHP application should provide a hyperlink to the Google URL that performs authentication. The static function `Zend_Gdata_AuthSub::getAuthSubTokenUri()` provides the correct URL. The arguments to this function include the URL to your PHP applicaion so that Google can redirect the user's browser back to your application after the user's credentials have been verified.

After Google's authentication server redirects the user's browser back to the current application, a GET request parameter is set, called `token`. The value of this parameter is a single-use token that can be used for authenticated access. This token can be converted into a multi-use token and stored in your session.

Then use the token value in a call to `Zend_Gdata_AuthSub::getHttpClient()`. This function returns an instance of Zend_Http_Client, with appropriate headers set so that subsequent requests your application submits using that Http Client are also authenticated.

Below is an example of PHP code for a web application to acquire authentication to use the Google Calendar service and create a Zend_Gdata client object using that authenticated Http Client.

```
$my_calendar = 'http://www.google.com/calendar/feeds/default/private/full';

if (!isset($_SESSION['cal_token'])) {
    if (isset($_GET['token'])) {
        // You can convert the single-use token to a session token.
        $session_token =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
        // Store the session token in our session.
        $_SESSION['cal_token'] = $session_token;
    } else {
        // Display link to generate single-use token
        $googleUri = Zend_Gdata_AuthSub::getAuthSubTokenUri(
            'http://'. $_SERVER['SERVER_NAME'] . $_SERVER['REQUEST_URI'],
            $my_calendar, 0, 1);
        echo "Click <a href='$googleUri'>here</a> " .
             "to authorize this application.";
        exit();
    }
}

// Create an authenticated HTTP Client to talk to Google.
$client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['cal_token']);

// Create a Gdata object using the authenticated Http Client
$cal = new Zend_Gdata_Calendar($client);
```

Revoking AuthSub authentication

To terminate the authenticated status of a given token, use the `Zend_Gdata_AuthSub::AuthSubRevokeToken()` static function. Otherwise, the token is still valid for some time.

```
// Carefully construct this value to avoid application security problems.
$php_self = htmlentities(substr($_SERVER['PHP_SELF'],
                         0,
                         strcspn($_SERVER['PHP_SELF'], "\n\r")),
                         ENT_QUOTES);

if (isset($_GET['logout'])) {
    Zend_Gdata_AuthSub::AuthSubRevokeToken($_SESSION['cal_token']);
    unset($_SESSION['cal_token']);
    header('Location: ' . $php_self);
    exit();
}
```

Security notes

The treatment of the `$php_self` variable in the example above is a general security guideline, it is not specific to Zend_Gdata. You should always filter content you output to http headers.

Regarding revoking authentication tokens, it is recommended to do this when the user is finished with her Google Data session. The possibility that someone can intercept the token and use it for malicious purposes is very small, but nevertheless it is a good practice to terminate authenticated access to any service.

Authenticating with ClientLogin

The ClientLogin mechanism enables you to write PHP application that acquire authenticated access to Google Services, specifying a user's credentials in the Http Client.

See http://code.google.com/apis/accounts/AuthForInstalledApps.html [http://code.google.com/apis/accounts/AuthForInstalledApps.html] for more information about Google Data ClientLogin authentication.

The Google documentation says the ClientLogin mechanism is appropriate for "installed applications" whereas the AuthSub mechanism is for "web applications." The difference is that AuthSub requires interaction from the user, and a browser interface that can react to redirection requests. The ClientLogin solution uses PHP code to supply the account credentials; the user is not required to enter her credentials interactively.

The account credentials supplied via the ClientLogin mechanism must be valid credentials for Google services, but they are not required to be those of the user who is using the PHP application.

Creating a ClientLogin authenticated Http Client

The process of creating an authenticated Http client using the ClientLogin mechanism is to call the static function `Zend_Gdata_ClientLogin::getHttpClient()` and pass the Google account credentials in plain text. The return value of this function is an object of class Zend_Http_Client.

The optional third parameter is the name of the Google Data service. For instance, this can be 'cl' for Google Calendar. The default is "xapi", which is recognized by Google Data servers as a generic service name.

The optional fourth parameter is an instance of Zend_Http_Client. This allows you to set options in the client, such as proxy server settings. If you pass `null` for this parameter, a generic Zend_Http_Client object is created.

The optional fifth parameter is a short string that Google Data servers use to identify the client application for logging purposes. By default this is string "Zend-ZendFramework";

The optional sixth parameter is a string ID for a CAPTCHA™ challenge that has been issued by the server. It is only necessary when logging in after receiving a CAPTCHA™ challenge from a previous login attempt.

The optional seventh parameter is a user's response to a CAPTCHA™ challenge that has been issued by the server. It is only necessary when logging in after receiving a CAPTCHA™ challenge from a previous login attempt.

Below is an example of PHP code for a web application to acquire authentication to use the Google Calendar service and create a Zend_Gdata client object using that authenticated Zend_Http_Client.

```
// Enter your Google account credentials
$email = 'johndoe@gmail.com';
$passwd = 'xxxxxxxx';
try {
   $client = Zend_Gdata_ClientLogin::getHttpClient($email, $passwd, 'cl');
} catch (Zend_Gdata_App_CaptchaRequiredException $cre) {
    echo 'URL of CAPTCHA image: ' . $cre->getCaptchaUrl() . "\n";
    echo 'Token ID: ' . $cre->getCaptchaToken() . "\n";
} catch (Zend_Gdata_App_AuthException $ae) {
   echo 'Problem authenticating: ' . $ae->exception() . "\n";
}

$cal = new Zend_Gdata_Calendar($client);
```

Terminating a ClientLogin authenticated Http Client

There is no method to revoke ClientLogin authentication as there is in the AuthSub token-based solution. The credentials used in the ClientLogin authentication are the login and password to a Google account, and therefore these can be used repeatedly in the future.

Using Google Calendar

You can use the `Zend_Gdata_Calendar` class to view, create, update, and delete events in the online Google Calendar service.

See http://code.google.com/apis/calendar/overview.html for more information about the Google Calendar API.

Connecting To The Calendar Service

The Google Calendar API, like all GData APIs, is based off of the Atom Publishing Protoco (APP), an XML based format for managing web-based resources. Traffic between a client and the Google Calendar servers occurs over HTTP and allows for both authenticated and unauthenticated connections.

Before any transactions can occur, this connection needs to be made. Creating a connection to the calendar servers involves two steps: creating an HTTP client and binding a `Zend_Gdata_Calendar` service instance to that client.

Authentication

The Google Calendar API allows access to both public and private calendar feeds. Public feeds do not require authentication, but are read-only and offer reduced functionality. Private feeds offers the most complete functionality but requires an authenticated connection to the calendar servers. There are three authentication schemes that are supported by Google Calendar:

- *ClientAuth* provides direct username/password authentication to the calendar servers. Since this scheme requires that users provide your application with their password, this authentication is only recommended when other authentication schemes are insufficient.

- *AuthSub* allows authentication to the calendar servers via a Google proxy server. This provides the same level of convenience as ClientAuth but without the security risk, making this an ideal choice for web-based applications.
- *MagicCookie* allows authentication based on a semi-random URL available from within the Google Calendar interface. This is the simplest authentication scheme to implement, but requires that users manually retrieve their secure URL before they can authenticate, doesn't provide access to calendar lists, and is limited to read-only access.

The `Zend_Gdata` library provides support for all three authentication schemes. The rest of this chapter will assume that you are familiar the authentication schemes available and how to create an appropriate authenticated connection. For more information, please see section the Authentication section of this manual or the Authentication Overview in the Google Data API Developer's Guide [http://code.google.com/apis/gdata/auth.html].

Creating A Service Instance

In order to interact with Google Calendar, this library provides the `Zend_Gdata_Calendar` service class. This class provides a common interface to the Google Data and Atom Publishing Protocol models and assists in marshaling requests to and from the calendar servers.

Once deciding on an authentication scheme, the next step is to create an instance of `Zend_Gdata_Calendar`. The class constructor takes an instance of `Zend_Http_Client` as a single argument. This provides an interface for AuthSub and ClientAuth authentication, as both of these require creation of a special authenticated HTTP client. If no arguments are provided, an unauthenticated instance of `Zend_Http_Client` will be automatically created.

The example below shows how to create a Calendar service class using ClientAuth authentication:

```
// Parameters for ClientAuth authentication
$service = Zend_Gdata_Calendar::AUTH_SERVICE_NAME;
$user = "sample.user@gmail.com";
$pass = "pa$$w0rd";

// Create an authenticated HTTP client
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);

// Create an instance of the Calendar service
$service = new Zend_Gdata_Calendar($client);
```

A Calendar service using AuthSub can be created in a similar, though slightly more lengthy fashion:

```
/*
 * Retrieve the current URL so that the AuthSub server knows where to
 * redirect the user after authentication is complete.
 */
function getCurrentUrl()
{
    global $_SERVER;
    // Filter php_self to avoid a security vulnerability.
    $php_request_uri =
        htmlentities(substr($_SERVER['REQUEST_URI'],
                            0,
                            strcspn($_SERVER['REQUEST_URI'], "\n\r")),
                            ENT_QUOTES);

    if (isset($_SERVER['HTTPS']) &&
        strtolower($_SERVER['HTTPS']) == 'on') {
        $protocol = 'https://';
    } else {
        $protocol = 'http://';
    }
    $host = $_SERVER['HTTP_HOST'];
    if ($_SERVER['HTTP_PORT'] != '' &&
        (($protocol == 'http://' && $_SERVER['HTTP_PORT'] != '80') ||
        ($protocol == 'https://' && $_SERVER['HTTP_PORT'] != '443'))) {
        $port = ':' . $_SERVER['HTTP_PORT'];
    } else {
        $port = '';
    }
    return $protocol . $host . $port . $php_request_uri;
}

/**
 * Obtain an AuthSub authenticated HTTP client, redirecting the user
 * to the AuthSub server to login if necessary.
 */
function getAuthSubHttpClient()
{
    global $_SESSION, $_GET;

    // If there is no AuthSub session or one-time token waiting for us,
    // redirect the user to the AuthSub server to get one.
    if (!isset($_SESSION['sessionToken']) && !isset($_GET['token'])) {
        // Parameters to give to AuthSub server
        $next = getCurrentUrl();
        $scope = "http://www.google.com/calendar/feeds/";
        $secure = false;
        $session = true;

        // Redirect the user to the AuthSub server to sign in

        $authSubUrl = Zend_Gdata_AuthSub::getAuthSubTokenUri($next,
                                                             $scope,
                                                             $secure,
                                                             $session);
         header("HTTP/1.0 307 Temporary redirect");

         header("Location: " . $authSubUrl);

         exit();
    }
    // Convert an AuthSub one-time token into a session token if needed
    if (!isset($_SESSION['sessionToken']) && isset($_GET['token'])) {
        $_SESSION['sessionToken'] =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
    }

    // At this point we are authenticated via AuthSub and can obtain an
    // authenticated HTTP client instance

    // Create an authenticated HTTP client
    $client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['sessionToken']);
    return $client;
}

// -> Script execution begins here <-

// Make sure that the user has a valid session, so we can record the
// AuthSub session token once it is available.
session_start();

// Create an instance of the Calendar service, redirecting the user
// to the AuthSub server if necessary.
$service = new Zend_Gdata_Calendar(getAuthSubHttpClient());
```

Finally, an unauthenticated server can be created for use with either public feeds or MagicCookie authentication:

```
// Create an instance of the Calendar service using an unauthenticated
// HTTP client

$service = new Zend_Gdata_Calendar();
```

Note that MagicCookie authentication is not supplied with the HTTP connection, but is instead specified along with the desired visibility when submitting queries. See the section on retrieving events below for an example.

Retrieving A Calendar List

The calendar service supports retrieving a list of calendars for the authenticated user. This is the same list of calendars which are displayed in the Google Calendar UI, except those marked as "`hidden`" are also available.

The calendar list is always private and must be accessed over an authenticated connection. It is not possible to retrieve another user's calendar list and it cannot be accessed using MagicCookie authentication. Attempting to access a calendar list without holding appropriate credentials will fail and result in a 401 (Authentication Required) status code.

```
$service = Zend_Gdata_Calendar::AUTH_SERVICE_NAME;
```

```
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Calendar($client);

try {
    $listFeed= $service->getCalendarListFeed();
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Calling `getCalendarListFeed()` creates a new instance of `Zend_Gdata_Calendar_ListFeed` containing each available calendar as an instance of `Zend_Gdata_Calendar_ListEntry`. After retrieving the feed, you can use the iterator and accessors contained within the feed to inspect the enclosed calendars.

```
echo "<h1>Calendar List Feed</h1>";
echo "<ul>";
foreach ($listFeed as $calendar) {
    echo "<li>" . $calendar->title .
         " (Event Feed: " . $calendar->id . ")</li>";
}
echo "</ul>";
```

Retrieving Events

Like the list of calendars, events are also retrieved using the `Zend_Gdata_Calendar` service class. The event list returned is of type `Zend_Gdata_Calendar_EventFeed` and contains each event as an instance of `Zend_Gdata_Calendar_EventEntry`. As before, the iterator and accessors contained within the event feed instance allow inspection of individual events.

Queries

When retrieving events using the Calendar API, specially constructed query URLs are used to describe what events should be returned. The `Zend_Gdata_Calendar_EventQuery` class simplifies this task by automatically constructing a query URL based on provided parameters. A full list of these parameters is available at the Queries section of the Google Data APIs Protocol Reference [http://code.google.com/apis/gdata/reference.html#Queries] . However, there are three parameters that are worth special attention:

- *User* is used to specify the user whose calendar is being searched for, and is specified as an email address. If no user is provided, "default" will be used instead to indicate the currently authenticated user (if authenticated).

- *Visibility* specifies whether a users public or private calendar should be searched. If using an unauthenticated session and no MagicCookie is available, only the public feed will be available.

- *Projection* specifies how much data should be returned by the server and in what format. In most cases you will want to use the "full" projection. Also available is the "basic" projection, which places most meta-data into each event's content field as human readable text, and the "composite" projection which includes complete text for any comments alongside each event. The "composite" view is often much larger than the "full" view.

Retrieving Events In Order Of Start Time

The example below illustrates the use of the `Zend_Gdata_Query` class and specifies the private visibility feed, which requires that an authenticated connection is available to the calendar servers. If a MagicCookie is being used for authentication, the visibility should be instead set to "`private-magicCookieValue`", where magicCookieValue is the random string obtained when viewing the private XML address in the Google Calendar UI. Events are requested chronologically by start time and only events occurring in the future are returned.

```
$query = $service->newEventQuery();
$query->setUser('default');
// Set to $query->setVisibility('private-magicCookieValue') if using
// MagicCookie auth
$query->setVisibility('private');
$query->setProjection('full');
$query->setOrderby('starttime');
$query->setFutureevents('true');

// Retrieve the event list from the calendar server
try {
    $eventFeed = $service->getCalendarEventFeed($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}

// Iterate through the list of events, outputting them as an HTML list
echo "<ul>";
foreach ($eventFeed as $event) {
    echo "<li>" . $event->title . " (Event ID: " . $event->id . ")</li>";
}
echo "</ul>";
```

Additional properties such as ID, author, when, event status, visibility, web content, and content, among others are available within `Zend_Gdata_Calendar_EventEntry`. Refer to the Zend Framework API Documentation [http://framework.zend.com/apidoc/core/] and the Calendar Protocol Reference [http://code.google.com/apis/gdata/reference.html] for a complete list.

Retrieving Events In A Specified Date Range

To print out all events within a certain range, for example from December 1, 2006 through December 15, 2007, add the following two lines to the previous sample. Take care to remove "`$query->setFutureevents('true')`", since´`futureevents` will override `startMin` and `startMax`.

```
$query->setStartMin('2006-12-01');
$query->setStartMax('2006-12-16');
```

Note that `startMin` is inclusive whereas `startMax` is exclusive. As a result, only events through 2006-12-15 23:59:59 will be returned.

Retrieving Events By Fulltext Query

To print out all events which contain a specific word, for example "dogfood", use the `setQuery()` method when creating the query.

```
$query->setQuery("dogfood");
```

Retrieving Individual Events

Individual events can be retrieved by specifying their event ID as part of the query. Instead of calling `getCalendarEventFeed()`, `getCalendarEventEntry()` should be called instead.

```
$query = $service->newEventQuery();
$query->setUser('default');
$query->setVisibility('private');
$query->setProjection('full');
$query->setEvent($eventId);

try {
    $event = $service->getCalendarEventEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

In a similar fashion, if the event URL is known, it can be passed directly into `getCalendarEntry()` to retrieve a specific event. In this case, no query object is required since the event URL contains all the necessary information to retrieve the event.

```
$eventURL =
    "http://www.google.com/calendar/feeds/default/private/full/g829on5sq4ag12se91d

try {
    $event = $service->getCalendarEventEntry($eventURL);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Creating Events

Creating Single-Occurrence Events

Events are added to a calendar by creating an instance of `Zend_Gdata_EventEntry` and populating it with the appropriate data. The calendar service instance (`Zend_Gdata_Calendar`) is then used to used to transparently covert the event into XML and POST it to the calendar server. Creating events requires either an AuthSub or ClientAuth authenticated connection to the calendar server.

At a minimum, the following attributes should be set:

- *Title* provides the headline that will appear above the event within the Google Calendar UI.
- *When* indicates the duration of the event and, optionally, any reminders that are associated with it. See the next section for more information on this attribute.

Other useful attributes that may optionally set include:

- *Author* provides information about the user who created the event.
- *Content* provides additional information about the event which appears when the event details are requested from within Google Calendar.
- *EventStatus* indicates whether the event is confirmed, tentative, or canceled.
- *Hidden* removes the event from the Google Calendar UI.
- *Transparency* indicates whether the event should be consume time on the user's free/busy list.
- *WebContent* allows links to external content to be provided within an event.
- *Where* indicates the location of the event.
- *Visibility* allows the event to be hidden from the public event lists.

For a complete list of event attributes, refer to the Zend Framework API Documentation [http://framework.zend.com/apidoc/core/] and the Calendar Protocol Reference [http://code.google.com/apis/gdata/reference.html]. Attributes that can contain multiple values, such as where, are implemented as arrays and need to be created accordingly. Be aware that all of these attributes require objects as parameters. Trying instead to populate them using strings or primitives will result in errors during conversion to XML.

Once the event has been populated, it can be uploaded to the calendar server by passing it as an argument to the calendar service's `insertEvent()` function.

```
// Create a new entry using the calendar service's magic factory method
$event= $service->newEventEntry();

// Populate the event with the desired information
// Note that each attribute is crated as an instance of a matching class
$event->title = $service->newTitle("My Event");
$event->where = array($service->newWhere("Mountain View, California"));
$event->content =
    $service->newContent(" This is my awesome event. RSVP required.");
```

```
// Set the date using RFC 3339 format.
$startDate = "2008-01-20";
$startTime = "14:00";
$endDate = "2008-01-20";
$endTime = "16:00";
$tzOffset = "-08";

$when = $service->newWhen();
$when->startTime = "{$startDate}T{$startTime}:00.000{$tzOffset}:00";
$when->endTime = "{$endDate}T{$endTime}:00.000{$tzOffset}:00";
$event->when = array($when);

// Upload the event to the calendar server
// A copy of the event as it is recorded on the server is returned
$newEvent = $service->insertEvent($event);
```

Event Schedules and Reminders

An event's starting time and duration are determined by the value of its when property, which contains the properties `startTime`, `endTime`, and `valueString`. `StartTime` and `EndTime` control the duration of the event, while `valueString` provides a way to store a friendly, human readable version of the duration such as "This Afternoon". Note that even when using `valueString`, `startTime` and `endTime` still must be be set to valid values.

All-day events can be scheduled by specifying only the date omitting the time when setting `startTime` and `endTime` . Likewise, zero-duration events can be specified by omitting the `endTime` . In all cases, date/time values should be provided in RFC3339 [http://www.ietf.org/rfc/rfc3339.txt] format.

```
// Schedule the event to occur on December 05, 2007 at 2 PM PST (UTC-8)
// with a duration of one hour.
$when = $service->newWhen();
$when->startTime = "2007-12-05T14:00:00-08:00";
$when->endTime="2007-12-05T15:00:00:00-08:00";

// Specify a optional human readable value for the above date
$when->valueString = "This Afternoon";

// Apply the when property to an event
$event->when = $when;
```

The `when` attribute also controls when reminders are sent to a user. Reminders are stored in an array and each event may have up to find reminders associated with it.

For a `reminder` to be valid, it needs to have two attributes set: `method` and a time. `Method` can accept one of the following strings: "alert", "email", or "sms". The time should be entered as an integer and can be set with either the property `minutes`, `hours`, `days`, or `absoluteTime`. However, a valid request may only have one of these attributes set. If a mixed time is desired, convert to the most precise unit available. For example, 1 hour and 30 minutes should be entered as 90 minutes.

```
// Create a new reminder object. It should be set to send an email
// to the user 10 minutes beforehand.
$reminder = $service->newReminder();
$reminder->method = "email";
$reminder->minutes = "10";

// Apply the reminder to an existing event's when property
$when = $event->when[0];
$when->reminders = array($reminder);
```

Creating Recurring Events

Recurring events are created the same way as single-occurrence events, except a recurrence attribute should be provided instead of a where attribute. The recurrence attribute should hold a string describing the event's recurrence pattern using properties defined in the iCalendar standard (RFC 2445 [http://www.ietf.org/rfc/rfc2445.txt]).

Exceptions to the recurrence pattern will usually be specified by a distinct `recurrenceException` attribute. However, the iCalendar standard provides a secondary format for defining recurrences, and the possibility that either may be used must be accounted for.

Due to the complexity of parsing recurrence patterns, further information on this them is outside the scope of this document. However, more information can be found in the Common Elements section of the Google Data APIs Developer Guide [http://code.google.com/apis/gdata/elements.html#gdRecurrence] , as well as in RFC 2445.

```
 // Create a new entry using the calendar service's magic factory method
$event= $service->newEventEntry();

// Populate the event with the desired information
// Note that each attribute is crated as an instance of a matching class
$event->title = $service->newTitle("My Recurring Event");
$event->where = array($service->newWhere("Palo Alto, California"));
$event->content =
    $service->newContent(' This is my other awesome event, ' .
                         ' occurring all-day every Tuesday from .
                         '2007-05-01 until 207-09-04. No RSVP required.');

// Set the duration and frequency by specifying a recurrence pattern.

$recurrence = "DTSTART;VALUE=DATE:20070501\r\n" .
        "DTEND;VALUE=DATE:20070502\r\n" .
        "RRULE:FREQ=WEEKLY;BYDAY=Tu;UNTIL=20070904\r\n";

$event->recurrence = $service->newRecurrence($recurrence);

// Upload the event to the calendar server
// A copy of the event as it is recorded on the server is returned
$newEvent = $service->insertEvent($event);
```

Using QuickAdd

QuickAdd is a feature which allows events to be created using free-form text entry. For example, the string "Dinner at Joe's Diner on Thursday" would create an event with the title "Dinner", location "Joe's Diner", and date "Thursday". To take advantage of QuickAdd, create a new `QuickAdd` property set to "true" and store the freeform text as a `content` property.

```
// Create a new entry using the calendar service's magic factory method
$event= $service->newEventEntry();

// Populate the event with the desired information
$event->content= $service->newContent("Dinner at Joe's Diner on Thursday");
$event->quickAdd = $service->newQuickAdd("true");

// Upload the event to the calendar server
// A copy of the event as it is recorded on the server is returned
$newEvent = $service->insertEvent($event);
```

Modifying Events

Once an instance of an event has been obtained, the event's attributes can be locally modified in the same way as when creating an event. Once all modifications are complete, calling the event's `save()` method will upload the changes to the calendar server and return a copy of the event as it was created on the server.

In the event another user has modified the event since the local copy was retrieved, `save()` will fail and the server will return a 409 (Conflict) status code. To resolve this a fresh copy of the event must be retrieved from the server before attempting to resubmit any modifications.

```
// Get the first event in the user's event list
$event = $eventFeed[0];

// Change the title to a new value
$event->title = $service->newTitle("Woof!");

// Upload the changes to the server
try {
    $event->save();
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Deleting Events

Calendar events can be deleted either by calling the calendar service's `delete()` method and providing the edit URL of an event or by calling an existing event's own `delete()` method.

In either case, the deleted event will still show up on a user's private event feed if an `updateMin` query parameter is provided. Deleted events can be distinguished from regular events because they will have their `eventStatus` property set to "http://schemas.google.com/g/2005#event.canceled".

```
// Option 1: Events can be deleted directly
$event->delete();
```

```
// Option 2: Events can be deleted supplying the edit URL of the event
// to the calendar service, if known
$service->delete($event->getEditLink()->href);
```

Accessing Event Comments

When using the full event view, comments are not directly stored within an entry. Instead, each event contains a URL to it's associated comment feed which must be manually requested.

Working with comments is fundamentally similar to working with events, with the only significant difference being that a different feed and event class should be used and that´ the additional meta-data for events such as where and when does not exist for comments. Specifically, the comment's author is stored in the `author` property, and the comment text is stored in the `content` property.

```
// Extract the comment URL from the first event in a user's feed list
$event = $eventFeed[0];
$commentUrl = $event->comments->feedLink->url;

// Retrieve the comment list for the event
try {
$commentFeed = $service->getFeed($commentUrl);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}

// Output each comment as an HTML list
echo "<ul>";
foreach ($commentFeed as $comment) {
    echo "<li><em>Comment By: " . $comment->author->name "</em><br/>" .
         $comment->content . "</li>";
}
echo "</ul>";
```

Using Google Documents List Data API

The Google Documents List Data API allows client applications to upload documents to Google Documents and list them in the form of Google Data API ("GData") feeds. Your client application can request a list of a user's documents, and query the content in an existing document.

See http://code.google.com/apis/documents/overview.html for more information about the Google Documents List API.

Get a List of Documents

You can get a list of the Google Documents for a particular user by using the `getDocumentListFeed` method of the docs service. The service will return a `Zend_Gdata_Docs_DocumentListFeed` object containing a list of documents associated with the authenticated user.

```
$service = Zend_Gdata_Docs::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$docs = new Zend_Gdata_Docs($client);
$feed = $docs->getDocumentListFeed();
```

The resulting `Zend_Gdata_Docs_DocumentListFeed` object represents the response from the server. This feed contains a list of `Zend_Gdata_Docs_DocumentListEntry` objects (`$feed->entries`), each of which represents a single Google Document.

Upload a Document

You can create a new Google Document by uploading a word processing document, spreadsheet, or presentation. This example is from the interactive Docs.php sample which comes with the library. It demonstrates uploading a file and printing information about the result from the server.

```
/**
 * Upload the specified document
 *
 * @param Zend_Gdata_Docs $docs The service object to use for communicating
 *     with the Google Documents server.
 * @param boolean $html True if output should be formatted for display in a
 *     web browser.
 * @param string $originalFileName The name of the file to be uploaded. The
 *     mime type of the file is determined from the extension on this file
 *     name. For example, test.csv is uploaded as a comma seperated volume
 *     and converted into a spreadsheet.
 * @param string $temporaryFileLocation (optional) The file in which the
 *     data for the document is stored. This is used when the file has been
 *     uploaded from the client's machine to the server and is stored in
 *     a temporary file which does not have an extension. If this parameter
 *     is null, the file is read from the originalFileName.
 */
function uploadDocument($docs, $html, $originalFileName,
                       $temporaryFileLocation) {
  $fileToUpload = $originalFileName;
  if ($temporaryFileLocation) {
    $fileToUpload = $temporaryFileLocation;
  }

  // Upload the file and convert it into a Google Document. The original
  // file name is used as the title of the document and the mime type
  // is determined based on the extension on the original file name.
  $newDocumentEntry = $docs->uploadFile($fileToUpload, $originalFileName,
      null, Zend_Gdata_Docs::DOCUMENTS_LIST_FEED_URI);

  echo "New Document Title: ";

  if ($html) {
      // Find the URL of the HTML view of this document.
      $alternateLink = '';
      foreach ($newDocumentEntry->link as $link) {
          if ($link->getRel() === 'alternate') {
              $alternateLink = $link->getHref();
          }
      }
      // Make the title link to the document on docs.google.com.
      echo "<a href=\"$alternateLink\">\n";
  }
  echo $newDocumentEntry->title."\n";
  if ($html) {echo "</a>\n";}
}
```

Searching the documents feed

You can search the Document List using some of the standard Google Data API query parameters [http://code.google.com/apis/gdata/reference.html#Queries]. Categories are used to restrict the type of document (word processor document, spreadsheet) returned. The full-text query string is used to search the content of all the documents. More detailed information on parameters specific to the Documents List can be found in the Documents List Data API Reference Guide [http://code.google.com/apis/documents/reference.html#Parameters].

Get a List of Word Processing Documents

You can also request a feed containing all of your documents of a specific type. For example, to see a list of your work processing documents, you would perform a category query as follows.

```
$feed = $docs->getDocumentListFeed(
    'http://docs.google.com/feeds/documents/private/full/-/document');
```

Get a List of Spreadsheets

To request a list of your Google Spreadsheets, use the following category query:

```
$feed = $docs->getDocumentListFeed(
    'http://docs.google.com/feeds/documents/private/full/-/spreadsheet');
```

Performing a text query

You can search the content of documents by using a `Zend_Gdata_Docs_Query` in your request. A Query object can be used to construct the query URI, with the search term being passed in as a parameter. Here is an example method which queries the documents list for documents which contain the search string:

```
$docsQuery = new Zend_Gdata_Docs_Query();
$docsQuery->setQuery($query);
$feed = $client->getDocumentListFeed($docsQuery);
```

Using Google Spreadsheets

The Google Spreadsheets data API allows client applications to view and update Spreadsheets content in the form of Google data API feeds. Your client application can request a list of a user's spreadsheets, edit or delete content in an existing Spreadsheets worksheet, and query the content in an existing Spreadsheets worksheet.

See http://code.google.com/apis/spreadsheets/overview.html for more information about the Google Spreadsheets API.

Create a Spreadsheet

The Spreadsheets data API does not currently provide a way to programatically create or delete a spreadsheet.

Get a List of Spreadsheets

You can get a list of spreadsheets for a particular user by using the `getSpreadsheetFeed` method of the Spreadsheets service. The service will return a `Zend_Gdata_Spreadsheets_SpreadsheetFeed` object containing a list of spreadsheets associated with the authenticated user.

```
$service = Zend_Gdata_Spreadsheets::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$spreadsheetService = new Zend_Gdata_Spreadsheets($client);
$feed = $spreadsheetService->getSpreadsheetFeed();
```

Get a List of Worksheets

A given spreadsheet may contain multiple worksheets. For each spreadsheet, there's a worksheets metafeed listing all the worksheets in that spreadsheet.

Given the spreadsheet key from the <id> of a `Zend_Gdata_Spreadsheets_SpreadsheetEntry` object you've already retrieved, you can fetch a feed containing a list of worksheets associated with that spreadsheet.

```
$query = new Zend_Gdata_Spreadsheets_DocumentQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$feed = $spreadsheetService->getWorksheetFeed($query);
```

The resulting `Zend_Gdata_Spreadsheets_WorksheetFeed` object feed represents the response from the server. Among other things, this feed contains a list of `Zend_Gdata_Spreadsheets_WorksheetEntry` objects (`$feed->entries`), each of which represents a single worksheet.

Interacting With List-based Feeds

A given worksheet generally contains multiple rows, each containing multiple cells. You can request data from the worksheet either as a list-based feed, in which each entry represents a row, or as a cell-based feed, in which each entry represents a single cell. For information on cell-based feeds, see Interacting with cell-based feeds.

The following sections describe how to get a list-based feed, add a row to a worksheet, and send queries with various query parameters.

The list feed makes some assumptions about how the data is laid out in the spreadsheet.

In particular, the list feed treats the first row of the worksheet as a header row; Spreadsheets dynamically creates XML elements named after the contents of header-row cells. Users who want to provide Gdata feeds should not put any data other than column headers in the first row of a worksheet.

The list feed contains all rows after the first row up to the first blank row. The first blank row terminates the data set. If expected data isn't appearing in a feed, check the worksheet manually to see whether there's an unexpected blank row in the middle of the data. In particular, if the second row of the spreadsheet is blank, then the list feed will contain no data.

A row in a list feed is as many columns wide as the worksheet itself.

Get a List-based Feed

To retrieve a worksheet's list feed, use the `getListFeed` method of the Spreadsheets service.

```
$query = new Zend_Gdata_Spreadsheets_ListQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$query->setWorksheetId($worksheetId);
$listFeed = $spreadsheetService->getListFeed($query);
```

The resulting `Zend_Gdata_Spreadsheets_ListFeed` object `$listfeed` represents a response from the server. Among other things, this feed contains an array of `Zend_Gdata_Spreadsheets_ListEntry` objects (`$listFeed->entries`), each of which represents a single row in a worksheet.

Each `Zend_Gdata_Spreadsheets_ListEntry` contains an array, `custom`, which contains the data for that row. You can extract and display this array:

```
$rowData = $listFeed->entries[1]->getCustom();
foreach($rowData as $customEntry) {
  echo $customEntry->getColumnName() . " = " . $customEntry->getText();
}
```

An alternate version of this array, `customByName`, allows direct access to an entry's cells by name. This is convenient when trying to access a specific header:

```
$customEntry = $listFeed->entries[1]->getCustomByName('my_heading');
echo $customEntry->getColumnName() . " = " . $customEntry->getText();
```

Reverse-sort Rows

By default, rows in the feed appear in the same order as the corresponding rows in the GUI; that is, they're in order by row number. To get rows in reverse order, set the reverse properties of the `Zend_Gdata_Spreadsheets_ListQuery` object to true:

```
$query = new Zend_Gdata_Spreadsheets_ListQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$query->setWorksheetId($worksheetId);
$query->setReverse('true');
$listFeed = $spreadsheetService->getListFeed($query);
```

Note that if you want to order (or reverse sort) by a particular column, rather than by position in the worksheet, you can set the `orderby` value of the `Zend_Gdata_Spreadsheets_ListQuery` object to `column:<the header of that column>`.

Send a Structured Query

You can set a `Zend_Gdata_Spreadsheets_ListQuery`'s `sq` value to produce a feed with entries that meet the specified criteria. For example, suppose you have a worksheet containing personnel data, in which each row represents information about a single person. You wish to retrieve all rows in which the person's name is "John" and the person's age is over 25. To do so, you would set `sq` as follows:

```
$query = new Zend_Gdata_Spreadsheets_ListQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$query->setWorksheetId($worksheetId);
```

```
$query->setSpreadsheetQuery('name=John and age>25');
$listFeed = $spreadsheetService->getListFeed($query);
```

Add a Row

Rows can be added to a spreadsheet by using the `insertRow` method of the Spreadsheet service.

```
$insertedListEntry = $spreadsheetService->insertRow($rowData,
                                                    $spreadsheetKey,
                                                    $worksheetId);
```

The `$rowData` parameter contains an array of column keys to data values. The method returns a `Zend_Gdata_Spreadsheets_SpreadsheetsEntry` object which represents the inserted row.

Spreadsheets inserts the new row immediately after the last row that appears in the list-based feed, which is to say immediately before the first entirely blank row.

Edit a Row

Once a `Zend_Gdata_Spreadsheets_ListEntry` object is fetched, its rows can be updated by using the `updateRow` method of the Spreadsheet service.

```
$updatedListEntry = $spreadsheetService->updateRow($oldListEntry,
                                                   $newRowData);
```

The `$oldListEntry` parameter contains the list entry to be updated. `$newRowData` contains an array of column keys to data values, to be used as the new row data. The method returns a `Zend_Gdata_Spreadsheets_SpreadsheetsEntry` object which represents the updated row.

Delete a Row

To delete a row, simply invoke `deleteRow` on the `Zend_Gdata_Spreadsheets` object with the existing entry to be deleted:

```
$spreadsheetService->deleteRow($listEntry);
```

Alternatively, you can call the `delete` method of the entry itself:

```
$listEntry->delete();
```

Interacting With Cell-based Feeds

In a cell-based feed, each entry represents a single cell.

Note that we don't recommend interacting with both a cell-based feed and a list-based feed for the same worksheet at the same time.

Get a Cell-based Feed

To retrieve a worksheet's cell feed, use the `getCellFeed` method of the Spreadsheets service.

```
$query = new Zend_Gdata_Spreadsheets_CellQuery();
$query->setSpreadsheetKey($spreadsheetKey);
$query->setWorksheetId($worksheetId);
$cellFeed = $spreadsheetService->getCellFeed($query);
```

The resulting `Zend_Gdata_Spreadsheets_CellFeed` object `$cellFeed` represents a response from the server. Among other things, this feed contains an array of `Zend_Gdata_Spreadsheets_CellEntry` objects (`$cellFeed>entries`), each of which represents a single cell in a worksheet. You can display this information:

```
foreach($cellFeed as $cellEntry) {
  $row = $cellEntry->cell->getRow();
  $col = $cellEntry->cell->getColumn();
  $val = $cellEntry->cell->getText();
  echo "$row, $col = $val\n";
}
```

Send a Cell Range Query

Suppose you wanted to retrieve the cells in the first column of a worksheet. You can request a cell feed containing only this column as follows:

```
$query = new Zend_Gdata_Spreadsheets_CellQuery();
$query->setMinCol(1);
$query->setMaxCol(1);
$query->setMinRow(2);
$feed = $spreadsheetService->getCellsFeed($query);
```

This requests all the data in column 1, starting with row 2.

Change Contents of a Cell

To modify the contents of a cell, call `updateCell` with the row, column, and new value of the cell.

```
$updatedCell = $spreadsheetService->updateCell($row,
                                               $col,
                                               $inputValue,
                                               $spreadsheetKey,
                                               $worksheetId);
```

The new data is placed in the specified cell in the worksheet. If the specified cell contains data already, it will be overwritten. Note: Use `updateCell` to change the data in a cell, even if the cell is empty.

Using Google Apps Provisioning

Google Apps is a service which allows domain administrators to offer their users managed access to Google services such as Mail, Calendar, and Docs & Spreadsheets. The Provisioning API offers a programatic interface to configure this service. Specifically, this API allows administrators the ability to create, retrieve, update, and delete user accounts, nicknames, and email lists.

This library implements version 2.0 of the Provisioning API. Access to your account via the Provisioning API must be manually enabled for each domain using the Google Apps control panel. Only certain account types are able to enable this feature.

For more information on the Google Apps Provisioning API, including instructions for enabling API access, refer to the Provisioning API V2.0 Reference [http://code.google.com/apis/calendar/overview.html].

Authentication

The Provisioning API does not support authentication via AuthSub and anonymous access is not permitted. All HTTP connections must be authenticated using ClientAuth authentication.

Setting the current domain

In order to use the Provisioning API, the domain being administered needs to be specified in all request URIs. In order to ease development, this information is stored within both the Gapps service and query classes to use when constructing requests.

Setting the domain for the service class

To set the domain for requests made by the service class, either call `setDomain()` or specify the domain when instantiating the service class. For example:

```
$domain = "example.com";
$gdata = new Zend_Gdata_Gapps($client, $domain);
```

Setting the domain for query classes

Setting the domain for requests made by query classes is similar to setting it for the service class-either call `setDomain()` or specify the domain when creating the query. For example:

```
$domain = "example.com";
$query = new Zend_Gdata_Gapps_UserQuery($domain, $arg);
```

When using a service class factory method to create a query, the service class will automatically set the query's domain to match its own domain. As a result, it is not necessary to specify the domain as part of the constructor arguments.

```
$domain = "example.com";
$gdata = new Zend_Gdata_Gapps($client, $domain);
$query = $gdata->newUserQuery($arg);
```

Interacting with users

Each user account on a Google Apps hosted domain is represented as an instance of Zend_Gdata_Gapps_UserEntry. This class provides access to all account properties including name, username, password, access rights, and current quota.

Creating a user account

User accounts can be created by calling the `createUser()` convenience method:

```
$gdata->createUser('foo', 'Random', 'User', '••••••••');
```

Users can also be created by instantiating UserEntry, providing a username, given name, family name, and password, then calling `insertUser()` on a service object to upload the entry to the server.

```
$user = $gdata->newUserEntry();
$user->login = $gdata->newLogin();
$user->login->username = 'foo';
$user->login->password = '••••••••';
$user->name = $gdata->newName();
$user->name->givenName = 'Random';
$user->name->familyName = 'User';
$user = $gdata->insertUser($user);
```

The user's password should normally be provided as cleartext. Optionally, the password can be provided as an SHA-1 digest if `login->passwordHashFunction` is set to 'SHA-1'.

Retrieving a user account

Individual user accounts can be retrieved by calling the `retrieveUser()` convenience method. If the user is not found, `null` will be returned.

```
$user = $gdata->retrieveUser('foo');

echo 'Username: ' . $user->login->userName . "\n";
echo 'Given Name: ' . $user->login->givenName . "\n";
echo 'Family Name: ' . $user->login->familyName . "\n";
echo 'Suspended: ' . ($user->login->suspended ? 'Yes' : 'No') . "\n";
echo 'Admin: ' . ($user->login->admin ? 'Yes' : 'No') . "\n"
echo 'Must Change Password: ' .
     ($user->login->changePasswordAtNextLogin ? 'Yes' : 'No') . "\n";
echo 'Has Agreed To Terms: ' .
     ($user->login->agreedToTerms ? 'Yes' : 'No') . "\n";
```

Users can also be retrieved by creating an instance of Zend_Gdata_Gapps_UserQuery, setting its username property to equal the username of the user that is to be retrieved, and calling `getUserEntry()` on a service object with that query.

```
$query = $gdata->newUserQuery('foo');
$user = $gdata->getUserEntry($query);

echo 'Username: ' . $user->login->userName . "\n";
echo 'Given Name: ' . $user->login->givenName . "\n";
echo 'Family Name: ' . $user->login->familyName . "\n";
echo 'Suspended: ' . ($user->login->suspended ? 'Yes' : 'No') . "\n";
echo 'Admin: ' . ($user->login->admin ? 'Yes' : 'No') . "\n"
echo 'Must Change Password: ' .
     ($user->login->changePasswordAtNextLogin ? 'Yes' : 'No') . "\n";
echo 'Has Agreed To Terms: ' .
     ($user->login->agreedToTerms ? 'Yes' : 'No') . "\n";
```

If the specified user cannot be located a ServiceException will be thrown with an error code of Zend_Gdata_Gapps_Error::ENTITY_DOES_NOT_EXIST. ServiceExceptions will be covered in the section called “Handling errors”.

Retrieving all users in a domain

To retrieve all users in a domain, call the `retrieveAllUsers()` convenience method.

```
$feed = $gdata->retrieveAllUsers();

foreach ($feed as $user) {
    echo "  * " . $user->login->username . ' (' . $user->name->givenName .
        ' ' . $user->name->familyName . ")\n";
}
```

This will create a Zend_Gdata_Gapps_UserFeed object which holds each user on the domain.

Alternatively, call `getUserFeed()` with no options. Keep in mind that on larger domains this feed may be paged by the server. For more information on paging, see the section called “Working with multi-page feeds”.

```
$feed = $gdata->getUserFeed();

foreach ($feed as $user) {
    echo "  * " . $user->login->username . ' (' . $user->name->givenName .
        ' ' . $user->name->familyName . ")\n";
}
```

Updating a user account

The easiest way to update a user account is to retrieve the user as described in the previous sections, make any desired changes, then call `save()` on that user. Any changes made will be propagated to the server.

```
$user = $gdata->retrieveUser('foo');
$user->name->givenName = 'Foo';
$user->name->familyName = 'Bar';
$user = $user->save();
```

Resetting a user's password

A user's password can be reset to a new value by updating the `login->password` property.

```
$user = $gdata->retrieveUser('foo');
$user->login->password = '••••••••';
$user = $user->save();
```

Note that it is not possible to recover a password in this manner as stored passwords are not made available via the Provisioning API for security reasons.

Forcing a user to change their password

A user can be forced to change their password at their next login by setting the `login->changePasswordAtNextLogin` property to `true`.

```
$user = $gdata->retrieveUser('foo');
$user->login->changePasswordAtNextLogin = true;
$user = $user->save();
```

Similarly, this can be undone by setting the `login->changePasswordAtNextLogin` property to `false`.

Suspending a user account

Users can be restricted from logging in without deleting their user account by instead *suspending* their user account. Accounts can be suspended or restored by using the `suspendUser()` and `restoreUser()` convenience methods:

```
$gdata->suspendUser('foo');
$gdata->restoreUser('foo');
```

Alternatively, you can set the UserEntry's `login->suspended` property to `true`.

```
$user = $gdata->retrieveUser('foo');
$user->login->suspended = true;
$user = $user->save();
```

To restore the user's access, set the `login->suspended` property to `false`.

Granting administrative rights

Users can be granted the ability to administer your domain by setting their `login->admin` property to `true`.

```
$user = $gdata->retrieveUser('foo');
$user->login->admin = true;
$user = $user->save();
```

And as expected, setting a user's `login->admin` property to `false` revokes their administrative rights.

Deleting user accounts

Deleting a user account to which you already hold a UserEntry is a simple as calling `delete()` on that entry.

```
$user = $gdata->retrieveUser('foo');
$user->delete();
```

If you do not have access to a UserEntry object for an account, use the `deleteUser()` convenience method.

```
$gdata->deleteUser('foo');
```

Interacting with nicknames

Nicknames serve as email aliases for existing users. Each nickname contains precisely two key properties: its name and its owner. Any email addressed to a nickname is forwarded to the user who owns that nickname.

Nicknames are represented as an instances of Zend_Gdata_Gapps_NicknameEntry.

Creating a nickname

Nicknames can be created by calling the `createNickname()` convenience method:

```
$gdata->createNickname('foo', 'bar');
```

Nicknames can also be created by instantiating NicknameEntry, providing the nickname with a name and an owner, then calling `insertNickname()` on a service object to upload the entry to the server.

```
$nickname = $gdata->newNicknameEntry();
$nickname->login = $gdata->newLogin('foo');
$nickname->nickname = $gdata->newNickname('bar');
$nickname = $gdata->insertNickname($nickname);
```

Retrieving a nickname

Nicknames can be retrieved by calling the `retrieveNickname()` convenience method. This will return `null` if a user is not found.

```
$nickname = $gdata->retrieveNickname('bar');

echo 'Nickname: ' . $nickname->nickname->name . "\n";
echo 'Owner: ' . $nickname->login->username . "\n";
```

Individual nicknames can also be retrieved by creating an instance of Zend_Gdata_Gapps_NicknameQuery, setting its nickname property to equal the nickname that is to be retrieved, and calling `getNicknameEntry()` on a service object with that query.

```
$query = $gdata->newNicknameQuery('bar');
$nickname = $gdata->getNicknameEntry($query);

echo 'Nickname: ' . $nickname->nickname->name . "\n";
echo 'Owner: ' . $nickname->login->username . "\n";
```

As with users, if no corresponding nickname is found a ServiceException will be thrown with an error code of Zend_Gdata_Gapps_Error::ENTITY_DOES_NOT_EXIST. Again, these will be discussed in the section called “Handling errors”.

Retrieving all nicknames for a user

To retrieve all nicknames associated with a given user, call the convenience method `retrieveNicknames()`.

```
$feed = $gdata->retrieveNicknames('foo');

foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . "\n";
}
```

This will create a Zend_Gdata_Gapps_NicknameFeed object which holds each nickname associated with the specified user.

Alternatively, create a new Zend_Gdata_Gapps_NicknameQuery, set its username property to the desired user, and submit the query by calling `getNicknameFeed()` on a service object.

```
$query = $gdata->newNicknameQuery();
$query->setUsername('foo');
$feed = $gdata->getNicknameFeed($query);

foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . "\n";
}
```

Retrieving all nicknames in a domain

To retrieve all nicknames in a feed, simply call the convenience method `retrieveAllNicknames()`

```
$feed = $gdata->retrieveAllNicknames();

foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . ' => ' .
        $nickname->login->username . "\n";
}
```

This will create a Zend_Gdata_Gapps_NicknameFeed object which holds each nickname on the domain.

Alternatively, call `getNicknameFeed()` on a service object with no arguments.

```
$feed = $gdata->getNicknameFeed();
```

```
foreach ($feed as $nickname) {
    echo '  * ' . $nickname->nickname->name . ' => ' .
        $nickname->login->username . "\n";
}
```

Deleting a nickname

Deleting a nickname to which you already hold a NicknameEntry for is a simple as calling `delete()` on that entry.

```
$nickname = $gdata->retrieveNickname('bar');
$nickname->delete();
```

For nicknames which you do not hold a NicknameEntry for, use the `deleteNickname()` convenience method.

```
$gdata->deleteNickname('bar');
```

Interacting with email lists

Email lists allow several users to retrieve email addressed to a single email address. Users do not need to be a member of this domain in order to subscribe to an email list provided their complete email address (including domain) is used.

Each email list on a domain is represented as an instance of Zend_Gdata_Gapps_EmailListEntry.

Creating an email list

Email lists can be created by calling the `createEmailList()` convenience method:

```
$gdata->createEmailList('friends');
```

Email lists can also be created by instantiating EmailListEntry, providing a name for the list, then calling `insertEmailList()` on a service object to upload the entry to the server.

```
$list = $gdata->newEmailListEntry();
$list->emailList = $gdata->newEmailList('friends');
$list = $gdata->insertEmailList($list);
```

Retrieving all email lists to which a recipient is subscribed

To retrieve all email lists to which a particular recipient is subscribed, call the `retrieveEmailLists()` convenience method:

```
$feed = $gdata->retrieveEmailLists('baz@somewhere.com');

foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

This will create a Zend_Gdata_Gapps_EmailListFeed object which holds each email list associated with the specified recipient.

Alternatively, create a new Zend_Gdata_Gapps_EmailListQuery, set its recipient property to the desired email address, and submit the query by calling `getEmailListFeed()` on a service object.

```
$query = $gdata->newEmailListQuery();
$query->setRecipient('baz@somewhere.com');
$feed = $gdata->getEmailListFeed($query);

foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

Retrieving all email lists in a domain

To retrieve all email lists in a domain, call the convenience method `retrieveAllEmailLists()`.

```
$feed = $gdata->retrieveAllEmailLists();

foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

This will create a Zend_Gdata_Gapps_EmailListFeed object which holds each email list on the domain.

Alternatively, call `getEmailListFeed()` on a service object with no arguments.

```
$feed = $gdata->getEmailListFeed();

foreach ($feed as $list) {
    echo '  * ' . $list->emailList->name . "\n";
}
```

Deleting an email list

To delete an email list, call the deleteEmailList() convenience method:

```
$gdata->deleteEmailList('friends');
```

Interacting with email list recipients

Each recipient subscribed to an email list is represented by an instance of Zend_Gdata_Gapps_EmailListRecipient. Through this class, individual recipients can be added and removed from email lists.

Adding a recipient to an email list

To add a recipient to an email list, simply call the `addRecipientToEmailList()` convenience method:

```
$gdata->addRecipientToEmailList('bar@somewhere.com', 'friends');
```

Retrieving the list of subscribers to an email list

The convenience method `retrieveAllRecipients()` can be used retrieve teh list of subscribers to an email list:

```
$feed = $gdata->retrieveAllRecipients('friends');

foreach ($feed as $recipient) {
    echo '  * ' . $recipient->who->email . "\n";
}
```

Alternatively, construct a new EmailListRecipientQuery, set its emailListName property to match the desired email list, and call `getEmailListRecipientFeed()` on a service object.

```
$query = $gdata->newEmailListRecipientQuery();
$query->setEmailListName('friends');
$feed = $gdata->getEmailListRecipientFeed($query);

foreach ($feed as $recipient) {
    echo '  * ' . $recipient->who->email . "\n";
}
```

This will create a Zend_Gdata_Gapps_EmailListRecipientFeed object which holds each recipient for the selected email list.

Removing a recipient from an email list

To remove a recipient from an email list, call the `removeRecipientFromEmailList()` convenience method:

```
$gdata->removeRecipientFromEmailList('baz@somewhere.com', 'friends');
```

Handling errors

In addition to the standard suite of exceptions thrown by Zend_Gdata, requests using the Provisioning API may also throw a `Zend_Gdata_Gapps_ServiceException`. These exceptions indicate that a API specific error occurred which prevents the request from completing.

Each ServiceException instance may hold one or more Error objects. Each of these objects contains an error code, reason, and (optionally) the input which triggered the exception. A complete list of known error codes is provided in the Zend Framework API documentation under Zend_Gdata_Gapps_Error. Additionally, the authoritative error list is available online at Google Apps Provisioning API V2.0 Reference: Appendix D [http://code.google.com/apis/apps/gdata_provisioning_api_v2.0_reference.html#appendix_d].

While the complete list of errors received is available within ServiceException as an array by calling `getErrors()`, often it is convenient to know if one specific error occurred. For these cases the presence of an error can be determined by calling `hasError()`.

The following example demonstrates how to detect if a requested resource doesn't exist and handle the fault gracefully:

```
function retrieveUser ($username) {
    $query = $gdata->newUserQuery($username);
    try {
        $user = $gdata->getUserEntry($query);
    } catch (Zend_Gdata_Gapps_ServiceException $e) {
        // Set the user to null if not found
        if ($e->hasError(Zend_Gdata_Gapps_Error::ENTITY_DOES_NOT_EXIST)) {
            $user = null;
        } else {
            throw $e;
        }
    }
    return $user;
}
```

Using Google Base

The Google Base data API is designed to enable developers to do two things:

- Query Google Base data to create applications and mashups.
- Input and manage Google Base items programmatically.

There are two item feeds: snippets feed and customer items feeds. The snippets feed contains all Google Base data and is available to anyone to query against without a need for authentication. The customer items feed is a customer-specific subset of data and only a customer/owner can access this feed to insert, update, or delete their own data. Queries are constructed the same way against both types of feeds.

See http://code.google.com/apis/base [http://code.google.com/apis/base/] for more information about the Google Base API.

Connect To The Base Service

The Google Base API, like all GData APIs, is based off of the Atom Publishing Protocol (APP), an XML based format for managing web-based resources. Traffic between a client and the Google Base servers occurs over HTTP and allows for both authenticated and unauthenticated connections.

Before any transactions can occur, this connection needs to be made. Creating a connection to the base servers involves two steps: creating an HTTP client and binding a `Zend_Gdata_Gbase` service instance to that client.

Authentication

The Google Base API allows access to both public and private base feeds. Public feeds do not require authentication, but are read-only and offer reduced functionality. Private feeds offers the most complete functionality but requires an authenticated connection to the base servers. There are three authentication schemes that are supported by Google Base:

- *ClientAuth* provides direct username/password authentication to the base servers. Since this scheme requires that users provide your application with their password, this authentication is only recommended when other authentication schemes are insufficient.
- *AuthSub* allows authentication to the base servers via a Google proxy server. This provides the same level of convenience as ClientAuth but without the security risk, making this an ideal choice for web-based applications.

The `Zend_Gdata` library provides support for all three authentication schemes. The rest of this chapter will assume that you are familiar the authentication schemes available and how to create an appropriate authenticated connection. For more information, please see section the section called “Google Data Client Authentication”. or the Authentication Overview in the Google Data API Developer's Guide [http://code.google.com/apis/gdata/auth.html].

Create A Service Instance

In order to interact with Google Base, this library provides the `Zend_Gdata_Gbase` service class. This class provides a common interface to the Google Data and Atom Publishing Protocol models and assists in marshaling requests to and from the base servers.

Once deciding on an authentication scheme, the next step is to create an instance of `Zend_Gdata_Gbase` . This class takes in an instance of `Zend_Http_Client` as a single argument. This provides an interface for AuthSub and ClientAuth authentication, as both of these creation of a special authenticated HTTP client. If no arguments are provided, an unauthenticated instance of `Zend_Http_Client` will be automatically created.

The example below shows how to create a Base service class using ClientAuth authentication:

```
// Parameters for ClientAuth authentication
$service = Zend_Gdata_Gbase::AUTH_SERVICE_NAME;
$user = "sample.user@gmail.com";
$pass = "pa$$w0rd";

// Create an authenticated HTTP client
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);

// Create an instance of the Base service
$service = new Zend_Gdata_Gbase($client);
```

A Base service using AuthSub can be created in a similar, though slightly more lengthy fashion:

```
/*
 * Retrieve the current URL so that the AuthSub server knows where to
 * redirect the user after authentication is complete.
 */
function getCurrentUrl()
{
    global $_SERVER;

    // Filter php_self to avoid a security vulnerability.
    $php_request_uri =
        htmlentities(substr($_SERVER['REQUEST_URI'],
                            0,
                            strcspn($_SERVER['REQUEST_URI'], "\n\r")),
                     ENT_QUOTES);

    if (isset($_SERVER['HTTPS']) &&
        strtolower($_SERVER['HTTPS']) == 'on') {
        $protocol = 'https://';
    } else {
        $protocol = 'http://';
    }
    $host = $_SERVER['HTTP_HOST'];
    if ($_SERVER['HTTP_PORT'] != '' &&
        (($protocol == 'http://' && $_SERVER['HTTP_PORT'] != '80') ||
        ($protocol == 'https://' && $_SERVER['HTTP_PORT'] != '443'))) {
        $port = ':' . $_SERVER['HTTP_PORT'];
    } else {
        $port = '';
    }
    return $protocol . $host . $port . $php_request_uri;
}

/**
 * Obtain an AuthSub authenticated HTTP client, redirecting the user
 * to the AuthSub server to login if necessary.
 */
function getAuthSubHttpClient()
{
    global $_SESSION, $_GET;

    // If there is no AuthSub session or one-time token waiting for us,
    // redirect the user to the AuthSub server to get one.
    if (!isset($_SESSION['sessionToken']) && !isset($_GET['token'])) {
        // Parameters to give to AuthSub server
        $next = getCurrentUrl();
        $scope = "http://www.google.com/base/feeds/items/";
        $secure = false;
        $session = true;

        // Redirect the user to the AuthSub server to sign in

        $authSubUrl = Zend_Gdata_AuthSub::getAuthSubTokenUri($next,
                                                             $scope,
                                                             $secure,
                                                             $session);
         header("HTTP/1.0 307 Temporary redirect");

         header("Location: " . $authSubUrl);

         exit();
    }

    // Convert an AuthSub one-time token into a session token if needed
    if (!isset($_SESSION['sessionToken']) && isset($_GET['token'])) {
        $_SESSION['sessionToken'] =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
    }

    // At this point we are authenticated via AuthSub and can obtain an
    // authenticated HTTP client instance

    // Create an authenticated HTTP client
    $client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['sessionToken']);
    return $client;
}

// -> Script execution begins here <-

// Make sure http://code.google.com/apis/gdata/reference.html#Queriesthat the user
// AuthSub session token once it is available.
session_start();

// Create an instance of the Base service, redirecting the user
// to the AuthSub server if necessary.
$service = new Zend_Gdata_Gbase(getAuthSubHttpClient());
```

Finally, an unauthenticated server can be created for use with snippets feeds:

```
// Create an instance of the Base service using an unauthenticated HTTP client
$service = new Zend_Gdata_Gbase();
```

Retrieve Items

You can query customer items feed or snippets feed to retrieve items. It involves two steps, sending a query and iterating through the returned feed.

Send a Structured Query

You can send a structured query to retrieve items from your own customer items feed or from the public snippets feed.

When retrieveing items using the Base API, specially constructed query URLs are used to describe what events should be returned. The `Zend_Gdata_Gbase_ItemQuery` and `Zend_Gdata_Gbase_SnippetQuery` classes simplify this task by automatically constructing a query URL based on provided parameters.

Query Customer Items Feed

To execute a query against the customer items feed, invoke `newItemQuery()` and `getGbaseItemFeed()` methods:

```
$service = new Zend_Gdata_Gbase($client);
$query = $service->newItemQuery();
$query->setBq('[title:Programming]');
$query->setOrderBy('modification_time');
$query->setSortOrder('descending');
$query->setMaxResults('5');
$feed = $service->getGbaseItemFeed($query);
```

A full list of these paremeters is available at the Query parameters section [http://code.google.com/apis/base/items-feed.html#QueParameters] of the Customer Items Feed documentation.

Query Snippets Feed

To execute a query against the public snippets feed, invoke `newSnippetQuery()` and `getGbaseSnippetFeed()` methods:

```
$service = new Zend_Gdata_Gbase();
$query = $service->newSnippetQuery();
$query->setBq('[title:Programming]');
$query->setOrderBy('modification_time');
$query->setSortOrder('descending');
```

```
$query->setMaxResults('5');
$feed = $service->getGbaseSnippetFeed($query);
```

A full list of these paremeters is available at the Query parameters section [http://code.google.com/apis/base/snippets-feed.html#Parameters] of the Snippets Feed documentation.

Iterate through the Items

Google Base items can contain item-specific attributes such as `<g:main_ingredient>` and `<g:weight>`.

To iterate through all attributes of a given item, invoke `getGbaseAttributes()` and iterate through the results:

```
foreach ($feed->entries as $entry) {
  // Get all attributes and print out the name and text value of each
  // attribute
  $baseAttributes = $entry->getGbaseAttributes();
  foreach ($baseAttributes as $attr) {
    echo "Attribute " . $attr->name . " : " . $attr->text . "<br>";
  }
}
```

Or, you can look for specific attribute name and iterate through the results that match:

```
foreach ($feed->entries as $entry) {
  // Print all main ingredients <g:main_ingredient>
  $baseAttributes = $entry->getGbaseAttribute("main_ingredient");
  foreach ($baseAttributes as $attr) {
    echo "Main ingredient: " . $attr->text . "<br>";
  }
}
```

Insert, Update, and Delete Customer Items

A customer/owner can access his own Customer Items feed to insert, update, or delete their items. These operations do not apply to the public snippets feed.

You can test a feed operation before it is actually executed by setting the dry-run flag (`$dryRun`) to `true`. Once you are sure that you want to submit the data, set it to `false` to execute the operation.

Insert an Item

Items can be added by using the `insertGbaseItem()` method for the Base service:

```
$service = new Zend_Gdata_Gbase($client);
$newEntry = $service->newItemEntry();

// Add title
$title = "PHP Developer Handbook";
$newEntry->title = $service->newTitle(trim($title));

// Add some content
$content = "Essential handbook for PHP developers.";
$newEntry->content = $service->newContent($content);
$newEntry->content->type = 'text';

// Define product type
$itemType = "Products";
$newEntry->itemType = $itemType;

// Add item specific attributes
$newEntry->addGbaseAttribute("product_type", "book", "text");
$newEntry->addGbaseAttribute("price", "12.99 USD", "floatUnit");
$newEntry->addGbaseAttribute("quantity", "10", "int");
$newEntry->addGbaseAttribute("weight", "2.2 lbs", "numberUnit");
$newEntry->addGbaseAttribute("condition", "New", "text");
$newEntry->addGbaseAttribute("author", "John Doe", "text");
$newEntry->addGbaseAttribute("edition", "First Edition", "text");
$newEntry->addGbaseAttribute("pages", "253", "number");
$newEntry->addGbaseAttribute("publisher", "My Press", "text");
$newEntry->addGbaseAttribute("year", "2007", "number");
$newEntry->addGbaseAttribute("payment_accepted", "Google Checkout", "text");

$dryRun = true;
$createdEntry = $service->insertGbaseItem($newEntry, $dryRun);
```

Modify an Item

You can update each attribute element of an item as you iterate through them:

```
// Update the title
$newTitle = "PHP Developer Handbook Second Edition";
$entry->title = $service->newTitle($newTitle);

// Find <g:price> attribute and update the price
$baseAttributes = $entry->getGbaseAttribute("price");
if (is_object($baseAttributes[0])) {
  $newPrice = "16.99 USD";
  $baseAttributes[0]->text = $newPrice;
}

// Find <g:pages> attribute and update the number of pages
$baseAttributes = $entry->getGbaseAttribute("pages");
if (is_object($baseAttributes[0])) {
  $newPages = "278";
  $baseAttributes[0]->text = $newPages;

  // Update the attribute type from "number" to "int"
  if ($baseAttributes[0]->type == "number") {
    $newType = "int";
    $baseAttributes[0]->type = $newType;
  }
}

// Remove <g:label> attributes
$baseAttributes = $entry->getGbaseAttribute("label");
foreach ($baseAttributes as $note) {
  $entry->removeGbaseAttribute($note);
}

// Add new attributes
$entry->addGbaseAttribute("note", "PHP 5", "text");
$entry->addGbaseAttribute("note", "Web Programming", "text");

// Save the changes by invoking save() on the entry object itself
$dryRun = true;
$entry->save($dryRun);

// Or, save the changes by calling updateGbaseItem() on the service object
// $dryRun = true;
// $service->updateGbaseItem($entry, $dryRun);
```

After making the changes, either invoke `save($dryRun)` method on the `Zend_Gdata_Gbase_ItemEntry` object or call `updateGbaseItem($entry, $dryRun)` method on the `Zend_Gdata_Gbase` object to save the changes.

Delete an Item

You can remove an item by calling `deleteGbaseItem()` method:

```
$dryRun = false;
$service->deleteGbaseItem($entry, $dryRun);
```

Alternatively, you can invoke `delete()` on the `Zend_Gdata_Gbase_ItemEntry` object:

```
$dryRun = false;
$entry->delete($dryRun);
```

Using the YouTube Data API

The YouTube Data API offers read and write access to YouTube's content. Users can perform unauthenticated requests to Google Data feeds to retrieve feeds of popular videos, comments, public information about YouTube user profiles, user playlists, favorites, subscriptions and so on.

For more information on the YouTube Data API, please refer to the official PHP Developer's Guide [http://code.google.com/apis/youtube/developers_guide_php.html] on code.google.com.

Authentication

The YouTube Data API allows read-only access to public data, which does not require authentication. For any write requests, a user needs to authenticate either using ClientLogin or AuthSub authentication. Please refer to the Authentication section in the PHP Developer's Guide [http://code.google.com/apis/youtube/developers_guide_php.html#Authentication] for more detail.

Developer Keys and Client ID

A developer key identifies the YouTube developer that is submitting an API request. A client ID identifies your application for logging and debugging purposes. Please visit http://code.google.com/apis/youtube/dashboard/ to obtain a developer key and client ID. The example below demonstrates how to pass the developer key and client ID to the Zend_Gdata_YouTube [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube.html] service object.

```
$yt = new Zend_Gdata_YouTube($httpClient, $applicationId, $clientId, $developerKey
```

Retrieving public video feeds

The YouTube Data API provides numerous feeds that return a list of videos, such as standard feeds, related videos, video responses, user's uploads, and user's favorites. For example, the user's uploads feed returns all videos uploaded by a specific user. See the YouTube API reference guide [http://code.google.com/apis/youtube/reference.html#Video_Feeds] for a detailed list of available feeds.

Searching for videos by metadata

You can retrieve a list of videos that match specified search criteria, using the YouTubeQuery class. The following query looks for videos which contain the word "cat" in their metadata, starting with the 10th video and displaying 20 videos per page, ordered by the view count.

```
$yt = new Zend_Gdata_YouTube();
$query = $yt->newVideoQuery();
$query->videoQuery = 'cat';
$query->startIndex = 10;
$query->maxResults = 20;
$query->orderBy = 'viewCount';

echo $query->queryUrl . "\n";
$videoFeed = $yt->getVideoFeed($query);
```

```
foreach ($videoFeed as $videoEntry) {
    echo "---------VIDEO----------\n";
    echo "Title: " . $videoEntry->getVideoTitle() . "\n";
    echo "\nDescription:\n";
    echo $videoEntry->getVideoDescription();
    echo "\n\n\n";
}
```

For more details on the different query parameters, please refer to the Reference Guide [http://code.google.com/apis/youtube/reference.html#Searching_for_videos]. The available helper functions in Zend_Gdata_YouTube_VideoQuery [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_VideoQuery.html] for each of these parameters are described in more detail in the PHP Developer's Guide [http://code.google.com/apis/youtube/developers_guide_php.html#SearchingVideos].

Searching for videos by categories and tags/keywords

Searching for videos in specific categories is done by generating a specially formatted URL [http://code.google.com/apis/youtube/reference.html#Category_search]. For example, to search for comedy videos which contain the keyword dog:

```
$yt = new Zend_Gdata_YouTube();
$query = $yt->newVideoQuery();
$query->category = 'Comedy/dog';

echo $query->queryUrl . "\n";
$videoFeed = $yt->getVideoFeed($query);
```

Retrieving standard feeds

The YouTube Data API has a number of standard feeds [http://code.google.com/apis/youtube/reference.html#Standard_feeds]. These standard feeds can be retrieved as Zend_Gdata_YouTube_VideoFeed [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_VideoFeed.html] objects using the specified URLs, using the predefined constants within the Zend_Gdata_YouTube [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube.html] class (Zend_Gdata_YouTube::STANDARD_TOP_RATED_URI for example) or using the predefined helper methods (see code listing below).

To retrieve the top rated videos using the helper method:

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getTopRatedVideoFeed();
```

There are also query parameters to specify the time period over which the standard feed is computed.

For example, to retrieve the top rated videos for today:

```
$yt = new Zend_Gdata_YouTube();
$query = $yt->newVideoQuery();
$query->setTime('today');
$videoFeed = $yt->getTopRatedVideoFeed($query);
```

Alternatively, you could just retrieve the feed using the URL:

```
$yt = new Zend_Gdata_YouTube();
$url = 'http://gdata.youtube.com/feeds/standardfeeds/top_rated?time=today'
$videoFeed = $yt->getVideoFeed($url);
```

Retrieving videos uploaded by a user

You can retrieve a list of videos uploaded by a particular user using a simple helper method. This example retrieves videos uploaded by the user 'liz'.

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getUserUploads('liz');
```

Retrieving videos favorited by a user

You can retrieve a list of a user's favorite videos using a simple helper method. This example retrieves videos favorited by the user 'liz'.

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getUserFavorites('liz');
```

Retrieving video responses for a video

You can retrieve a list of a video's video responses using a simple helper method. This example retrieves video response for a video with the ID 'abc123813abc'.

```
$yt = new Zend_Gdata_YouTube();
$videoFeed = $yt->getVideoResponseFeed('abc123813abc');
```

Retrieving video comments

The comments for each YouTube video can be retrieved in several ways. To retrieve the comments for the video with the ID 'abc123813abc', use the following code:

```
$yt = new Zend_Gdata_YouTube();
$commentFeed = $yt->getVideoCommentFeed('abc123813abc');

foreach ($commentFeed as $commentEntry) {
    echo $commentEntry->title->text . "\n";
    echo $commentEntry->content->text . "\n\n\n";
}
```

Comments can also be retrieved for a video if you have a copy of the Zend_Gdata_YouTube_VideoEntry [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_VideoEntry.html] object:

```
$yt = new Zend_Gdata_YouTube();
$videoEntry = $yt->getVideoEntry('abc123813abc');
// we don't know the video ID in this example, but we do have the URL
$commentFeed = $yt->getVideoCommentFeed(null,
                                        $videoEntry->comments->href);
```

Retrieving playlist feeds

The YouTube Data API provides information about users, including profiles, playlists, subscriptions, and more.

Retrieving the playlists of a user

The library provides a helper method to retrieve the playlists associated with a given user. To retrieve the playlists for the user 'liz':

```
$yt = new Zend_Gdata_YouTube();
$playlistListFeed = $yt->getPlaylistListFeed('liz');

foreach ($playlistListFeed as $playlistEntry) {
    echo $playlistEntry->title->text . "\n";
    echo $playlistEntry->description->text . "\n";
    echo $playlistEntry->getPlaylistVideoFeedUrl() . "\n\n\n";
}
```

Retrieving a specific playlist

The library provides a helper method to retrieve the videos associated with a given playlist. To retrieve the playlists for a specific playlist entry:

```
$feedUrl = $playlistEntry->getPlaylistVideoFeedUrl();
$playlistVideoFeed = $yt->getPlaylistVideoFeed($feedUrl);
```

Retrieving a list of a user's subscriptions

A user can have several types of subscriptions: channel subscription, tag subscription, or favorites subscription. A Zend_Gdata_YouTube_SubscriptionEntry [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_SubscriptionEntry.html] is used to represent individual subscriptions.

To retrieve all subscriptions for the user 'liz':

```
$yt = new Zend_Gdata_YouTube();
$subscriptionFeed = $yt->getSubscriptionFeed('liz');

foreach ($subscriptionFeed as $subscriptionEntry) {
    echo $subscriptionEntry->title->text . "\n";
}
```

Retrieving a user's profile

You can retrieve the public profile information for any YouTube user. To retrieve the profile for the user 'liz':

```
$yt = new Zend_Gdata_YouTube();
$userProfile = $yt->getUserProfile('liz');
echo "username: " . $userProfile->username->text . "\n";
echo "age: " . $userProfile->age->text . "\n";
echo "hometown: " . $userProfile->hometown->text . "\n";
```

Uploading Videos to YouTube

Please make sure to review the diagrams in the protocol guide [http://code.google.com/apis/youtube/developers_guide_protocol.html#Process_Flows_for_Uploading_Videos] on code.google.com for a high-level overview of the upload process. Uploading videos can be done in one of two ways: either by uploading the video directly or by sending just the video meta-data and having a user upload the video through an HTML form.

In order to upload a video directly, you must first construct a new Zend_Gdata_YouTube_VideoEntry [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_VideoEntry.html] object and specify some required meta-data The following example shows uploading the Quicktime video "mytestmovie.mov" to YouTube with the following properties:

Table 20.1. Metadata used in the code-sample below

Property	**Value**
Title	My Test Movie
Category	Autos
Keywords	cars, funny
Description	My description
Filename	mytestmovie.mov
File MIME type	video/quicktime
Video private?	false
Video location	37, -122 (lat, long)
Developer Tags	mydevelopertag, anotherdevelopertag

The code below creates a blank Zend_Gdata_YouTube_VideoEntry [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_VideoEntry.html] to be uploaded. A Zend_Gdata_App_MediaFileSource [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_App_MediaFileSource.html] object is then used to hold the actual video file. Under the hood, the Zend_Gdata_YouTube_Extension_MediaGroup [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_Extension_MediaGroup.html] object is used to hold all of the video's meta-data. Our helper methods detailed below allow you to just set the video meta-data without having to worry about the media group object. The $uploadUrl is the location where the new entry gets posted to. This can be specified either with the $userName of the currently authenticated user, or, alternatively, you can simply use the string 'default' to refer to the currently authenticated user.

```
$yt = new Zend_Gdata_YouTube($httpClient);
$myVideoEntry = new Zend_Gdata_YouTube_VideoEntry();

$filesource = $yt->newMediaFileSource('mytestmovie.mov');
$filesource->setContentType('video/quicktime');
$filesource->setSlug('mytestmovie.mov');

$myVideoEntry->setMediaSource($filesource);

$myVideoEntry->setVideoTitle('My Test Movie');
$myVideoEntry->setVideoDescription('My Test Movie');
// Note that category must be a valid YouTube category !
$myVideoEntry->setVideoCategory('Comedy');

// Set keywords, note that this must be a comma separated string
// and that each keyword cannot contain whitespace
$myVideoEntry->SetVideoTags('cars, funny');

// Optionally set some developer tags
```

```
$myVideoEntry->setVideoDeveloperTags(array('mydevelopertag',
                                           'anotherdevelopertag'));

// Optionally set the video's location
$yt->registerPackage('Zend_Gdata_Geo');
$yt->registerPackage('Zend_Gdata_Geo_Extension');
$where = $yt->newGeoRssWhere();
$position = $yt->newGmlPos('37.0 -122.0');
$where->point = $yt->newGmlPoint($position);
$myVideoEntry->setWhere($where);

// Upload URI for the currently authenticated user
$uploadUrl =
    'http://uploads.gdata.youtube.com/feeds/users/default/uploads';

// Try to upload the video, catching a Zend_Gdata_App_HttpException
// if availableor just a regular Zend_Gdata_App_Exception

try {
    $newEntry = $yt->insertEntry($myVideoEntry,
                                 $uploadUrl,
                                 'Zend_Gdata_YouTube_VideoEntry');
} catch (Zend_Gdata_App_HttpException $httpException) {
    echo $httpException->getRawResponseBody();
} catch (Zend_Gdata_App_Exception $e) {
    echo $e->getMessage();
}
```

To upload a video as private, simply use: $myVideoEntry->setVideoPrivate(); prior to performing the upload. $videoEntry->isVideoPrivate() can be used to check whether a video entry is private or not.

Browser-based upload

Browser-based uploading is performed almost identically to direct uploading, except that you do not attach a Zend_Gdata_App_MediaFileSource [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_App_MediaFileSource.html] object to the Zend_Gdata_YouTube_VideoEntry [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_VideoEntry.html] you are constructing. Instead you simply submit all of your video's meta-data to receive back a token element which can be used to construct an HTML upload form.

```
$yt = new Zend_Gdata_YouTube($httpClient);

$myVideoEntry= new Zend_Gdata_YouTube_VideoEntry();
$myVideoEntry->setVideoTitle('My Test Movie');
$myVideoEntry->setVideoDescription('My Test Movie');

// Note that category must be a valid YouTube category
$myVideoEntry->setVideoCategory('Comedy');
$myVideoEntry->SetVideoTags('cars, funny');
```

```
$tokenHandlerUrl = 'http://gdata.youtube.com/action/GetUploadToken';
$tokenArray = $yt->getFormUploadToken($myVideoEntry, $tokenHandlerUrl);
$tokenValue = $tokenArray['token'];
$postUrl = $tokenArray['url'];
```

The above code prints out a link and a token that is used to construct an HTML form to display in the user's browser. A simple example form is shown below with $tokenValue representing the content of the returned token element, as shown being retrieved from $myVideoEntry above. In order for the user to be redirected to your website after submitting the form, make sure to append a $nextUrl parameter to the $postUrl above, which functions in the same way as the $next parameter of an AuthSub link. The only difference is that here, instead of a single-use token, a status and an id variable are returned in the URL.

```
// place to redirect user after upload
$nextUrl = 'http://mysite.com/youtube_uploads';

$form = '<form action="'. $postUrl .'?nexturl='. $nextUrl .
        '" method="post" enctype="multipart/form-data">'.
        '<input name="file" type="file"/>'.
        '<input name="token" type="hidden" value="'. $tokenValue .'"/>'.
        '<input value="Upload Video File" type="submit" />'.
        '</form>';
```

Checking upload status

After uploading a video, it will immediately be visible in an authenticated user's uploads feed. However, it will not be public on the site until it has been processed. Videos that have been rejected or failed to upload successfully will also only be in the authenticated user's uploads feed. The following code checks the status of a Zend_Gdata_YouTube_VideoEntry [http://framework.zend.com/apidoc/core/Zend_Gdata/Zend_Gdata_YouTube_VideoEntry.html] to see if it is not live yet or if it has been rejected.

```
try {
    $control = $videoEntry->getControl();
} catch (Zend_Gdata_App_Exception $e) {
    echo $e->getMessage();
}

if ($control instanceof Zend_Gdata_App_Extension_Control) {
    if ($control->getDraft() != null &&
        $control->getDraft()->getText() == 'yes') {
        $state = $videoEntry->getVideoState();

        if ($state instanceof Zend_Gdata_YouTube_Extension_State) {
            print 'Upload status: '
                  . $state->getName()
                  .' '.  $state->getText();
        } else {
            print 'Not able to retrieve the video status information'
                  .' yet. ' . "Please try again shortly.\n";
        }
    }
}
```

Other Functions

In addition to the functionality described above, the YouTube API contains many other functions that allow you to modify video meta-data, delete video entries and use the full range of community features on the site. Some of the community features that can be modified through the API include: ratings, comments, playlists, subscriptions, user profiles, contacts and messages.

Please refer to the full documentation available in the PHP Developer's Guide [http://code.google.com/apis/youtube/developers_guide_php.html] on code.google.com.

Using Picasa Web Albums

Picasa Web Albums is a service which allows users to maintain albums of their own pictures, and browse the albums and pictures of others. The API offers a programatic interface to this service, allowing users to add to, update, and remove from their albums, as well as providing the ability to tag and comment on photos.

Access to public albums and photos is not restricted by account, however, a user must be logged in for non-read-only access.

For more information on the API, including instructions for enabling API access, refer to the Picasa Web Albums Data API Overview [http://code.google.com/apis/picasaweb/overview.html].

Authentication

The API provides authentication via AuthSub (recommended) and ClientAuth. HTTP connections must be authenticated for write support, but non-authenticated connections have read-only access.

Connecting To The Service

The Picasa Web Albums API, like all GData APIs, is based off of the Atom Publishing Protocol (APP), an XML based format for managing web-based resources. Traffic between a client and the servers occurs over HTTP and allows for both authenticated and unauthenticated connections.

Before any transactions can occur, this connection needs to be made. Creating a connection to the Picasa servers involves two steps: creating an HTTP client and binding a `Zend_Gdata_Photos` service instance to that client.

Authentication

The Google Picasa API allows access to both public and private photo feeds. Public feeds do not require authentication, but are read-only and offer reduced functionality. Private feeds offers the most complete functionality but requires an authenticated connection to the Picasa servers. There are three authentication schemes that are supported by Google Picasa :

- *ClientAuth* provides direct username/password authentication to the Picasa servers. Since this scheme requires that users provide your application with their password, this authentication is only recommended when other authentication schemes are insufficient.
- *AuthSub* allows authentication to the Picasa servers via a Google proxy server. This provides the same level of convenience as ClientAuth but without the security risk, making this an ideal choice for web-based applications.

The `Zend_Gdata` library provides support for both authentication schemes. The rest of this chapter will assume that you are familiar the authentication schemes available and how to create an appropriate authenticated connection. For more information, please see section the Authentication section of this manual or the Authentication Overview in the Google Data API Developer's Guide [http://code.google.com/apis/gdata/auth.html].

Creating A Service Instance

In order to interact with the servers, this library provides the `Zend_Gdata_Photos` service class. This class provides a common interface to the Google Data and Atom Publishing Protocol models and assists in marshaling requests to and from the servers.

Once deciding on an authentication scheme, the next step is to create an instance of `Zend_Gdata_Photos`. The class constructor takes an instance of `Zend_Http_Client` as a single argument. This provides an interface for AuthSub and ClientAuth authentication, as both of these require creation of a special authenticated HTTP client. If no arguments are provided, an unauthenticated instance of `Zend_Http_Client` will be automatically created.

The example below shows how to create a service class using ClientAuth authentication:

```
// Parameters for ClientAuth authentication
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$user = "sample.user@gmail.com";
$pass = "pa$$w0rd";

// Create an authenticated HTTP client
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);

// Create an instance of the service
$service = new Zend_Gdata_Photos($client);
```

A service instance using AuthSub can be created in a similar, though slightly more lengthy fashion:

```
session_start();

/**
 * Returns the full URL of the current page, based upon env variables
 *
 * Env variables used:
 * $_SERVER['HTTPS'] = (on|off|)
 * $_SERVER['HTTP_HOST'] = value of the Host: header
 * $_SERVER['SERVER_PORT'] = port number (only used if not http/80,https/443)
 * $_SERVER['REQUEST_URI'] = the URI after the method of the HTTP request
```

```
     *
     * @return string Current URL
     */
    function getCurrentUrl()
    {
        global $_SERVER;

        /**
         * Filter php_self to avoid a security vulnerability.
         */
        $php_request_uri = htmlentities(substr($_SERVER['REQUEST_URI'], 0,
        strcspn($_SERVER['REQUEST_URI'], "\n\r")), ENT_QUOTES);

        if (isset($_SERVER['HTTPS']) && strtolower($_SERVER['HTTPS']) == 'on') {
            $protocol = 'https://';
        } else {
            $protocol = 'http://';
        }
        $host = $_SERVER['HTTP_HOST'];
        if ($_SERVER['SERVER_PORT'] != '' &&
            (($protocol == 'http://' && $_SERVER['SERVER_PORT'] != '80') ||
            ($protocol == 'https://' && $_SERVER['SERVER_PORT'] != '443'))) {
                $port = ':' . $_SERVER['SERVER_PORT'];
        } else {
            $port = '';
        }
        return $protocol . $host . $port . $php_request_uri;
    }

    /**
     * Returns the AuthSub URL which the user must visit to authenticate requests
     * from this application.
     *
     * Uses getCurrentUrl() to get the next URL which the user will be redirected
     * to after successfully authenticating with the Google service.
     *
     * @return string AuthSub URL
     */
    function getAuthSubUrl()
    {
        $next = getCurrentUrl();
        $scope = 'http://picasaweb.google.com/data';
        $secure = false;
        $session = true;
        return Zend_Gdata_AuthSub::getAuthSubTokenUri($next, $scope, $secure,
            $session);
    }

    /**
     * Returns a HTTP client object with the appropriate headers for communicating
     * with Google using AuthSub authentication.
     *
     * Uses the $_SESSION['sessionToken'] to store the AuthSub session token after
     * it is obtained.  The single use token supplied in the URL when redirected
```

```
 * after the user succesfully authenticated to Google is retrieved from the
 * $_GET['token'] variable.
 *
 * @return Zend_Http_Client
 */
function getAuthSubHttpClient()
{
    global $_SESSION, $_GET;
    if (!isset($_SESSION['sessionToken']) && isset($_GET['token'])) {
        $_SESSION['sessionToken'] =
            Zend_Gdata_AuthSub::getAuthSubSessionToken($_GET['token']);
    }
    $client = Zend_Gdata_AuthSub::getHttpClient($_SESSION['sessionToken']);
    return $client;
}

/**
 * Create a new instance of the service, redirecting the user
 * to the AuthSub server if necessary.
 */
$service = new Zend_Gdata_Photos(getAuthSubHttpClient());
```

Finally, an unauthenticated server can be created for use with public feeds:

```
// Create an instance of the service using an unauthenticated HTTP client
$service = new Zend_Gdata_Photos();
```

Understanding and Constructing Queries

The primary method to request data from the service is by constructing a query. There are query classes for each of the following types:

- *User* is used to specify the user whose data is being searched for, and is specified as a username. If no user is provided, "default" will be used instead to indicate the currently authenticated user (if authenticated).
- *Album* is used to specify the album which is being searched for, and is specified as either an id, or an album name.
- *Photo* is used to specify the photo which is being searched for, and is specified as an id.

A new `UserQuery` can be constructed as followed:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_UserQuery();
```

```
$query->setUser("sample.user");
```

For each query, a number of parameters limiting the search can be requested, or specified, with get(Parameter) and set(Parameter), respectively. They are as follows:

- *Projection* sets the format of the data returned in the feed, as either "api" or "base". Normally, "api" is desired. The default is "api".
- *Type* sets the type of element to be returned, as either "feed" or "entry". The default is "feed".
- *Access* sets the visibility of items to be returned, as "all", "public", or "private". The default is "all". Non-public elements will only be returned if the query is searching for the authenticated user.
- *Tag* sets a tag filter for returned items. When a tag is set, only items tagged with this value will return.
- *Kind* sets the kind of elements to return. When kind is specified, only entries that match this value will be returned.
- *ImgMax* sets the maximum image size for entries returned. Only image entries smaller than this value will be returned.
- *Thumbsize* sets the thumbsize of entries that are returned. Any retrieved entry will have a thumbsize equal to this value.
- *User* sets the user whose data is being searched for. The default is "default".
- *AlbumId* sets the id of the album being searched for. This element only applies to album and photo queries. In the case of photo queries, this specifies the album that contains the requested photo. The album id is mutually exclusive with the album's name. Setting one unsets the other.
- *AlbumName* sets the name of the album being searched for. This element only applies to the album and photo queries. In the case of photo queries, this specifies the album that contains the requested photo. The album name is mutually exclusive with the album's id. Setting one unsets the other.
- *PhotoId* sets the id of the photo being searched for. This element only applies to photo queries.

Retrieving Feeds And Entries

The service has functions to retrieve a feed, or individual entries, for users, albums, and individual photos.

Retrieving A User

The service supports retrieving a user feed and list of the user's content. If the requested user is also the authenticated user, entries marked as "`hidden`" will also be returned.

The user feed can be accessed by passing the username to the `getUserFeed` method:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

try {
    $userFeed = $service->getUserFeed("sample.user");
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Or, the feed can be accessed by constructing a query, first:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_UserQuery();
$query->setUser("sample.user");

try {
    $userFeed = $service->getUserFeed(null, $query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Constructing a query also provides the ability to request a user entry object:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_UserQuery();
$query->setUser("sample.user");
$query->setType("entry");

try {
    $userEntry = $service->getUserEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Retrieving An Album

The service supports retrieving an album feed and a list of the album's content.

The album feed is accessed by constructing a query object and passing it to `getAlbumFeed`:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);
```

```
$query = new Zend_Gdata_Photos_AlbumQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");

try {
    $albumFeed = $service->getAlbumFeed($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Alternatively, the query object can be given an album name with `setAlbumName`. Setting the album name is mutually exclusive with setting the album id, and setting one will unset the other.

Constructing a query also provides the ability to request an album entry object:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_AlbumQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setType("entry");

try {
    $albumEntry = $service->getAlbumEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Retrieving A Photo

The service supports retrieving a photo feed and a list of associated comments and tags.

The photo feed is accessed by constructing a query object and passing it to `getPhotoFeed`:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");

try {
    $photoFeed = $service->getPhotoFeed($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Constructing a query also provides the ability to request a photo entry object:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");
$query->setType("entry");

try {
    $photoEntry = $service->getPhotoEntry($query);
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Retrieving A Comment

The service supports retrieving comments from a feed of a different type. By setting a query to return a kind of "comment", a feed request can return comments associated with a specific user, album, or photo.

Performing an action on each of the comments on a given photo can be accomplished as follows:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");
$query->setKind("comment");

try {
    $photoFeed = $service->getPhotoFeed($query);

    foreach ($photoFeed as $entry) {
        if ($entry instanceof Zend_Gdata_Photos_CommentEntry) {
            // Do something with the comment
        }
    }
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Retrieving A Tag

The service supports retrieving tags from a feed of a different type. By setting a query to return a kind of "tag", a feed request can return tags associated with a specific photo.

Performing an action on each of the tags on a given photo can be accomplished as follows:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$query = new Zend_Gdata_Photos_PhotoQuery();
$query->setUser("sample.user");
$query->setAlbumId("1");
$query->setPhotoId("100");
$query->setKind("tag");

try {
    $photoFeed = $service->getPhotoFeed($query);

    foreach ($photoFeed as $entry) {
        if ($entry instanceof Zend_Gdata_Photos_TagEntry) {
            // Do something with the tag
        }
    }
} catch (Zend_Gdata_App_Exception $e) {
    echo "Error: " . $e->getResponse();
}
```

Creating Entries

The service has functions to create albums, photos, comments, and tags.

Creating An Album

The service supports creating a new album for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$entry = new Zend_Gdata_Photos_AlbumEntry();
$entry->setTitle($service->newTitle("test album"));
```

```
$service->insertAlbumEntry($entry);
```

Creating A Photo

The service supports creating a new photo for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

// $photo is the name of a file uploaded via an HTML form

$fd = $service->newMediaFileSource($photo["tmp_name"]);
$fd->setContentType($photo["type"]);

$entry = new Zend_Gdata_Photos_PhotoEntry();
$entry->setMediaSource($fd);
$entry->setTitle($service->newTitle($photo["name"]));

$albumQuery = new Zend_Gdata_Photos_AlbumQuery;
$albumQuery->setUser("sample.user");
$albumQuery->setAlbumId("1");

$albumEntry = $service->getAlbumEntry($albumQuery);

$service->insertPhotoEntry($entry, $albumEntry);
```

Creating A Comment

The service supports creating a new comment for a photo:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$entry = new Zend_Gdata_Photos_CommentEntry();
$entry->setTitle($service->newTitle("comment"));
$entry->setContent($service->newContent("comment"));

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$photoEntry = $service->getPhotoEntry($photoQuery);

$service->insertCommentEntry($entry, $photoEntry);
```

Creating A Tag

The service supports creating a new tag for a photo:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$entry = new Zend_Gdata_Photos_TagEntry();
$entry->setTitle($service->newTitle("tag"));

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$photoEntry = $service->getPhotoEntry($photoQuery);

$service->insertTagEntry($entry, $photoEntry);
```

Deleting Entries

The service has functions to delete albums, photos, comments, and tags.

Deleting An Album

The service supports deleting an album for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$albumQuery = new Zend_Gdata_Photos_AlbumQuery;
$albumQuery->setUser("sample.user");
$albumQuery->setAlbumId("1");
$albumQuery->setType('entry');

$entry = $service->getAlbumEntry($albumQuery);

$service->deleteAlbumEntry($entry, true);
```

Deleting A Photo

The service supports deleting a photo for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$entry = $service->getPhotoEntry($photoQuery);

$service->deletePhotoEntry($entry, true);
```

Deleting A Comment

The service supports deleting a comment for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setType('entry');

$path = $photoQuery->getQueryUrl() . '/commentid/' . "1000";

$entry = $service->getCommentEntry($path);

$service->deleteCommentEntry($entry, true);
```

Deleting A Tag

The service supports deleting a tag for an authenticated user:

```
$service = Zend_Gdata_Photos::AUTH_SERVICE_NAME;
$client = Zend_Gdata_ClientLogin::getHttpClient($user, $pass, $service);
$service = new Zend_Gdata_Photos($client);

$photoQuery = new Zend_Gdata_Photos_PhotoQuery;
```

```
$photoQuery->setUser("sample.user");
$photoQuery->setAlbumId("1");
$photoQuery->setPhotoId("100");
$photoQuery->setKind("tag");
$query = $photoQuery->getQueryUrl();

$photoFeed = $service->getPhotoFeed($query);

foreach ($photoFeed as $entry) {
    if ($entry instanceof Zend_Gdata_Photos_TagEntry) {
        if ($entry->getContent() == $tagContent) {
            $tagEntry = $entry;
        }
    }
}

$service->deleteTagEntry($tagEntry, true);
```

Optimistic Concurrency (Notes On Deletion)

GData feeds, including those of the Picasa Web Albums service, implement optimistic concurrency, a versioning system that prevents users from overwriting changes, inadvertently. When deleting a entry through the service class, if the entry has been modified since it was last fetched, an exception will be thrown, unless explicitly set otherwise (in which case the deletion is retried on the updated entry).

An example of how to handle versioning during a deletion is shown by `deleteAlbumEntry`:

```
// $album is the albumEntry to be deleted
try {
    $this->delete($album);
} catch (Zend_Gdata_App_HttpException $e) {
    if ($e->getResponse->getStatus() === 409) {
        $entry =
            new Zend_Gdata_Photos_AlbumEntry($e->getResponse()->getBody());
        $this->delete($entry->getLink('edit')->href);
    } else {
        throw $e;
    }
}
```

Catching Gdata Exceptions

The `Zend_Gdata_App_Exception` class is a base class for exceptions thrown by Zend_Gdata. You can catch any exception thrown by Zend_Gdata by catching Zend_Gdata_App_Exception.

```
try {
    $client =
        Zend_Gdata_ClientLogin::getHttpClient($username, $password);
} catch(Zend_Gdata_App_Exception $ex) {
    // Report the exception to the user
    die($ex->getMessage());
}
```

The following exception subclasses are used by Zend_Gdata:

- `Zend_Gdata_App_AuthException` indicates that the user's account credentials were not valid.
- `Zend_Gdata_App_BadMethodCallException` indicates that a method was called for a service that does not support the method. For example, the CodeSearch service does not support `post()`.
- `Zend_Gdata_App_HttpException` indicates that an HTTP request was not successful. Provides the ability to get the full Zend_Http_Response object to determine the exact cause of the failure in cases where `$e->getMessage()` does not provide enough details.
- `Zend_Gdata_App_InvalidArgumentException` is thrown when the application provides a value that is not valid in a given context. For example, specifying a Calendar visibility value of "banana", or fetching a Blogger feed without specifying any blog name.
- `Zend_Gdata_App_CaptchaRequiredException` is thrown when a ClientLogin attempt receives a CAPTCHA™ challenge from the authentication service. This exception contains a token ID and a URL to a CAPTCHA™ challenge image. The image is a visual puzzle that should be displayed to the user. After collecting the user's response to the challenge image, the response can be included with the next ClientLogin attempt.The user can alternatively be directed to this website: https://www.google.com/accounts/DisplayUnlockCaptcha Further information can be found in the ClientLogin documentation.

You can use these exception subclasses to handle specific exceptions differently. See the API documentation for information on which exception subclasses are thrown by which methods in Zend_Gdata.

```
try {
    $client = Zend_Gdata_ClientLogin::getHttpClient($username,
                                                     $password,
                                                     $service);
} catch(Zend_Gdata_App_AuthException $authEx) {
    // The user's credentials were incorrect.
    // It would be appropriate to give the user a second try.
    ...
} catch(Zend_Gdata_App_HttpException $httpEx) {
    // Google Data servers cannot be contacted.
    die($httpEx->getMessage);}
```

Chapter 21. Zend_Http

Zend_Http_Client - Introduction

Introduction

Zend_Http_Client provides an easy interface for preforming Hyper-Text Transfer Protocol (HTTP) requests. Zend_Http_Client supports 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 the section called “Zend_Http_Response”).

The class constructor optionally accepts a URL as it's first parameter (can be either a string or a Zend_Uri_Http object), and an optional array of configuration parameters. Both can be left out, and set later using the setUri() and setConfig() methods.

Example 21.1. Instantiating a Zend_Http_Client object

```
$client = new Zend_Http_Client('http://example.org', array(
    'maxredirects' => 0,
    'timeout'      => 30));

// This is actually exactly the same:
$client = new Zend_Http_Client();
$client->setUri('http://example.org');
$client->setConfig(array(
    'maxredirects' => 0,
    'timeout'      => 30));
```

Configuration Parameters

The constructor and setConfig() method accept an associative array of configuration parameters. Setting these parameters is optional, as they all have default values.

Table 21.1. Zend_Http_Client configuration parameters

Parameter	Description	Expected Values	Default Value
maxredirects	Maximum number of redirections to follow (0 = none)	integer	5
strict	Whether perform validation on header names. When set to false, validation functions will be skipped. Usually this should not be changed	boolean	true
strictredirects	Whether to strictly follow the RFC when redirecting (see the section called “HTTP Redirections”)	boolean	false
useragent	User agent identifier string (sent in request headers)	string	'Zend_Http_Client'
timeout	Connection timeout (seconds)	integer	10
httpversion	HTTP protocol version (usually '1.1' or '1.0')	string	'1.1'
adapter	Connection adapter class to use (see the section called “Zend_Http_Client - Connection Adapters”)	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 performned.	boolean	false
storeresponse	Whether to store last response for later retrieval with getLastResponse(). If set to false getLastResponse() will return null.	boolean	true

Performing Basic HTTP Requests

Performing simple HTTP requests is very easily done using the request() method, and rarely needs more than three lines of code:

Example 21.2. Performing a Simple GET Request

```
$client = new Zend_Http_Client('http://example.org');
$response = $client->request();
```

The request() method takes one optional parameter - the request method. This can be either GET, POST, PUT, HEAD, DELETE, TRACE, OPTIONS or CONNECT as defined by the HTTP protocol [1]. For convenience, these are all defined as class constants: Zend_Http_Request::GET, Zend_Http_Request::POST and so on.

If no method is specified, the method set by the last setMethod() call is used. If setMethod() was never called, the default request method is GET (see the above example).

[1] See RFC 2616 - http://www.w3.org/Protocols/rfc2616/rfc2616.html.

Example 21.3. Using Request Methods Other Than GET

```
// Preforming a POST request
$response = $client->request('POST');

// Yet another way of preforming a POST request
$client->setMethod(Zend_Http_Client::POST);
$response = $client->request();
```

Adding GET and POST 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 parameter's name as it's first parameter, and the GET parameter's value as it's second parameter. For convenience, the setParameterGet() method can also accept a single associative array of name => value GET variables - which may be more comfortable when several GET parameters need to be set.

Example 21.4. Setting GET Parameters

```
// Setting a get parameter using the setParameterGet method
$client->setParameterGet('knight', 'lancelot');

// This is equivalent to setting such URL:
$client->setUri('http://example.com/index.php?knight=lancelot');

// Adding several parameters with one call
$client->setParameterGet(array(
    'first_name'  => 'Bender',
    'middle_name' => 'Bending'
    'made_in'     => 'Mexico',
));
```

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 similar to the setParameterGet() method in structure.

Example 21.5. Setting POST Parameters

```
// Setting a POST parameter
$client->setParameterPost('language', 'fr');

// Setting several POST parameters, one of them with several values
$client->setParameterPost(array(
    'language'  => 'es',
    'country'   => 'ar',
    'selection' => array(45, 32, 80)
));
```

Note that when sending POST requests, you can set both GET and POST parameters. On the other hand, while setting POST parameters for a non-POST request will not trigger and error, it is useless. Unless the request is a POST request, POST parameters are simply ignored.

Accessing Last Request and Response

Zend_Http_Client provides methods of accessing the last request sent and last response received by the client object. `Zend_Http_Client->getLastRequest()` takes no parameters and returns the last HTTP request sent by the client as a string. Similarly, `Zend_Http_Client->getLastResponse()` returns the last HTTP response received by the client as a Zend_Http_Response object.

Zend_Http_Client - Advanced Usage

HTTP Redirections

By default, Zend_Http_Client automatically handles HTTP redirections, and will follow up to 5 redirections. This can be changed by setting the 'maxredirects' configuration parameter.

According to the HTTP/1.1 RFC, HTTP 301 and 302 responses should be treated by the client by resending the same request to the specified location - using the same request method. However, most clients to not implement this and always use a GET request when redirecting. By default, Zend_Http_Client does the same - when redirecting on a 301 or 302 response, all GET and POST parameters are reset, and a GET request is sent to the new location. This behavior can be changed by setting the 'strictredirects' configuration parameter to boolean TRUE:

Example 21.6. Forcing RFC 2616 Strict Redirections on 301 and 302 Responses

```
// Strict Redirections
$client->setConfig(array('strictredirects' => true));

// Non-strict Redirections
$client->setConfig(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. This is done using the setCookie() method. This method can be used in several ways:

Example 21.7. Setting Cookies Using setCookie()

```
// Easy and simple: by providing a cookie name and cookie value
$client->setCookie('flavor', 'chocolate chips');

// By directly providing a raw cookie string (name=value)
// Note that the value must be already URL encoded
$client->setCookie('flavor=chocolate%20chips');

// By providing a Zend_Http_Cookie object
$cookie = Zend_Http_Cookie::fromString('flavor=chocolate%20chips');
$client->setCookie($cookie);
```

For more information about Zend_Http_Cookie objects, see the section called “Zend_Http_Cookie and Zend_Http_CookieJar”.

Zend_Http_Client also provides the means for cookie stickiness - that is having the client internally store all sent and received cookies, and resend them automatically on subsequent requests. This is useful, for example when you need to log in to a remote site first and receive and authentication or session ID cookie before sending further requests.

Example 21.8. Enabling Cookie Stickiness

```
// To turn cookie stickiness on, set a Cookie Jar
$client->setCookieJar();

// First request: log in and start a session
$client->setUri('http://example.com/login.php');
$client->setParameterPost('user', 'h4x0r');
$client->setParameterPost('password', '1337');
$client->request('POST');

// The Cookie Jar automatically stores the cookies set
// in the response, like a session ID cookie.

// Now we can send our next request - the stored cookies
// will be automatically sent.
$client->setUri('http://example.com/read_member_news.php');
$client->request('GET');
```

For more information about the Zend_Http_CookieJar class, see the section called "The Zend_Http_CookieJar Class: Instantiation".

Setting Custom Request Headers

Setting custom headers can be done by using the setHeaders() method. This method is quite diverse and can be used in several ways, as the following example shows:

Example 21.9. Setting A Single Custom Request Header

```
// Setting a single header, overwriting any previous value
$client->setHeaders('Host', 'www.example.com');

// Another way of doing the exact same thing
$client->setHeaders('Host: www.example.com');

// Setting several values for the same header (useful mostly for Cookie headers):
$client->setHeaders('Cookie', array(
    'PHPSESSID=1234567890abcdef1234567890abcdef',
    'language=he'
));
```

setHeader() can also be easily used to set multiple headers in one call, by providing an array of headers as a single parameter:

Example 21.10. Setting Multiple Custom Request Headers

```
// Setting multiple headers, overwriting any previous value
$client->setHeaders(array(
    'Host' => 'www.example.com',
    'Accept-encoding' => 'gzip,deflate',
    'X-Powered-By' => 'Zend Framework'));

// The array can also contain full array strings:
$client->setHeaders(array(
    'Host: www.example.com',
    'Accept-encoding: gzip,deflate',
    'X-Powered-By: Zend Framework'));
```

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.

Example 21.11. Using setFileUpload to Upload Files

```
// Uploading arbitrary data as a file
$text = 'this is some plain text';
$client->setFileUpload('some_text.txt', 'upload', $text, 'text/plain');

// Uploading an existing file
$client->setFileUpload('/tmp/Backup.tar.gz', 'bufile');

// Send the files
$client->submit('POST');
```

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 guesses automatically if possible - and if not, the content type will be set to 'application/octet-stream'.

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 requests body on other request methods.

Sending Raw POST Data

You can use a Zend_Http_Client to send raw POST data using the setRawData() method. This method takes two parameters: the first is the data to send in the request body. The second optional parameter is the content-type of the data. While this parameter is optional, you should usually set it before sending the request - either using setRawData(), or with another method: setEncType().

Example 21.12. Sending Raw POST Data

```
$xml = '<book>' .
       '  <title>Islands in the Stream</title>' .
       '  <author>Ernest Hemingway</author>' .
       '  <year>1970</year>' .
       '</book>';

$client->setRawData($xml, 'text/xml')->request('POST');

// Another way to do the same thing:
$client->setRawData($xml)->setEncType('text/xml')->request('POST');
```

The data should be available on the server side through PHP's $HTTP_RAW_POST_DATA variable or through the php://input stream.

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. The 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).

Example 21.13. Setting HTTP Authentication User and Password

```
// Using basic authentication
$client->setAuth('shahar', 'myPassword!', Zend_Http_Client::AUTH_BASIC);

// Since basic auth is default, you can just do this:
$client->setAuth('shahar', 'myPassword!');
```

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 request-specific headers are reset and are not reused in the next request.

Reseting parameters

Note that non-request specific headers are not reset when the resetParameters method is used. As a matter of fact, only the 'Content-length' and 'Content-type' headers are reset. This allows you to set-and-forget headers like 'Accept-language' and 'Accept-encoding'

Another feature designed specifically for consecutive requests is the Cookie Jar object. Cookie Jars allow you to automatically save cookies set by the server in the first request, and send them 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 Cookie Jar object in the user's session. This way, you will only need to authenticate the user once every session.

Example 21.14. Performing consecutive requests with one client

```
// First, instantiate the client
$client = new Zend_Http_Client('http://www.example.com/fetchdata.php', array(
    'keepalive' => true
));

// Do we have the cookies stored in our session?
if (isset($_SESSION['cookiejar']) &&
    $_SESSION['cookiejar'] instanceof Zend_Http_CookieJar)) {

    $client->setCookieJar($_SESSION['cookiejar']);
} else {
    // If we don't, authenticate and store cookies
    $client->setCookieJar();
    $client->setUri('http://www.example.com/login.php');
    $client->setParameterPost(array(
        'user' => 'shahar',
        'pass' => 'somesecret'
    ));
    $client->request(Zend_Http_Client::POST);

    // Now, clear parameters and set the URI to the original one
    // (note that the cookies that were set by the server are now
    // stored in the jar)
    $client->resetParameters();
    $client->setUri('http://www.example.com/fetchdata.php');
}

$response = $client->request(Zend_Http_Client::GET);

// Store cookies in session, for next page
$_SESSION['cookiejar'] = $client->getCookieJar();
```

Zend_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 three built-in connection adapters:

- `Zend_Http_Client_Adapter_Socket` (default)
- `Zend_Http_Client_Adapter_Proxy`

- `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_Test`). You can also set the adapter later, using the Zend_Http_Client->setConfig() 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->setConfig()` or passed to the client constructor.

Table 21.2. Zend_Http_Client_Adapter_Socket configuration parameters

Parameter	Description	Expected Type	Default Value
persistent	Whether to use persistent TCP connections	boolean	false
ssltransport	SSL transport layer (eg. 'sslv2', 'tls')	string	ssl
sslcert	Path to a PEM encoded SSL certificate	string	null
sslpassphrase	Passphrase for the SSL certificate file	string	null

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 section called “Configuration Parameters” - otherwise persistent connections might have little or no effect.

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 [http://www.php.net/manual/en/transports.php#transports.inet].

Example 21.15. Changing the HTTPS transport layer

```
// Set the configuration parameters
$config = array(
    'adapter'      => 'Zend_Http_Client_Adapter_Socket',
    'ssltransport' => 'tls'
);

// Instantiate a client object
$client = Zend_Http_Client('https://www.example.com', $config);

// The following request will be sent over a TLS secure connection.
$response = $client->request();
```

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)
```

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 21.3. Zend_Http_Client configuration parameters

Parameter	Description	Expected Type	Example Value
proxy_host	Proxy server address	string	'proxy.myhost.com' or '10.1.2.3'
proxy_port	Proxy server TCP port	integer	8080 (default) or 81
proxy_user	Proxy user name, if required	string	'shahar' or '' for none (default)
proxy_pass	Proxy password, if required	string	'secret' or '' for none (default)
proxy_auth	Proxy HTTP authentication type	string	Zend_Http_Client::AUTH_BASIC (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.

Example 21.16. Using Zend_Http_Client behind a proxy server

```
// Set the configuration parameters
$config = array(
    'adapter'    => 'Zend_Http_Client_Adapter_Proxy',
    'proxy_host' => 'proxy.int.zend.com',
    'proxy_port' => 8000,
    'proxy_user' => 'shahar.e',
    'proxy_pass' => 'bananashaped'
);

// Instantiate a client object
$client = Zend_Http_Client('http://www.example.com', $config);

// Continue working...
```

As mentioned, if proxy_host is not set or is set to a blank string, the connection will fall back to a regular direct connection. This allows you to easily write your application in a way that allows a proxy to be used optionally, according to a configuration parameter.

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() method. 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.

Example 21.17. Testing Against a Single HTTP Response Stub

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Set the expected response
$adapter->setResponse(
    "HTTP/1.1 200 OK"        . "\r\n" .
    "Content-type: text/xml" . "\r\n" .
                               "\r\n" .
    '<?xml version="1.0" encoding="UTF-8"?>' .
    '<rss version="2.0" ' .
    '     xmlns:content="http://purl.org/rss/1.0/modules/content/"' .
    '     xmlns:wfw="http://wellformedweb.org/CommentAPI/"' .
    '     xmlns:dc="http://purl.org/dc/elements/1.1/">' .
    '  <channel>' .
    '    <title>Premature Optimization</title>' .
    // and so on...
    '</rss>');

$response = $client->request('GET');
// .. 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.

Example 21.18. Testing Against Multiple HTTP Response Stubs

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Set the first expected response
$adapter->setResponse(
    "HTTP/1.1 302 Found"      . "\r\n" .
    "Location: /"             . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
    '  <head><title>Moved</title></head>' .
    '  <body><p>This page has moved.</p></body>' .
    '</html>');

// Set the next successive response
$adapter->addResponse(
    "HTTP/1.1 200 OK"         . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
    '  <head><title>My Pet Store Home Page</title></head>' .
    '  <body><p>...</p></body>' .
    '</html>');

// inject the http client object ($client) into your object
// being tested and then test your object's behavior below
```

The setResponse() method clears any responses in the Zend_Http_Client_Adapter_Test's buffer and sets the first response that will be returned. The addResponse() method will add successive responses.

The responses will be replayed in the order that they were added. If more requests are made than the number of responses stored, the responses will cycle again in order.

In the example above, the adapter is configured to test your object's behavior when it encounters a 302 redirect. Depending on your application, following a redirect may or may not be desired behavior. In our example, we expect that the redirect will be followed and we configure the test adapter to help us test this. The initial 302 response is set up with the setResponse() method and the 200 response to be returned next is added with the addResponse() method. After configuring the test adapter, inject the HTTP client containing the adapter into your object under test and test its behavior.

Creating your own connection adapters

You can create your own connection adapters and use them. 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_Interface 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:

Example 21.19. Creating your own connection adapter

```
class MyApp_Http_Client_Adapter_BananaProtocol
    implements Zend_Http_Client_Adapter_Interface
{
    /**
     * Set the configuration array for the adapter
     *
     * @param array $config
     */
    public function setConfig($config = array())
    {
        // This rarely changes - you should usually copy the
        // implementation in Zend_Http_Client_Adapter_Socket.
    }

    /**
     * Connect to the remote server
     *
     * @param string  $host
     * @param int     $port
     * @param boolean $secure
     */
    public function connect($host, $port = 80, $secure = false)
    {
        // Set up the connection to the remote server
    }

    /**
     * Send request to the remote server
     *
     * @param string        $method
     * @param Zend_Uri_Http $url
     * @param string        $http_ver
     * @param array         $headers
     * @param string        $body
     * @return string Request as text
     */
    public function write($method,
                          $url,
                          $http_ver = '1.1',
                          $headers = array(),
                          $body = '')
    {
        // Send request to the remote server.
        // This function is expected to return the full request
        // (headers and body) as a string
    }

    /**
     * Read response from server
     *
     * @return string
     */
    public function read()
    {
        // Read response from remote server and return it as a string
    }

    /**
     * Close the connection to the server
     *
     */
    public function close()
    {
        // Close the connection to the remote server - called last.
    }
}

// Then, you could use this adapter:
$client = new Zend_Http_Client(array(
    'adapter' => 'MyApp_Http_Client_Adapter_BananaProtocol'
));
```

Zend_Http_Cookie and Zend_Http_CookieJar

Introduction

Zend_Http_Cookie, as expected, is a class that represents an HTTP cookie. It provides methods for parsing HTTP response strings, collecting cookies, and easily accessing their properties. It also allows checking if a cookie matches against a specific scenario, IE a request URL, expiration time, secure connection, etc.

Zend_Http_CookieJar is an object usually used by Zend_Http_Client to hold a set of Zend_Http_Cookie objects. The idea is that if a Zend_Http_CookieJar object is attached to a Zend_Http_Client object, all cookies going from and into the client through HTTP requests and responses will be stored by the CookieJar object. Then, when the client will send another request, it will first ask the CookieJar object for all cookies matching the request. These will be added to the request headers automatically. This is highly useful in cases where you need to maintain a user session over consecutive HTTP requests, automatically sending the session ID cookies when required. Additionally, the Zend_Http_CookieJar object can be serialized and stored in $_SESSION when needed.

Instantiating Zend_Http_Cookie Objects

Instantiating a Cookie object can be done in two ways:

- Through the constructor, using the following syntax: `new Zend_Http_Cookie(string $name, string $value, string $domain, [int $expires, [string $path, [boolean $secure]]]);`
  - `$name`: The name of the cookie (eg. 'PHPSESSID') (required)
  - `$value`: The value of the cookie (required)

    - `$domain`: The cookie's domain (eg. '.example.com') (required)
    - `$expires`: Cookie expiration time, as UNIX time stamp (optional, defaults to null). If not set, cookie will be treated as a 'session cookie' with no expiration time.
    - `$path`: Cookie path, eg. '/foo/bar/' (optional, defaults to '/')
    - `$secure`: Boolean, Whether the cookie is to be sent over secure (HTTPS) connections only (optional, defaults to boolean FALSE)

- By calling the fromString() static method, with a cookie string as represented in the 'Set-Cookie' HTTP response header or 'Cookie' HTTP request header. In this case, the cookie value must already be encoded. When the cookie string does not contain a 'domain' part, you must provide a reference URI according to which the cookie's domain and path will be set.

Example 21.20. Instantiating a Zend_Http_Cookie object

```
// First, using the constructor. This cookie will expire in 2 hours
$cookie = new Zend_Http_Cookie('foo',
                               'bar',
                               '.example.com',
                               time() + 7200,
                               '/path');

// You can also take the HTTP response Set-Cookie header and use it.
// This cookie is similar to the previous one, only it will not expire, and
// will only be sent over secure connections
$cookie = Zend_Http_Cookie::fromString('foo=bar; domain=.example.com; ' .
                                       'path=/path; secure');

// If the cookie's domain is not set, you have to manually specify it
$cookie = Zend_Http_Cookie::fromString('foo=bar; secure;',
                                       'http://www.example.com/path');
```

Note

When instantiating a cookie object using the Zend_Http_Cookie::fromString() method, the cookie value is expected to be URL encoded, as cookie strings should be. However, when using the constructor, the cookie value string is expected to be the real, decoded value.

A cookie object can be transferred back into a string, using the __toString() magic method. This method will produce a HTTP request "Cookie" header string, showing the cookie's name and value, and terminated by a semicolon (';'). The value will be URL encoded, as expected in a Cookie header:

Example 21.21. Stringifying a Zend_Http_Cookie object

```
// Create a new cookie
$cookie = new Zend_Http_Cookie('foo',
                               'two words',
                               '.example.com',
                               time() + 7200,
                               '/path');

// Will print out 'foo=two+words;' :
echo $cookie->__toString();

// This is actually the same:
echo (string) $cookie;

// In PHP 5.2 and higher, this also works:
echo $cookie;
```

Zend_Http_Cookie getter methods

Once a Zend_Http_Cookie object is instantiated, it provides several getter methods to get the different properties of the HTTP cookie:

- `string getName()`: Get the name of the cookie
- `string getValue()`: Get the real, decoded value of the cookie
- `string getDomain()`: Get the cookie's domain
- `string getPath()`: Get the cookie's path, which defaults to '/'
- `int getExpiryTime()`: Get the cookie's expiration time, as UNIX time stamp. If the cookie has no expiration time set, will return NULL.

Additionally, several boolean tester methods are provided:

- `boolean isSecure()`: Check whether the cookie is set to be sent over secure connections only. Generally speaking, if true the cookie should only be sent over HTTPS.
- `boolean isExpired(int $time = null)`: Check whether the cookie is expired or not. If the cookie has no expiration time, will always return true. If $time is provided, it will override the current time stamp as the time to check the cookie against.
- `boolean isSessionCookie()`: Check whether the cookie is a "session cookie" - that is a cookie with no expiration time, which is meant to expire when the session ends.

Example 21.22. Using getter methods with Zend_Http_Cookie

```
// First, create the cookie
$cookie =
    Zend_Http_Cookie::fromString('foo=two+words; ' +
                                 'domain=.example.com; ' +
                                 'path=/somedir; ' +
                                 'secure; ' +
                                 'expires=Wednesday, 28-Feb-05 20:41:22 UTC');

echo $cookie->getName();   // Will echo 'foo'
echo $cookie->getValue();  // will echo 'two words'
echo $cookie->getDomain(); // Will echo '.example.com'
echo $cookie->getPath();   // Will echo '/'

echo date('Y-m-d', $cookie->getExpiryTime());
// Will echo '2005-02-28'

echo ($cookie->isExpired() ? 'Yes' : 'No');
// Will echo 'Yes'

echo ($cookie->isExpired(strtotime('2005-01-01') ? 'Yes' : 'No');
// Will echo 'No'

echo ($cookie->isSessionCookie() ? 'Yes' : 'No');
// Will echo 'No'
```

Zend_Http_Cookie: Matching against a scenario

The only real logic contained in a Zend_Http_Cookie object, is in the match() method. This method is used to test a cookie against a given HTTP request scenario, in order to tell whether the cookie should be sent in this request or not. The method has the following syntax and parameters: `boolean Zend_Http_Cookie->match(mixed $uri, [boolean $matchSessionCookies, [int $now]]);`

- `mixed $uri`: A Zend_Uri_Http object with a domain name and path to be checked. Optionally, a string representing a valid HTTP URL can be passed instead. The cookie will match if the URL's scheme (HTTP or HTTPS), domain and path all match.
- `boolean $matchSessionCookies`: Whether session cookies should be matched or not. Defaults to true. If set to false, cookies with no expiration time will never match.
- `int $now`: Time (represented as UNIX time stamp) to check a cookie against for expiration. If not specified, will default to the current time.

Example 21.23. Matching cookies

```
// Create the cookie object - first, a secure session cookie
$cookie = Zend_Http_Cookie::fromString('foo=two+words; ' +
                                       'domain=.example.com; ' +
                                       'path=/somedir; ' +
                                       'secure;');

$cookie->match('https://www.example.com/somedir/foo.php');
// Will return true

$cookie->match('http://www.example.com/somedir/foo.php');
// Will return false, because the connection is not secure

$cookie->match('https://otherexample.com/somedir/foo.php');
// Will return false, because the domain is wrong

$cookie->match('https://example.com/foo.php');
// Will return false, because the path is wrong

$cookie->match('https://www.example.com/somedir/foo.php', false);
// Will return false, because session cookies are not matched

$cookie->match('https://sub.domain.example.com/somedir/otherdir/foo.php');
// Will return true

// Create another cookie object - now, not secure, with expiration time
// in two hours
$cookie = Zend_Http_Cookie::fromString('foo=two+words; ' +
                                       'domain=www.example.com; ' +
                                       'expires='
                                       . date(DATE_COOKIE, time() + 7200));

$cookie->match('http://www.example.com/');
// Will return true

$cookie->match('https://www.example.com/');
// Will return true - non secure cookies can go over secure connections
// as well!

$cookie->match('http://subdomain.example.com/');
// Will return false, because the domain is wrong

$cookie->match('http://www.example.com/', true, time() + (3 * 3600));
// Will return false, because we added a time offset of +3 hours to
// current time
```

The Zend_Http_CookieJar Class: Instantiation

In most cases, there is no need to directly instantiate a Zend_Http_CookieJar object. If you want to attach a new cookie jar to your Zend_Http_Client object, just call the Zend_Http_Client->setCookieJar() method, and a new, empty cookie jar will be attached to your client. You could later get this cookie jar using Zend_Http_Client->getCookieJar().

If you still wish to manually instantiate a CookieJar object, you can do so by calling "new Zend_Http_CookieJar()" directly - the constructor method does not take any parameters. Another way to instantiate a CookieJar object is to use the static Zend_Http_CookieJar::fromResponse() method. This method takes two parameters: a Zend_Http_Response object, and a reference URI, as either a string or a Zend_Uri_Http object. This method will return a new Zend_Http_CookieJar object, already containing the cookies set by the passed HTTP response. The reference URI will be used to set the cookie's domain and path, if they are not defined in the Set-Cookie headers.

Adding Cookies to a Zend_Http_CookieJar object

Usually, the Zend_Http_Client object you attached your CookieJar object to will automatically add cookies set by HTTP responses to your jar. If you wish to manually add cookies to your jar, this can be done by using two methods:

- `Zend_Http_CookieJar->addCookie($cookie[, $ref_uri])`: Add a single cookie to the jar. $cookie can be either a Zend_Http_Cookie object or a string, which will be converted automatically into a Cookie object. If a string is provided, you should also provide $ref_uri - which is a reference URI either as a string or Zend_Uri_Http object, to use as the cookie's default domain and path.
- `Zend_Http_CookieJar->addCookiesFromResponse($response, $ref_uri)`: Add all cookies set in a single HTTP response to the jar. $response is expected to be a Zend_Http_Response object with Set-Cookie headers. $ref_uri is the request URI, either as a string or a Zend_Uri_Http object, according to which the cookies' default domain and path will be set.

Retrieving Cookies From a Zend_Http_CookieJar object

Just like with adding cookies, there is usually no need to manually fetch cookies from a CookieJar object. Your Zend_Http_Client object will automatically fetch the cookies required for an HTTP request for you. However, you can still use 3 provided methods to fetch cookies from the jar object: `getCookie()`, `getAllCookies()`, and `getMatchingCookies()`.

It is important to note that each one of these methods takes a special parameter, which sets the return type of the method. This parameter can have 3 values:

- `Zend_Http_CookieJar::COOKIE_OBJECT`: Return a Zend_Http_Cookie object. If the method returns more than one cookie, an array of objects will be returned.
- `Zend_Http_CookieJar::COOKIE_STRING_ARRAY`: Return cookies as strings, in a "foo=bar" format, suitable for sending in a HTTP request "Cookie" header. If more than one cookie is returned, an array of strings is returned.
- `Zend_Http_CookieJar::COOKIE_STRING_CONCAT`: Similar to COOKIE_STRING_ARRAY, but if more than one cookie is returned, this method will concatenate all cookies into a single, long string separated by semicolons (;), and return it. This is especially useful if you want to directly send all matching cookies in a single HTTP request "Cookie" header.

The structure of the different cookie-fetching methods is described below:

- `Zend_Http_CookieJar->getCookie($uri, $cookie_name[, $ret_as])`: Get a single cookie from the jar, according to it's URI (domain and path) and name. $uri is either a string or a Zend_Uri_Http object representing the URI. $cookie_name is a string identifying the cookie name. $ret_as specifies the return type as described above. $ret_type is optional, and defaults to COOKIE_OBJECT.
- `Zend_Http_CookieJar->getAllCookies($ret_as)`: Get all cookies from the jar. $ret_as specifies the return type as described above. If not specified, $ret_type defaults to COOKIE_OBJECT.
- `Zend_Http_CookieJar->getMatchingCookies($uri[, $matchSessionCookies[, $ret_as[, $now]]])`: Get all cookies from the jar that match a specified scenario, that is a URI and expiration time.
  - `$uri` is either a Zend_Uri_Http object or a string specifying the connection type (secure or non-secure), domain and path to match against.
  - `$matchSessionCookies` is a boolean telling whether to match session cookies or not. Session cookies are cookies that have no specified expiration time. Defaults to true.
  - `$ret_as` specifies the return type as described above. If not specified, defaults to COOKIE_OBJECT.
  - `$now` is an integer representing the UNIX time stamp to consider as "now" - that is any cookies who are set to expire before this time will not be matched. If not specified, defaults to the current time.

  You can read more about cookie matching here: the section called “Zend_Http_Cookie: Matching against a scenario”.

Zend_Http_Response

Introduction

Zend_Http_Response provides easy access to an HTTP responses message, as well as a set of static methods for parsing HTTP response messages. Usually, Zend_Http_Response is used as an object returned by a Zend_Http_Client request.

In most cases, a Zend_Http_Response object will be instantiated using the factory() method, which reads a string containing an HTTP response message, and returns a new Zend_Http_Response object:

Example 21.24. Instantiating a Zend_Http_Response object using the factory method

```
$str = '';
$sock = fsockopen('www.example.com', 80);
$req =     "GET / HTTP/1.1\r\n" .
        "Host: www.example.com\r\n" .
        "Connection: close\r\n" .
        "\r\n";

fwrite($sock, $req);
while ($buff = fread($sock, 1024))
    $str .= $sock;

$response = Zend_Http_Response::factory($str);
```

You can also use the contractor method to create a new response object, by specifying all the parameters of the response:

```
public function __construct($code, $headers, $body = null, $version =
'1.1', $message = null)
```

- `$code`: The HTTP response code (eg. 200, 404, etc.)
- `$headers`: An associative array of HTTP response headers (eg. 'Host' => 'example.com')
- `$body`: The response body as a string
- `$version`: The HTTP response version (usually 1.0 or 1.1)
- `$message`: The HTTP response message (eg 'OK', 'Internal Server Error'). If not specified, the message will be set according to the response code

Boolean Tester Methods

Once a Zend_Http_Response object is instantiated, it provides several methods that can be used to test the type of the response. These all return Boolean true or false:

- `Boolean isSuccessful()`: Whether the request was successful or not. Returns TRUE for HTTP 1xx and 2xx response codes
- `Boolean isError()`: Whether the response code implies an error or not. Returns TRUE for HTTP 4xx (client errors) and 5xx (server errors) response codes
- `Boolean isRedirect()`: Whether the response is a redirection response or not. Returns TRUE for HTTP 3xx response codes

Example 21.25. Using the isError() method to validate a response

```
if ($response->isError()) {
  echo "Error transmitting data.\n"
  echo "Server reply was: " . $response->getStatus() .
       " " . $response->getMessage() . "\n";
}
// .. process the response here...
```

Accessor Methods

The main goal of the response object is to provide easy access to various response parameters.

- `int getStatus()`: Get the HTTP response status code (eg. 200, 504, etc.)
- `string getMessage()`: Get the HTTP response status message (eg. "Not Found", "Authorization Required")
- `string getBody()`: Get the fully decoded HTTP response body
- `string getRawBody()`: Get the raw, possibly encoded HTTP response body. If the body was decoded using GZIP encoding for example, it will not be decoded.
- `array getHeaders()`: Get the HTTP response headers as an associative array (eg. 'Content-type' => 'text/html')
- `string|array getHeader($header)`: Get a specific HTTP response header, specified by $header
- `string getHeadersAsString($status_line = true, $br = "\n")`: Get the entire set of headers as a string. If $status_line is true (default), the first status line (eg. "HTTP/1.1 200 OK") will also be returned. Lines are broken with the $br parameter (Can be, for example, "<br />")
- `string asString($br = "\n")`: Get the entire response message as a string. Lines are broken with the $br parameter (Can be, for example, "<br />")

Example 21.26. Using Zend_Http_Response Accessor Methods

```
if ($response->getStatus() == 200) {
  echo "The request returned the following information:<br />";
  echo $response->getBody();
} else {
  echo "An error occurred while fetching data:<br />";
  echo $response->getStatus() . ": " . $response->getMessage();
}
```

Always check return value

Since a response can contain several instances of the same header, the getHeader() method and getHeaders() method may return either a single string, or an array of strings for each header. You should always check whether the returned value is a string or array.

Example 21.27. Accessing Response Headers

```
$ctype = $response->getHeader('Content-type');
if (is_array($ctype)) $ctype = $ctype[0];

$body = $response->getBody();
if ($ctype == 'text/html' || $ctype == 'text/xml') {
  $body = htmlentities($body);
}

echo $body;
```

Static HTTP Response Parsers

The Zend_Http_Response class also includes several internally-used methods for processing and parsing HTTP response messages. These methods are all exposed as static methods, which means they can be used externally, even if you do not need to instantiate a response object, and just want to extract a specific part of the response.

- `int Zend_Http_Response::extractCode($response_str)`: Extract and return the HTTP response code (eg. 200 or 404) from $response_str
- `string Zend_Http_Response::extractMessage($response_str)`: Extract and return the HTTP response message (eg. "OK" or "File Not Found") from $response_str
- `string Zend_Http_Response::extractVersion($response_str)`:: Extract and return the HTTP version (eg. 1.1 or 1.0) from $response_str
- `array Zend_Http_Response::extractHeaders($response_str)`: Extract and return the HTTP response headers from $response_str as an array
- `string Zend_Http_Response::extractBody($response_str)`: Extract and return the HTTP response body from $response_str
- `string Zend_Http_Response::responseCodeAsText($code = null, $http11 = true)`: Get the standard HTTP response message for a response code $code. For example, will return "Internal Server Error" if $code is 500. If $http11 is true (default), will return HTTP/1.1 standard messages - otherwise HTTP/1.0 messages will be returned. If $code is not specified, this method will return all known HTTP response codes as an associative (code => message) array.

Apart from parser methods, the class also includes a set of decoders for common HTTP response transfer encodings:

- `string Zend_Http_Response::decodeChunkedBody($body)`: Decode a complete "Content-Transfer-Encoding: Chunked" body

- `string Zend_Http_Response::decodeGzip($body)`: Decode a "Content-Encoding: gzip" body
- `string Zend_Http_Response::decodeDeflate($body)`: Decode a "Content-Encoding: deflate" body

Chapter 22. Zend_InfoCard

Introduction

The `Zend_InfoCard` component implements relying-party support for Information Cards. Infomation Cards are used for identity management on the internet and authentication of users to web sites (called relying parties).

Detailed information about information cards and their importance to the internet identity metasystem can be found on the IdentityBlog [http://www.identityblog.com/]

Basic Theory of Usage

Usage of `Zend_InfoCard` can be done one of two ways: either as part of the larger `Zend_Auth` component via the `Zend_InfoCard` authentication adapter or as a stand-alone component. In both cases an information card can be requested from a user by using the following HTML block in your HTML login form:

```
<form action="http://example.com/server" method="POST">
  <input type='image' src='/images/ic.png' align='center'
        width='120px' style='cursor:pointer' />
  <object type="application/x-informationCard"
          name="xmlToken">
   <param name="tokenType"
         value="urn:oasis:names:tc:SAML:1.0:assertion" />
   <param name="requiredClaims"
         value="http://.../claims/privatepersonalidentifier
         http://.../claims/givenname
         http://.../claims/surname" />
 </object>
</form>
```

In the example above, the `requiredClaims` `<param>` tag is used to identify pieces of information known as claims (i.e. person's first name, last name) which the web site (a.k.a "relying party") needs in order a user to authenticate using an information card. For your reference, the full URI (for instance the `givenname` claim) is as follows: `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/givenname`

When the above HTML is activated by a user (clicks on it), the browser will bring up a card selection program which not only shows them which information cards meet the requirements of the site, but also allows them to select which information card to use if multiple meet the criteria. This information card is transmitted as an XML document to the specified `POST` URL and is ready to be processed by the `Zend_InfoCard` component.

Note, Information cards can only be `HTTP POST`ed to SSL-encrypted URLs. Please consult your web server's documentation on how to set up SSL encryption.

Using as part of Zend_Auth

In order to use the component as part of the `Zend_Auth` authentication system, you must use the provided `Zend_Auth_Adapter_InfoCard` to do so (not available in the standalone `Zend_InfoCard` distribution). An example of its usage is shown below:

```
<?php
if (isset($_POST['xmlToken'])) {

    $adapter = new Zend_Auth_Adapter_InfoCard($_POST['xmlToken']);

    $adapter->addCertificatePair('/usr/local/Zend/apache2/conf/server.key',
                                 '/usr/local/Zend/apache2/conf/server.crt');

    $auth = Zend_Auth::getInstance();

    $result = $auth->authenticate($adapter);

    switch ($result->getCode()) {
        case Zend_Auth_Result::SUCCESS:
            $claims = $result->getIdentity();
            print "Given Name: {$claims->givenname}<br />";
            print "Surname: {$claims->surname}<br />";
            print "Email Address: {$claims->emailaddress}<br />";
            print "PPI: {$claims->getCardID()}<br />";
            break;
        case Zend_Auth_Result::FAILURE_CREDENTIAL_INVALID:
            print "The Credential you provided did not pass validation";
            break;
        default:
        case Zend_Auth_Result::FAILURE:
            print "There was an error processing your credentials.";
            break;
    }

    if (count($result->getMessages()) > 0) {
        print "<pre>";
        var_dump($result->getMessages());
        print "</pre>";
    }

}
?>
<hr />
<div id="login" style="font-family: arial; font-size: 2em;">
<p>Simple Login Demo</p>
 <form method="post">
  <input type="submit" value="Login" />
   <object type="application/x-informationCard" name="xmlToken">
    <param name="tokenType"
          value="urn:oasis:names:tc:SAML:1.0:assertion" />
    <param name="requiredClaims"
```

```
          value="http://.../claims/givenname
                 http://.../claims/surname
                 http://.../claims/emailaddress
                 http://.../claims/privatepersonalidentifier" />
  </object>
 </form>
</div>
```

In the example above, we first create an instance of the `Zend_Auth_Adapter_InfoCard` and pass the XML data posted by the card selector into it. Once an instance has been created you must then provide at least one SSL certificate public/private key pair used by the web server which received the `HTTP POST`. These files are used to validate the destination of the information posted to the server and are a requirement when using Information Cards.

Once the adapter has been configured you can then use the standard `Zend_Auth` facilities to validate the provided information card token and authenticate the user by examining the identity provided by the `getIdentity()` method.

Using the Zend_InfoCard component standalone

It is also possible to use the Zend_InfoCard component as a standalone component by interacting with the `Zend_InfoCard` class directly. Using the Zend_InfoCard class is very similar to its use with the `Zend_Auth` component. An example of its use is shown below:

```
<?php
if (isset($_POST['xmlToken'])) {
    $infocard = new Zend_InfoCard();
    $infocard->addCertificatePair('/usr/local/Zend/apache2/conf/server.key',
                                  '/usr/local/Zend/apache2/conf/server.crt');

    $claims = $infocard->process($_POST['xmlToken']);

    if($claims->isValid()) {
        print "Given Name: {$claims->givenname}<br />";
        print "Surname: {$claims->surname}<br />";
        print "Email Address: {$claims->emailaddress}<br />";
        print "PPI: {$claims->getCardID()}<br />";
    } else {
        print "Error Validating identity: {$claims->getErrorMsg()}";
    }
}
?>
<hr />
<div id="login" style="font-family: arial; font-size: 2em;">
 <p>Simple Login Demo</p>
 <form method="post">
  <input type="submit" value="Login" />
   <object type="application/x-informationCard" name="xmlToken">
    <param name="tokenType"
          value="urn:oasis:names:tc:SAML:1.0:assertion" />
    <param name="requiredClaims"
```

```
            value="http://.../claims/givenname
                   http://.../claims/surname
                   http://.../claims/emailaddress
                   http://.../claims/privatepersonalidentifier" />
    </object>
  </form>
</div>
```

In the example above we use the `Zend_InfoCard` component indepdently to validate the token provided by the user. As was the case with the `Zend_Auth_Adapter_InfoCard`, we create an instance of `Zend_InfoCard` and then set one or more SSL certificate public/private key pairs used by the web server. Once configured we can use the `process()` method to process the information card and return the results

Working with a Claims object

Regardless of if the `Zend_InfoCard` component is used as a standalone component or as part of `Zend_Auth` via the `Zend_Auth_Adapter_InfoCard`, in both cases the ultimate result of the processing of an information card is a `Zend_InfoCard_Claims` object. This object contains the assertions (a.k.a. claims) made by the submitting user based on the data requested by your web site when the user authenticated. As shown in the examples above, the validitiy of the information card can be ascertained by calling the `Zend_InfoCard_Claims::isVaild()` method. Claims themselves can either be retrieved by simply accessing the identifier desired (i.e. `givenname`) as a property of the object or through the `getClaim()` method.

In most cases you will never need to use the `getClaim()` method. However, if your `requiredClaims` mandate that you request claims from multiple different sources/namespaces then you will need to extract them explictally using this method (simply pass it the full URI of the claim to retrieve its value from within the information card). Generally speaking however, the `Zend_InfoCard` component will set the default URI for claims to be the one used the most frequently within the information card itself and the simplified property-access method can be used.

As part of the validation process, it is up to the developer to examine the issuing source of the claims contained within the information card and decide if that source is a trusted source of information. To do so, the `getIssuer()` method is provided within the `Zend_InfoCard_Claims` object which returns the URI of the issuer of the information card claims.

Attaching Information Cards to existing accounts

It is possible to add support for information cards to an existing authentication system by storing the private personal identifier (PPI) to a previously traditionally-authenticated account and including at least the `http://schemas.xmlsoap.org/ws/2005/05/identity/claims/privatepersonalidentifier` claim as part of the `requiredClaims` of the request. If this claim is requested then the `Zend_InfoCard_Claims` object will provide a unique identifier for the specific card that was submitted by calling the `getCardID()` method.

An example of how to attach an information card to an existing traditional-authentication account is shown below:

```
// ...
public function submitinfocardAction()
```

```
{
    if (!isset($_REQUEST['xmlToken'])) {
        throw new ZBlog_Exception('Expected an encrypted token ' .
                                  'but was not provided');
    }

    $infoCard = new Zend_InfoCard();
    $infoCard->addCertificatePair(SSL_CERTIFICATE_PRIVATE,
                                  SSL_CERTIFICATE_PUB);

    try {
        $claims = $infoCard->process($request['xmlToken']);
    } catch(Zend_InfoCard_Exception $e) {
        // TODO Error processing your request
        throw $e;
    }

    if ($claims->isValid()) {
        $db = ZBlog_Data::getAdapter();

        $ppi = $db->quote($claims->getCardID());
        $fullname = $db->quote("{$claims->givenname} {$claims->surname}");

        $query = "UPDATE blogusers
                     SET ppi = $ppi,
                         real_name = $fullname
                   WHERE username='administrator'";

        try {
            $db->query($query);
        } catch(Exception $e) {
            // TODO Failed to store in DB
        }

        $this->view->render();
        return;
    } else {
        throw new
            ZBlog_Exception("Infomation card failed security checks");
    }
}
```

Creating Zend_InfoCard adapters

The `Zend_InfoCard` component was designed to allow for growth in the information card standard through the use of a modular architecture. At this time many of these hooks are unused and can be ignored, however there is one aspect which should be implemented in any serious information card implementation: The `Zend_InfoCard_Adapter`.

The `Zend_InfoCard` adapter is used as a callback mechanism within the component to perform various tasks, such as storing and retrieving Assertion IDs for information cards when they are processed by the component. While storing the assertion IDs of submitted information cards is not necessary, failing to do so opens up the possibility of the authentication scheme being compromised through a replay attack.

To prevent this, one must implement the `Zend_InfoCard_Adapter_Interface` and then set an instance of this interface prior to calling either the `process()` (standalone) or `authenticate()` method (as a `Zend_Auth` adapter. To set this interface the `setAdapter()` method is used. In the example below we set a `Zend_InfoCard` adapter and use it within our application:

```
class myAdapter implements Zend_InfoCard_Adapter_Interface
{
    public function storeAssertion($assertionURI,
                                   $assertionID,
                                   $conditions)
    {
        /* Store the assertion and its conditions by ID and URI */
    }

    public function retrieveAssertion($assertionURI, $assertionID)
    {
        /* Retrieve the assertion by URI and ID */
    }

    public function removeAssertion($assertionURI, $assertionID)
    {
        /* Delete a given assertion by URI/ID */
    }
}

$adapter  = new myAdapter();

$infoCard = new Zend_InfoCard();
$infoCard->addCertificatePair(SSL_PRIVATE, SSL_PUB);
$infoCard->setAdapter($adapter);

$claims = $infoCard->process($_POST['xmlToken']);
```

Chapter 23. Zend_Json

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 [http://www.json.org/].

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.

Basic Usage

Usage of `Zend_Json` involves using the two public static methods available: `Zend_Json::encode()` and `Zend_Json::decode()`.

```
// Retrieve a value:
$phpNative = Zend_Json::decode($encodedValue);

// Encode it to return to the client:
$json = Zend_Json::encode($phpNative);
```

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::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 closesly 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` will decode JSON objects as associative arrays. However, if you desire an object returned, you can specify this:

```
// Decode JSON objects as PHP objects
```

```
$phpNative = Zend_Json::decode($encodedValue, Zend_Json::TYPE_OBJECT);
```

Any objects thus decoded are returned as `StdClass` objects with properties corresponding to the key/value pairs in the JSON notation.

The recommendation of the 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`.

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 [http://www.ibm.com/developerworks/xml/library/x-xml2jsonphp/].

`Zend_Json` includes a static function called `Zend_Json::fromXml()`. This function will generate JSON from a given XML input. This function takes any aribitrary XML string as an input parameter. It also takes an optional boolean input parameter to instruct the conversion logic to ignore or not ignore the XML attributes during the conversion process. If this optional input parameter is not given, then the default behavior is to ignore the XML attributes. This function call is made as shown below:

```
// fromXml function simply takes a String containing XML contents
// as input.
$jsonContents = Zend_Json::fromXml($xmlStringContents, true);
```

`Zend_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 the Zend Framework. They can be used to test the functionality of the xml2json feature.

The following is a simple example that shows both the XML input string passed to and the JSON output string returned as a result from the `Zend_Json::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::fromXml()` function:

```
<?xml version="1.0" encoding="UTF-8"?>
<books>
    <book id="1">
        <title>Code Generation in Action</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>Manning</publisher>
    </book>

    <book id="2">
```

```
        <title>PHP Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>

    <book id="3">
        <title>Podcasting Hacks</title>
        <author><first>Jack</first><last>Herrington</last></author>
        <publisher>O'Reilly</publisher>
    </book>
</books>
```

JSON output string returned from `Zend_Json::fromXml()` function:

```
{
   "books" : {
      "book" : [ {
         "@attributes" : {
            "id" : "1"
         },
         "title" : "Code Generation in Action",
         "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "Manning"
      }, {
         "@attributes" : {
            "id" : "2"
         },
         "title" : "PHP Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }, {
         "@attributes" : {
            "id" : "3"
         },
         "title" : "Podcasting Hacks", "author" : {
            "first" : "Jack", "last" : "Herrington"
         },
         "publisher" : "O'Reilly"
      }
   ]}
}
```

More details about this xml2json feature can be found in the original proposal itself. Take a look at the Zend_xml2json proposal [http://tinyurl.com/2tfa8z].

Zend_Json_Server - JSON-RPC server

`Zend_Json_Server` is a JSON-RPC [http://groups.google.com/group/json-rpc/] server implementation. It supports both the JSON-RPC version 1 specification [http://json-rpc.org/wiki/specification] as well as the version 2 specification [http://groups.google.com/group/json-rpc/web/json-rpc-1-2-proposal]; additionally, it provides a PHP implemention of the Service Mapping Description (SMD) specification [http://groups.google.com/group/json-schema/web/service-mapping-description-proposal] 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 [http://us.php.net/manual/en/function.soap-soapserver-construct.php] API. This means that 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 the section called “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.

Example 23.1. Zend_Json_Server Usage

First, let's define a class we wish to expose via the JSON-RPC server. We'll call the class 'Calculator', and define methods for 'add', 'subtract', 'multiply', and 'divide':

```
/**
 * Calculator - sample class to expose via JSON-RPC
 */
class Calculator
{
    /**
     * Return sum of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function add($x, $y)
    {
        return $x + $y;
    }

    /**
     * Return difference of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function subtract($x, $y)
    {
        return $x - $y;
    }

    /**
     * Return product of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return int
     */
    public function multiply($x, $y)
    {
        return $x * $y;
    }

    /**
     * Return the product of division of two variables
     *
     * @param  int $x
     * @param  int $y
     * @return float
     */
    public function divide($x, $y)
    {
        return $x / $y;
    }
}
```

Note that each method has a docblock with entries indicating each parameter and its type, as well as an entry for the return value. This is *absolutely critical* when utilizing `Zend_Json_Server` -- or any other server component in Zend Framework, for that matter.

Now we'll create a script to handle the requests:

```
$server = new Zend_Json_Server();

// Indicate what functionality is available:
$server->setClass('Calculator');

// Handle the request:
$server->handle();
```

However, this will not address the issue of returning an SMD so that the JSON-RPC client can autodiscover methods. That can be accomplished by determining the HTTP request method, and then specifying some server metadata:

```
$server = new Zend_Json_Server();
$server->setClass('Calculator');

if ('GET' == $_SERVER['REQUEST_METHOD']) {
    // Indicate the URL endpoint, and the JSON-RPC version used:
    $server->setTarget('/json-rpc.php')
           ->setEnvelope(Zend_Json_Server_Smd::ENV_JSONRPC_2);

    // Grab the SMD
    $smd = $server->getServiceMap();

    // Return the SMD to the client
    header('Content-Type: application/json');
    echo $smd;
    return;
}

$server->handle();
```

If utilizing the JSON-RPC server with Dojo toolkit, you will also need to set a special compatibility flag to ensure that the two interoperate properly:

```
$server = new Zend_Json_Server();
$server->setClass('Calculator');

if ('GET' == $_SERVER['REQUEST_METHOD']) {
    $server->setTarget('/json-rpc.php')
           ->setEnvelope(Zend_Json_Server_Smd::ENV_JSONRPC_2);
    $smd = $server->getServiceMap();

    // Set Dojo compatibility:
    $smd->setDojoCompatible(true);

    header('Content-Type: application/json');
    echo $smd;
    return;
}

$server->handle();
```

Advanced Details

While most functionality for `Zend_Json_Server` is spelled out in Example 23.1, “Zend_Json_Server Usage”, more advanced functionality is available.

Zend_Json_Server

`Zend_Json_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 [http://xmlrpc-epi.sourceforge.net/specs/rfc.fault_codes.php]. `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 [http://groups.google.com/group/json-schema/web/service-mapping-description-proposal] 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 object 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` 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($envelopeType)`: Set the request envelope that should be used to access the service. Currently, supports the constants `Zend_Json_Server_Smd::ENV_JSONRPC_1` and `Zend_Json_Server_Smd::ENV_JSONRPC_1`.
- `getEnvelope()`: Get the current request envelope.
- `setContentType($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 compatability 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 overal SMD to which the service is attached).

- `getTarget()`: Get the URL endpoint of the service.
- `setEnvelope($envelopeType)`: Set the service envelope (currently, only envelopes supported by `Zend_Json_Server_Smd` are allowed).
- `getEnvelope()`: Retrieve the service envelope type.
- `addParam($type, array $options = array(), $order = null)`: Add a parameter to the service. By default, only the parameter type is necessary. However, you may also specify the order, as well as options such as:
  - *name*: the parameter name
  - *optional*: whether or not the parameter is optional
  - *default*: a default value for the parameter
  - *description*: text describing the parameter
- `addParams(array $params)`: Add several parameters at once; each param should be an assoc array containing minimally the key 'type' describing the parameter type, and optionally the key 'order'; any other keys will be passed as `$options` to `addOption()`.
- `setParams(array $params)`: Set many parameters at once, overwriting any existing parameters.
- `getParams()`: Retrieve all currently set parameters.
- `setReturn($type)`: Set the return value type of the service.
- `getReturn()`: Get the return value type of the service.
- `toArray()`: Cast the service to an array.
- `toJson()`: Cast the service to a JSON representation.

Chapter 24. Zend_Layout

Introduction

`Zend_Layout` implements a classic Two Step View pattern, allowing developers to wrap application content within another view, usually representing the site template. Such templates are often termed *layouts* by other projects, and Zend Framework has adopted this term for consistency.

The main goals of `Zend_Layout` are as follows:

- Automate selection and rendering of layouts when used with the Zend Framework MVC components.
- Provide separate scope for layout related variables and content.
- Allow configuration, including layout name, layout script resolution (inflection), and layout script path.
- Allow disabling layouts, changing layout scripts, and other states; allow these actions from within action controllers and view scripts.
- Follow same script resolution rules (inflection) as the ViewRenderer, but allow them to also use different rules.
- Allow usage without Zend Framework MVC components.

Zend_Layout Quick Start

There are two primary use cases for `Zend_Layout`: with the Zend Framework MVC, and without.

Layout scripts

In both cases, however, you'll need to create a layout script. Layout scripts simply utilize Zend_View (or whatever view implementation you are using). Layout variables are registered with a `Zend_Layout` placeholder, and may be accessed via the placeholder helper or by fetching them as object properties of the layout object via the layout helper.

As an example:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>My Site</title>
</head>
<body>
<?php
    // fetch 'content' key using layout helper:
    echo $this->layout()->content;

    // fetch 'foo' key using placeholder helper:
```

```
    echo $this->placeholder('Zend_Layout')->foo;

    // fetch layout object and retrieve various keys from it:
    $layout = $this->layout();
    echo $layout->bar;
    echo $layout->baz;
?>
</body>
</html>
```

Because `Zend_Layout` utilizes `Zend_View` for rendering, you can also use any view helpers registered, and also have access to any previously assigned view variables. Particularly useful are the various placeholder helpers, as they allow you to retrieve content for areas such as the <head> section, navigation, etc.:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <?= $this->headTitle() ?>
    <?= $this->headScript() ?>
    <?= $this->headStyle() ?>
</head>
<body>
    <?= $this->render('header.phtml') ?>

    <div id="nav"><?= $this->placeholder('nav') ?></div>

    <div id="content"><?= $this->layout()->content ?></div>

    <?= $this->render('footer.phtml') ?>
</body>
</html>
```

Using Zend_Layout with the Zend Framework MVC

`Zend_Controller` offers a rich set of functionality for extension via its front controller plugins and action controller helpers. `Zend_View` also has helpers. `Zend_Layout` takes advantage of these various extension points when used with the MVC components.

`Zend_Layout::startMvc()` creates an instance of `Zend_Layout` with any optional configuration you provide it. It then registers a front controller plugin that renders the layout with any application content once the dispatch loop is done, and registers an action helper to allow access to the layout object from your action controllers. Additionally, you may at any time grab the layout instance from within a view script using the `layout` view helper.

First, let's look at how to initialize Zend_Layout for use with the MVC:

```
// In your bootstrap:
Zend_Layout::startMvc();
```

`startMvc()` can take an optional array of options or `Zend_Config` object to customize the instance; these options are detailed in the section called “Zend_Layout Configuration Options”.

In an action controller, you may then access the layout instance as an action helper:

```
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // disable layouts for this action:
        $this->_helper->layout->disableLayout();
    }

    public function bazAction()
    {
        // use different layout script with this action:
        $this->_helper->layout->setLayout('foobaz');
    };
}
```

In your view scripts, you can then access the layout object via the `layout` view helper. This view helper is slightly different than others in that it takes no arguments, and returns an object instead of a string value. This allows you to immediately call methods on the layout object:

```
<?php $this->layout()->setLayout('foo'); // set alternate layout ?>
```

At any time, you can fetch the `Zend_Layout` instance registered with the MVC via the `getMvcInstance()` static method:

```
// Returns null if startMvc() has not first been called
$layout = Zend_Layout::getMvcInstance();
```

Finally, `Zend_Layout`'s front controller plugin has one important feature in addition to rendering the layout: it retrieves all named segments from the response object and assigns them as layout variables, assigning the 'default' segment to the variable 'content'. This allows you to access your application content and render it in your view scripts.

As an example, let's say your code first hits `FooController::indexAction()`, which renders some content to the default response segment, and then forwards to `NavController::menuAction()`, which renders content to the 'nav' response segment. Finally, you forward to `CommentControl-`

`ler::fetchAction()` and fetch some comments, but render those to the default response segment as well (which appends content to that segment). Your view script could then render each separately:

```
<body>
    <!-- renders /nav/menu -->
    <div id="nav"><?= $this->layout()->nav ?></div>

    <!-- renders /foo/index + /comment/fetch -->
    <div id="content"><?= $this->layout()->content ?></div>
</body>
```

This feature is particularly useful when used in conjunction with the ActionStack action helper and plugin, which you can use to setup a stack of actions through which to loop, and thus create widgetized pages.

Using Zend_Layout as a Standalone Component

As a standalone component, Zend_Layout does not offer nearly as many features or as much convenience as when used with the MVC. However, it still has two chief benefits:

- Scoping of layout variables.
- Isolation of layout view script from other view scripts.

When used as a standalone component, simply instantiate the layout object, use the various accessors to set state, set variables as object properties, and render the layout:

```
$layout = new Zend_Layout();

// Set a layout script path:
$layout->setLayoutPath('/path/to/layouts');

// set some variables:
$layout->content = $content;
$layout->nav     = $nav;

// choose a different layout script:
$layout->setLayout('foo');

// render final layout
echo $layout->render();
```

Sample Layout

Sometimes a picture is worth a thousand words. The following is a sample layout script showing how it might all come together.

The actual order of elements may vary, depending on the CSS you've setup; for instance, if you're using absolute positioning, you may be able to have the navigation displayed later in the document, but still show up at the top; the same could be said for the sidebar or header. The actual mechanics of pulling the content remain the same, however.

Zend_Layout Configuration Options

`Zend_Layout` has a variety of configuration options. These may be set by calling the appropriate accessors, passing an array or `Zend_Config` object to the constructor or `startMvc()`, passing an array of options to `setOptions()`, or passing a `Zend_Config` object to `setConfig()`.

- *layout*: the layout to use. Uses the current inflector to resolve the name provided to the appropriate layout view script. By default, this value is 'layout' and resolves to 'layout.phtml'. Accessors are `setLayout()` and `getLayout()`.
- *layoutPath*: the base path to layout view scripts. Accessors are `setLayoutPath()` and `getLayoutPath()`.
- *contentKey*: the layout variable used for default content (when used with the MVC). Default value is 'content'. Accessors are `setContentKey()` and `getContentKey()`.
- *mvcSuccessfulActionOnly*: when using the MVC, if an action throws an exception and this flag is true, the layout will not be rendered (this is to prevent double-rendering of the layout when the ErrorHandler plugin is in use). By default, the flat is true. Accessors are `setMvcSuccessfulActionOnly()` and `getMvcSuccessfulActionOnly()`.
- *view*: the view object to use when rendering. When used with the MVC, `Zend_Layout` will attempt to use the view object registered with the ViewRenderer if no view object has been passed to it explicitly. Accessors are `setView()` and `getView()`.
- *helperClass*: the action helper class to use when using `Zend_Layout` with the MVC components. By default, this is `Zend_Layout_Controller_Action_Helper_Layout`. Accessors are `setHelperClass()` and `getHelperClass()`.
- *pluginClass*: the front controller plugin class to use when using `Zend_Layout` with the MVC components. By default, this is `Zend_Layout_Controller_Plugin_Layout`. Accessors are `setPluginClass()` and `getPluginClass()`.
- *inflector*: the inflector to use when resolving layout names to layout view script paths; see the `Zend_Layout` inflector documentation for more details. Accessors are `setInflector()` and `getInflector()`.

helperClass and pluginClass must be passed to startMvc()

In order for the `helperClass` and `pluginClass` settings to have effect, they must be passed in as options to `startMvc()`; if set later, they have no affect.

Examples

The following examples assume the following `$options` array and `$config` object:

```
$options = array(
    'layout'     => 'foo',
    'layoutPath' => '/path/to/layouts',
    'contentKey' => 'CONTENT',           // ignored when MVC not used
);
```

```
/**
[layout]
layout = "foo"
layoutPath = "/path/to/layouts"
contentKey = "CONTENT"
*/
$config = new Zend_Config_Ini('/path/to/layout.ini', 'layout');
```

Example 24.1. Passing options to the constructor or startMvc()

Both the constructor and the `startMvc()` static method can accept either an array of options or a `Zend_Config` object with options in order to configure the `Zend_Layout` instance.

First, let's look at passing an array:

```
// Using constructor:
$layout = new Zend_Layout($options);

// Using startMvc():
$layout = Zend_Layout::startMvc($options);
```

And now using a config object:

```
$config = new Zend_Config_Ini('/path/to/layout.ini', 'layout');

// Using constructor:
$layout = new Zend_Layout($config);

// Using startMvc():
$layout = Zend_Layout::startMvc($config);
```

Basically, this is the easiest way to customize your `Zend_Layout` instance.

Example 24.2. Using setOption() and setConfig()

Sometimes you need to configure the `Zend_Layout` object after it has already been instantiated; `setOptions()` and `setConfig()` give you a quick and easy way to do so:

```
// Using an array of options:
$layout->setOptions($options);

// Using a Zend_Config object:
$layout->setConfig($options);
```

Note, however, that certain options, such as `pluginClass` and `helperClass`, will have no affect when passed using this method; they need to be passed to the constructor or `startMvc()` method.

Example 24.3. Using Accessors

Finally, you can also configure your `Zend_Layout` instance via accessors. All accessors implement a fluent interface, meaning their calls may be chained:

```
$layout->setLayout('foo')
       ->setLayoutPath('/path/to/layouts')
       ->setContentKey('CONTENT');
```

Zend_Layout Advanced Usage

`Zend_Layout` has a number of use cases for the advanced developer who wishes to adapt it for different view implementations, file system layouts, and more.

The major points of extension are:

- *Custom view objects.* `Zend_Layout` allows you to utilize any class that implements `Zend_View_Interface`.
- *Custom front controller plugins.* `Zend_Layout` ships with a standard front controller plugin that automates rendering of layouts prior to returning the response. You can substitute your own plugin.
- *Custom action helpers.* `Zend_Layout` ships with a standard action helper that should be suitable for most needs as it is a dumb proxy to the layout object itself.
- *Custom layout script path resolution*. `Zend_Layout` allows you to use your own inflector for layout script path resolution, or simply to modify the attached inflector to specify your own inflection rules.

Custom View Objects

`Zend_Layout` allows you to use any class implementing `Zend_View_Interface` or extending `Zend_View_Abstract` for rendering your layout script. Simply pass in your custom view object as a parameter to the constructor/`startMvc()`, or set it using the `setView()` accessor:

```
$view = new My_Custom_View();
$layout->setView($view);
```

Not all Zend_View implementations are equal

While `Zend_Layout` allows you to use any class implementing `Zend_View_Interface`, you may run into issues if they can not utilize the various `Zend_View` helpers, particularly the layout and placeholder helpers. This is because `Zend_Layout` makes variables set in the object available via itself and placeholders.

If you need to use a custom `Zend_View` implementation that does not support these helpers, you will need to find a way to get the layout variables to the view. This can be done by either extending the `Zend_Layout` object and altering the `render()` method to pass variables to the view, or creating your own plugin class that passes them prior to rendering the layout.

Alternately, if your view implementation supports any sort of plugin capability, you can access the variables via the 'Zend_Layout' placeholder, using the placeholder helper:

```
$placeholders = new Zend_View_Helper_Placeholder();
$layoutVars   = $placeholders->placeholder('Zend_Layout')->getArrayCopy();
```

Custom Front Controller Plugins

When used with the MVC components, `Zend_Layout` registers a front controller plugin that renders the layout as the last action prior to exiting the dispatch loop. In most cases, the default plugin will be suitable, but should you desire to write your own, you can specify the name of the plugin class to load by passing the `pluginClass` option to the `startMvc()` method.

Any plugin class you write for this purpose will need to extend `Zend_Controller_Plugin_Abstract`, and should accept a layout object instance as an argument to the constructor. Otherwise, the details of your implementation are up to you.

The default plugin class used is `Zend_Layout_Controller_Plugin_Layout`.

Custom Action Helpers

When used with the MVC components, `Zend_Layout` registers an action controller helper with the helper broker. The default helper, `Zend_Layout_Controller_Action_Helper_Layout`, acts as a dumb proxy to the layout object instance itself, and should be suitable for most use cases.

Should you feel the need to write custom functionality, simply write an action helper class extending `Zend_Controller_Action_Helper_Abstract` and pass the class name as the `helperClass` option to the `startMvc()` method. Details of the implementation are up to you.

Custom Layout Script Path Resolution: Using the Inflector

`Zend_Layout` uses `Zend_Filter_Inflector` to establish a filter chain for translating a layout name to a layout script path. By default, it uses the rules 'CamelCaseToDash' followed by 'StringToLower', and the suffix 'phtml' to transform the name to a path. As some examples:

- 'foo' will be transformed to 'foo.phtml'.
- 'FooBarBaz' will be transformed to 'foo-bar-baz.phtml'.

You have three options for modifying inflection: modify the inflection target and/or view suffix via `Zend_Layout` accessors, modify the inflector rules and target of the inflector associated with the `Zend_Layout` instance, or create your own inflector instance and pass it to `Zend_Layout::setInflector()`.

Example 24.4. Using Zend_Layout accessors to modify the inflector

The default `Zend_Layout` inflector uses static references for the target and view script suffix, and has accessors for setting these values.

```
// Set the inflector target:
$layout->setInflectorTarget('layouts/:script.:suffix');

// Set the layout view script suffix:
$layout->setViewSuffix('php');
```

Example 24.5. Direct modification of Zend_Layout inflector

Inflectors have a target and one or more rules. The default target used with `Zend_Layout` is ':script.:suffix'; ':script' is passed the registered layout name, while ':suffix' is a static rule of the inflector.

Let's say you want the layout script to end in the suffix 'html', and that you want to separate MixedCase and camelCased words with underscores instead of dashes, and not lowercase the name. Additionally, you want it to look in a 'layouts' subdirectory for the script.

```
$layout->getInflector()->setTarget('layouts/:script.:suffix')
                       ->setStaticRule('suffix', 'html')
                       ->setFilterRule(array('CamelCaseToUnderscore'));
```

Example 24.6. Custom inflectors

In most cases, modifying the existing inflector will be enough. However, you may have an inflector you wish to use in several places, with different objects of different types. `Zend_Layout` supports this.

```
$inflector = new Zend_Filter_Inflector('layouts/:script.:suffix');
$inflector->addRules(array(
    ':script' => array('CamelCaseToUnderscore'),
    'suffix'  => 'html'
));
$layout->setInflector($inflector);
```

Inflection can be disabled

Inflection can be disabled and enabled using accessors on the `Zend_Layout` object. This can be useful if you want to specify an absolute path for a layout view script, or know that the mechanism you will be using for specifying the layout script does not need inflection. Simply use the `enableInflection()` and `disableInflection()` methods.

Chapter 25. Zend_Ldap

Introduction

Minimal Functionality

Currently this class is designed only to satisfy the limited functionality necessary for the `Zend_Auth_Adapter_Ldap` authentication adapter. Operations such as searching, creating, modifying or renaming entries in the directory are currently not supported and will be defined at a later time.

`Zend_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 two classes, `Zend_Ldap` and `Zend_Ldap_Exception`. The `Zend_Ldap` class conceptually represents a binding to a single LDAP server. The parameters for binding may be provided explicitly or in the form of an options array.

Using the `Zend_Ldap` class depends on the type of LDAP server and is best summarized with some simple examples.

If you are using OpenLDAP, a simple example looks like the following (note that the `bindRequiresDn` option is important if you are *not* using AD):

```
$options = array(
    'host' => 's0.foo.net',
    'username' => 'CN=user1,DC=foo,DC=net',
    'password' => 'pass1',
    'bindRequiresDn' => true,
    'accountDomainName' => 'foo.net',
    'baseDn' => 'OU=Sales,DC=foo,DC=net',
);
$ldap = new Zend_Ldap($options);
$acctname = $ldap->getCanonicalAccountName('abaker',
                                            Zend_Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

If you are using Microsoft AD a simple example is:

```
$options = array(
    'host' => 'dc1.w.net',
    'useStartTls' => true,
    'username' => 'user1@w.net',
    'password' => 'pass1',
    'accountDomainName' => 'w.net',
    'accountDomainNameShort' => 'W',
    'baseDn' => 'CN=Users,DC=w,DC=net',
);
$ldap = new Zend_Ldap($options);
$acctname = $ldap->getCanonicalAccountName('bcarter',
                                           Zend_Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

Note that we use the `getCanonicalAccountName()` method to retrieve the account DN here only because that is what exercises the most of what little code is currently present in this class.

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` 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_Auth_Adapter_Ldap`, which passes the username supplied by the user directly to `bind()`.

The following example illustrates how the non-DN username '`abaker`' can be used with `bind()`:

```
$options = array(
        'host' => 's0.foo.net',
        'username' => 'CN=user1,DC=foo,DC=net',
        'password' => 'pass1',
        'bindRequiresDn' => true,
        'accountDomainName' => 'foo.net',
        'baseDn' => 'OU=Sales,DC=foo,DC=net',
);
$ldap = new Zend_Ldap($options);
$ldap->bind('abaker', 'moonbike55');
$acctname = $ldap->getCanonicalAccountName('abaker',
                                           Zend_Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

The `bind()` call in this example sees that the username '`abaker`' is not in DN form, finds `bindRequiresDn` is `true`, uses '`CN=user1,DC=foo,DC=net`' and '`pass1`' to bind, retrieves the DN for '`abaker`', unbinds and then rebinds with the newly discovered '`CN=Alice Baker,OU=Sales,DC=foo,DC=net`'.

Zend_Ldap Options

The `Zend_Ldap` component accepts an array of options either supplied to the constructor or through the `setOptions()` method. The permitted options are as follows:

Table 25.1. Zend_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.
password	The default credentials password (used only with username above).
bindRequiresDn	If `true`, this instructs `Zend_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 `printf()` [http://php.net/printf] style expression that must contain one '`%s`' to accomodate 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.

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 25.2. `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` component by itself makes no attempt to authenticate with multiple servers. However, `Zend_Ldap` is specifically designed to handle this scenario gracefully. The required technique is to simply iterate over an array of arrays of server 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 `W\bcarter` or `cdavis` - the `bind()` method will only succeed if the credentials were successfully used in the bind.

Consider the following example that illustrates the technique required to implement multi-domain authentication and failover:

```
$acctname = 'W\\user2';
$password = 'pass2';

$multiOptions = array(
    'server1' => array(
        'host' => 's0.foo.net',
        'username' => 'CN=user1,DC=foo,DC=net',
        'password' => 'pass1',
        'bindRequiresDn' => true,
        'accountDomainName' => 'foo.net',
        'accountDomainNameShort' => 'FOO',
        'accountCanonicalForm' => 4, // ACCT_FORM_PRINCIPAL
        'baseDn' => 'OU=Sales,DC=foo,DC=net',
    ),
    'server2' => array(
        'host' => 'dc1.w.net',
        'useSsl' => true,
        'username' => 'user1@w.net',
        'password' => 'pass1',
        'accountDomainName' => 'w.net',
        'accountDomainNameShort' => 'W',
        'accountCanonicalForm' => 4, // ACCT_FORM_PRINCIPAL
        'baseDn' => 'CN=Users,DC=w,DC=net',
    ),
);

$ldap = new Zend_Ldap();

foreach ($multiOptions as $name => $options) {

    echo "Trying to bind using server options for '$name'\n";

    $ldap->setOptions($options);
    try {
        $ldap->bind($acctname, $password);
        $acctname = $ldap->getCanonicalAccountName($acctname);
        echo "SUCCESS: authenticated $acctname\n";
        return;
    } catch (Zend_Ldap_Exception $zle) {
        echo '  ' . $zle->getMessage() . "\n";
        if ($zle->getCode() === Zend_Ldap_Exception::LDAP_X_DOMAIN_MISMATCH) {
            continue;
        }
    }
}
```

If the bind fails for any reason, the next set of server options is tried.

The `getCanonicalAccountName` call gets the canonical account name that the application would presumably use to associate data with such as preferences. The `accountCanonicalForm = 4` in all server options ensures that the canonical form is consistent regardless of which server was ultimately used.

The special `LDAP_X_DOMAIN_MISMATCH` exception occurs when an account name with a domain component was supplied (e.g., `abaker@foo.net` or `FOO\abaker` 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_Auth_Adapter_Ldap`. In fact, we recommend that you simply use that authentication adapter for multi-domain + failover LDAP based authentication (or copy the code).

Chapter 26. Zend_Loader

Loading Files and Classes Dynamically

The Zend_Loader class includes methods to help you load files dynamically.

Zend_Loader vs. require_once()

The `Zend_Loader` methods are best used if the filename you need to load is variable. For example, if it is based on a parameter from user input or method argument. If you are loading a file or a class whose name is constant, there is no benefit to using `Zend_Loader` over using traditional PHP functions such as `require_once()` [http://php.net/require_once].

Loading Files

The static method `Zend_Loader::loadFile()` loads a PHP file. The file loaded may contain any PHP code. The method is a wrapper for the PHP function `include()` [http://php.net/include]. This method throws `Zend_Exception` on failure, for example if the specified file does not exist.

Example 26.1. Example of loadFile() method

```
Zend_Loader::loadFile($filename, $dirs=null, $once=false);
```

The `$filename` argument specifies the filename to load, which must not contain any path information. A security check is performed on `$filename`. The `$filename` may only contain alphanumeric characters, dashes ("-"), underscores ("_"), or periods ("."). No such restriction is placed on the `$dirs` argument.

The `$dirs` argument specifies directories to search for the file. If `NULL`, only the `include_path` is searched. If a string or an array, the directory or directories specified will be searched, and then the `include_path`.

The `$once` argument is a boolean. If `TRUE`, `Zend_Loader::loadFile()` uses the PHP function `include_once()` [http://php.net/include] for loading the file, otherwise the PHP function `include()` [http://php.net/include_once] is used.

Loading Classes

The static method `Zend_Loader::loadClass($class, $dirs)` loads a PHP file and then checks for the existance of the class.

Example 26.2. Example of loadClass() method

```
Zend_Loader::loadClass('Container_Tree',
    array(
        '/home/production/mylib',
        '/home/production/myapp'
    )
);
```

The string specifying the class is converted to a relative path by substituting directory separates for underscores, and appending '.php'. In the example above, 'Container_Tree' becomes 'Container/Tree.php'.

If `$dirs` is a string or an array, `Zend_Loader::loadClass()` searches the directories in the order supplied. The first matching file is loaded. If the file does not exist in the specified `$dirs`, then the `include_path` for the PHP environment is searched.

If the file is not found or the class does not exist after the load, `Zend_Loader::loadClass()` throws a `Zend_Exception`.

`Zend_Loader::loadFile()` is used for loading, so the class name may only contain alphanumeric characters and the hyphen ('-'), underscore ('_'), and period ('.').

Testing if a File is Readable

The static method `Zend_Loader::isReadable($pathname)` returns `TRUE` if a file at the specified pathname exists and is readable, `FALSE` otherwise.

Example 26.3. Example of isReadable() method

```
if (Zend_Loader::isReadable($filename)) {
    // do something with $filename
}
```

The `$filename` argument specifies the filename to check. This may contain path information. This method is a wrapper for the PHP function `is_readable()` [http://php.net/is_readable]. The PHP function does not search the `include_path`, while `Zend_Loader::isReadable()` does.

Using the Autoloader

The `Zend_Loader` class contains a method you can register with the PHP SPL autoloader. `Zend_Loader::autoload()` is the callback method. As a convenience, `Zend_Loader` provides the `registerAutoload()` function register its `autoload()` method. If the `spl_autoload` extension is not present in your PHP environment, then `registerAutoload()` method throws a `Zend_Exception`.

Example 26.4. Example of registering the autoloader callback method

```
Zend_Loader::registerAutoload();
```

After registering the Zend Framework autoload callback, you can reference classes from the Zend Framework without having to load them explicitly. The `autoload()` method uses `Zend_Loader::loadClass()` automatically when you reference a class.

If you have extended the `Zend_Loader` class, you can give an optional argument to `registerAutoload()`, to specify the class from which to register an `autoload()` method.

Example 26.5. Example of registering the autoload callback method from an extended class

Because of the semantics of static function references in PHP, you must implement code for both `loadClass()` and `autoload()`, and the `autoload()` must call `self::loadClass()`. If your `autoload()` method delegates to its parent to call `self::loadClass()`, then it calls the method of that name in the parent class, not the subclass.

```
class My_Loader extends Zend_Loader
{
    public static function loadClass($class, $dirs = null)
    {
        parent::loadClass($class, $dirs);
    }

    public static function autoload($class)
    {
        try {
            self::loadClass($class);
            return $class;
        } catch (Exception $e) {
            return false;
        }
    }
}

Zend_Loader::registerAutoload('My_Loader');
```

You can remove an autoload callback. The `registerAutoload()` has an optional second argument, which is `true` by default. If this argument is `false`, the autoload callback in unregistered from the SPL autoload stack instead of registered.

Loading Plugins

A number of Zend Framework components are pluggable, and allow loading of dynamic functionality by specifying a class prefix and path to class files that are not necessarily on the `include_path` or do not necessarily follow traditional naming conventions. `Zend_Loader_PluginLoader` provides common functionality for this process.

The basic usage of the `PluginLoader` follows Zend Framework naming conventions of one class per file, using the underscore as a directory separator when resolving paths. It allows passing an optional class prefix to prepend when determining if a particular plugin class is loaded. Additionally, paths are searched in LIFO order. Due to the LIFO search and the class prefixes, this allows you to namespace your plugins, and thus override plugins from paths registered earlier.

Basic Use Case

First, let's assume the following directory structure and class files, and that the toplevel directory and library directory are on the include_path:

```
application/
    modules/
        foo/
            views/
                helpers/
                    FormLabel.php
                    FormSubmit.php
        bar/
            views/
                helpers/
                    FormSubmit.php
library/
    Zend/
        View/
            Helper/
                FormLabel.php
                FormSubmit.php
                FormText.php
```

Now, let's create a plugin loader to address the various view helper repositories available:

```
$loader = new Zend_Loader_PluginLoader();
$loader->addPrefixPath('Zend_View_Helper', 'Zend/View/Helper/')
       ->addPrefixPath('Foo_View_Helper',
                       'application/modules/foo/views/helpers')
       ->addPrefixPath('Bar_View_Helper',
                       'application/modules/bar/views/helpers');
```

We can then load a given view helper using just the portion of the class name following the prefixes as defined when adding the paths:

```
// load 'FormText' helper:
$formTextClass = $loader->load('FormText'); // 'Zend_View_Helper_FormText';
```

```
// load 'FormLabel' helper:
$formLabelClass = $loader->load('FormLabel'); // 'Foo_View_Helper_FormLabel'

// load 'FormSubmit' helper:
$formSubmitClass = $loader->load('FormSubmit'); // 'Bar_View_Helper_FormSubmit'
```

Once the class is loaded, we can now instantiate it.

Multiple paths may be registered for a given prefix

In some cases, you may use the same prefix for multiple paths. `Zend_Loader_PluginLoader` actually registers an array of paths for each given prefix; the last one registered will be the first one checked. This is particularly useful if you are utilizing incubator components.

Paths may be defined at instantiation

You may optionally provide an array of prefix / path pairs (or prefix / paths -- plural paths are allowed) as a parameter to the constructor:

```
$loader = new Zend_Loader_PluginLoader(array(
    'Zend_View_Helper' => 'Zend/View/Helper/',
    'Foo_View_Helper'  => 'application/modules/foo/views/helpers',
    'Bar_View_Helper'  => 'application/modules/bar/views/helpers'
));
```

`Zend_Loader_PluginLoader` also optionally allows you to share plugins across plugin-aware objects, without needing to utilize a singleton instance. It does so via a static registry. Indicate the registry name at instantiation as the second parameter to the constructor:

```
// Store plugins in static registry 'foobar':
$loader = new Zend_Loader_PluginLoader(array(), 'foobar');
```

Other components that instantiate the `PluginLoader` using the same registry name will then have access to already loaded paths and plugins.

Manipulating Plugin Paths

The example in the previous section shows how to add paths to a plugin loader. What if you want to determine the paths already loaded, or remove one or more?

- `getPaths($prefix = null)` returns all paths as prefix / path pairs if no `$prefix` is provided, or just the paths registered for a given prefix if a `$prefix` is present.
- `clearPaths($prefix = null)` will clear all registered paths by default, or only those associated with a given prefix, if the `$prefix` is provided and present in the stack.

- `removePrefixPath($prefix, $path = null)` allows you to selectively remove a specific path associated with a given prefix. If no `$path` is provided, all paths for that prefix are removed. If a `$path` is provided and exists for that prefix, only that path will be removed.

Testing for Plugins and Retrieving Class Names

Sometimes you simply want to determine if a plugin class has been loaded before you perform an action. `isLoaded()` takes a plugin name, and returns the status.

Another common use case for the `PluginLoader` is to determine fully qualified plugin class names of loaded classes; `getClassName()` provides this functionality. Typically, this would be used in conjunction with `isLoaded()`:

```
if ($loader->isLoaded('Adapter')) {
    $class   = $loader->getClassName('Adapter');
    $adapter = call_user_func(array($class, 'getInstance'));
}
```

Chapter 27. Zend_Locale

Introduction

`Zend_Locale` is the Frameworks answer to the question, "How can the same application be used around the whole world?" Most people will say, "That's easy. Let's translate all our output to several languages." However, using simple translation tables to map phrases from one language to another is not sufficient. Different regions will have different conventions for first names, surnames, salutory titles, formatting of numbers, dates, times, currencies, etc.

We need Localization [http://en.wikipedia.org/wiki/L10n] and complementary Internationalization [http://en.wikipedia.org/wiki/L10n] . Both are often abbreviated to `L10n` and `I18n`. Internationalization refers more to support for use of systems, regardless of special needs unique to groups of users related by language, region, number format conventions, financial conventions, time and date conventions, etc. Localization involves adding explicit support to systems for special needs of these unique groups, such as language translation, and support for local customs or conventions for communicating plurals, dates, times, currencies, names, symbols, sorting and ordering, etc. `L10n` and `I18n` compliment each other. The Zend Framework provides support for these through a combination of components, including Zend_Locale, Zend_Date, Zend_Measure, Zend_Translate, Zend_Currency, and Zend_TimeSync.

Zend_Locale and setLocale()

PHP's documentation [http://php.net/setlocale] states that `setlocale()` is not threadsave because it is maintained per process and not per thread. This means that, in multithreaded environments, you can have the problem that the locale changes while the script never has changed the locale itself. This can lead to unexpected behaviour when you use `setlocale()` in your scripts.

When you are using `Zend_Locale` you will not have this limitations, because `Zend_Locale` is not related to or coupled with PHP's `setlocale()`.

What is Localization

Localization means that an application (or homepage) can be used from different users which speak different languages. But as you already have expected Localization means more than only translating strings. It includes

- `Zend_Locale` - Backend support of locales available for localization support within other ZF components.
- `Zend_Translate` - Translating of strings.
- `Zend_Date` - Localization of dates, times.
- `Zend_Calendar` - Localization of calendars (support for non-Gregorian calendar systems)
- `Zend_Currency` - Localization of currencies.
- `Zend_Locale_Format` - Parsing and generating localized numbers.
- `Zend_Locale_Data` - Retrieve localized standard strings as country names, language names and more from the CLDR [http://unicode.org/cldr/] .

- TODO - Localization of collations

What is a Locale? [http://unicode.org/reports/tr35/#Locale]

Each computer user makes use of Locales, even when they don't know it. Applications lacking localization support, normally have implicit support for one particular locale (the locale of the author). When a class or function makes use of localization, we say it is locale-aware. How does the code know which localization the user is expecting?

A locale string or object identifying a supported locale gives Zend_Locale and it's subclasses access to information about the language and region expected by the user. Correct formatting, normalization, and conversions are made based on this information.

How are Locales Represented?

Locale identifiers consist of information about the user's language and preferred/primary geographic region (e.g. state or province of home or workplace). The locale identifier strings used in the Zend Framework are internationally defined standard abbreviations of language and region, written as language_REGION. Both the language and region parts are abbreviated to alphabetic, ASCII characters.

Note

Be aware that there exist not only locales with 2 characters as most people think. Also there are languages and regions which are not only abbreviated with 2 characters. Therefor you should NOT strip the region and language yourself, but use Zend_Locale when you want to strip language or region from a locale string. Otherwise you could have unexpected behaviour within your code when you do this yourself.

A user from USA would expect the language English and the region USA, yielding the locale identifier "en_US". A user in Germany would expect the language German and the region Germany, yielding the locale identifier "de_DE". See the list of pre-defined locale and region combinations [http://unicode.org/cldr/data/diff/supplemental/languages_and_territories.html] , if you need to select a specific locale within the Zend Framework.

Example 27.1. Choosing a specific locale

```
$locale = new Zend_Locale('de_DE'); // German language _ Germany
```

A German user in America might expect the language German and the region USA, but these non-standard mixes are not supported directly as recognized "locales". Instead, if an invalid combination is used, then it will automatically be truncated by dropping the region code. For example, "de_IS" would be truncated to "de", and "xh_RU" would be truncated to "xh", because neither of these combinations are valid. Additionally, if the base language code is not supported (e.g. "zz_US") or does not exist, then a default "root" locale will be used. The "root" locale has default definitions for internationally recognized representations of dates, times, numbers, currencies, etc. The truncation process depends on the requested information, since some combinations of language and region might be valid for one type of data (e.g. dates), but not for another (e.g. currency format).

Beware of historical changes, as ZF components do not know about or attempt to track the numerous timezone changes made over many years by many regions. For example, we can see a historical list [http://www.statoids.com/tus.html] showing dozens of changes made by governments to when and if a particular region observes Daylight Savings Time, and even which timezone a particular geographic area belongs. Thus, when performing date math, the math performed by ZF components will not adjust for these changes, but instead will give the correct time for the timezone using current, modern rules for DST and timezone assignment for geographic regions.

Selecting the Right Locale

For most situations, `new Zend_Locale()` will automatically select the correct locale, with preference given to information provided by the user's web browser. However, if `new Zend_Locale(Zend_Locale::ENVIRONMENT)` is used, then preference will be given to using the host server's environment configuration, as described below.

Example 27.2. Automatically selecting a locale

```
$locale  = new Zend_Locale();

// default behavior, same as above
$locale1 = new Zend_Locale(Zend_Locale::BROWSER);

// prefer settings on host server
$locale2 = new Zend_Locale(Zend_Locale::ENVIRONMENT);

// perfer framework app default settings
$locale3 = new Zend_Locale(Zend_Locale::FRAMEWORK);
```

The seach algorithm used by `Zend_Locale` for automatic selection of a locale uses three sources of information:

1. const `Zend_Locale::BROWSER` - The user's Web browser provides information with each request, which is published by PHP in the global variable `HTTP_ACCEPT_LANGUAGE`. If no matching locale can be found, then preference is given to `ENVIRONMENT` and lastly `FRAMEWORK`.
2. const `Zend_Locale::ENVIRONMENT` - PHP publishes the host server's locale via the PHP internal function `setlocale()`. If no matching locale can be found, then preference is given to FRAMEWORK and lastly BROWSER.
3. const `Zend_Locale::FRAMEWORK` - When the Zend Framework has a standardized way of specifying component defaults (planned, but not yet available), then using this constant during instantiation will give preference to choosing a locale based on these defaults. If no matching locale can be found, then preference is given to `ENVIRONMENT` and lastly `BROWSER`.

Usage of automatic Locales

`Zend_Locale` provides three additionally locales. These locales do not belong to any language or region. They are "automatic" locales which means that they have the same effect as the method `getDefault()` but without the negative effects like creating an instance. These "automatic" locales can be used anywhere, where also a standard locale and also the definition of a locale, it's string representation, can be used. This offers simplicity for situations like working with locales which are provided by a browser.

There are three locales which have a slightly different behaviour:

1. `'browser'` - `Zend_Locale` should work with the information which is provided by the user's Web browser. It is published by PHP in the global variable `HTTP_ACCEPT_LANGUAGE`.

   If a user provides more than one locale within his browser, `Zend_Locale` will use the first found locale. If the user does not provide a locale or the script is being called from the commandline the automatic locale `'environment'` will automatically be used and returned.

2. `'environment'` - `Zend_Locale` should work with the information which is provided by the host server. It is published by PHP via the internal function `setlocale()`.

   If a environment provides more than one locale, `Zend_Locale` will use the first found locale. If the host does not provide a locale the automatic locale `'browser'` will automatically be used and returned.

3. `'auto'` - `Zend_Locale` should automatically detect any locale which can be worked with. It will first search for a users locale and then, if not successfull, search for the host locale.

   If no locale can be detected, it will throw an exception and tell you that the automatical detection has been failed.

Example 27.3. Using automatic locales

```
// without automatic detection
//$locale = new Zend_Locale(Zend_Locale::BROWSER);
//$date = new Zend_Date($locale);

// with automatic detection
$date = new Zend_Date('auto');
```

Using a default Locale

In some environments it is not possible to detect a locale automatically. You can expect this behaviour when you get an request from commandline or the requesting browser has no language tag set and additionally your server has the default locale 'C' set or another properitary locale.

In such cases `Zend_Locale` will normally throw an exception with a message that the automatic detection of any locale was not successfull. You have two options to handle such a situation. Either through setting a new locale per hand, or defining a default locale.

Example 27.4. Handling locale exceptions

```
// within the bootstrap file
try {
    $locale = new Zend_Locale('auto');
} catch (Zend_Locale_Exception $e) {
    $locale = new Zend_Locale('de');
}

// within your model/controller
$date = new Zend_Date($locale);
```

But this has one big negative effect. You will have to set your locale object within every class using `Zend_Locale`. This could become very unhandy if you are using multiple classes.

Since Zend Framework Release 1.5 there is a much better way to handle this. You can set a default locale which the static `setDefault()` method. Of course, every unknown or not full qualified locale will also throw an exception. `setDefault()` should be the first call before you initiate any class using `Zend_Locale`. See the following example for details:

Example 27.5. Setting a default locale

```
// within the bootstrap file
Zend_Locale::setDefault('de');

// within your model/controller
$date = new Zend_Date();
```

In the case that no locale can be detected, automatically the locale **de** will be used. Otherwise, the detected locale will be used.

ZF Locale-Aware Classes

In the ZF, locale-aware classes rely on `Zend_Locale` to automatically select a locale, as explained above. For example, in a ZF web application, constructing a date using `Zend_Date` without specifying a locale results in an object with a locale based on information provided by the current user's web browser.

Example 27.6. Dates default to correct locale of web users

```
$date = new Zend_Date('2006',Zend_Date::YEAR);
```

To override this default behavior, and force locale-aware ZF components to use specific locales, regardless of the origin of your website visitors, explicitly specify a locale as the third argument to the constructor.

Example 27.7. Overriding default locale selection

```
$usLocale = new Zend_Locale('en_US');
$date = new Zend_Date('2006', Zend_Date::YEAR, $usLocale);
$temp = new Zend_Measure_Temperature('100,10',
                                     Zend_Measure::TEMPERATURE,
                                     $usLocale);
```

If you know many objects should all use the same default locale, explicitly specify the default locale to avoid the overhead of each object determining the default locale.

Example 27.8. Performance optimization when using a default locale

```
$locale = new Zend_Locale();
$date = new Zend_Date('2006', Zend_Date::YEAR, $locale);
$temp = new Zend_Measure_Temperature('100,10',
                                     Zend_Measure::TEMPERATURE,
                                     $locale);
```

Application wide locale

Zend Framework allows the usage of an application wide locale. You simply set an instance of `Zend_Locale` to the registry with the key 'Zend_Locale'. Then this instance will be used within all locale aware classes of Zend Framework. This way you set one locale within your registry and then you can forget about setting it again. It will automatically be used in all other classes. See the below example for the right usage:

Example 27.9. Usage of an application wide locale

```
// within your bootstrap
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// within your model or controller
$date = new Zend_Date();
// print $date->getLocale();
echo $date->getDate();
```

Zend_Locale_Format::setOptions(array $options)

The 'precision' option of a value is used to truncate or stretch extra digits. A value of '-1' disables modification of the number of digits in the fractional part of the value. The 'locale' option helps when parsing numbers and dates using separators and month names. The date format 'format_type' option selects between CLDR/ISO date format specifier tokens and PHP's date() tokens. The 'fix_date' option enables or disables heuristics that attempt to correct invalid dates. The 'number_format' option specifies a default number format for use with `toNumber()` (see the section called “Number localization”).

The 'date_format' option can be used to specify a default date format string, but beware of using getDate(), checkdateFormat() and getTime() after using setOptions() with a 'date_format'. To use these four methods with the default date format for a locale, use array('date_format' => null, 'locale' => $locale) for their options.

Example 27.10. Dates default to correct locale of web users

```
Zend_Locale_Format::setOptions(array('locale' => 'en_US',
                                     'fix_date' => true,
                                     'format_type' => 'php'));
```

For working with the standard definitions of a locale the option Zend_Locale_Format::STANDARD can be used. Setting the option Zend_Locale_Format::STANDARD for date_format uses the standard definitions from the actual set locale. Setting it for number_format uses the standard number format for this locale. And setting it for locale uses the standard locale for this environment or browser.

Example 27.11. Using STANDARD definitions for setOptions()

```
Zend_Locale_Format::setOptions(array('locale' => 'en_US',
                                     'date_format' => 'dd.MMMM.YYYY'));
// overriding the global set date format
$date = Zend_Locale_Format::getDate('2007-04-20,
                                    array('date_format' =>
                                              Zend_Locale_Format::STANDARD);

// global setting of the standard locale
Zend_Locale_Format::setOptions(array('locale' => Zend_Locale_Format::STANDARD,
                                     'date_format' => 'dd.MMMM.YYYY'));
```

Speed up Zend_Locale and its subclasses

`Zend_Locale` and its subclasses can be speed up by the usage of `Zend_Cache`. Use the static method `Zend_Locale::setCache($cache)` if you are using `Zend_Locale`. `Zend_Locale_Format` can be speed up the using the option `cache` within `Zend_Locale_Format::setOptions(array('cache' => $adapter));`. If you are using both classes you should only set the cache for `Zend_Locale`, otherwise the last set cache will overwrite the previous set cache. For convenience there is also static method `Zend_Locale::getCache()`.

Using Zend_Locale

`Zend_Locale` also provides localized information about locales for each locale, including localized names for other locales, days of the week, month names, etc.

Copying, Cloning, and Serializing Locale Objects

Use object cloning [http://php.net/language.oop5.cloning] to duplicate a locale object exactly and efficiently. Most locale-aware methods also accept string representations of locales, such as the result of `$locale->toString()`.

Example 27.12. clone

```
$locale = new Zend_Locale('ar');

// Save the $locale object as a serialization
$serializedLocale = $locale->serialize();
// re-create the original object
$localeObject = unserialize($serializedLocale);

// Obtain a string identification of the locale
$stringLocale = $locale->toString();

// Make a cloned copy of the $local object
$copiedLocale = clone $locale;

print "copied: ", $copiedLocale->toString();

// PHP automatically calls toString() via __toString()
print "copied: ", $copiedLocale;
```

Equality

`Zend_Locale` also provides a convenience function to compare two locales. All locale-aware classes should provide a similar equality check.

Example 27.13. Check for equal locales

```
$locale = new Zend_Locale();
$mylocale = new Zend_Locale('en_US');

// Check if locales are equal
if ($locale->equals($mylocale)) {
    print "Locales are equal";
}
```

Default locales

The method `getDefault()` returns an array of relevant locales using information from the user's web browser (if available), information from the environment of the host server, and ZF settings. As with the constructor for `Zend_Locale`, the first parameter selects a preference of which information to consider

(BROWSER, ENVIRONMENT, or FRAMEWORK) first. The second parameter toggles between returning all matching locales or only the first/best match. Locale-aware components normally use only the first locale. A quality rating is included, when available.

Example 27.14. Get default locales

```
$locale = new Zend_Locale();

// Return all default locales
$found = $locale->getDefault();
print_r($found);

// Return only browser locales
$found2 = $locale->getDefault(Zend_Locale::BROWSER,TRUE);
print_r($found2);
```

To obtain only the default locales relevent to the BROWSER, ENVIRONMENT, or FRAMEWORK , use the corresponding method:

- getEnvironment()
- getBrowser()
- getLocale()

Set a new locale

A new locale can be set with the function setLocale(). This function takes a locale string as parameter. If no locale is given, a locale is automatically selected . Since Zend_Locale objects are "light", this method exists primarily to cause side-effects for code that have references to the existing instance object.

Example 27.15. setLocale

```
$locale = new Zend_Locale();

// Actual locale
print $locale->toString();

// new locale
$locale->setLocale('aa_DJ');
print $locale->toString();
```

Getting the language and region

Use getLanguage() to obtain a string containing the two character language code from the string locale identifier. Use getRegion() to obtain a string containing the two character region code from the string locale identifier.

Example 27.16. getLanguage and getRegion

```
$locale = new Zend_Locale();

// if locale is 'de_AT' then 'de' will be returned as language
print $locale->getLanguage();

// if locale is 'de_AT' then 'AT' will be returned as region
print $locale->getRegion();
```

Obtaining localized strings

`getTranslationList()` gives you access to localized informations of several types. These information are useful if you want to display localized data to a customer without the need of translating it. They are already available for your usage.

The requested list of information is always returned as named array. If you want to give more than one value to a explicit type where you wish to receive values from, you have to give an array instead of multiple values.

Example 27.17. getTranslationList

```
$locale = new Zend_Locale('de_AT');
$list = $locale->getTranslationList('language');

print_r ($list);
// example key -> value pairs...
// [de] -> Deutsch
// [en] -> Englisch

// use one of the returned key as value for the getTranslation() method
// of another language
print $locale->getTranslation('de', 'language', 'zh');
// returns the translation for the language 'de' in chinese
```

You can receive this informations for all languages. But not all of the informations are completly available for all languages. Some of these types are also available through an own function for simplicity. See this list for detailed informations.

Table 27.1. Details for getTranslationList($type = null, $locale = null, $value = null)

Type	Description
Language	Returns a localized list of all languages. The language part of the locale is returned as key and the translation as value. For your convinience use the `getLanguageTranslationList()` method
Script	Returns a localized list of all scripts. The script is returned as key and the translation as value. For your convinience use the `getScriptTranslationList()` method
Territory	Returns a localized list of all territories. This contains countries, continents and territories. To get only territories and continents use '1' as value. To get only countries use '2' as value. The country part of the locale is used as key where applicable. In the other case the official ISO code for this territory is used. The translated territory is returned as value. For your convinience use the `getCountryTranslationList()` method to receive all countries and the `getTerritoryTranslationList()` method to receive all territories without countries. When you omit the value you will get a list with both.
Variant	Returns a localized list of known variants of scripts. The variant is returned as key and the translation as value
Key	Returns a localized list of known keys. This keys are generic values used in translation. These are normally calendar, collation and currency. The key is returned as array key and the translation as value
Type	Returns a localized list of known types of keys. These are variants of types of calendar representations and types of collations. When you use 'collation' as value you will get all types of collations returned. When you use 'calendar' as value you will get all types of calendars returned. When you omit the value you will get a list all both returned. The type is used as key and the translation as value
Layout	Returns a list of rules which describes how to format special text parts
Characters	Returns a list of allowed characters within this locale
Delimiters	Returns a list of allowed quoting characters for this locale
Measurement	Returns a list of known measurement values. This list is depreciated
Months	Returns a list of all month representations within this locale. There are several different represenations which are all returned as sub array. If you omit the value you will get a list of all months from the 'gregorian' calendar returned. You can give any known calendar as value to get a list of months from this calendar returned. Use Zend_Date for simplicity
Month	Returns a localized list of all month names for this locale. If you omit the value you will get the normally used gregorian full name of the months where each month number is used as key and the translated month is returned as value. You can get the months for different calendars and formats if you give an array as value. The first array entry has to be the calendar, the second the used context and the third the width to return. Use Zend_Date for simplicity
Days	Returns a list of all day representations within this locale. There are several different represenations which are all returned as sub array. If you omit the value you will get a list of all days from the 'gregorian' calendar returned. You can give any known calendar as value to get a list of days from this calendar returned. Use Zend_Date for simplicity

Type	Description
Day	Returns a localized list of all day names for this locale. If you omit the value you will get the normally used gregorian full name of the days where the english day abbreviation is used as key and the translated day is returned as value. You can get the days for different calendars and formats if you give an array as value. The first array entry has to be the calendar, the second the used context and the third the width to return. Use Zend_Date for simplicity
Week	Returns a list of values used for proper week calculations within a locale. Use Zend_Date for simplicity
Quarters	Returns a list of all quarter representations within this locale. There are several different represenations which are all returned as sub array. If you omit the value you will get a list of all quarters from the 'gregorian' calendar returned. You can give any known calendar as value to get a list of quarters from this calendar returned
Quarter	Returns a localized list of all quarter names for this locale. If you omit the value you will get the normally used gregorian full name of the quarters where each quarter number is used as key and the translated quarter is returned as value. You can get the quarters for different calendars and formats if you give an array as value. The first array entry has to be the calendar, the second the used context and the third the width to return
Eras	Returns a list of all era representations within this locale. If you omit the value you will get a list of all eras from the 'gregorian' calendar returned. You can give any known calendar as value to get a list of eras from this calendar returned
Era	Returns a localized list of all era names for this locale. If you omit the value you will get the normally used gregorian full name of the eras where each era number is used as key and the translated era is returned as value. You can get the eras for different calendars and formats if you give an array as value. The first array entry has to be the calendar and the second the width to return
Date	Returns a localized list of all date formats for this locale. The name of the dateformat is used as key and the format itself as value.If you omit the value you will get the date formats for the gregorian calendar returned. You can get the date formats for different calendars if you give the wished calendar as string. Use Zend_Date for simplicity
Time	Returns a localized list of all time formats for this locale. The name of the timeformat is used as key and the format itself as value. If you omit the value you will get the time formats for the gregorian calendar returned. You can get the time formats for different calendars if you give the wished calendar as string. Use Zend_Date for simplicity
DateTime	Returns a localized list of all known date-time formats for this locale. The name of the date-time format is used as key and the format itself as value. If you omit the value you will get the date-time formats for the gregorian calendar returned. You can get the date-time formats for different calendars if you give the wished calendar as string. Use Zend_Date for simplicity
Field	Returns a localized list of date fields which can be used to display calendars or date strings like 'month' or 'year' in a wished language. If you omit the value you will get this list for the gregorian calendar returned. You can get the list for different calendars if you give the wished calendar as string

Type	Description
Relative	Returns a localized list of relative dates which can be used to display textual relative dates like 'yesterday' or 'tomorrow' in a wished language. If you omit the value you will get this list for the gregorian calendar returned. You can get the list for different calendars if you give the wished calendar as string
Symbols	Returns a localized list of characters used for number representations
NameToCurrency	Returns a localized list of names for currencies. The currency is used as key and the translated name as value. Use Zend_Currency for simplicity
CurrencyToName	Returns a list of currencies for localized names. The translated name is used as key and the currency as value. Use Zend_Currency for simplicity
CurrencySymbol	Returns a list of known localized currency symbols for currencies. The currency is used as key and the symbol as value. Use Zend_Currency for simplicity
Question	Returns a list of localized strings for acceptance ('yes') and negotation ('no'). Use Zend_Locale's getQuestion method for simplicity
CurrencyFraction	Returns a list of fractions for currency values. The currency is used as key and the fraction as integer value. Use Zend_Currency for simplicity
CurrencyRounding	Returns a list of how to round which currency. The currency is used as key and the rounding as integer value. Use Zend_Currency for simplicity
CurrencyToRegion	Returns a list of currencies which are known to be used within a region. The ISO3166 value ('region') is used as array key and the ISO4217 value ('currency') as array value. Use Zend_Currency for simplicity
RegionToCurrency	Returns a list of regions where a currency is used . The ISO4217 value ('currency') is used as array key and the ISO3166 value ('region') as array value. When a currency is used in several regions these regions are seperated with a whitespace. Use Zend_Currency for simplicity
RegionToTerritory	Returns a list of territories with the countries or sub territories which are included within that territory. The ISO territory code ('territory') is used as array key and the ISO3166 value ('region') as array value. When a territory contains several regions these regions are seperated with a whitespace
TerritoryToRegion	Returns a list of regions and the territories where these regions are located. The ISO3166 code ('region') is used as array key and the ISO territory code ('territory') as array value. When a region is located in several territories these territories are seperated with a whitespace
ScriptToLanguage	Returns a list of scripts which are used within a language. The language code is used as array key and the script code as array value. When a language contains several scripts these scripts are seperated with a whitespace
LanguageToScript	Returns a list of languages which are using a script. The script code is used as array key and the language code as array value. When a script is used in several languages these languages are seperated with a whitespace
TerritoryToLanguage	Returns a list of countries which are using a language. The country code is used as array key and the language code as array value. When a language is used in several countries these countries are seperated with a whitespace
LanguageToTerritory	Returns a list of countries and the languages spoken within these countries. The country code is used as array key and the language code as array value. When a territory is using several languages these languages are seperated with a whitespace
TimezoneToWindows	Returns a list of windows timezones and the related ISO timezone. The windows timezone is used as array key and the ISO timezone as array value

Type	Description
WindowsToTimezone	Returns a list of ISO timezones and the related windows timezone. The ISO timezone is used as array key and the windows timezone as array value
TerritoryToTimezone	Returns a list of regions or territories and the related ISO timezone. The ISO timezone is used as array key and the territory code as array value
TimezoneToTerritory	Returns a list of timezones and the related region or territory code. The region or territory code is used as array key and the ISO timezone as array value
CityToTimezone	Returns a localized list of cities which can be used as translation for a related timezone. Not for all timezones is a translation available, but for a user is the real city written in his languages more accurate than the ISO name of this timezone. The ISO timezone is used as array key and the translated city as array value
TimezoneToCity	Returns a list of timezones for localized city names. The localized city is used as array key and the ISO timezone name as array value

If you are in need of a single translated value, you can use the `getTranslation()` method. It returns always a string but it accepts some different types than the `getTranslationList()` method. Also value is the same as before with one difference. You have to give the detail you want to get returned as additional value.

Note

Because you have almost always give a value as detail this parameter has to be given as first parameter. This differs from the `getTranslationList()` method.

See the following table for detailed information:

Table 27.2. Details for getTranslation($value = null, $type = null, $locale = null)

Type	Description
Language	Returns a translation for a language. To select the wished translation you must give the language code as value. For your convinience use the `getLanguageTranslation($value)` method
Script	Returns a translation for a script. To select the wished translation you must give the script code as value. For your convinience use the `getScriptTranslation($value)` method
Territory or **Country**	Returns a translation for a territory. This can be countries, continents and territories. To select the wished variant you must give the territory code as value. For your convinience use the `getCountryTranslation($value)` method.
Variant	Returns a translation for a script variant. To select the wished variant you must give the variant code as value
Key	Returns translation for a known keys. This keys are generic values used in translation. These are normally calendar, collation and currency. To select the wished key you must give the key code as value
DateChars	Returns a character table which contains all characters used when displaying dates
DefaultCalendar	Returns the default calendar for the given locale. For most locales this will be 'gregorian'. Use Zend_Date for simplicity
MonthContext	Returns the default context for months which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
DefaultMonth	Returns the default format for months which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Month	Returns a translation for a month. You have to give the number of the month as integer value. It has to be between 1 and 12. If you want to receive data for other calendars, contexts or formats, then you must give an array instead of an integer with the expected values. The array has to look like this: `array( 'calendar', 'context', 'format', 'month number')`. If you give only an integer then the default values are the 'gregorian' calendar, the context 'format' and the format 'wide'. Use Zend_Date for simplicity
DayContext	Returns the default context for ´days which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
DefaultDay	Returns the default format for days which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Day	Returns a translation for a day. You have to give the english abbreviation of the day as string value ('sun', 'mon', etc.). If you want to receive data for other calendars, contexts or format, then you must give an array instead of an integer with the expected values. The array has to look like this: `array('calendar', 'context', 'format', 'day abbreviation')`. If you give only an string then the default values are the 'gregorian' calendar, the context 'format' and the format 'wide'. Use Zend_Date for simplicity

Type	Description
Quarter	Returns a translation for a quarter. You have to give the number of the quarter as integer and it has to be between 1 and 4. If you want to receive data for other calendars, contexts or formats, then you must give an array instead of an integer with the expected values. The array has to look like this: `array('calendar', 'context', 'format', 'quarter number')`. If you give only an string then the default values are the 'gregorian' calendar, the context 'format' and the format 'wide'
Am	Returns a translation for 'AM' in a expected locale. If you want to receive data for other calendars an string with the expected calendar. If you omit the value then the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Pm	Returns a translation for 'PM' in a expected locale. If you want to receive data for other calendars an string with the expected calendar. If you omit the value then the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Era	Returns a translation for an era within a locale. You have to give the era number as string or integer. If you want to receive data for other calendars or formats, then you must give an array instead of the era number with the expected values. The array has to look like this: `array('calendar', 'format', 'era number')`. If you give only an string then the default values are the 'gregorian' calendar and the 'abbr' format
DefaultDate	Returns the default date format which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Date	Returns the date format for an given calendar or format within a locale. If you omit the value then the 'gregorian' calendar will be used with the 'medium' format. If you give a string then the 'gregorian' calendar will be used with the given format. Or you can also give an array which will have to look like this: `array('calendar', 'format')`. Use Zend_Date for simplicity
DefaultTime	Returns the default time format which is used within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Time	Returns the time format for an given calendar or format within a locale. If you omit the value then the 'gregorian' calendar will be used with the 'medium' format. If you give a string then the 'gregorian' calendar will be used with the given format. Or you can also give an array which will have to look like this: `array('calendar', 'format')`. Use Zend_Date for simplicity
DateTime	Returns the datetime format for the given locale which indicates how to display date with times in the same string within the given calendar. If you omit the value the 'gregorian' calendar will be used. Use Zend_Date for simplicity
Field	Returns a translated date field which can be used to display calendars or date strings like 'month' or 'year' in a wished language. You must give the field which has to be returned as string. In this case the 'gregorian' calendar will be used. You can get the field for other calendar formats if you give an array which has to look like this: `array('calendar', 'date field')`
Relative	Returns a translated date which is relative to today which can include date strings like 'yesterday' or 'tomorrow' in a wished language. You have to give the number of days relative to tomorrow to receive the expected string. Yesterday would be '-1', tomorrow '1' and so on. This will use the 'gregorian' calendar. If you want to get relative dates for other calendars you will have to give an array which has to look like this: `array('calendar', 'relative days')`. Use Zend_Date for simplicity

Type	Description
DecimalNumber	Returns the format for decimal numbers within a given locale. Use Zend_Locale_Format for simplicity
ScientificNumber	Returns the format for scientific numbers within a given locale
PercentNumber	Returns the format for percentage numbers within a given locale
CurrencyNumber	Returns the format for displaying currency numbers within a given locale. Use Zend_Currency for simplicity
NameToCurrency	Returns the translated name for a given currency. The currency has to be given in ISO format which is for example 'EUR' for the currency 'euro'. Use Zend_Currency for simplicity
CurrencyToName	Returns a currency for a given localized name. Use Zend_Currency for simplicity
CurrencySymbol	Returns the used symbol for a currency within a given locale. Not for all currencies exists a symbol. Use Zend_Currency for simplicity
Question	Returns a localized string for acceptance ('yes') and negotation ('no'). You have to give either 'yes' or 'no' as value to receive the expected string. Use Zend_Locale's getQuestion method for simplicity
CurrencyFraction	Returns the fraction to use for a given currency. You must give the currency as ISO value. Use Zend_Currency for simplicity
CurrencyRounding	Returns how to round a given currency. You must give the currency as ISO value. If you omit the currency then the 'DEFAULT' rounding will be returned. Use Zend_Currency for simplicity
CurrencyToRegion	Returns the currency for a given region. The region code has to be given as ISO3166 string for example 'AT' for austria. Use Zend_Currency for simplicity
RegionToCurrency	Returns the regions where a currency is used. The currency has to be given as ISO4217 code for example 'EUR' for euro. When a currency is used in multiple regions, these regions are seperated with a whitespace character. Use Zend_Currency for simplicity
RegionToTerritory	Returns the regions for a given territory. The territory has to be given as ISO4217 string for example '001' for world. The regions within this territory are seperated with a whitespace character
TerritoryToRegion	Returns the territories where a given region is located. The region has to be given in ISO3166 string for example 'AT' for austria. When a region is located in multiple territories then these territories are seperated with a whitespace character
ScriptToLanguage	Returns the scripts which are used within a given language. The language has to be given as ISO language code for example 'en' for english. When multiple scripts are used within a language then these scripts are seperated with a whitespace character
LanguageToScript	Returns the languages which are used within a given script. The script has to be given as ISO script code for example 'Latn' for latin. When a script is used in multiple languages then these languages are seperated with a whitespace character
TerritoryToLanguage	Returns the territories where a given language is used. The language has to be given as ISO language code for example 'en' for english. When multiple territories exist where this language is used then these territories are seperated with a whitespace character
LanguageToTerritory	Returns the languages which are used within a given territory. The territory has to be given as ISO3166 code for example 'IT' for italia. When a language is used in multiple territories then these territories are seperated with a whitespace character

Type	Description
TimezoneToWindows	Returns a ISO timezone for a given windows timezone
WindowsToTimezone	Returns a windows timezone for a given ISO timezone
TerritoryToTimezone	Returns the territory for a given ISO timezone
TimezoneToTerritory	Returns the ISO timezone for a given territory
CityToTimezone	Returns the localized city for a given ISO timezone. Not for all timezones does a city translation exist
TimezoneToCity	Returns the ISO timezone for a given localized city name. Not for all cities does a timezone exist

Note

With Zend Framework 1.5 several old types have been renamed. This has to be done because of several new types, some misspelling and to increase the usability. See this table for a list of old to new types:

Table 27.3. Differences between ZF 1.0 and ZF 1.5

Old type	New type
Country	Territory (with value '2')
Calendar	Type (with value 'calendar')
Month_Short	Month (with array('gregorian', 'format', 'abbreviated')
Month_Narrow	Month (with array('gregorian', 'stand-alone', 'narrow')
Month_Complete	Months
Day_Short	Day (with array('gregorian', 'format', 'abbreviated')
Day_Narrow	Day (with array('gregorian', 'stand-alone', 'narrow')
DateFormat	Date
TimeFormat	Time
Timezones	CityToTimezone
Currency	NameToCurrency
Currency_Sign	CurrencySymbol
Currency_Detail	CurrencyToRegion
Territory_Detail	TerritoryToRegion
Language_Detail	LanguageToTerritory

The example below demonstrates how to obtain the names of things in different languages.

Example 27.18. getTranslationList

```
$locale = new Zend_Locale('en_US');
// prints the names of all countries in German language
print_r($locale->getTranslationList('country', 'de'));
```

The next example shows how to find the name of a language in another language, when the two letter iso country code is not known.

Example 27.19. Converting country name in one language to another

```
require 'Zend/Locale.php';
$locale = new Zend_Locale('en_US');
$code2name = $locale->getLanguageTranslationList();
$name2code = array_flip($code2name);
$frenchCode = $name2code['French'];
echo $locale->getLanguageTranslation($frenchCode, 'de_AT');
// output is the German name of the French language
```

To gain some familiarity with what is available, try the example and examine the output.

Example 27.20. All available translations

```
// obtain a list of all the translation lists
$lists = $locale->getTranslationList();

// show all translation lists available (lots of output, all in English language)
foreach ($lists as $list) {
    echo "List $list = ";
    print_r($locale->getTranslationList($list));
}
```

To generate a list of all languages known by Zend_Locale, with each language name shown in its own language, try the example below in a web page. Similarly, `getCountryTranslationList()` and `getCountryTranslation()` could be used to create a table mapping your native language names for regions to the names of the regions shown in another language. Use a `try .. catch` block to handle exceptions that occur when using a locale that does not exist. Not all languages are also locales. In the example, below exceptions are ignored to prevent early termination.

Example 27.21. All Languages written in their native language

```
$sourceLanguage = null; // set to your native language code
$locale = new Zend_Locale($sourceLanguage);
$list = $locale->getLanguageTranslationList();

foreach($list as $language => $content) {
    try {
        $output = $locale->getLanguageTranslation($language, $language);
        if (is_string($output)) {
            print "\n<br>[".$language."] ".$output;
        }
    } catch (Exception $e) {
        continue;
    }
}
```

Obtaining translations for "yes" and "no"

Frequently, programs need to solicit a "yes" or "no" response from the user. Use `getQuestion()` to obtain an array containing the correct word(s) or regex strings to use for prompting the user in a particular $locale (defaults to the current object's locale). The returned array will contain the following informations :

- **yes and no**: A generic string representation for yes and no responses. This will contain the first and most generic response from yesarray and noarray.

  yesarray and noarray: An array with all known yes and no responses. Several languages have more than just two responses. In general this is the full string and it's abbreviation.

  yesexpr and noexpr: An generated regex which allows you to handle user response, and search for yes or no.

All of this informations are of course localized and depend on the set locale. See the following example for the informations you can receive:

Example 27.22. getQuestion()

```
$locale = new Zend_Locale();
// Question strings
print_r($locale->getQuestion('de'));

- - - Output - - -

Array
(
    [yes] => ja
    [no] => nein
    [yesarray] => Array
        (
            [0] => ja
            [1] => j
        )

    [noarray] => Array
        (
            [0] => nein
            [1] => n
        )

    [yesexpr] => ^([jJ][aA]?)|([jJ]?)
    [noexpr] => ^([nN]([eE][iI][nN])?)|([nN]?)
)
```

Note

Until 1.0.3 **yesabbr** from the underlaying locale data was also available. Since 1.5 this information is no longer standalone available, but you will find the information from it within **yesarray**.

Get a list of all known locales

Sometimes you will want to get a list of all known locales. This can be used for several tasks like the creation of a selectbox. For this purpose you can use the static `getLocaleList()` method which will return a list of all known locales.

Example 27.23. getLocaleList()

```
$localelist = Zend_Locale::getLocaleList();
```

Note

Note that the locales are returned as key of the array you will receive. The value is always a boolean true.

Normalization and Localization

`Zend_Locale_Format` is a internal component used by Zend_Locale. All locale aware classes use `Zend_Locale_Format` for normalization and localization of numbers and dates. Normalization involves parsing input from a variety of data respresentations, like dates, into a standardized, structured representation, such as a PHP array with year, month, and day elements.

The exact same string containing a number or a date might mean different things to people with different customs and conventions. Disambiguation of numbers and dates requires rules about how to interpret these strings and normalize the values into a standardized data structure. Thus, all methods in `Zend_Locale_Format` require a locale in order to parse the input data.

Default "root" Locale

If no locale is specified, then normalization and localization will use the standard "root" locale, which might yield unexpected behavior, if the input originated in a different locale, or output for a specific locale was expected.

Number normalization: getNumber($input, Array $options)

There are many number systems [http://en.wikipedia.org/wiki/Numeral] different from the common decimal system [http://en.wikipedia.org/wiki/Decimal] (e.g. "3.14"). Numbers can be normalized with the `getNumber()` function to obtain the standard decimal representation. For all number-related discussions in this manual, Arabic/European numerals (0,1,2,3,4,5,6,7,8,9) [http://en.wikipedia.org/wiki/Arabic_numerals] are implied, unless explicitly stated otherwise. The options array may contain a 'locale' to define grouping and decimal characters. The array may also have a 'precision' to truncate excess digits from the result.

Example 27.24. Number normalization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getNumber('13.524,678',
                                        array('locale' => $locale,
                                              'precision' => 3)
                                       );

print $number; // will return 13524.678
```

Precision and Calculations

Since `getNumber($value, array $options = array())` can normalize extremely large numbers, check the result carefully before using finite precision calculations, such as ordinary PHP math operations. For example, `if ((string)int_val($number) != $number) { use BCMath [http://www.php.net/bc] or GMP [http://www.php.net/gmp]` . Most PHP installations support the BCMath extension.

Also, the precision of the resulting decimal representation can be rounded to a desired length with `getNumber()` with the option `'precision'`. If no precision is given, no rounding occurs. Use only PHP integers to specify the precision.

If the resulting decimal representation should be truncated to a desired length instead of rounded the option `'number_format'` can be used instead. Define the length of the decimal representation with the desired length of zeros. The result will then not be rounded. So if the defined precision within `number_format` is zero the value "1.6" will return "1", not "2. See the example nearby:

Example 27.25. Number normalization with precision

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getNumber('13.524,678',
                                        array('precision' => 1,
                                              'locale' => $locale)
                                       );
print $number; // will return 13524.7

$number = Zend_Locale_Format::getNumber('13.524,678',
                                        array('number_format' => '#.00',
                                              'locale' => $locale)
                                       );
print $number; // will return 13524.67
```

Number localization

`toNumber($value, array $options = array())` can localize numbers to the following supported locales . This function will return a localized string of the given number in a conventional format for a specific locale. The 'number_format' option explicitly specifies a non-default number format for use with `toNumber()`.

Example 27.26. Number localization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toNumber(13547.36,
                                       array('locale' => $locale));

// will return 13.547,36
print $number;
```

Unlimited length

`toNumber()` can localize numbers with unlimited length. It is not related to integer or float limitations.

The same way as within `getNumber()`, `toNumber()` handles precision. If no precision is given, the complete localized number will be returned.

Example 27.27. Number localization with precision

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toNumber(13547.3678, array('precision' => 2, 'locale

// will return 13.547,37
print $number;
```

Using the option 'number_format' a self defined format for generating a number can be defined. The format itself has to be given in CLDR format as described below. The locale is used to get seperation, precission and other number formatting signs from it. German for example defines ',' as precission seperation and in english the '.' sign is used.

Table 27.4. Format tokens for self generated number formats

Token	Description	Example format	Generated output
#0	Generates a number without precission and seperation	#0	1234567
,	Generates a seperation with the length from seperation to next seperation or to 0	#,##0	1,234,567
#,##,##0	Generates a standard seperation of 3 and all following seperations with 2	#,##,##0	12,34,567
.	Generates a precission	#0.#	1234567.1234
0	Generates a precission with a defined length	#0.00	1234567.12

Example 27.28. Using a self defined number format

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toNumber(13547.3678,
                                       array('number_format' => '#,#0.00',
                                             'locale' => 'de')
                                      );

// will return 1.35.47,36
print $number;

$number = Zend_Locale_Format::toNumber(13547.3,
                                       array('number_format' => '#,##0.00',
                                             'locale' => 'de')
                                       );

// will return 13.547,30
print $number;
```

Number testing

`isNumber($value, array $options = array())` checks if a given string is a number and returns true or false.

Example 27.29. Number testing

```
$locale = new Zend_Locale();
if (Zend_Locale_Format::isNumber('13.445,36', array('locale' => 'de_AT')) {
    print "Number";
} else {
    print "not a Number";
}
```

Float value normalization

Floating point values can be parsed with the `getFloat($value, array $options = array())` function. A floating point value will be returned.

Example 27.30. Floating point value normalization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getFloat('13.524,678',
                                       array('precision' => 2,
                                             'locale' => $locale)
                                      );

// will return 13524.68
print $number;
```

Floating point value localization

`toFloat()` can localize floating point values. This function will return a localized string of the given number.

Example 27.31. Floating point value localization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toFloat(13547.3655,
                                      array('precision' => 1,
                                            'locale' => $locale)
                                     );

// will return 13.547,4
print $number;
```

Floating point value testing

`isFloat($value, array $options = array())` checks if a given string is a floating point value and returns true or false.

Example 27.32. Floating point value testing

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::isFloat('13.445,36', array('locale' => $locale)) {
    print "float";
} else {
    print "not a float";
}
```

Integer value normalization

Integer values can be parsed with the `getInteger()` function. A integer value will be returned.

Example 27.33. Integer value normalization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::getInteger('13.524,678',
                                         array('locale' => $locale));

// will return 13524
print $number;
```

Integer point value localization

`toInteger($value, array $options = array())` can localize integer values. This function will return a localized string of the given number.

Example 27.34. Integer value localization

```
$locale = new Zend_Locale('de_AT');
$number = Zend_Locale_Format::toInteger(13547.3655,
                                        array('locale' => $locale));

// will return 13.547
print $number;
```

Integer value testing

`isInteger($value, array $options = array())` checks if a given string is a integer value and returns true or false.

Example 27.35. Integer value testing

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::isInteger('13.445', array('locale' => $locale)) {
    print "integer";
} else {
    print "not a integer";
}
```

Numeral System Conversion

`Zend_Locale_Format::convertNumerals()` converts digits between different numeral systems [http://en.wikipedia.org/wiki/Arabic_numerals] , including the standard Arabic/European/Latin numeral system (0,1,2,3,4,5,6,7,8,9), not to be confused with Eastern Arabic numerals [http://en.wikipedia.org/wiki/Eastern_Arabic_numerals] sometimes used with the Arabic language to express numerals. Attempts to use an unsupported numeral system will result in an exception, to avoid accidentally performing an incorrect conversion due to a spelling error. All characters in the input, which are not numerals for the selected numeral system, are copied to the output with no conversion provided for unit separator characters. `Zend_Locale*` components rely on the data provided by CLDR (see their list of scripts grouped by language [http://unicode.org/cldr/data/diff/supplemental/languages_and_scripts.html?sortby=date]).

In CLDR and hereafter, the Europena/Latin numerals will be referred to as "Latin" or by the assigned 4-letter code "Latn". Also, the CLDR refers to this numeral systems as "scripts".

Suppose a web form collected a numeric input expressed using Eastern Arabic digits " ". Most software and PHP functions expect input using Arabic numerals. Fortunately, converting this input to it's equivalent Latin numerals "100" requires little effort using `convertNumerals($inputNumeralString, $sourceNumeralSystem, $destNumeralSystem)` , which returns the `$input` with numerals in the script `$sourceNumeralSystem` converted to the script `$destNumeralSystem`.

Example 27.36. Converting numerals from Eastern Arabic scripts to European/Latin scripts

```
$arabicScript = "    ";   // Arabic for "100" (one hundred)
$latinScript = Zend_Locale_Format::convertNumerals($arabicScript,
                                                   'Arab',
                                                   'Latn');

print "\nOriginal:   " . $arabicScript;
print "\nNormalized: " . $latinScript;
```

Similarly, any of the supported numeral systems may be converted to any other supported numeral system.

Example 27.37. Converting numerals from Latin script to Eastern Arabic script

```
$latinScript = '123';
$arabicScript = Zend_Locale_Format::convertNumerals($latinScript,
                                                    'Latn',
                                                    'Arab');

print "\nOriginal:  " . $latinScript;
print "\nLocalized: " . $arabicScript;
```

Example 27.38. Getting 4 letter CLDR script code using a native-language name of the script

```
function getScriptCode($scriptName, $locale)
{
    $scripts2names = Zend_Locale_Data::getList($locale, 'script');
    $names2scripts = array_flip($scripts2names);
    return $names2scripts[$scriptName];
}
echo getScriptCode('Latin', 'en'); // outputs "Latn"
echo getScriptCode('Tamil', 'en'); // outputs "Taml"
echo getScriptCode('tamoul', 'fr'); // outputs "Taml"
```

List of supported numeral systems

Table 27.5. List of supported numeral systems

Notation Name	Script
Arabic	Arab
Balinese	Bali
Bengali	Beng
Devanagari	Deva
Gujarati	Gujr
Gurmukhi	Guru
Kannada	Knda
Khmer	Khmr
Lao	Laoo
Limbu	Limb
Malayalam	Mlym
Mongolian	Mong
Myanmar	Mymr
New_Tai_Lue	Talu
Nko	Nkoo
Oriya	Orya
Tamil	Taml
Telugu	Telu
Thai	Tale
Tibetan	Tibt

Working with Dates and Times

`Zend_Locale_Format` provides several methods for working with dates and times to help convert and normalize between different formats for different locales. Use `Zend_Date` for manipulating dates, and working with date strings that already conform to one of the many internationally recognized standard formats, or one of the localized date formats supported by `Zend_Date` . Using an existing, pre-defined format offers advantages, including the use of well-tested code, and the assurance of some degree of portability and interoperability (depending on the standard used). The examples below do not follow these recommendations, since using non-standard date formats would needlessly increase the difficulty of understanding these examples.

Normalizing Dates and Times

The `getDate()` method parses strings containing dates in localized formats. The results are returned in a structured array, with well-defined keys for each part of the date. In addition, the array will contain a key 'date_format' showing the format string used to parse the input date string. Since a localized date string may not contain all parts of a date/time, the key-value pairs are optional. For example, if only the year, month, and day is given, then all time values are supressed from the returned array, and vice-versa if only hour, minute, and second were given as input. If no date or time can be found within the given input, an exception will be thrown.

If `setOption(array('fix_date' => true))` is set the `getDate()` method adds a key 'fixed' with a whole number value indicating if the input date string required "fixing" by rearranging the day, month, or year in the input to fit the format used.

Table 27.6. Key values for getDate() with option 'fix_date'

value	meaning
0	nothing to fix
1	fixed false month
2	swapped day and year
3	swapped month and year
4	swapped month and day

For those needing to specify explicitly the format of the date string, the following format token specifiers are supported. If an invalid format specifier is used, such as the PHP 'i' specifier when in ISO format mode, then an error will be thrown by the methods in Zend_Locale_Format that support user-defined formats.

These specifiers (below) are a small subset of the full "ISO" set supported by Zend_Date's `toString()`. If you need to use PHP `date()` compatible format specifiers, then first call `setOptions(array('format_type' => 'php'))`. And if you want to convert only one special format string from PHP `date()` compatible format to "ISO" format use `convertPhpToIsoFormat()`. Currently, the only practical difference relates to the specifier for minutes ('m' using the ISO default, and 'i' using the PHP date format).

Table 27.7. Return values

getDate() format character	Array key	Returned value	Minimum	Maximum
d	day	integer	1	31
M	month	integer	1	12
y	year	integer	no limit	PHP integer's maximum
h	hour	integer	0	PHP integer's maximum
m	minute	integer	0	PHP integer's maximum
s	second	integer	0	PHP integer's maximum

Example 27.39. Normalizing a date

```
$dateString = Zend_Locale_Format::getDate('13.04.2006',
                                          array('date_format' =>
                                                    'dd.MM.yyyy')
                                         );

// creates a Zend_Date object for this date
$dateObject = Zend_Date('13.04.2006',
                         array('date_format' => 'dd.MM.yyyy'));

print_r($dateString); // outputs:

Array
(
    [format] => dd.MM.yyyy
    [day] => 13
    [month] => 4
    [year] => 2006
)

// alternatively, some types of problems with input data can be automatically corr
$date2 = Zend_Locale_Format::getDate('04.13.2006',
                                     array('date_format' => 'dd.MM.yyyy',
                                           'fix_date' => true)
                                    );

print_r($date); // outputs:

Array
(
    [format] => dd.MM.yyyy
    [day] => 13
    [month] => 4
    [year] => 2006
    [fixed] => 4
)
```

Since `getDate()` is "locale-aware", specifying the `$locale` is sufficient for date strings adhering to that locale's format. The option '`fix_date`' uses simple tests to determine if the day or month is not valid, and then applies heuristics to try and correct any detected problems. Note the use of '`Zend_Locale_Format::STANDARD`' as the value for '`date_format`' to prevent the use of a class-wide default date format set using `setOptions()`. This forces getDate to use the default date format for `$locale`.

Example 27.40. Normalizing a date by locale

```
$locale = new Zend_Locale('de_AT');
$date = Zend_Locale_Format::getDate('13.04.2006',
                                    array('date_format' =>
                                              Zend_Locale_Format::STANDARD,
                                          'locale' => $locale)
                                   );

print_r ($date);
```

A complete date and time is returned when the input contains both a date and time in the expected format.

Example 27.41. Normalizing a date with time

```
$locale = new Zend_Locale('de_AT');
$date = Zend_Locale_Format::getDate('13.04.2005 22:14:55', array('date_format' =>

print_r ($date);
```

If a specific format is desired, specify the `$format` argument, without giving a `$locale`. Only single-letter codes (H, m, s, y, M, d), and MMMM and EEEE are supported in the `$format`.

Example 27.42. Normalizing a userdefined date

```
$date = Zend_Locale_Format::getDate('13200504T551422',
                                    array('date_format' =>
                                              'ddyyyyMM ssmmHH')
                                   );

print_r ($date);
```

The format can include the following signs :

Table 27.8. Format definition

Format Letter	Description
d or dd	1 or 2 digit day
M or MM	1 or 2 digit month
y or yy	1 or 2 digit year
yyyy	4 digit year
h	1 or 2 digit hour
m	1 or 2 digit minute
s	1 or 2 digit second

Examples for proper formats are

Table 27.9. Example formats

Formats	Input	Output
dd.MM.yy	1.4.6	['day'] => 1, ['month'] => 4, ['year'] => 6
dd.MM.yy	01.04.2006	['day'] => 1, ['month'] => 4, ['year'] => 2006
yyyyMMdd	1.4.6	['day'] => 6, ['month'] => 4, ['year'] => 1

Database date format

To parse a database date value (f.e. MySql or MsSql), use Zend_Date's ISO_8601 format instead of getDate().

The option '`fix_date`' uses simple tests to determine if the day or month is not valid, and then applies heuristics to try and correct any detected problems. `getDate()` automatically detects and corrects some kinds of problems with input, such as misplacing the year:

Example 27.43. Automatic correction of input dates

```
$date = Zend_Locale_Format::getDate('41.10.20',
                                    array('date_format' => 'ddMMyy',
                                          'fix_date' => true)
                                   );

// instead of 41 for the day, the 41 will be returned as year value
print_r ($date);
```

Testing Dates

Use `checkDateFormat($inputString, array('date_format' => $format, $locale))` to check if a given string contains all expected date parts. The `checkDateFormat()` method uses `getDate()`, but without the option `'fixdate'` to avoid returning true when the input fails to conform to the date format. If errors are detected in the input, such as swapped values for months and days, the option `'fixdate'` method will apply heuristics to "correct" dates before determining their validity.

Example 27.44. Date testing

```
$locale = new Zend_Locale('de_AT');
// using the default date format for 'de_AT', is this a valid date?
if (Zend_Locale_Format::checkDateFormat('13.Apr.2006',
                                        array('date_format' =>
                                                  Zend_Locale_Format::STANDARD,
                                              $locale)
                                       ) {
    print "date";
} else {
    print "not a date";
}
```

Normalizing a Time

Normally, a time will be returned with a date, if the input contains both. If the proper format is not known, but the locale relevant to the user input is known, then `getTime()` should be used, because it uses the default time format for the selected locale.

Example 27.45. Normalize an unknown time

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::getTime('13:44:42',
                                array('date_format' =>
                                          Zend_Locale_Format::STANDARD,
                                      'locale' => $locale)) {
    print "time";
} else {
    print "not a time";
}
```

Testing Times

Use `checkDateFormat()` to check if a given string contains a proper time. The usage is exact the same as with checking Dates, only `date_format` should contain the parts which you expect to have.

Example 27.46. Testing a time

```
$locale = new Zend_Locale('de_AT');
if (Zend_Locale_Format::checkDateFormat('13:44:42',
                                        array('date_format' => 'HH:mm:ss',
                                              'locale' => $locale)) {
    print "time";
} else {
    print "not a time";
}
```

Supported locales

`Zend_Locale` provides information on several locales. The following table shows all languages and their related locales, sorted by language:

Table 27.10. List of all supported languages

Language	Locale	Region
Afar	aa	---
	aa_DJ	Djibouti
	aa_ER	Eritrea
	aa_ET	Ethiopia
Afrikaans	af	---
	af_NA	Namibia
	af_ZA	South Africa
Akan	ak	---
	ak_GH	Ghana
Amharic	am	---
	am_ET	Ethiopia
Arabic	ar	---
	ar_AE	United Arab Emirates
	ar_BH	Bahrain
	ar_DZ	Algeria
	ar_EG	Egypt
	ar_IQ	Iraq
	ar_JO	Jordan
	ar_KW	Kuwait
	ar_LB	Lebanon
	ar_LY	Libya
	ar_MA	Morocco
	ar_OM	Oman
	ar_QA	Qatar
	ar_SA	Saudi Arabia
	ar_SD	Sudan
	ar_SY	Syria
	ar_TN	Tunisia
	ar_YE	Yemen
Assamese	as	---
	as_IN	India
Azerbaijani	az	---
	az_AZ	Azerbaijan
Belarusian	be	---
	be_BY	Belarus
Bulgarian	bg	---
	bg_BG	Bulgaria

Language	Locale	Region
Bengali	bn	---
	bn_BD	Bangladesh
	bn_IN	India
Bosnian	bs	---
	bs_BA	Bosnia and Herzegovina
Blin	byn	---
	byn_ER	Eritrea
Catalan	ca	---
	ca_ES	Spain
Atsam	cch	---
	cch_NG	Nigeria
Coptic	cop	---
Czech	cs	---
	cs_CZ	Czech Republic
Welsh	cy	---
	cy_GB	United Kingdom
Danish	da	---
	da_DK	Denmark
German	de	---
	de_AT	Austria
	de_BE	Belgium
	de_CH	Switzerland
	de_DE	Germany
	de_LI	Liechtenstein
	de_LU	Luxembourg
Divehi	dv	---
	dv_MV	Maldives
Dzongkha	dz	---
	dz_BT	Bhutan
Ewe	ee	---
	ee_GH	Ghana
	ee_TG	Togo
Greek	el	---
	el_CY	Cyprus
	el_GR	Greece

Language	**Locale**	**Region**
English	en	---
	en_AS	American Samoa
	en_AU	Australia
	en_BE	Belgium
	en_BW	Botswana
	en_BZ	Belize
	en_CA	Canada
	en_GB	United Kingdom
	en_GU	Guam
	en_HK	Hong Kong
	en_IE	Ireland
	en_IN	India
	en_JM	Jamaica
	en_MH	Marshall Islands
	en_MP	Northern Mariana Islands
	en_MT	Malta
	en_NA	Namibia
	en_NZ	New Zealand
	en_PH	Philippines
	en_PK	Pakistan
	en_SG	Singapore
	en_TT	Trinidad and Tobago
	en_UM	United States Minor Outlying Islands
	en_US	United States
	en_VI	U.S. Virgin Islands
	en_ZA	South Africa
	en_ZW	Zimbabwe
Esperanto	eo	---

Language	**Locale**	**Region**
Spanish	es	---
	es_AR	Argentina
	es_BO	Bolivia
	es_CL	Chile
	es_CO	Colombia
	es_CR	Costa Rica
	es_DO	Dominican Republic
	es_EC	Ecuador
	es_ES	Spain
	es_GT	Guatemala
	es_HN	Honduras
	es_MX	Mexico
	es_NI	Nicaragua
	es_PA	Panama
	es_PE	Peru
	es_PR	Puerto Rico
	es_PY	Paraguay
	es_SV	El Salvador
	es_US	United States
	es_UY	Uruguay
	es_VE	Venezuela
Estonian	et	---
	et_EE	Estonia
Basque	eu	---
	eu_ES	Spain
Persian	fa	---
	fa_AF	Afghanistan
	fa_IR	Iran
Finnish	fi	---
	fi_FI	Finland
Filipino	fil	---
	fil_PH	Philippines
Faroese	fo	---
	fo_FO	Faroe Islands

Language	**Locale**	**Region**
French	fr	---
	fr_BE	Belgium
	fr_CA	Canada
	fr_CH	Switzerland
	fr_FR	France
	fr_LU	Luxembourg
	fr_MC	Monaco
	fr_SN	Senegal
Friulian	fur	---
	fur_IT	Italy
Irish	ga	---
	ga_IE	Ireland
Ga	gaa	---
	gaa_GH	Ghana
Geez	gez	---
	gez_ER	Eritrea
	gez_ET	Ethiopia
Gallegan	gl	---
	gl_ES	Spain
Gujarati	gu	---
	gu_IN	India
Manx	gv	---
	gv_GB	United Kingdom
Hausa	ha	---
	ha_GH	Ghana
	ha_NE	Niger
	ha_NG	Nigeria
	ha_SD	Sudan
Hawaiian	haw	---
	haw_US	United States
Hebrew	he	---
	he_IL	Israel
Hindi	hi	---
	hi_IN	India
Croatian	hr	---
	hr_HR	Croatia
Hungarian	hu	---
	hu_HU	Hungary

Language	**Locale**	**Region**
Armenian	hy	---
Interlingua	ia	---
Indonesian	id	---
	id_ID	Indonesia
Igbo	ig	---
	ig_NG	Nigeria
Sichuan Yi	ii	---
	ii_CN	China
Indonesian	in	---
Icelandic	is	---
	is_IS	Iceland
Italian	it	---
	it_CH	Switzerland
	it_IT	Italy
Inuktitut	iu	---
Hebrew	iw	---
Japanese	ja	---
	ja_JP	Japan
Georgian	ka	---
	ka_GE	Georgia
Jju	kaj	---
	kaj_NG	Nigeria
Kamba	kam	---
	kam_KE	Kenya
Tyap	kcg	---
	kcg_NG	Nigeria
Koro	kfo	---
	kfo_CI	Ivory Coast
Kazakh	kk	---
	kk_KZ	Kazakhstan
Kalaallisut	kl	---
	kl_GL	Greenland
Khmer	km	---
	km_KH	Cambodia
Kannada	kn	---
	kn_IN	India
Korean	ko	---
	ko_KR	South Korea

Language	**Locale**	**Region**
Konkani	kok	---
	kok_IN	India
Kpelle	kpe	---
	kpe_GN	Guinea
	kpe_LR	Liberia
Kurdish	ku	---
	ku_TR	Turkey
Cornish	kw	---
	kw_GB	United Kingdom
Kirghiz	ky	---
	ky_KG	Kyrgyzstan
Lingala	ln	---
	ln_CD	Congo - Kinshasa
	ln_CG	Congo - Brazzaville
Lao	lo	---
	lo_LA	Laos
Lithuanian	lt	---
	lt_LT	Lithuania
Latvian	lv	---
	lv_LV	Latvia
Macedonian	mk	---
	mk_MK	Macedonia
Malayalam	ml	---
	ml_IN	India
Mongolian	mn	---
	mn_CN	China
	mn_MN	Mongolia
Romanian	mo	---
Marathi	mr	---
	mr_IN	India
Malay	ms	---
	ms_BN	Brunei
	ms_MY	Malaysia
Maltese	mt	---
	mt_MT	Malta
Burmese	my	---
	my_MM	Myanmar

Language	**Locale**	**Region**
Norwegian Bokmal	nb	---
	nb_NO	Norway
Nepali	ne	---
	ne_IN	India
	ne_NP	Nepal
Dutch	nl	---
	nl_BE	Belgium
	nl_NL	Netherlands
Norwegian Nynorsk	nn	---
	nn_NO	Norway
Norwegian	no	---
South Ndebele	nr	---
	nr_ZA	South Africa
Northern Sotho	nso	---
	nso_ZA	South Africa
Nyanja	ny	---
	ny_MW	Malawi
Oromo	om	---
	om_ET	Ethiopia
	om_KE	Kenya
Oriya	or	---
	or_IN	India
Punjabi	pa	---
	pa_IN	India
	pa_PK	Pakistan
Polish	pl	---
	pl_PL	Poland
Pashto	ps	---
	ps_AF	Afghanistan
Portuguese	pt	---
	pt_BR	Brazil
	pt_PT	Portugal
Romanian	ro	---
	ro_MD	Moldova
	ro_RO	Romania
Russian	ru	---
	ru_RU	Russia
	ru_UA	Ukraine

Language	Locale	Region
Kinyarwanda	rw	---
	rw_RW	Rwanda
Sanskrit	sa	---
	sa_IN	India
Northern Sami	se	---
	se_FI	Finland
	se_NO	Norway
Serbo-Croatian	sh	---
	sh_BA	Bosnia and Herzegovina
	sh_CS	Serbia and Montenegro
	sh_YU	Serbia
Sinhala	si	---
	si_LK	Sri Lanka
Sidamo	sid	---
	sid_ET	Ethiopia
Slovak	sk	---
	sk_SK	Slovakia
Slovenian	sl	---
	sl_SI	Slovenia
Somali	so	---
	so_DJ	Djibouti
	so_ET	Ethiopia
	so_KE	Kenya
	so_SO	Somalia
Albanian	sq	---
	sq_AL	Albania
Serbian	sr	---
	sr_BA	Bosnia and Herzegovina
	sr_CS	Serbia and Montenegro
	sr_ME	Montenegro
	sr_RS	Serbia
	sr_YU	Serbia
Swati	ss	---
	ss_SZ	Swaziland
	ss_ZA	South Africa
Southern Sotho	st	---
	st_LS	Lesotho
	st_ZA	South Africa

Language	**Locale**	**Region**
Swedish	sv	---
	sv_FI	Finland
	sv_SE	Sweden
Swahili	sw	---
	sw_KE	Kenya
	sw_TZ	Tanzania
Syriac	syr	---
	syr_SY	Syria
Tamil	ta	---
	ta_IN	India
Telugu	te	---
	te_IN	India
Tajik	tg	---
	tg_TJ	Tajikistan
Thai	th	---
	th_TH	Thailand
Tigrinya	ti	---
	ti_ER	Eritrea
	ti_ET	Ethiopia
Tigre	tig	---
	tig_ER	Eritrea
Tagalog	tl	---
Tswana	tn	---
	tn_ZA	South Africa
Tonga	to	---
	to_TO	Tonga
Turkish	tr	---
	tr_TR	Turkey
Tsonga	ts	---
	ts_ZA	South Africa
Tatar	tt	---
	tt_RU	Russia
Uighur	ug	---
	ug_CN	China
Ukrainian	uk	---
	uk_UA	Ukraine

Language	**Locale**	**Region**
Urdu	ur	---
	ur_IN	India
	ur_PK	Pakistan
Uzbek	uz	---
	uz_AF	Afghanistan
	uz_UZ	Uzbekistan
Venda	ve	---
	ve_ZA	South Africa
Vietnamese	vi	---
	vi_VN	Vietnam
Walamo	wal	---
	wal_ET	Ethiopia
Wolof	wo	---
	wo_SN	Senegal
Xhosa	xh	---
	xh_ZA	South Africa
Yoruba	yo	---
	yo_NG	Nigeria
Chinese	zh	---
	zh_CN	China
	zh_HK	Hong Kong
	zh_MO	Macau
	zh_SG	Singapore
	zh_TW	Taiwan
Zulu	zu	---
	zu_ZA	South Africa

Chapter 28. Zend_Log

Overview

`Zend_Log` 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 Log (instance of `Zend_Log`) is the object that your application uses the most. You can have as many Log objects as you like; they do not interact. A Log object must contain at least one Writer, and can optionally contain one or more Filters.
- A Writer (inherits from `Zend_Log_Writer_Abstract`) is responsible for saving data to storage.
- A Filter (implements `Zend_Log_Filter_Interface`) blocks log data from being saved. A filter may be applied to an individual Writer, or to a Log where it is applied before all Writers. In either case, filters may be chained.
- A Formatter (implements `Zend_Log_Formatter_Interface`) 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 Log instance:

```
$logger = new Zend_Log();
$writer = new Zend_Log_Writer_Stream('php://output');

$logger->addWriter($writer);
```

It is important to note that the Log must have at least one Writer. You can add any number of Writers using the Log's `addWriter()` method.

Alternatively, you can pass a Writer directly to constructor of Log as a shortcut:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$logger = new Zend_Log($writer);
```

The Log is now ready to use.

Logging Messages

To log a message, call the `log()` method of a Log instance and pass it the message with a corresponding priority:

```
$logger->log('Informational message', Zend_Log::INFO);
```

The first parameter of the `log()` method is a string `message` and the second parameter is an integer `priority`. The priority must be one of the priorities recognized by the Log instance. This is explained in the next section.

A shortcut is also available. Instead of calling the `log()` method, you can call a method by the same name as the priority:

```
$logger->log('Informational message', Zend_Log::INFO);
$logger->info('Informational message');

$logger->log('Emergency message', Zend_Log::EMERG);
$logger->emerg('Emergency message');
```

Destroying a Log

If the Log object is no longer needed, set the variable containing it to `null` to destroy it. This will automatically call the `shutdown()` instance method of each attached Writer before the Log object is destroyed:

```
$logger = null;
```

Explicitly destroying the log in this way is optional and is performed automatically at PHP shutdown.

Using Built-in Priorities

The `Zend_Log` class defines the following priorities:

```
EMERG   = 0;  // Emergency: system is unusable
ALERT   = 1;  // Alert: action must be taken immediately
CRIT    = 2;  // Critical: critical conditions
ERR     = 3;  // Error: error conditions
WARN    = 4;  // Warning: warning conditions
NOTICE  = 5;  // Notice: normal but significant condition
INFO    = 6;  // Informational: informational messages
DEBUG   = 7;  // Debug: debug messages
```

These priorities are always available, and a convenience method of the same name is available for each one.

The priorities are not arbitrary. They come from the BSD `syslog` protocol, which is described in RFC-3164 [http://tools.ietf.org/html/rfc3164]. The names and corresponding priority numbers are also compatible with another PHP logging system, PEAR Log [http://pear.php.net/package/log], which perhaps promotes interoperability between it and `Zend_Log`.

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.

Adding User-defined Priorities

User-defined priorities can be added at runtime using the Log's `addPriority()` method:

```
$logger->addPriority('FOO', 8);
```

The snippet above creates a new priority, `FOO`, whose value is `8`. The new priority is then available for logging:

```
$logger->log('Foo message', 8);
$logger->foo('Foo Message');
```

New priorities cannot overwrite existing ones.

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. However, knowledge of the `event` array is required for adding an item that does not exist in the default set above.

To add a new item to every future event, call the `setEventItem()` method giving a key and a value:

```
$logger->setEventItem('pid', getmypid());
```

The example above sets a new item named `pid` and populates it with the PID of the current process. Once a new item has been set, it is available automatically to all writers along with all of the other data event data during logging. An item can be overwritten at any time by calling the `setEventItem()` method again.

Setting a new event item with `setEventItem()` causes the new item to be sent to all writers of the logger. However, this does not guarantee that the writers actually record the item. This is because the writers won't know what to do with it unless a formatter object is informed of the new item. Please see the section on Formatters to learn more.

Writers

A Writer is an object that inherits from `Zend_Log_Writer_Abstract`. 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 [http://www.php.net/stream].

To write log data to the PHP output buffer, use the URL `php://output`. Alternatively, you can may like to send log data directly to a stream like `STDERR` (`php://stderr`).

```
$writer = new Zend_Log_Writer_Stream('php://output');
$logger = new Zend_Log($writer);

$logger->info('Informational message');
```

To write data to a file, use one of the Filesystem URLs [http://www.php.net/manual/en/wrappers.php#wrappers.file]:

```
$writer = new Zend_Log_Writer_Stream('/path/to/logfile');
$logger = new Zend_Log($writer);

$logger->info('Informational message');
```

By default, the stream opens in the append mode (`"a"`). To open it with a different mode, the Zend_Log_Writer_Stream constructor accepts an optional second parameter for the mode.

The constructor of `Zend_Log_Writer_Stream` also accepts an existing stream resource:

```
$stream = @fopen('/path/to/logfile', 'a', false);
if (! $stream) {
    throw new Exception('Failed to open stream');
}

$writer = new Zend_Log_Writer_Stream($stream);
$logger = new Zend_Log($writer);

$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

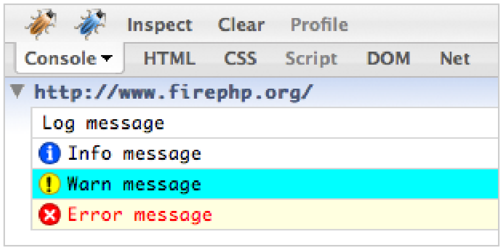
Zend_Log_Writer_Db writes log information to a database table using Zend_Db. The constructor of Zend_Log_Writer_Db receives a Zend_Db_Adapter instance, a table name, and a mapping of database columns to event data items:

```
$params = array ('host'     => '127.0.0.1',
                 'username' => 'malory',
                 'password' => '******',
                 'dbname'   => 'camelot');
$db = Zend_Db::factory('PDO_MYSQL', $params);

$columnMapping = array('lvl' => 'priority', 'msg' => 'message');
$writer = new Zend_Log_Writer_Db($db, 'log_table_name', $columnMapping);

$logger = new Zend_Log($writer);

$logger->info('Informational message');
```

The example above writes a single row of log data to the database table named log_table_name table. The database column named lvl receives the priority number and the column named msg receives the log message.

Writing to Firebug

Zend_Log_Writer_Firebug sends log data to the Firebug [http://www.getfirebug.com/] Console [http://getfirebug.com/logging.html].



```
http://www.firephp.org/
Log message
Info message
Warn message
Error message
```

All data is sent via the Zend_Wildfire_Channel_HttpHeaders component which uses HTTP headers to ensure the page content is not disturbed. Debugging AJAX requests that require clean JSON and XML responses is possible with this approach.

Requirements:

- Firefox Browser ideally version 3 but version 2 is also supported.
- Firebug Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/1843.
- FirePHP Firefox Extension which you can download from https://addons.mozilla.org/en-US/firefox/addon/6149.

Example 28.1. Logging with `Zend_Controller_Front`

```
// Place this in your bootstrap file before dispatching your front controller
$writer = new Zend_Log_Writer_Firebug();
$logger = new Zend_Log($writer);

// Use this in your model, view and controller files
$logger->log('This is a log message!', Zend_Log::INFO);
```

Example 28.2. Logging without `Zend_Controller_Front`

```
$writer = new Zend_Log_Writer_Firebug();
$logger = new Zend_Log($writer);

$request = new Zend_Controller_Request_Http();
$response = new Zend_Controller_Response_Http();
$channel = Zend_Wildfire_Channel_HttpHeaders::getInstance();
$channel->setRequest($request);
$channel->setResponse($response);

// Now you can make calls to the logger

$logger->log('This is a log message!', Zend_Log::INFO);

// Flush log data to browser
$channel->flush();
$response->sendHeaders();
```

Setting Styles for Priorities

Built-in and user-defined priorities can be styled with the `setPriorityStyle()` method.

```
$logger->addPriority('FOO', 8);
$writer->setPriorityStyle(8, 'TRACE');
$logger->foo('Foo Message');
```

The default style for user-defined priorities can be set with the `setDefaultPriorityStyle()` method.

```
$writer->setDefaultPriorityStyle('TRACE');
```

The supported styles are as follows:

Table 28.1. Firebug Logging Styles

Style	Description
`LOG`	Displays a plain log message
`INFO`	Displays an info log message
`WARN`	Displays a warning log message
`ERROR`	Displays an error log message that increments Firebug's error count
`TRACE`	Displays a log message with an expandable stack trace
`EXCEPTION`	Displays an error long message with an expandable stack trace
`TABLE`	Displays a log message with an expandable table

Preparing data for Logging

While any PHP variable can be logged with the built-in priorities, some special formatting is required if using some of the more specialized log styles.

The `LOG`, `INFO`, `WARN`, `ERROR` and `TRACE` styles require no special formatting.

Exception Logging

To log a `Zend_Exception` simply pass the exception object to the logger. It does not matter which priority or style you have set as the exception is automatically recognized.

```
$exception = new Zend_Exception('Test exception');
$logger->err($exception);
```

Table Logging

You can also log data and format it in a table style. Columns are automatically recognized and the first row of data automatically becomes the header.

```
$writer->setPriorityStyle(8, 'TABLE');
$logger->addPriority('TABLE', 8);

$table = array('Summary line for the table',
             array(
                 array('Column 1', 'Column 2'),
                 array('Row 1 c 1',' Row 1 c 2'),
                 array('Row 2 c 1',' Row 2 c 2')
             )
            );
$logger->table($table);
```

Stubbing Out the Writer

The `Zend_Log_Writer_Null` is a stub that does not write log data to anything. It is useful for disabling logging or stubbing out logging during tests:

```
$writer = new Zend_Log_Writer_Null;
$logger = new Zend_Log($writer);

// goes nowhere
$logger->info('Informational message');
```

Testing with the Mock

The `Zend_Log_Writer_Mock` is a very simple writer that records the raw data it receives in an array exposed as a public property.

```
$mock = new Zend_Log_Writer_Mock;
$logger = new Zend_Log($mock);

$logger->info('Informational message');

var_dump($mock->events[0]);

// Array
// (
//     [timestamp] => 2007-04-06T07:16:37-07:00
//     [message] => Informational message
//     [priority] => 6
//     [priorityName] => INFO
// )
```

To clear the events logged by the mock, simply set `$mock->events = array()`.

Compositing Writers

There is no composite Writer object. However, a Log instance can write to any number of Writers. To do this, use the `addWriter()` method:

```
$writer1 = new Zend_Log_Writer_Stream('/path/to/first/logfile');
$writer2 = new Zend_Log_Writer_Stream('/path/to/second/logfile');

$logger = new Zend_Log();
$logger->addWriter($writer1);
$logger->addWriter($writer2);

// goes to both writers
```

```
$logger->info('Informational message');
```

Formatters

A Formatter is an object that is responsible for taking an `event` array describing a log event and outputting a string with a formatted log line.

Some Writers are not line-oriented and cannot use a Formatter. An example is the Database Writer, which inserts the event items directly into database columns. For Writers that cannot support a Formatter, an exception is thrown if you attempt to set a Formatter.

Simple Formatting

`Zend_Log_Formatter_Simple` is the default formatter. It is configured automatically when you specify no formatter. The default configuration is equivalent to the following:

```
$format = '%timestamp% %priorityName% (%priority%): %message%' . PHP_EOL;
$formatter = new Zend_Log_Formatter_Simple($format);
```

A formatter is set on an individual Writer object using the Writer's `setFormatter()` method:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$formatter = new Zend_Log_Formatter_Simple('hello %message%' . PHP_EOL);
$writer->setFormatter($formatter);

$logger = new Zend_Log();
$logger->addWriter($writer);

$logger->info('there');

// outputs "hello there"
```

The constructor of `Zend_Log_Formatter_Simple` accepts a single parameter: the format string. This string contains keys surrounded by percent signs (e.g. `%message%`). The format string may contain any key from the event data array. You can retrieve the default keys by using the DEFAULT_FORMAT constant from `Zend_Log_Formatter_Simple`.

Formatting to XML

`Zend_Log_Formatter_Xml` formats log data into XML strings. By default, it automatically logs all items in the event data array:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$formatter = new Zend_Log_Formatter_Xml();
```

```
$writer->setFormatter($formatter);

$logger = new Zend_Log();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above outputs the following XML (space added for clarity):

```
<logEntry>
  <timestamp>2007-04-06T07:24:37-07:00</timestamp>
  <message>informational message</message>
  <priority>6</priority>
  <priorityName>INFO</priorityName>
</logEntry>
```

It's possible to customize the root element as well as specify a mapping of XML elements to the items in the event data array. The constructor of `Zend_Log_Formatter_Xml` accepts a string with the name of the root element as the first parameter and an associative array with the element mapping as the second parameter:

```
$writer = new Zend_Log_Writer_Stream('php://output');
$formatter = new Zend_Log_Formatter_Xml('log',
                                        array('msg' => 'message',
                                              'level' => 'priorityName')
                                       );
$writer->setFormatter($formatter);

$logger = new Zend_Log();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above changes the root element from its default of `logEntry` to `log`. It also maps the element `msg` to the event data item `message`. This results in the following output:

```
<log>
  <msg>informational message</msg>
  <level>INFO</level>
</log>
```

Filters

A Filter object blocks a message from being written to the log.

Filtering for All Writers

To filter before all writers, you can add any number of Filters to a Log object using the `addFilter()` method:

```
$logger = new Zend_Log();

$writer = new Zend_Log_Writer_Stream('php://output');
$logger->addWriter($writer);

$filter = new Zend_Log_Filter_Priority(Zend_Log::CRIT);
$logger->addFilter($filter);

// blocked
$logger->info('Informational message');

// logged
$logger->emerg('Emergency message');
```

When you add one or more Filters to the Log object, the message must pass through all of the Filters before any Writers receives it.

Filtering for a Writer Instance

To filter only on a specific Writer instance, use the `addFilter` method of that Writer:

```
$logger = new Zend_Log();

$writer1 = new Zend_Log_Writer_Stream('/path/to/first/logfile');
$logger->addWriter($writer1);

$writer2 = new Zend_Log_Writer_Stream('/path/to/second/logfile');
$logger->addWriter($writer2);

// add a filter only to writer2
$filter = new Zend_Log_Filter_Priority(Zend_Log::CRIT);
$writer2->addFilter($filter);

// logged to writer1, blocked from writer2
$logger->info('Informational message');

// logged by both writers
$logger->emerg('Emergency message');
```

Chapter 29. Zend_Mail

Introduction

Getting started

Zend_Mail provides generalized functionality to compose and send both text and MIME-compliant multipart e-mail messages. Mail can be sent with Zend_Mail via the default Zend_Mail_Transport_Sendmail transport or via Zend_Mail_Transport_Smtp.

Example 29.1. Simple E-Mail with Zend_Mail

A simple e-mail consists of some recipients, a subject, a body and a sender. To send such a mail using Zend_Mail_Transport_Sendmail, do the following:

```
$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

Minimum definitions

In order to send an e-mail with Zend_Mail you have to specify at least one recipient, a sender (e.g., with setFrom()), and a message body (text and/or HTML).

For most mail attributes there are "get" methods to read the information stored in the mail object. For further details, please refer to the API documentation. A special one is getRecipients(). It returns an array with all recipient e-mail addresses that were added prior to the method call.

For security reasons, Zend_Mail filters all header fields to prevent header injection with newline (\n) characters.

You also can use most methods of the Zend_Mail object with a convenient fluent interface. A fluent interface means that each method returns a reference to the object on which it was called, so you can immediately call another method.

```
$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.')
    ->setFrom('somebody@example.com', 'Some Sender')
    ->addTo('somebody_else@example.com', 'Some Recipient')
    ->setSubject('TestSubject')
    ->send();
```

Configuring the default sendmail transport

The default transport for a `Zend_Mail` instance is `Zend_Mail_Transport_Sendmail`. It is essentially a wrapper to the PHP `mail()` [http://php.net/mail] function. If you wish to pass additional parameters to the `mail()` [http://php.net/mail] function, simply create a new transport instance and pass your parameters to the constructor. The new transport instance can then act as the default `Zend_Mail` transport, or it can be passed to the `send()` method of `Zend_Mail`.

Example 29.2. Passing additional parameters to the Zend_Mail_Transport_Sendmail transport

This example shows how to change the Return-Path of the `mail()` [http://php.net/mail] function.

```
$tr = new Zend_Mail_Transport_Sendmail('-freturn_to_me@example.com');
Zend_Mail::setDefaultTransport($tr);

$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

Safe mode restrictions

The optional additional parameters will be cause the `mail()` [http://php.net/mail] function to fail if PHP is running in safe mode.

Sending via SMTP

To send mail via SMTP, `Zend_Mail_Transport_Smtp` needs to be created and registered with `Zend_Mail` before the `send()` method is called. For all remaining `Zend_Mail::send()` calls in the current script, the SMTP transport will then be used:

Example 29.3. Sending E-Mail via SMTP

```
$tr = new Zend_Mail_Transport_Smtp('mail.example.com');
Zend_Mail::setDefaultTransport($tr);
```

The `setDefaultTransport()` method and the constructor of `Zend_Mail_Transport_Smtp` are not expensive. These two lines can be processed at script setup time (e.g., config.inc or similar) to configure the behaviour of the `Zend_Mail` class for the rest of the script. This keeps configuration information out of the application logic - whether mail is sent via SMTP or `mail()` [http://php.net/mail], what mail server to use, etc.

Sending Multiple Mails per SMTP Connection

By default a single SMTP transport creates a single connection and re-uses it for the lifetime of the script execution. You may send multiple e-mails through this SMTP connection. A RSET command is issued before each delivery to ensure the correct SMTP handshake is followed.

Example 29.4. Sending Multiple Mails per SMTP Connection

```
// Create transport
$transport = new Zend_Mail_Transport_Smtp('localhost');

// Loop through messages
for ($i = 0; $i > 5; $i++) {
    $mail = new Zend_Mail();
    $mail->addTo('studio@peptolab.com', 'Test');
    $mail->setFrom('studio@peptolab.com', 'Test');
    $mail->setSubject(
        'Demonstration - Sending Multiple Mails per SMTP Connection'
    );
    $mail->setBodyText('...Your message here...');
    $mail->send($transport);
}
```

If you wish to have a separate connection for each mail delivery, you will need to create and destroy your transport before and after each `send()` method is called. Or alternatively, you can manipulate the connection between each delivery by accessing the transport's protocol object.

Example 29.5. Manually controlling the transport connection

```
// Create transport
$transport = new Zend_Mail_Transport_Smtp();

$protocol = new Zend_Mail_Protocol_Smtp('localhost');
$protocol->connect();
$protocol->helo('localhost');

$transport-&gt;setConnection($protocol);

// Loop through messages
for ($i = 0; $i > 5; $i++) {
    $mail = new Zend_Mail();
    $mail->addTo('studio@peptolab.com', 'Test');
    $mail->setFrom('studio@peptolab.com', 'Test');
    $mail->setSubject(
        'Demonstration - Sending Multiple Mails per SMTP Connection'
    );
    $mail->setBodyText('...Your message here...');

    // Manually control the connection
    $protocol->rset();
    $mail->send($transport);
}

$protocol->quit();
$protocol->disconnect();
```

Using Different Transports

In case you want to send different e-mails through different connections, you can also pass the transport object directly to `send()` without a prior call to `setDefaultTransport()`. The passed object will override the default transport for the actual `send()` request:

Example 29.6. Using Different Transports

```
$mail = new Zend_Mail();
// build message...
$tr1 = new Zend_Mail_Transport_Smtp('server@example.com');
$tr2 = new Zend_Mail_Transport_Smtp('other_server@example.com');
$mail->send($tr1);
$mail->send($tr2);
$mail->send();  // use default again
```

Additional transports

Additional transports can be written by implementing `Zend_Mail_Transport_Interface`.

HTML E-Mail

To send an e-mail in HTML format, set the body using the method `setBodyHTML()` instead of `setBodyText()`. The MIME content type will automatically be set to `text/html` then. If you use both HTML and Text bodies, a multipart/alternative MIME message will automatically be generated:

Example 29.7. Sending HTML E-Mail

```
$mail = new Zend_Mail();
$mail->setBodyText('My Nice Test Text');
$mail->setBodyHtml('My Nice <b>Test</b> Text');
$mail->setFrom('somebody@example.com', 'Some Sender');
$mail->addTo('somebody_else@example.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send();
```

Attachments

Files can be attached to an e-mail using the `createAttachment()` method. The default behaviour of `Zend_Mail` is to assume the attachment is a binary object (application/octet-stream), should be transferred with base64 encoding, and is handled as an attachment. These assumptions can be overridden by passing more parameters to `createAttachment()`:

Example 29.8. E-Mail Messages with Attachments

```
$mail = new Zend_Mail();
// build message...
$mail->createAttachment($someBinaryString);
$mail->createAttachment($myImage,
                        'image/gif',
                        Zend_Mime::DISPOSITION_INLINE,
                        Zend_Mime::ENCODING_8BIT);
```

If you want more control over the MIME part generated for this attachment you can use the return value of `createAttachment()` to modify its attributes. The `createAttachment()` method returns a `Zend_Mime_Part` object:

```
$mail = new Zend_Mail();

$at = $mail->createAttachment($myImage);
$at->type        = 'image/gif';
```

```
$at->disposition = Zend_Mime::DISPOSITION_INLINE;
$at->encoding    = Zend_Mime::ENCODING_8BIT;
$at->filename    = 'test.gif';

$mail->send();
```

An alternative is to create an instance of `Zend_Mime_Part` and add it with `addAttachment()`:

```
$mail = new Zend_Mail();

$at = new Zend_Mime_Part($myImage);
$at->type        = 'image/gif';
$at->disposition = Zend_Mime::DISPOSITION_INLINE;
$at->encoding    = Zend_Mime::ENCODING_8BIT;
$at->filename    = 'test.gif';

$mail->addAttachment($at);

$mail->send();
```

Adding Recipients

Recipients can be added in three ways:

- `addTo()`: Adds a recipient to the mail with a "To" header
- `addCc()`: Adds a recipient to the mail with a "Cc" header
- `addBcc()`: Adds a recipient to the mail not visible in the header.

Additional parameter

`addTo()` and `addCc()` accept a second optional parameter that is used as a human-readable name of the recipient for the header.

Controlling the MIME Boundary

In a multipart message a MIME boundary for separating the different parts of the message is normally generated at random. In some cases, however, you might want to specify the MIME boundary that is used. This can be done using the `setMimeBoundary()` method, as in the following example:

Example 29.9. Changing the MIME Boundary

```
$mail = new Zend_Mail();
$mail->setMimeBoundary('=_' . md5(microtime(1) . $someId++));
// build message...
```

Additional Headers

Arbitrary mail headers can be set by using the `addHeader()` method. It requires two parameters containing the name and the value of the header field. A third optional parameter determines if the header should have only one or multiple values:

Example 29.10. Adding E-Mail Message Headers

```
$mail = new Zend_Mail();
$mail->addHeader('X-MailGenerator', 'MyCoolApplication');
$mail->addHeader('X-greetingsTo', 'Mom', true); // multiple values
$mail->addHeader('X-greetingsTo', 'Dad', true);
```

Character Sets

`Zend_Mail` does not check for the correct character set of the mail parts. When instantiating `Zend_Mail`, a charset for the e-mail itself may be given. It defaults to `iso-8859-1`. The application has to make sure that all parts added to that mail object have their content encoded in the correct character set. When creating a new mail part, a different charset can be given for each part.

Only in text format

Character sets are only applicable for message parts in text format.

Encoding

Text and HTML message bodies are encoded with the quotedprintable mechanism by default. All other attachments are encoded via base64 if no other encoding is given in the `addAttachment()` call or assigned to the MIME part object later. 7Bit and 8Bit encoding currently only pass on the binary content data.

`Zend_Mail_Transport_Smtp` encodes lines starting with one dot or two dots so that the mail does not violate the SMTP protocol.

SMTP Authentication

`Zend_Mail` supports the use of SMTP Authentication, which can be enabled be passing the 'auth' parameter to the configuration array in the `Zend_Mail_Transport_Smtp` constructor. The available built-in Authentication methods are PLAIN, LOGIN and CRAM-MD5 which all expect a 'username' and 'password' value in the configuration array.

Example 29.11. Enabling authentication within Zend_Mail_Transport_Smtp

```
$config = array('auth' => 'login',
                'username' => 'myusername',
                'password' => 'password');

$transport = new Zend_Mail_Transport_Smtp('mail.server.com', $config);

$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('sender@test.com', 'Some Sender');
$mail->addTo('recipient@test.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send($transport);
```

Authentication types

The authentication type is case-insensitive but has no punctuation. E.g. to use CRAM-MD5 you would pass 'auth' => 'crammd5' in the `Zend_Mail_Transport_Smtp` constructor.

Securing SMTP Transport

`Zend_Mail` also supports the use of either TLS or SSL to secure a SMTP connection. This can be enabled be passing the 'ssl' parameter to the configuration array in the `Zend_Mail_Transport_Smtp` constructor with a value of either 'ssl' or 'tls'. A port can optionally be supplied, otherwise it defaults to 25 for TLS or 465 for SSL.

Example 29.12. Enabling a secure connection within Zend_Mail_Transport_Smtp

```
$config = array('ssl' => 'tls',
                'port' => 25); // Optional port number supplied

$transport = new Zend_Mail_Transport_Smtp('mail.server.com', $config);

$mail = new Zend_Mail();
$mail->setBodyText('This is the text of the mail.');
$mail->setFrom('sender@test.com', 'Some Sender');
$mail->addTo('recipient@test.com', 'Some Recipient');
$mail->setSubject('TestSubject');
$mail->send($transport);
```

Reading Mail Messages

`Zend_Mail` can read mail messages from several local or remote mail storages. All of them have the same basic API to count and fetch messages and some of them implement additional interfaces for not so common features. For a feature overview of the implemented storages see the following table.

Table 29.1. Mail Read Feature Overview

Feature	Mbox	Maildir	Pop3	IMAP
Storage type	local	local	remote	remote
Fetch message	Yes	Yes	Yes	Yes
Fetch mime-part	emulated	emulated	emulated	emulated
Folders	Yes	Yes	No	Yes
Create message/folder	No	todo	No	todo
Flags	No	Yes	No	Yes
Quota	No	Yes	No	No

Simple example using Pop3

```
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'localhost',
                                         'user'     => 'test',
                                         'password' => 'test'));

echo $mail->countMessages() . " messages found\n";
foreach ($mail as $message) {
    echo "Mail from '{$message->from}': {$message->subject}\n";
}
```

Opening a local storage

Mbox and Maildir are the two supported formats for local mail storages, both in their most simple formats.

If you want to read from a Mbox file you only need to give the filename to the constructor of `Zend_Mail_Storage_Mbox`:

```
$mail = new Zend_Mail_Storage_Mbox(array('filename' =>
                                             '/home/test/mail/inbox'));
```

Maildir is very similar but needs a dirname:

```
$mail = new Zend_Mail_Storage_Maildir(array('dirname' =>
                                                '/home/test/mail/'));
```

Both constructors throw a `Zend_Mail_Exception` if the storage can't be read.

Opening a remote storage

For remote storages the two most popular protocols are supported: Pop3 and Imap. Both need at least a host and a user to connect and login. The default password is an empty string, the default port as given in the protocol RFC.

```
// connecting with Pop3
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com'
                                         'user'     => 'test',
                                         'password' => 'test'));

// connecting with Imap
$mail = new Zend_Mail_Storage_Imap(array('host'     => 'example.com'
                                         'user'     => 'test',
                                         'password' => 'test'));

// example for a none standard port
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com',
                                         'port'     => 1120
                                         'user'     => 'test',
                                         'password' => 'test'));
```

For both storages SSL and TLS are supported. If you use SSL the default port changes as given in the RFC.

```
// examples for Zend_Mail_Storage_Pop3, same works for Zend_Mail_Storage_Imap

// use SSL on different port (default is 995 for Pop3 and 993 for Imap)
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com'
                                         'user'     => 'test',
                                         'password' => 'test',
                                         'ssl'      => 'SSL'));

// use TLS
$mail = new Zend_Mail_Storage_Pop3(array('host'     => 'example.com'
                                         'user'     => 'test',
                                         'password' => 'test',
                                         'ssl'      => 'TLS'));
```

Both constructors can throw `Zend_Mail_Exception` or `Zend_Mail_Protocol_Exception` (extends `Zend_Mail_Exception`), depending on the type of error.

Fetching messages and simple methods

Once you've opened the storage messages can be fetched. You need the message number, which is a counter starting with 1 for the first message. To fetch the message you use the method `getMessage()`:

```
$message = $mail->getMessage($messageNum);
```

Array access is also supported, but won't supported any additional parameters that could be added to `getMessage()`. As long as you don't mind and can live with defaults you may use:

```
$message = $mail[$messageNum];
```

For iterating over all messages the Iterator interface is implemented:

```
foreach ($mail as $messageNum => $message) {
    // do stuff ...
}
```

To count the messages in the storage you can either use the method `countMessages()` or use array access:

```
// method
$maxMessage = $mail->countMessages();

// array access
$maxMessage = count($mail);
```

To remove a mail you use the method `removeMessage()` or again array access:

```
// method
$mail->removeMessage($messageNum);

// array access
unset($mail[$messageNum]);
```

Working with messages

After you fetched the messages with `getMessage()` you want to fetch headers, the content or single parts of a multipart message. All headers can be accessed via properties or the method `getHeader()` if you want more control or have unusual header names. The header names are lower-cased internally, thus the case of the header name in the mail message doesn't matter. Also headers with a dash can be written in camel-case.

```
// get the message object
$message = $mail->getMessage(1);

// output subject of message
echo $message->subject . "\n";

// get content-type header
$type = $message->contentType;
```

If you have multiple headers with the same name i.e. the Received headers you might want it as array instead of a string, which is possible with the `getHeader()` method.

```
// get header as property - the result is always a string,
// with new lines between the single occurrences in the message
$received = $message->received;

// the same via getHeader() method
$received = $message->getHeader('received', 'string');

// better an array with a single entry for every occurrences
$received = $message->getHeader('received', 'array');
foreach ($received as $line) {
    // do stuff
}

// if you don't define a format you'll get the internal representation
// (string for single headers, array for multiple)
$received = $message->getHeader('received');
if (is_string($received)) {
    // only one received header found in message
}
```

The method `getHeaders()` returns all headers as array with the lower-cased name as key and the value as array for multiple headers or as string for single headers.

```
// dump all headers
foreach ($message->getHeaders() as $name => $value) {
    if (is_string($value)) {
        echo "$name: $value\n";
        continue;
    }
    foreach ($value as $entry) {
        echo "$name: $entry\n";
    }
}
```

If you don't have a multipart message fetching the content is easy done via `getContent()`. Unlike the headers the content is only fetched when needed (aka late-fetch).

```
// output message content for HTML
echo '<pre>';
echo $message->getContent();
echo '</pre>';
```

Checking for a multipart message is done with the method `isMultipart()`. If you have multipart message you can get an instance of `Zend_Mail_Part` with the method `getPart()`. `Zend_Mail_Part` is the base class of `Zend_Mail_Message`, so you have the same methods: `getHeader()`, `getHeaders()`, `getContent()`, `getPart()`, `isMultipart` and the properties for headers.

```
// get the first none multipart part
$part = $message;
while ($part->isMultipart()) {
    $part = $message->getPart(1);
}
echo 'Type of this part is ' . strtok($part->contentType, ';') . "\n";
echo "Content:\n";
echo $part->getContent();
```

`Zend_Mail_Part` also implements `RecursiveIterator`, which makes it easy to scan through all parts. And for easy output it also implements the magic method `__toString()`, which returns the content.

```
// output first text/plain part
$foundPart = null;
foreach (new RecursiveIteratorIterator($mail->getMessage(1)) as $part) {
    try {
        if (strtok($part->contentType, ';') == 'text/plain') {
            $foundPart = $part;
            break;
        }
    } catch (Zend_Mail_Exception $e) {
        // ignore
    }
}
if (!$foundPart) {
    echo 'no plain text part found';
} else {
    echo "plain text part: \n" . $foundPart;
}
```

Checking for flags

Maildir and IMAP support storing flags. The class `Zend_Mail_Storage` has constants for all known maildir and IMAP system flags, named `Zend_Mail_Storage::FLAG_<flagname>`. To check for flags `Zend_Mail_Message` has a method called `hasFlag()`. With `getFlags()` you'll get all set flags.

```
// find unread messages
echo "Unread mails:\n";
foreach ($mail as $message) {
    if ($message->hasFlag(Zend_Mail_Storage::FLAG_SEEN)) {
        continue;
    }
    // mark recent/new mails
    if ($message->hasFlag(Zend_Mail_Storage::FLAG_RECENT)) {
        echo '! ';
    } else {
        echo '  ';
    }
    echo $message->subject . "\n";
}


// check for known flags
$flags = $message->getFlags();
echo "Message is flagged as: ";
foreach ($flags as $flag) {
    switch ($flag) {
        case Zend_Mail_Storage::FLAG_ANSWERED:
            echo 'Answered ';
            break;
        case Zend_Mail_Storage::FLAG_FLAGGED:
            echo 'Flagged ';
            break;

        // ...
        // check for other flags
        // ...

        default:
            echo $flag . '(unknown flag) ';
    }
}
```

As IMAP allows user or client defined flags you could get flags, that don't have a constant in `Zend_Mail_Storage`. Instead they are returned as string and can be checked the same way with `hasFlag()`.

```
// check message for client defined flags $IsSpam, $SpamTested
if (!$message->hasFlag('$SpamTested')) {
    echo 'message has not been tested for spam';
} else if ($message->hasFlag('$IsSpam')) {
    echo 'this message is spam';
} else {
    echo 'this message is ham';
}
```

Using folders

All storages, except Pop3, support folders, also called mailboxes. The interface implemented by all storages supporting folders is called `Zend_Mail_Storage_Folder_Interface`. Also all of these classes have an additional optional parameter called `folder`, which is the folder selected after login, in the constructor.

For the local storages you need to use separate classes called `Zend_Mail_Storage_Folder_Mbox` or `Zend_Mail_Storage_Folder_Maildir`. Both need one parameter called `dirname` with the name of the base dir. The format for maildir is as defined in maildir++ (with a dot as default delimiter), Mbox is a directory hierarchy with Mbox files. If you don't have a Mbox file called INBOX in your Mbox base dir you need to set an other folder in the constructor.

`Zend_Mail_Storage_Imap` already supports folders by default. Examples for opening these storages:

```
// mbox with folders
$mail = new Zend_Mail_Storage_Folder_Mbox(array('dirname' =>
                                                    '/home/test/mail/'));

// mbox with a default folder not called INBOX, also works
// with Zend_Mail_Storage_Folder_Maildir and Zend_Mail_Storage_Imap
$mail = new Zend_Mail_Storage_Folder_Mbox(array('dirname' =>
                                                    '/home/test/mail/',
                                                'folder'  =>
                                                    'Archive'));

// maildir with folders
$mail = new Zend_Mail_Storage_Folder_Maildir(array('dirname' =>
                                                       '/home/test/mail/'));

// maildir with colon as delimiter, as suggested in Maildir++
$mail = new Zend_Mail_Storage_Folder_Maildir(array('dirname' =>
                                                       '/home/test/mail/',
                                                   'delim'   => ':'));

// imap is the same with and without folders
$mail = new Zend_Mail_Storage_Imap(array('host'     => 'example.com'
                                         'user'     => 'test',
                                         'password' => 'test'));
```

With the method getFolders($root = null) you can get the folder hierarchy starting with the root folder or the given folder. It's returned as instance of `Zend_Mail_Storage_Folder`, which implements `RecursiveIterator` and all children are also instances of `Zend_Mail_Storage_Folder`. Each of these instances has a local and a global name returned by the methods `getLocalName()` and `getGlobalName()`. The global name is the absolute name from the root folder (including delimiters), the local name is the name in the parent folder.

Table 29.2. Mail Folder Names

Global Name	Local Name
/INBOX	INBOX
/Archive/2005	2005
List.ZF.General	General

If you use the iterator the key of the current element is the local name. The global name is also returned by the magic method `__toString()`. Some folders may not be selectable, which means they can't store messages and selecting them results in an error. This can be checked with the method `isSelectable()`. So it's very easy to output the whole tree in a view:

```
$folders = new RecursiveIteratorIterator($this->mail->getFolders(),
                                         RecursiveIteratorIterator::SELF_FIRST);
echo '<select name="folder">';
foreach ($folders as $localName => $folder) {
    $localName = str_pad('', $folders->getDepth(), '-', STR_PAD_LEFT) .
                 $localName;
    echo '<option';
    if (!$folder->isSelectable()) {
        echo ' disabled="disabled"';
    }
    echo ' value="' . htmlspecialchars($folder) . '">'
        . htmlspecialchars($localName) . '</option>';
}
echo '</select>';
```

The current selected folders is returned by the method `getSelectedFolder()`. Changing the folder is done with the method `selectFolder()`, which needs the global name as parameter. If you want to avoid to write delimiters you can also use the properties of a `Zend_Mail_Storage_Folder` instance:

```
// depending on your mail storage and its settings $rootFolder->Archive->2005
// is the same as:
//   /Archive/2005
//  Archive:2005
//  INBOX.Archive.2005
//  ...
$folder = $mail->getFolders()->Archive->2005;
echo 'Last folder was ' . $mail->getSelectedFolder() . "new folder is $folder\n";
$mail->selectFolder($folder);
```

Advanced Use

Using NOOP

If you're using a remote storage and have some long tasks you might need to keep the connection alive via noop:

```
foreach ($mail as $message) {

    // do some calculations ...

    $mail->noop(); // keep alive

    // do something else ...

    $mail->noop(); // keep alive
}
```

Caching instances

`Zend_Mail_Storage_Mbox`, `Zend_Mail_Storage_Folder_Mbox`, `Zend_Mail_Storage_Maildir` and `Zend_Mail_Storage_Folder_Maildir` implement the magic methods `__sleep()` and `__wakeup()`, which means they are serializable. This avoids parsing the files or directory tree more than once. The disadvantage is that your Mbox or Maildir storage should not change. Some easy checks are done, like reparsing the current Mbox file if the modification time changes or reparsing the folder structure if a folder has vanished (which still results in an error, but you can search for an other folder afterwards). It's better if you have something like a signal file for changes and check it before using the cached instance.

```
// there's no specific cache handler/class used here,
// change the code to match your cache handler
$signal_file = '/home/test/.mail.last_change';
$mbox_basedir = '/home/test/mail/';
$cache_id = 'example mail cache ' . $mbox_basedir . $signal_file;

$cache = new Your_Cache_Class();
if (!$cache->isCached($cache_id) ||
    filemtime($signal_file) > $cache->getMTime($cache_id)) {
    $mail = new Zend_Mail_Storage_Folder_Pop3(array('dirname' =>
                                                        $mbox_basedir));
} else {
    $mail = $cache->get($cache_id);
}

// do stuff ...

$cache->set($cache_id, $mail);
```

Extending Protocol Classes

Remote storages use two classes: `Zend_Mail_Storage_<Name>` and `Zend_Mail_Protocol_<Name>`. The protocol class translates the protocol commands and responses from and to PHP, like methods for the commands or variables with different structures for data. The other/main class implements the common interface.

If you need additional protocol features you can extend the protocol class and use it in the constructor of the main class. As an example assume we need to knock different ports before we can connect to POP3.

```
class Example_Mail_Exception extends Zend_Mail_Exception
{
}

class Example_Mail_Protocol_Exception extends Zend_Mail_Protocol_Exception
{
}

class Example_Mail_Protocol_Pop3_Knock extends Zend_Mail_Protocol_Pop3
{
    private $host, $port;

    public function __construct($host, $port = null)
    {
        // no auto connect in this class
        $this->host = $host;
        $this->port = $port;
    }

    public function knock($port)
    {
        $sock = @fsockopen($this->host, $port);
        if ($sock) {
            fclose($sock);
        }
    }

    public function connect($host = null, $port = null, $ssl = false)
    {
        if ($host === null) {
            $host = $this->host;
        }
        if ($port === null) {
            $port = $this->port;
        }
        parent::connect($host, $port);
    }
}

class Example_Mail_Pop3_Knock extends Zend_Mail_Storage_Pop3
{
    public function __construct(array $params)
    {
        // ... check $params here! ...
        $protocol = new Example_Mail_Protocol_Pop3_Knock($params['host']);

        // do our "special" thing
        foreach ((array)$params['knock_ports'] as $port) {
            $protocol->knock($port);
        }

        // get to correct state
        $protocol->connect($params['host'], $params['port']);
        $protocol->login($params['user'], $params['password']);

        // initialize parent
        parent::__construct($protocol);
    }
}

$mail = new Example_Mail_Pop3_Knock(array('host'        => 'localhost',
                                          'user'        => 'test',
                                          'password'    => 'test',
                                          'knock_ports' =>
                                              array(1101, 1105, 1111)));
```

As you see we always assume we're connected, logged in and, if supported, a folder is selected in the constructor of the main class. Thus if you assign your own protocol class you always need to make sure that's done or the next method will fail if the server doesn't allow it in the current state.

Using Quota (since 1.5)

`Zend_Mail_Storage_Writable_Maildir` has support for Maildir++ quotas. It's disabled by default, but it's possible to use it manually, if the automatic checks are not desired (this means `appendMessage()`, `removeMessage()` and `copyMessage()` do no checks and do not add entry to the maildirsize file). If enabled an exception is thrown if you try to write to the maildir if it's already over quota.

There are three methods used for quotas: `getQuota()`, `setQuota()` and `checkQuota()`:

```
$mail = new Zend_Mail_Storage_Writable_Maildir(array('dirname' =>
                                                   '/home/test/mail/'));
$mail->setQuota(true); // true to enable, false to disable
echo 'Quota check is now ', $mail->getQuota() ? 'enabled' : 'disabled', "\n";
// check quota can be used even if quota checks are disabled
echo 'You are ', $mail->checkQuota() ? 'over quota' : 'not over quota', "\n";
```

`checkQuota()` can also return a more detailed response:

```
$quota = $mail->checkQuota(true);
echo 'You are ', $quota['over_quota'] ? 'over quota' : 'not over quota', "\n";
echo 'You have ', $quota['count'], ' of ', $quota['quota']['count'], ' messages an
echo $quota['size'], ' of ', $quota['quota']['size'], ' octets';
```

If you want to specify your own quota instead of using the one specified in the maildirsize file you can do with `setQuota()`:

```
// message count and octet size supported, order does matter
$quota = $mail->setQuota(array('size' => 10000, 'count' => 100));
```

To add your own quota checks use single letters as key and they are preserved (but obviously not checked). It's also possible to extend `Zend_Mail_Storage_Writable_Maildir` to define your own quota only if the maildirsize file is missing (which can happen in Maildir++):

```
class Example_Mail_Storage_Maildir extends Zend_Mail_Storage_Writable_Maildir {
 // getQuota is called with $fromStorage = true by quota checks
 public function getQuota($fromStorage = false) {
  try {
   return parent::getQuota($fromStorage);
  } catch (Zend_Mail_Storage_Exception $e) {
   if (!$fromStorage) {
    // unknown error:
    throw $e;
   }
   // maildirsize file must be missing

   list($count, $size) = get_quota_from_somewhere_else();
   return array('count' => $count, 'size' => $size);
  }
 }
}
```

Chapter 30. Zend_Measure

Introduction

Zend_Measure_* classes provide a generic and easy way for working with measurements. Using Zend_Measure_* classes, you can convert measurements into different units of the same type. They can be added, subtracted and compared against each other. From a given input made in the user's native language, the unit of measurement can be automatically extracted. Numerous units of measurement are supported.

Example 30.1. Converting measurements

The following introductory example shows automatic conversion of units of measurement. To convert a measurement, its value and its type have to be known. The value can be an integer, a float, or even a string containing a number. Conversions are only possible for units of the same type (mass, area, temperature, velocity, etc.), not between types.

```
$locale = new Zend_Locale('en');
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::METER, $locale);

// Convert meters to yards
echo $unit->convertTo(Zend_Measure_Length::YARD);
```

Zend_Measure_* includes support for many different units of measurement. The units of measurement all have a unified notation: Zend_Measure_<TYPE>::NAME_OF_UNIT, where <TYPE> corresponds to a well-known physical or numerical property. . Every unit of measurement consists of a conversion factor and a display unit. A detailed list can be found in the chapter Types of measurements .

Example 30.2. The meter measurement

The meter is used for measuring lengths, so its type constant can be found in the Length class. To refer to this unit of measurement, the notation Length::METER must be used. The display unit is m.

```
echo Zend_Measure_Length::STANDARD;  // outputs 'Length::METER'
echo Zend_Measure_Length::KILOMETER; // outputs 'Length::KILOMETER'

$unit = new Zend_Measure_Length(100,'METER');
echo $unit;
// outputs '100 m'
```

Creation of Measurements

When creating a measurement object, Zend_Measure_* methods expect the input/original measurement data value as the first parameter. This can be a numeric argument , a string without units, or a `localized string` with unit(s) specified. The second parameter defines the type of the measurement. Both parameters are mandatory. The language may optionally be specified as the third parameter.

Creating measurements from integers and floats

In addition to integer data values, floating point types may be used, but "simple decimal fractions like 0.1 or 0.7 cannot be converted into their internal binary counterparts without a little loss of precision," [http://www.php.net/float] sometimes giving surprising results. Also, do not compare two "float" type numbers for equality.

Example 30.3. Creation using integer and floating values

```
$measurement = 1234.7;
$unit = new Zend_Measure_Length((integer)$measurement, Zend_Measure_Length::STANDA
echo $unit;
// outputs '1234 m' (meters)

$unit = new Zend_Measure_Length($measurement, Zend_Measure_Length::STANDARD);
echo $unit;
// outputs '1234.7 m' (meters)
```

Creating measurements from strings

Many measurements received as input to ZF applications can only be passed to `Zend_Measure_*` classes as strings, such as numbers written using roman numerals [http://en.wikipedia.org/wiki/Roman_numerals] or extremely large binary values that exceed the precision of PHP's native integer and float types. Since integers can be denoted using strings, if there is any risk of losing precision due to limitations of PHP's native integer and float types, using strings instead. `Zend_Measure_Number` uses the BCMath extension to support arbitrary precision, as shown in the example below, to avoid limitations in many PHP functions, such as `bin2dec()` [http://php.net/bin2dec] .

Example 30.4. Creation using strings

```
$mystring = "10010100111010111010100001011011101010001";
$unit = new Zend_Measure_Number($mystring, Zend_Measure_Number::BINARY);

echo $unit;
```

Usually, `Zend_Measure_*` can automatically extract the desired measurement embedded in an arbitrary string. Only the first identifiable number denoted using standard European/Latin digits (0,1,2,3,4,5,6,7,8,9) will be used for measurement creation. If there are more numerals later in the string, the rest of these numerals will be ignored.

Example 30.5. Arbitrary text input containing measurements

```
$mystring = "My house is 125m² in size";
$unit = new Zend_Measure_Area($mystring, Zend_Measure_Area::STANDARD);
echo $unit; // outputs "125 m²in size";

$mystring = "My house is 125m² in size, it has 5 rooms of 25m² each.";
$unit = new Zend_Measure_Area($mystring, Zend_Measure_Area::STANDARD);
echo $unit; // outputs "125 m² in size";
```

Measurements from localized strings

When a string is entered in a localized notation, the correct interpretation can not be determined without knowing the intended locale. The division of decimal digits with "." and grouping of thousands with "," is common in the English language, but not so in other languages. For example, the English number "1,234.50" would be interpreted as meaning "1.2345" in German. To deal with such problems, the locale-aware `Zend_Measure_*` family of classes offer the possibility to specify a language or region to disambiguate the input data and properly interpret the intended semantic value.

Example 30.6. Localized string

```
$locale = new Zend_Locale('de');
$mystring = "The boat is 1,234.50 long.";
$unit = new Zend_Measure_Length($mystring, Zend_Measure_Length::STANDARD, $locale)
echo $unit; // outputs "1.234 m"


$mystring = "The boat is 1,234.50 long.";
$unit = new Zend_Measure_Length($mystring, Zend_Measure_Length::STANDARD, 'en_US')
echo $unit; // outputs "1234.50 m"
```

Since Zend Framework 1.6 `Zend_Measure` does also support the usage of an application wide locale. You can simply set a `Zend_Locale` instance to the registry like shown below. With this notation you can forget about setting the locale manually with each instance when you want to use the same locale multiple times.

```
// in your bootstrap file
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// somewhere in your application
$length = new Zend_Measure_Length(Zend_Measure_Length::METER();
```

Outputting measurements

Measurements can be output in a number of different ways.

`Automatic output`

`Outputting values`

`Output with unit of measurement`

`Output as localized string`

Automatic output

`Zend_Measure` supports outputting of strings automatically.

Example 30.7. Automatic output

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89 Meter";
$unit = new Zend_Measure_Length($mystring,Zend_Measure_Length::STANDARD, $locale);

echo $unit;
```

Measurement output

Output can be achieved simply by using `echo` [http://php.net/echo] or `print` [http://php.net/print] .

Outputting values

The value of a measurement can be output using `getValue()`.

Example 30.8. Output a value

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89 Meter";
$unit = new Zend_Measure_Length($mystring,Zend_Measure_Length::STANDARD, $locale);

echo $unit->getValue();
```

The `getValue()` method accepts an optional parameter '`round`' which allows to define a precision for the generated output. The standard precision is '`2`'.

Output with unit of measurement

The function `getType()` returns the current unit of measurement.

Example 30.9. Outputting units

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89";
$unit = new Zend_Measure_Weight($mystring,Zend_Measure_Weight::POUND, $locale);

echo $unit->getType();
```

Output as localized string

Outputtig a string in a format common in the users' country is usually desirable. For example, the measurement "1234567.8" would become "1.234.567,8" for Germany. This functionality will be supported in a future release.

Manipulating Measurements

Parsing and normalization of input, combined with output to localized notations makes data accessible to users in different locales. Many additional methods exist in `Zend_Measure_*` components to manipulate and work with this data, after it has been normalized.

- `Convert`
- `Add and subtract`
- `Compare to boolean`
- `Compare to greater/smaller`
- `Manually change values`
- `Manually change types`

Convert

Probably the most important feature is the conversion into different units of measurement. The conversion of a unit can be done any number of times using the method `convertTo()`. Units of measurement can only be converted to other units of the same type (class). Therefore, it is not possible to convert (e.g.) a length into a weight, which would might encourage poor programming practices and allow errors to propagate without exceptions.

The `convertTo` method accepts an optional parameter. With this parameter you can define an precision for the returned output. The standard precision is '`2`'.

Example 30.10. Convert

```
$locale = new Zend_Locale('de');
$mystring = "1.234.567,89";
$unit = new Zend_Measure_Weight($mystring,'POND', $locale);

print "Kilo:".$unit->convertTo('KILOGRAM');

// constants are considered "better practice" than strings
print "Ton:".$unit->convertTo(Zend_Measure_Weight::TON);

// define a precision for the output
print "Ton:".$unit->convertTo(Zend_Measure_Weight::TON, 3);
```

Add and subtract

Measurements can be added together using `add()` and subtracted using `sub()`. Each addition will create a new object for the result. The actual object will never be changed by the class. The new object will be of the same type as the originating object. Dynamic objects support a fluid style of programming, where complex sequences of operations can be nested without risk of side-effects altering the input objects.

Example 30.11. Adding units

```
// Define objects
$unit = new Zend_Measure_Length(200, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

// Add $unit2 to $unit
$sum = $unit->add($unit2);

echo $sum; // outputs "300 cm"
```

Automatic conversion

Adding one object to another will automatically convert it to the correct unit. It is not neccessary to call `convertTo()` before adding different units.

Example 30.12. Subtract

Subtraction of measurements works just like addition.

```
// Define objects
$unit = new Zend_Measure_Length(200, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

// Subtract $unit2 from $unit
$sum = $unit->sub($unit2);

echo $sum;
```

Compare

Measurements can also be compared, but without automatic unit conversion. Thus, `equals()` returns `TRUE`, only if both the value and the unit of measure are identical.

Example 30.13. Different measurements

```
// Define measurements
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

if ($unit->equals($unit2)) {
    print "Both measurements are identical";
} else {
    print "These are different measurements";
}
```

Example 30.14. Identical measurements

```
// Define measurements
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);

$unit2->setType(Zend_Measure_Length::CENTIMETER);

if ($unit->equals($unit2)) {
    print "Both measurements are identical";
} else {
    print "These are different measurements";
}
```

Compare

To determine if a measurement is less than or greater than another, use `compare()`, which returns 0, -1 or 1 depending on the difference between the two objects. Identical measurements will return 0. Lesser ones will return a negative, greater ones a positive value.

Example 30.15. Difference

```
$unit = new Zend_Measure_Length(100, Zend_Measure_Length::CENTIMETER);
$unit2 = new Zend_Measure_Length(1, Zend_Measure_Length::METER);
$unit3 = new Zend_Measure_Length(1.2, Zend_Measure_Length::METER);

print "Equal:".$unit2->compare($unit);
print "Lesser:".$unit2->compare($unit3);
print "Greater:".$unit3->compare($unit2);
```

Manually change values

To change the value of a measurement explicitly, use `setValue()`. to overwrite the current value. The parameters are the same as the constructor.

Example 30.16. Changing a value

```
$locale = new Zend_Locale('de_AT');
$unit = new Zend_Measure_Length(1,Zend_Measure_Length::METER);

$unit->setValue(1.2);
echo $unit;

$unit->setValue(1.2, Zend_Measure_Length::KILOMETER);
echo $unit;

$unit->setValue("1.234,56", Zend_Measure_Length::MILLIMETER,$locale);
echo $unit;
```

Manually change types

To change the type of a measurement without altering its value use `setType()`.

Example 30.17. Changing the type

```
$unit = new Zend_Measure_Length(1,Zend_Measure_Length::METER);
echo $unit; // outputs "1 m"

$unit->setType(Zend_Measure_Length::KILOMETER);
echo $unit; // outputs "1000 km"
```

Types of measurements

All supported measurement types are listed below, each with an example of the standard usage for such measurements.

Table 30.1. List of measurement types

Typ	Class	Standardunit	Description
Acceleration	Zend_Measure_Acceleration	Meter per square second | `m/s²`	`Zend_Measure_Acceleration` covers the physical factor of acceleration.
Angle	Zend_Measure_Angle	Radiant | `rad`	`Zend_Measure_Angle` covers angular dimensions.
Area	Zend_Measure_Area	Square meter | `m²`	`Zend_Measure_Area` covers square measures.
Binary	Zend_Measure_Binary	Byte | `b`	`Zend_Measure_Binary` covers binary convertions.
Capacitance	Zend_Measure_Capacitance	Farad | `F`	`Zend_Measure_Capacitance` covers physical factor of capacitance.
Cooking volumes	Zend_Measure_Cooking_Volume	Cubic meter | `m³`	`Zend_Measure_Cooking_Volume` covers volumes which are used for cooking or written in cookbooks.
Cooking weights	Zend_Measure_Cooking_Weight	Gram | `g`	`Zend_Measure_Cooking_Weight` covers the weights which are used for cooking or written in cookbooks.
Current	Zend_Measure_Current	Ampere | `A`	`Zend_Measure_Current` covers the physical factor of current.
Density	Zend_Measure_Density	Kilogram per cubic meter | `kg/m³`	`Zend_Measure_Density` covers the physical factor of density.
Energy	Zend_Measure_Energy	Joule | `J`	`Zend_Measure_Energy` covers the physical factor of energy.
Force	Zend_Measure_Force	Newton | `N`	`Zend_Measure_Force` covers the physical factor of force.
Flow (mass)	Zend_Measure_Flow_Mass	Kilogram per second | `kg/s`	`Zend_Measure_Flow_Mass` covers the physical factor of flow rate. The weight of the flowing mass is used as reference point within this class.
Flow (mole)	Zend_Measure_Flow_Mole	Mole per second | `mol/s`	`Zend_Measure_Flow_Mole` covers the physical factor of flow rate. The density of the flowing mass is used as reference point within this class.
Flow (volume)	Zend_Measure_Flow_Volume	Cubic meter per second | `m³/s`	`Zend_Measure_Flow_Volume` covers the physical factor of flow rate. The volume of the flowing mass is used as reference point within this class.
Frequency	Zend_Measure_Frequency	Hertz | `Hz`	`Zend_Measure_Frequency` covers the physical factor of frequency.
Illumination	Zend_Measure_Illumination	Lux | `lx`	`Zend_Measure_Illumination` covers the physical factor of light density.
Length	Zend_Measure_Length	Meter | `m`	`Zend_Measure_Length` covers the physical factor of length.
Lightness	Zend_Measure_Lightness	Candela per square meter | `cd/m²`	`Zend_Measure_Ligntness` covers the physical factor of light energy.

Typ	Class	Standardunit	Description
Number	Zend_Measure_Number	Decimal | `(10)`	`Zend_Measure_Number` converts between number formats.
Power	Zend_Measure_Power	Watt | `W`	`Zend_Measure_Power` covers the physical factor of power.
Pressure	Zend_Measure_Pressure	Newton per square meter | `N/m²`	`Zend_Measure_Pressure` covers the physical factor of pressure.
Speed	Zend_Measure_Speed	Meter per second | `m/s`	`Zend_Measure_Speed` covers the physical factor of speed.
Temperature	Zend_Measure_Temperature	Kelvin | `K`	`Zend_Measure_Temperature` covers the physical factor of temperature.
Time	Zend_Measure_Time	Second | `s`	`Zend_Measure_Time` covers the physical factor of time.
Torque	Zend_Measure_Torque	Newton meter | `Nm`	`Zend_Measure_Torque` covers the physical factor of torque.
Viscosity (dynamic)	Zend_Measure_Viscosity_Dynamic	Kilogram per meter second | `kg/ms`	`Zend_Measure_Viscosity_Dynamic` covers the physical factor of viscosity. The weight of the fluid is used as reference point within this class.
Viscosity (kinematic)	Zend_Measure_Viscosity_Kinematic	Square meter per second | `m²/s`	`Zend_Measure_Viscosity_Kinematic` covers the physical factor of viscosity. The distance of the flown fluid is used as reference point within this class.
Volume	Zend_Measure_Volume	Cubic meter | `m³`	`Zend_Measure_Volume` covers the physical factor of volume (content).
Weight	Zend_Measure_Weight	Kilogram | `kg`	`Zend_Measure_Weight` covers the physical factor of weight.

Hints for Zend_Measure_Binary

Some popular binary conventions, include terms like kilo-, mega-, giga, etc. in normal language use imply base 10, such as 1000 or 10^3. However, in the binary format for computers these terms have to be seen for a convertion factor of 1024 instead of 1000. To preclude confusions a few years ago the notation BI was introduced. Instead of kilobyte, kibibyte for kilo-binary-byte should be used.

In the class BINARY both notations can be found, such as `KILOBYTE = 1024 - binary conputer conversion KIBIBYTE = 1024 - new notation KILO_BINARY_BYTE = 1024 - new`, or the notation, long format `KILOBYTE_SI = 1000 - SI notation for kilo (1000)`. DVDs for example are marked with the SI-notation, but almost all harddisks are marked in computer binary notation.

Hints for Zend_Measure_Number

The best known number format is the decimal system. Additionaly this class supports the octal system, the hexadecimal system, the binary system, the roman number system and some other less popular systems. Note that only the decimal part of numbers is handled. Any fractional part will be stripped.

Roman numbers

For the roman numbersystem digits greater 4000 are supported. In reality these digits are shown with a crossbeam on top of the digit. As the crossbeam can not be shown within the computer, an underline has to be used instead of it.

```
$great = '_X';
$locale = new Zend_Locale('en');
$unit = new Zend_Measure_Number($great,Zend_Measure_Number::ROMAN, $locale);

// convert to the decimal system
echo $unit->convertTo(Zend_Measure_Number::DECIMAL);
```

Chapter 31. Zend_Memory

Overview

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 Zend_Cache backends as storage providers.

Example 31.1. Using Zend_Memory component

`Zend_Memory::factory()` instantiates the memory manager object with specified backend options.

```
$backendOptions = array(
    'cache_dir' => './tmp/' // Directory where to put the swapped memory blocks
);

$memoryManager = Zend_Memory::factory('File', $backendOptions);

$loadedFiles = array();

for ($count = 0; $count < 10000; $count++) {
    $f = fopen($fileNames[$count], 'rb');
    $data = fread($f, filesize($fileNames[$count]));
    $fclose($f);

    $loadedFiles[] = $memoryManager->create($data);
}

echo $loadedFiles[$index1]->value;

$loadedFiles[$index2]->value = $newValue;

$loadedFiles[$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:

```
$memObject = $memoryManager->create($data);

echo $memObject->value;

$memObject->value = $newValue;

$memObject->value[$index] = '_';

echo ord($memObject->value[$index1]);

$memObject->value = substr($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 the section called “MinSize” for more details.

Memory Manager

Creating a Memory Manager

You can create new a memory manager (`Zend_Memory_Manager` object) using the `Zend_Memory::factory($backendName [, $backendOprions])` method.

The first argument `$backendName` is a string that names one of the backend implementations supported by Zend_Cache.

The second argument `$backendOptions` is an optional backend options array.

```
$backendOptions = array(
    'cache_dir' => './tmp/' // Directory where to put the swapped memory blocks
);

$memoryManager = Zend_Memory::factory('File', $backendOptions);
```

Zend_Memory uses Zend_Cache backends as storage providers.

You may use the special name '`None`' as a backend name, in addition to standard Zend_Cache backends.

```
$memoryManager = Zend_Memory::factory('None');
```

If you use '`None`' as the backend name, then the memory manager never swaps memory blocks. This is useful if you know that memory is not limited or the overall size of objects never reaches the memory limit.

The '`None`' backend doesn't need any option specified.

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_Manager::create([$data])` method:

```
$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_Manager::createLocked([$data])` method:

```
$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:

```
function foo()
{
    global $memoryManager, $memList;

    ...

    $memObject1 = $memoryManager->create($data1);
    $memObject2 = $memoryManager->create($data2);
    $memObject3 = $memoryManager->create($data3);

    ...

    $memList[] = $memObject3;

    ...

    unset($memObject2); // $memObject2 is destroyed here

    ...
    // $memObject1 is destroyed here
    // but $memObject3 object is still referenced by $memList and is not destroyed
}
```

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:

```
$oldLimit = $memoryManager->getMemoryLimit();  // Get memory limit in bytes
$memoryManager->setMemoryLimit($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:

```
$oldMinSize = $memoryManager->getMinSize();  // Get MinSize in bytes
$memoryManager->setMinSize($newSize);        // Set MinSize limit in bytes
```

The default minimum size value is 16KB (16384 bytes).

Memory Objects

Movable

Create movable memory objects using the `create([$data])` method of the memory manager:

```
$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:

```
$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:

```
$memObject = $memoryManager->create($data);

echo $memObject->value;

$memObject->value = $newValue;

$memObject->value[$index] = '_';

echo ord($memObject->value[$index1]);

$memObject->value = substr($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

```
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:

```
$memObject = $memoryManager->create($data);

$value = &$memObject->getRef();

for ($count = 0; $count < strlen($value); $count++) {
    $char = $value[$count];
    ...
}
```

touch() method

```
public function touch();
```

The `touch()` method should be used in common with `getRef()`. It signals that object value has been changed:

```
$memObject = $memoryManager->create($data);
...

$value = &$memObject->getRef();

for ($count = 0; $count < strlen($value); $count++) {
    ...
    if ($condition) {
        $value[$count] = $char;
    }
    ...
}

$memObject->touch();
```

lock() method

```
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 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:

```
$memObject1 = $memoryManager->create($data1);
$memObject2 = $memoryManager->create($data2);
...

$memObject1->lock();
$memObject2->lock();

$value1 = &$memObject1->getRef();
$value2 = &$memObject2->getRef();

for ($count = 0; $count < strlen($value2); $count++) {
    $value1 .= $value2[$count];
}

$memObject1->touch();
$memObject1->unlock();
$memObject2->unlock();
```

unlock() method

```
public function unlock();
```

`unlock()` method unlocks object when it's no longer necessary to be locked. See the example above.

isLocked() method

```
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.

Chapter 32. Zend_Mime

Zend_Mime

Introduction

`Zend_Mime` is a support class for handling multipart MIME messages. It is used by `Zend_Mail` and `Zend_Mime_Message` and may be used by applications requiring MIME support.

Static Methods and Constants

`Zend_Mime` provides a simple set of static helper methods to work with MIME:

- `Zend_Mime::isPrintable()`: Returns TRUE if the given string contains no unprintable characters, FALSE otherwise.
- `Zend_Mime::encodeBase64()`: Encodes a string into base64 encoding.
- `Zend_Mime::encodeQuotedPrintable()`: Encodes a string with the quoted-printable mechanism.

`Zend_Mime` defines a set of constants commonly used with MIME Messages:

- `Zend_Mime::TYPE_OCTETSTREAM`: 'application/octet-stream'
- `Zend_Mime::TYPE_TEXT`: 'text/plain'
- `Zend_Mime::TYPE_HTML`: 'text/html'
- `Zend_Mime::ENCODING_7BIT`: '7bit'
- `Zend_Mime::ENCODING_8BIT`: '8bit'
- `Zend_Mime::ENCODING_QUOTEDPRINTABLE`: 'quoted-printable'
- `Zend_Mime::ENCODING_BASE64`: 'base64'
- `Zend_Mime::DISPOSITION_ATTACHMENT`: 'attachment'
- `Zend_Mime::DISPOSITION_INLINE`: 'inline'

Instantiating Zend_Mime

When Instantiating a `Zend_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` 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.

Zend_Mime_Message

Introduction

`Zend_Mime_Message` represents a MIME compliant message that can contain one or more seperate 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 (experimental). 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` Object to generate a boundary. If you need to define the boundary or want to change the behaviour of the `Zend_Mime` object used by `Zend_Mime_Message`, you can instantiate the `Zend_Mime` object yourself and then register it to `Zend_Mime_Message`. Usually you will not need to do this. `->setMime(Zend_Mime $mime)` sets a special instance of `Zend_Mime` to be used by this `Zend_Mime_Message`

`->getMime()` returns the instance of `Zend_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 (experimental)

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()`

Zend_Mime_Part

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:

```
public $type = ZMime::TYPE_OCTETSTREAM;
public $encoding = ZMime::ENCODING_8BIT;
public $id;
public $disposition;
public $filename;
public $description;
public $charset;
```

Methods for rendering the message part to a string

`getContent()` returns the encoded content of the MimePart as a string using the encoding specified in the attribute $encoding. Valid values are ZMime::ENCODING_* Characterset conversions are not performed.

`getHeaders()` returns the Mime-Headers for the MimePart as generated from the information in the publicly accessable 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.

Chapter 33. Zend_OpenId

Introduction

Zend_OpenId is a Zend Framework component that provides a simple API for building OpenID-enabled sites and identity providers.

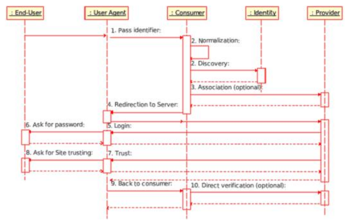
What is OpenID?

OpenID is a set of protocols for user-centric digital identities. These protocols allow to create an identity online, using an identity provider. This identity can be used anywhere that OpenID is supported. Using OpenID-enabled sites, web users do not need to remember traditional authentication tokens such as username and password. All OpenID-enabled sites accept a single OpenID identity. This identity is typically a URL. It may be the URL of the user's personal page, blog or other resource that may provide additional information about them. No more need for many passwords and different user names - just one identifier for all Internet services. OpenID is an open, decentralized, and free user centric solution. A user may choose which OpenID provider to use, or even create their own personal identity server. No central authority is needed to approve or register OpenID-enabled sites or identity providers.

For more information about OpenID visit OpenID official site [http://www.openid.net/] and look into the OpenID Book by Rafeeq Rehman [http://www.openidbook.com/].

How Does it Work?

The main purpose of the Zend_OpenId components is to implement an OpenID authentication protocol as described in the following diagram:



1. Authentication is initiated by the end-user, who passes their OpenID identifier to the OpenID consumer through a User-Agent.

2. The OpenID consumer performs normalization of the user-supplied identifier, and discovery on it. As result, it gets the following: a claimed identifier, OpenID provider URL and an OpenID protocol version.

3. The OpenID client establishes an optional association with the server using Diffie-Hellman keys. As a result, both parties get a common "shared secret" that is used for signing and verification of the following (subsequent) messages.

4. The OpenID consumer redirects the User-Agent to the OpenID provider's URL with an OpenID authentication request.

5. The OpenID Provider checks if the user-Agent is already authenticated and offers to do so if needed.

6. The end user enters the required password.

7. The OpenID Provider checks if it is allowed to pass the user identity to the given consumer, and asks the user if needed.
8. The end user allows or disallows passing his identity.
9. The OpenID Provider redirects the User-Agent back to the OpenID consumer with an "authentication approved" or "failed" request.
10. The OpenID consumer verifies the information received from the provider by using the "shared secret" it got on step 3 or by sending additional direct request to the OpenID provider.

Zend_OpenId Structure

`Zend_OpenId` consists of two sub packages. The first one is `Zend_OpenId_Consumer` for developing OpenID-enabled sites and the second `Zend_OpenId_Provider` for developing OpenID servers. They are completely independent of each other and may be used separately.

The only common parts of these sub packages are the OpenID Simple Registration Extension implemented by `Zend_OpenId_Extension_Sreg` class and the set of utility functions implemented by `Zend_OpenId` class.

Note

`Zend_OpenId` takes advantage of the GMP extension [http://php.net/gmp], where available. Consider enabling the GMP extension for better performance when using `Zend_OpenId`.

Supported Standards

The `Zend_OpenId` component conforms to the following standards:

- OpenID Authentication protocol version 1.1
- OpenID Authentication protocol version 2.0 draft 11
- OpenID Simple Registration Extension version 1.0
- OpenID Simple Registration Extension version 1.1 draft 1

Zend_OpenId_Consumer Basics

`Zend_OpenId_Consumer` is used to implement the OpenID authentication schema on web sites.

OpenID Authentication

From a site developers point of view, the OpenID authentication process consists of three steps:

1. Show OpenID authentication form.
2. Accept OpenID identity and pass it to the OpenID provider.
3. Verify response from the OpenID provider.

In actual fact the OpenID authentication protocol performs more steps, but most of them are encapsulated inside the `Zend_OpenId_Consumer`, and they are transparent to the developer.

The OpenID authentication process is initiated by the end-user by filling in their identification into the appropriate form and submiting it. The following example shows a simple form that accepts an OpenID identifier. Note that the example shows only a login.

Example 33.1. The Simple OpenID Login form

```
<html><body>
<form method="post" action="example-1_2.php"><fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier">
<input type="submit" name="openid_action" value="login">
</fieldset></form></body></html>
```

On submit this form passes the OpenID identity to the following PHP script that performs a second step of authentication. The only thing the PHP script needs to do in this step is call the `Zend_OpenId_Consumer::login()` method. The first argument of this method is an accepted OpenID identity and the second is a URL of a script that handles the third and last step of authentication.

Example 33.2. The Authentication Request Handler

```
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->login($_POST['openid_identifier'], 'example-1_3.php')) {
    die("OpenID login failed.");
}
```

The `Zend_OpenId_Consumer::login()` performs discovery on a given identifier and on success, finds out the address of the identity provider and its local identifier. Then, it creates an association to the given provider so that both the site and provider know the same secret that is used to sign the following messages. Then it passes an authentication request to the provider. Note this request redirects the end-user's web browser to an OpenID server site, where users are able to continue the authentication process.

An OpenID Server usually asks users for; their password (if they weren't previously logged-in), if the user trusts this site and what information may be returned to the site. These interactions are not visible to the OpenID-enabled site so there is no what for it to get the user's password or other information that was not opened.

On success, `Zend_OpenId_Consumer::login()` never returns, because it performs an HTTP redirection, however in case of error it may return false. Errors may occure due to an invalid identity, dead provider, communication error, etc

The third step of authentication is initiated by a response from the OpenID provider, after it has already authenticated the user's password. This response is passed indirectly, as an HTTP redirection of the end-user's web browser. And the only thing that site must do is to check if this response is valid.

Example 33.3. The Authentication Response Verifier

```
$consumer = new Zend_OpenId_Consumer();
if ($consumer->verify($_GET, $id)) {
    echo "VALID " . htmlspecialchars($id);
} else {
    echo "INVALID " . htmlspecialchars($id);
}
```

This check is performed using the `Zend_OpenId_Consumer::verify` method, that takes the whole array of the HTTP request's arguments and checks if this response is properly signed by an appropriate OpenID provider. It also may assign the claimed OpenID identity that was entered by end-user in the first step into the second (optional) argument.

Combine all Steps in One Page

The following example combines all three steps together. It doesn't provide any additional functionality. The only advantage is that now developers don't need to specify any URL's of scripts that handle the next step. By default, all steps use the same URL. However, the script now includes a dispatch code that calls appropriate code for each step of authentication.

Example 33.4. The Complete OpenID Login Script

```
<?php
$status = "";
if (isset($_POST['openid_action']) &&
    $_POST['openid_action'] == "login" &&
    !empty($_POST['openid_identifier'])) {

    $consumer = new Zend_OpenId_Consumer();
    if (!$consumer->login($_POST['openid_identifier'])) {
        $status = "OpenID login failed.<br>";
    }
} else if (isset($_GET['openid_mode'])) {
    if ($_GET['openid_mode'] == "id_res") {
        $consumer = new Zend_OpenId_Consumer();
        if ($consumer->verify($_GET, $id)) {
            $status = "VALID " . htmlspecialchars($id);
        } else {
            $status = "INVALID " . htmlspecialchars($id);
        }
    } else if ($_GET['openid_mode'] == "cancel") {
        $status = "CANCELED";
    }
}
?>
<html><body>
<?php echo "$status<br>";?>
<form method="post"><fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier" value="">
<input type="submit" name="openid_action" value="login">
</fieldset></form></body></html>
```

In addition, this code differenciates between canceled and wrong authentication responses. The provider retuns a canceled responce in cases when an identity provider doesn't know the supplied identity or the user is not logged-in or they don't trust the site. A wrong response assumes that the responce is wrong or incorrectly signed.

Realm

When an OpenID-enabled site passes authentication requests to a provider, it identifies itself with a realm URL. This URL may be considered as a root of a trusted site. If the user trusts the URL they will also trust to matched and subsequent URLs.

By default, the realm URL is automatically set to the URL of the directory where the login script is. This decision is useful for most, but not all cases. Sometimes a whole site and not directory is used, or even a combination of several servers from one domain.

To implement this ability, developers may pass the realm value as a third argument to the `Zend_OpenId_Consumer::login` method. In the following example the single interaction asks for trusted access to all php.net sites.

Example 33.5. Authentication Request for Specified Realm

```
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->login($_POST['openid_identifier'],
                      'example-3_3.php',
                      'http://*.php.net/')) {
    die("OpenID login failed.");
}
```

The example below only implements the second step of authentication, the first and third steps are the same as in the first example.

Immediate Check

In some situations it is necissary to see if a user is already logged-in into a trusted OpenID server without any interaction with the user. The `Zend_OpenId_Consumer::check` method does precisely that. It is executed with exactly the same arguments as `Zend_OpenId_Consumer::login` but it doesn't show the user any OpenID server pages. Therefore from the users point of view it is transparent and it seems as if they never left the site. The third step succeedes if user is already logged-in and trusted to the site otherwise it will fail.

Example 33.6. Immediate Check without Interaction

```
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->check($_POST['openid_identifier'], 'example-4_3.php')) {
    die("OpenID login failed.");
}
```

The example below only implements the second step of authentication, first and third steps are the same as in the first example.

Zend_OpenId_Consumer_Storage

There are three steps to the OpenID authentication procedure, each step is performed by a separate HTTP request. To store information between requests `Zend_OpenId_Consumer` uses internal storage.

Developers may not care about this storage because by default `Zend_OpenId_Consumer` uses file-based storage under /tmp similar to PHP sessions. However, this storage may be not suitable in all cases. Some may want to store information in a database while others may need to use common storage suitable for big web-farms. Fortunately, developers may easily replace the default storage with their own. The only thing to implement is it's own storage class as a child of the `Zend_OpenId_Consumer_Storage` method and pass it as a first argument to the `Zend_OpenId_Consumer` constructor.

The following example demonstrates a simple storage that uses `Zend_Db` as the backend containing three groups of functions. the first is for working with associations, the second is to cache discovery information and the third is to check responce uniqueness. The class is implemented in such a way that it can be easily used with existing or new databases. If necessary, it will create database tables if they don't exist.

Example 33.7. Databse Storage

```
class DbStorage extends Zend_OpenId_Consumer_Storage
{
    private $_db;
    private $_association_table;
    private $_discovery_table;
    private $_nonce_table;

    public function __construct($db,
                                $association_table = "association",
                                $discovery_table = "discovery",
                                $nonce_table = "nonce")
    {
        $this->_db = $db;
        $this->_association_table = $association_table;
        $this->_discovery_table = $discovery_table;
        $this->_nonce_table = $nonce_table;
        $tables = $this->_db->listTables();
        if (!in_array($association_table, $tables)) {
            $this->_db->getConnection()->exec(
                "create table $association_table (" .
                " url     varchar(256) not null primary key," .
                " handle  varchar(256) not null," .
                " macFunc char(16) not null," .
                " secret  varchar(256) not null," .
                " expires timestamp" .
                ")");
        }
        if (!in_array($discovery_table, $tables)) {
            $this->_db->getConnection()->exec(
                "create table $discovery_table (" .
                " id      varchar(256) not null primary key," .
                " realId  varchar(256) not null," .
                " server  varchar(256) not null," .
                " version float," .
                " expires timestamp" .
                ")");
        }
        if (!in_array($nonce_table, $tables)) {
            $this->_db->getConnection()->exec(
                "create table $nonce_table (" .
                " nonce   varchar(256) not null primary key," .
                " created timestamp default current_timestamp" .
                ")");
        }
    }

    public function addAssociation($url,
                                   $handle,
                                   $macFunc,
                                   $secret,
                                  $expires)
    {
        $table = $this->_association_table;
        $secret = base64_encode($secret);
        $this->_db
             ->query('insert into ' .
                     $table (url, handle, macFunc, secret, expires) " .
                     "values ('$url', '$handle', '$macFunc', '$secret', $expires)"
        return true;
    }

    public function getAssociation($url,
                                   &$handle,
                                   &$macFunc,
                                   &$secret,
                                   &$expires)
    {
        $table = $this->_association_table;
        $this->_db->query("delete from $table where expires < " . time());
        $res = $this->_db->fetchRow('select handle, macFunc, secret, expires ' .
                                    "from $table where url = '$url'");
        if (is_array($res)) {
            $handle  = $res['handle'];
            $macFunc = $res['macFunc'];
            $secret  = base64_decode($res['secret']);
            $expires = $res['expires'];
            return true;
        }
        return false;
    }

    public function getAssociationByHandle($handle,
                                           &$url,
                                           &$macFunc,
                                           &$secret,
                                           &$expires)
    {
        $table = $this->_association_table;
        $this->_db->query("delete from $table where expires < " . time());
        $res = $this->_db
                ->fetchRow('select url, macFunc, secret, expires ' .
                           "from $table where handle = '$handle'");
        if (is_array($res)) {
            $url     = $res['url'];
            $macFunc = $res['macFunc'];
            $secret  = base64_decode($res['secret']);
            $expires = $res['expires'];
            return true;
        }
        return false;
    }

    public function delAssociation($url)
    {
        $table = $this->_association_table;
        $this->_db->query("delete from $table where url = '$url'");
        return true;
    }

    public function addDiscoveryInfo($id,
                                      $realId,
                                      $server,
                                      $version,
                                      $expires)
    {
        $table = $this->_discovery_table;
        $this->_db
             ->query("insert into $table (id, realId, server, version, expires) "
                     "values ('$id', '$realId', '$server', $version, $expires)");
        return true;
    }

    public function getDiscoveryInfo($id,
                                      &$realId,
                                      &$server,
                                      &$version,
                                      &$expires)
    {
        $table = $this->_discovery_table;
        $this->_db->query("delete from $table where expires < " . time());
        $res = $this->_db
                    ->fetchRow('select realId, server, version, expires ' .
                               "from $table where id = '$id'");
        if (is_array($res)) {
            $realId  = $res['realId'];
            $server  = $res['server'];
            $version = $res['version'];
            $expires = $res['expires'];
            return true;
        }
        return false;
    }

    public function delDiscoveryInfo($id)
    {
        $table = $this->_discovery_table;
        $this->_db->query("delete from $table where id = '$id'");
        return true;
    }

    public function isUniqueNonce($nonce)
    {
        $table = $this->_nonce_table;
        try {
            $ret = $this->_db
                        ->query("insert into $table (nonce) values ('$nonce')");
        } catch (Zend_Db_Statement_Exception $e) {
            return false;
        }
        return true;
    }

    public function purgeNonces($date=null)
    {
    }
}

$db = Zend_Db::factory('Pdo_Sqlite',
    array('dbname'=>'/tmp/openid_consumer.db'));
$storage = new DbStorage($db);
$consumer = new Zend_OpenId_Consumer($storage);
```

The example doesn't include OpenID authentication code itself, but it is based on the same logic as in the previous or following examples.

Simple Registration Extension

In addition to authentication, the OpenID can be used for light-weight profile exchange. This feature is not covered by OpenID authentication specification but by the OpenID Simple Registration Extension protocol. This protocol allows OpenID-enabled sites to ask for information about an end-user from OpenID providers. Such information may include:

- *nickname* - any UTF-8 string that the end user wants to use as a nickname.
- *email* - the email address of the end user as specified in section 3.4.1 of RFC2822.
- *fullname* - a UTF-8 string representation of the end user's full name.
- *dob* - the end user's date of birth as YYYY-MM-DD. Any values whose representation uses fewer than the specified number of digits should be zero-padded. The length of this value must always be 10. If the end user does not want to reveal any particular component of this value, it must be set to zero. For instance, if a end user wants to specify that his date of birth is in 1980, but not the month or day, the value returned shall be "1980-00-00".
- *gender* - the end user's gender, "M" for male, "F" for female.
- *postcode* - UTF-8 string that should conform to the end user's country's postal system.
- *country* - the End User's country of residence as specified by ISO3166.
- *language* - end User's preferred language as specified by ISO639.
- *timezone* - ASCII string from TimeZone database. For example, "Europe/Paris" or "America/Los_Angeles".

An OpenID-enabled web site may ask for any combination of these fields. It may also strictly require some information and allow end-users to provide or hide other information. The following example creates an object of the `Zend_OpenId_Extension_Sreg` class that requires a *nickname* and optionally ask for *email* and *fullname*.

Example 33.8. Sending Requests with a Simple Registration Extension

```
$sreg = new Zend_OpenId_Extension_Sreg(array(
    'nickname'=>true,
    'email'=>false,
    'fullname'=>false), null, 1.1);
$consumer = new Zend_OpenId_Consumer();
if (!$consumer->login($_POST['openid_identifier'],
                      'example-6_3.php',
                      null,
                      $sreg)) {
    die("OpenID login failed.");
}
```

As you can see the `Zend_OpenId_Extension_Sreg` constructor accepts an array of asked fields. This array has the names of fields as indexes and requirements flag as values. *true* means the field is required and *false* means the field is optional. The `Zend_OpenId_Consumer::login` accepts extensions or list of extensions as a fourth argument.

On the third step of authentication, the `Zend_OpenId_Extension_Sreg` object should be passed to `Zend_OpenId_Consumer::verify`. Then on successful authentication `Zend_OpenId_Extension_Sreg::getProperties` will return an associative array of requested fields.

Example 33.9. Verifying Responses with a Simple Registration Extension

```
$sreg = new Zend_OpenId_Extension_Sreg(array(
    'nickname'=>true,
    'email'=>false,
    'fullname'=>false), null, 1.1);
$consumer = new Zend_OpenId_Consumer();
if ($consumer->verify($_GET, $id, $sreg)) {
    echo "VALID " . htmlspecialchars($id) ."<br>\n";
    $data = $sreg->getProperties();
    if (isset($data['nickname'])) {
        echo "nickname: " . htmlspecialchars($data['nickname']) . "<br>\n";
    }
    if (isset($data['email'])) {
        echo "email: " . htmlspecialchars($data['email']) . "<br>\n";
    }
    if (isset($data['fullname'])) {
        echo "fullname: " . htmlspecialchars($data['fullname']) . "<br>\n";
    }
} else {
    echo "INVALID " . htmlspecialchars($id);
}
```

If `Zend_OpenId_Extension_Sreg` was created without any arguments, the user code should check for the existence of the required data itself. However, if the object is created with the same list of required fields as on the second step, it will automatically check for the existence of required data. In this case, `Zend_OpenId_Consumer::verify` will return *false* if any of the required fields are missing.

By default, `Zend_OpenId_Extension_Sreg` uses version 1.0, because the specification for version 1.1 is not yet finalized. However, some libraries don't fully support version 1.0. For example, www.my-openid.com requires an SREG namespace in requests which is only available in 1.1. To work with this server, explicitly set the version to 1.1 in the `Zend_OpenId_Extension_Sreg` constructor.

The second argument of the `Zend_OpenId_Extension_Sreg` constructor is a policy URL, that should be provided to the end-user by the identity provider.

Integration with Zend_Auth

Zend Framework provides a special class to support user authentication - `Zend_Auth`. This class can be used together with `Zend_OpenId_Consumer`. The following example shows how `OpenIdAdapter` implements the `Zend_Auth_Adapter_Interface` with the `authenticate` method.This performs an authentication query and verification.

The big difference between this adapter and existing ones, is that it works on two HTTP requests and includes a dispatch code to perform the second or third step of OpenID authentication.

Example 33.10. Zend_Auth Adapter for OpenID

```
<?php
class OpenIdAdapter implements Zend_Auth_Adapter_Interface {
    private $_id = null;

    public function __construct($id = null) {
        $this->_id = $id;
    }

    public function authenticate() {
        $id = $this->_id;
        if (!empty($id)) {
            $consumer = new Zend_OpenId_Consumer();
            if (!$consumer->login($id)) {
                $ret = false;
                $msg = "Authentication failed.";
            }
        } else {
            $consumer = new Zend_OpenId_Consumer();
            if ($consumer->verify($_GET, $id)) {
                $ret = true;
                $msg = "Authentication successful";
            } else {
                $ret = false;
                $msg = "Authentication failed";
            }
        }
        return new Zend_Auth_Result($ret, $id, array($msg));
    }
}

$status = "";
$auth = Zend_Auth::getInstance();
if ((isset($_POST['openid_action']) &&
     $_POST['openid_action'] == "login" &&
     !empty($_POST['openid_identifier'])) ||
    isset($_GET['openid_mode'])) {
    $adapter = new OpenIdAdapter(@$_POST['openid_identifier']);
    $result = $auth->authenticate($adapter);
    if ($result->isValid()) {
        Zend_OpenId::redirect(Zend_OpenId::selfURL());
    } else {
        $auth->clearIdentity();
        foreach ($result->getMessages() as $message) {
            $status .= "$message<br>\n";
        }
    }
} else if ($auth->hasIdentity()) {
    if (isset($_POST['openid_action']) &&
        $_POST['openid_action'] == "logout") {
        $auth->clearIdentity();
    } else {
        $status = "Yoy are logged-in as " . $auth->getIdentity() . "<br>\n";
    }
}
?>
<html><body>
<?php echo htmlspecialchars($status);?>
<form method="post"><fieldset>
<legend>OpenID Login</legend>
<input type="text" name="openid_identifier" value="">
<input type="submit" name="openid_action" value="login">
<input type="submit" name="openid_action" value="logout">
</fieldset></form></body></html>
```

With `Zend_Auth` the end-user's identity is saved in the session's data. It may be checked with `Zend_Auth::hasIdentity` and `Zend_Auth::getIdentity`.

Integration with Zend_Controller

Finally a couple of words about integration into Model-View-Controller applications. Such Zend Framework applications are implemented using the `Zend_Controller` class and they use objects of the `Zend_Controller_Response_Http` class to prepare HTTP responses and send them back to the end user's web-browser.

`Zend_OpenId_Consumer` doesn't provide any GUI capabilities but it performs HTTP redirections on success of `Zend_OpenId_Consumer::login` and `Zend_OpenId_Consumer::check`. These redirections, may work incorrectly or not work at all if some data was already sent to the web-browser. To properly perform HTTP redirection in MVC code the real `Zend_Controller_Response_Http` should be sent to `Zend_OpenId_Consumer::login` or `Zend_OpenId_Consumer::check` as the last argument.

Zend_OpenId_Provider

The `Zend_OpenId_Provider` is used to implement OpenID servers. This chapter provides very basic examples demonstrating how to build a working server. However, for implementation of a production OpenID server (like www.myopenid.com [http://www.myopenid.com]) you may be required to deal with more complex issues.

Quick Start

The following identity includes the code for creating a user account using `Zend_OpenId_Provider::register`. The link element with `rel="openid.server"` points to our own server script. If you submit this identity to an OpenID-enabled site, it will perform authentication on this server.

The code before <html> is just a trick that automatically creates a user account. You wont need such code when using real identities.

Example 33.11. The Identity

```
<?php
define("TEST_SERVER", Zend_OpenId::absoluteURL("example-8.php"));
define("TEST_ID", Zend_OpenId::selfURL());
define("TEST_PASSWORD", "123");
$server = new Zend_OpenId_Provider();
if (!$server->hasUser(TEST_ID)) {
    $server->register(TEST_ID, TEST_PASSWORD);
}
?>
<html><head>
<link rel="openid.server" href="<?php echo TEST_SERVER;?>" />
</head><body>
<?php echo TEST_ID;?>
</body></html>
```

The following identity server script handles two kinds of requests from OpenID-enabled sites (for association and authentication). Both of them are handled by the same method `Zend_OpenId_Provider::handle`. The two arguments to `Zend_OpenId_Provider` are URLs of login and trust pages, these ask for interaction from the end-user.

On success, the method `Zend_OpenId_Provider::handle` returns a string that should be passed back to the OpenID-enabled site. On failure, it returns `false` - in this example it will return a HTTP 403 response. You will get it if you try to open this page by web-browser, because it sends a non-OpenID conformed request.

Example 33.12. Simple Identity Provider

```
$server = new Zend_OpenId_Provider("example-8-login.php",
                                   "example-8-trust.php");
$ret = $server->handle();
if (is_string($ret)) {
    echo $ret;
} else if ($ret !== true) {
    header('HTTP/1.0 403 Forbidden');
    echo 'Forbidden';
}
```

It is a good idea to use a secure connection (HTTPS) for this and especially for the following interactive scripts, to prevent password disclosure.

The following script implements a login screen for an identity server `Zend_OpenId_Provider` and redirects to this page when a required user has not yet logged-in. On this page, users enter a password to login.

You should use the password "123" that was used during a tricky user registration from an identity script.

On submit, the script calls `Zend_OpenId_Provider::login` with the accepted end-user's identity and password, then redirects it back to the main identity provider's script. On success, the `Zend_OpenId_Provider::login` establishes a session between the end-user and the identity-provider and stores the information about logged-in user. So, all following requests from the same end-user won't require login procedure (even if they come from another OpenID enabled web-site).

Note that this session is between end-user and identity provider only. OpenID enabled sites know nothing about it.

Example 33.13. Simple Login Screen

```
<?php
$server = new Zend_OpenId_Provider();

if ($_SERVER['REQUEST_METHOD'] == 'POST' &&
    isset($_POST['openid_action']) &&
    $_POST['openid_action'] === 'login' &&
    isset($_POST['openid_identifier']) &&
    isset($_POST['openid_password'])) {
    $server->login($_POST['openid_identifier'],
                   $_POST['openid_password']);
    Zend_OpenId::redirect("example-8.php", $_GET);
}
?>
<html><body>
<form method="post"><fieldset>
<legend>OpenID Login</legend>
<table border=0>
<tr><td>Name:</td><td><input type="text" name="openid_identifier"
value="<?php
echo htmlspecialchars($_GET['openid_identity']);
?>"></td></tr>
<tr><td>Password:</td><td><input type="text"
name="openid_password" value=""></td></tr>
<tr><td>&nbsp;</td><td><input type="submit"
name="openid_action" value="login"></td></tr>
</table></fieldset></form></body></html>
```

The fact that the user is logged-in doesn't mean that the authentication must succeed. The user may decide to trust or not to trust particular OpenID enabled sites. The following trust screen allows the end-user to make that choise. This choise may be done only for current requests or "forever". In the last case information about trusted/untrusted sites is stored in an internal database and all following authentication requests from this site will be handled automatically, without user interaction.

Example 33.14. Simple Trust Screen

```
<?php
$server = new Zend_OpenId_Provider();

if ($_SERVER['REQUEST_METHOD'] == 'POST' &&
    isset($_POST['openid_action']) &&
    $_POST['openid_action'] === 'trust') {

    if (isset($_POST['allow'])) {
        if (isset($_POST['forever'])) {
            $server->allowSite($server->getSiteRoot($_GET));
        }
        $server->respondToConsumer($_GET);
    } else if (isset($_POST['deny'])) {
        if (isset($_POST['forever'])) {
            $server->denySite($server->getSiteRoot($_GET));
        }
        Zend_OpenId::redirect($_GET['openid_return_to'],
                              array('openid.mode'=>'cancel'));
    }
}
?>
<html><body>
<p>A site identifying as
<a href="<?php echo htmlspecialchars($server->getSiteRoot($_GET));?>">
<?php echo htmlspecialchars($server->getSiteRoot($_GET));?></a>
has asked us for confirmation that
<a href="<?php echo htmlspecialchars($server->getLoggedInUser());?>">
<?php echo htmlspecialchars($server->getLoggedInUser());?></a>
is your identity URL.</p>
<form method="post">
<input type="checkbox" name="forever">
<label for="forever">forever</label><br>
<input type="hidden" name="openid_action" value="trust">
<input type="submit" name="allow" value="Allow">
<input type="submit" name="deny" value="Deny">
</form></body></html>
```

Production OpenID servers usually support Simple Registration Extension that allows consumers to ask some information about user from provider. In this case the trust page is usually extended with ability to enter requested fields or to select user profile.

Combine all together

It is possible to combine all provider functions in one script. In this case login and trust URLs are omitted, and `Zend_OpenId_Provider` assumes that they point to the same page with additional "openid.action" GET argument.

The following example is not complete. It doesn't provide GUI for end-user like it should, but performs automatic login and trusting instead. It is done just to simplify the example, and real server must include code from previous examples.

Example 33.15. All together

```
$server = new Zend_OpenId_Provider();

define("TEST_ID", Zend_OpenId::absoluteURL("example-9-id.php"));
define("TEST_PASSWORD", "123");

if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'login') {
    $server->login(TEST_ID, TEST_PASSWORD);
    unset($_GET['openid_action']);
    Zend_OpenId::redirect(Zend_OpenId::selfUrl(), $_GET);
} else if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'trust') {
    unset($_GET['openid_action']);
    $server->respondToConsumer($_GET);
} else {
    $ret = $server->handle();
    if (is_string($ret)) {
        echo $ret;
    } else if ($ret !== true) {
        header('HTTP/1.0 403 Forbidden');
        echo 'Forbidden';
    }
}
```

If you compare this example with previous example divided to separate page, in addition to dispatch code you will see only the one difference - `unset($_GET['openid_action'])`. This `unset` is necessary to route next request to main handler.

Simple Registration Extension

The following identity page makes a trick again. It creates new user account and associates it with profile (nickname and password). Such tricks aren't needed in real life where end-user registers on OpenID server and fill-in their profiles, but implementing this GUI is not a subject of this manual.

Example 33.16. Identity with Profile

```
<?php
define("TEST_SERVER", Zend_OpenId::absoluteURL("example-10.php"));
define("TEST_ID", Zend_OpenId::selfURL());
define("TEST_PASSWORD", "123");
$server = new Zend_OpenId_Provider();
if (!$server->hasUser(TEST_ID)) {
    $server->register(TEST_ID, TEST_PASSWORD);
    $server->login(TEST_ID, TEST_PASSWORD);
    $sreg = new Zend_OpenId_Extension_Sreg(array(
        'nickname' =>'test',
        'email' => 'test@test.com'
    ));
    $root = Zend_OpenId::absoluteURL(".");
    Zend_OpenId::normalizeUrl($root);
    $server->allowSite($root, $sreg);
    $server->logout();
}
?>
<html><head>
<link rel="openid.server" href="<?php echo TEST_SERVER;?>" />
</head><body>
<?php echo TEST_ID;?>
</body></html>
```

You should pass this identity to OpenID-enabled site (use Simple Registration Extension example from previous chapter) and it will use the following OpenID server script.

It is a variation from previous "All together" example. It uses the same automatic login mechanism, but it doesn't contain any code for trust page. The user already trusts "forever" to example scripts. This trust was made by `Zend_OpenId_Provider::alowSite` method in identity script. The same method associated profile with trusted URL and this profile will be returned automatically on request from this trusted URL.

The only thing necessary to make Simple Registration Extension work is passing object of `Zend_OpenId_Extension_Sreg` as second argument to `Zend_OpenId_Provider::handle`.

Example 33.17. Provider with SREG

```
$server = new Zend_OpenId_Provider();
$sreg = new Zend_OpenId_Extension_Sreg();

define("TEST_ID", Zend_OpenId::absoluteURL("example-10-id.php"));
define("TEST_PASSWORD", "123");

if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'login') {
    $server->login(TEST_ID, TEST_PASSWORD);
    unset($_GET['openid_action']);
    Zend_OpenId::redirect(Zend_OpenId::selfUrl(), $_GET);
} else if ($_SERVER['REQUEST_METHOD'] == 'GET' &&
    isset($_GET['openid_action']) &&
    $_GET['openid_action'] === 'trust') {
   echo "UNTRUSTED DATA" ;
} else {
    $ret = $server->handle(null, $sreg);
    if (is_string($ret)) {
        echo $ret;
    } else if ($ret !== true) {
        header('HTTP/1.0 403 Forbidden');
        echo 'Forbidden';
    }
}
```

What Else?

Building OpenID servers is less usual tasks then building OpenID-enabled sites, so this manual don't try to cover all `Zend_OpenId_Provider` features as it was done for `Zend_OpenId_Consumer`.

In two words in additional it provides:

- a set of methods to build end-user's GUI interface that allows users to register, manage their trusted sites and profiles.
- an abstraction storage layer to store information about users, their sites and profiles. It also stores associations between provider and OpenID-enabled sites. This layer is very similar to the `Zend_OpenId_Consumer`'s one. It also uses file storage by default but may be substituted with another implementation.
- an abstraction user-association layer that may associate end-user's web browser with logged-in identity

`Zend_OpenId_Provider` doesn't try to cover all possible features that can be implemented by OpenID server (like digital certificates), but it can be easily extended using `Zend_OpenId_Extension`s or by creating a child class.

Chapter 34. Zend_Paginator

Introduction

`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

Usage

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 34.1. Adapters for `Zend_Paginator`

Adapter	Description
Array	Use a PHP array
DbSelect	Use a `Zend_Db_Select` instance
Iterator	Use an `Iterator` [http://www.php.net/~helly/php/ext/spl/interfaceIterator.html] instance
Null	Do not use `Zend_Paginator` to manage data pagination. You can still take advantage of the pagination control feature.

To create an instance of `Zend_Paginator`, you must supply an adapter to the constructor:

```
$paginator = new Zend_Paginator(new Zend_Paginator_Adapter_Array($array));
```

For convenience, you may take advantage of the static `factory()` method for the adapters packaged with Zend Framework:

```
$paginator = Zend_Paginator::factory($array);
```

Note

In the case of the Null adapter, in lieu of a data collection you must supply an item count to its constructor.

Although the instance is technically usable in this state, you'll need to tell the paginator what page number the user requested in order to allow him to advance through the paginated data:

```
$paginator->setCurrentPageNumber($pageNumber);
```

The simplest way to keep track of this value is through a URL. Although we recommend using a `Zend_Controller_Router_Interface`-compatible router to handle this, it is not a requirement.

The following is an example route you might use in an INI configuration file:

```
routes.example.route = articles/:articleName/:pageNumber
routes.example.defaults.controller = articles
routes.example.defaults.action = view
routes.example.defaults.pageNumber = 1
routes.example.reqs.articleName = \w+
routes.example.reqs.pageNumber = \d+
```

With the above route (and using Zend Framework MVC components), you might set the current page number like this:

```
$paginator->setCurrentPageNumber($this->_getParam('pageNumber'));
```

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_View` with the ViewRenderer action helper, the following will work:

```
$this->view->paginator = $paginator;
```

Rendering pages with view scripts

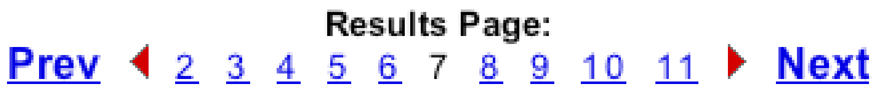
The view script is used to render the page items (if you're using `Zend_Paginator` to do so) and display the pagination control.

Because `Zend_Paginator` implements the SPL interface `IteratorAggregate` [http://www.php.net/~helly/php/ext/spl/interfaceIteratorAggregate.html], looping over your items and displaying them is simple.

```
<html>
<body>
<h1>Example</h1>
<?php if (count($this->paginator)): ?>
<ul>
<?php foreach ($this->paginator as $item): ?>
  <li><?= $item; ?></li>
<?php endforeach; ?>
</ul>
<?php endif; ?>

<?= $this->paginationControl($this->paginator, 'Sliding', 'my_pagination_control.p
</body>
</html>
```

Notice the view helper call near the end. PaginationControl takes the paginator instance, an optional scrolling style, and an optional view partial.

Despite being optional, the latter two parameters are very important. Whereas the view partial is used to determine how the pagination control should *look*, the scrolling style is used to control how it should *behave*. Say the view partial is in the style of a search pagination control, like the one below:



What happens when the user clicks the "next" link a few times? Well, any number of things could happen. The current page number could stay in the middle as you click through (as it does on Yahoo!), or it could advance to the end of the page range and then appear again on the left when the user clicks "next" one more time. The page numbers might even expand and contract as the user advances (or "scrolls") through them (as they do on Google).

There are four scrolling styles packaged with Zend Framework:

Table 34.2. 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.

By setting the default view partial, default scrolling style, and view instance, you can eliminate the calls to PaginationControl completely:

```
Zend_Paginator::setDefaultScrollingStyle('Sliding');
Zend_View_Helper_PaginationControl::setDefaultViewPartial('my_pagination_control.p
$paginator->setView($view);
```

When all of these values are set, you can render the pagination control inside your view script with a simple echo statement:

```
<?= $this->paginator; ?>
```

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 ($this->pageCount): ?>
<div id="paginationControl">
<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?= $this->url(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="<?= $this->url(array('page' => $page)); ?>"><?= $page; ?></a> |
  <?php else: ?>
    <?= $page; ?> |
  <?php endif; ?>
<?php endforeach; ?>

<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?= $this->url(array('page' => $this->next)); ?>">Next &gt;</a>
<?php else: ?>
  <span class="disabled">Next &gt;</span>
<?php endif; ?>
</div>
<?php endif; ?>
```

Item pagination

```
<!--
See http://developer.yahoo.com/ypatterns/pattern.php?pattern=itempagination
-->

<?php if ($this->pageCount): ?>
<div id="paginationControl">
<?= $this->firstItemNumber; ?> - <?= $this->lastItemNumber; ?>
of <?= $this->totalItemCount; ?>

<!-- First page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?= $this->url(array('page' => $this->first)); ?>">First</a> |
<?php else: ?>
  <span class="disabled">First</span> |
<?php endif; ?>

<!-- Previous page link -->
<?php if (isset($this->previous)): ?>
  <a href="<?= $this->url(array('page' => $this->previous)); ?>">&lt; Previous</a>
<?php else: ?>
  <span class="disabled">&lt; Previous</span> |
<?php endif; ?>

<!-- Next page link -->
<?php if (isset($this->next)): ?>
  <a href="<?= $this->url(array('page' => $this->next)); ?>">Next &gt;</a> |
<?php else: ?>
  <span class="disabled">Next &gt;</span> |
<?php endif; ?>

<!-- Last page link -->
<?php if (isset($this->next)): ?>
  <a href="<?= $this->url(array('page' => $this->last)); ?>">Last</a>
<?php else: ?>
  <span class="disabled">Last</span>
<?php endif; ?>

</div>
<?php endif; ?>
```

Dropdown pagination

```
<?php if ($this->pageCount): ?>
<select id="paginationControl" size="1">
<?php foreach ($this->pagesInRange as $page): ?>
  <?php $selected = ($page == $this->current) ? ' selected="selected"' : ''; ?>
  <option value="<?= $this->url(array('page' => $page)); ?>"<?= $selected ?>><?= $
<?php endforeach; ?>
```

```
</select>
<?php endif; ?>

<script type="text/javascript" src="http://ajax.googleapis.com/ajax/libs/prototype
<script type="text/javascript">
$('paginationControl').observe('change', function() {
    window.location = this.options[this.selectedIndex].value;
})
</script>
```

Listing of properties

The following options are available to pagination control view partials:

Table 34.3. 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
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 34.4. Configuration methods for `Zend_Paginator`

Method	Description
setCurrentPageNumber	Sets the current page number (default 1).
setItemCountPerPage	Sets the maximum number of items to display on a page (default 10).
setPageRange	Sets the number of items to display in the pagination control (default 10). Note: Most of the time this number will be adhered to exactly, but scrolling styles do have the option of only using it as a guideline or starting value (e.g., Elastic).
setView	Sets the view instance, for rendering convenience.

Advanced usage

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_Interface`. 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` [http://www.php.net/~helly/php/ext/spl/interfaceCountable.html], you're familiar with `count()`. As used with `Zend_Paginator`, this is the total number of items in the data collection.

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_Interface`) 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_Interface`, which defines a single method, `getPages()`. Specifically,

```
public function getPages(Zend_Paginator $paginator, $pageRange = null);
```

This method should calculate a lower and upper bound for page numbers within the range of so-called "local" pages (that is, pages that are nearby the current page).

Unless it extends another scrolling style (see `Zend_Paginator_ScrollingStyle_Elastic` for an example), your custom scrolling style will inevitably end with something similar to the following line of code:

```
return $paginator->getPagesInRange($lowerBound, $upperBound);
```

There's nothing special about this call; it's merely a convenience method to check the validity of the lower and upper bound and return an array of the range to the paginator.

When you're ready to use your new scrolling style, you'll need to tell `Zend_Paginator` what directory to look in. To do that, do the following:

```
$prefix = 'My_Paginator_ScrollingStyle';
$path   = 'My/Paginator/ScrollingStyle/';
Zend_Paginator::addScrollingStylePrefixPath($prefix, $path);
```

Chapter 35. Zend_Pdf

Introduction.

Zend_Pdf module is a PDF (Portable Document Format) manipulation engine written entirely in PHP 5. It can load existing documents, create new, modify and save modified documents. Thus it can help any PHP-driven application dynamically prepare documents in a PDF by modifying existing template or generating document from a scratch. Zend_Pdf module supports the following features:

- Create new document or load existing one. [1]
- Retrieving specified revision of the document.
- Manipulate pages within document. Changing page order, adding new pages, removing pages from a document.
- Different drawing primitives (lines, rectangles, polygons, circles, ellipses and sectors).
- Text drawing using any of the 14 standard (built-in) fonts or your own custom TrueType fonts.
- Rotations.
- Image drawing. [2]
- Incremental PDF file update.

Creating and loading PDF documents.

`Zend_Pdf` class represents PDF document itself and provides document level functionality.

To create new document new `Zend_Pdf` object should be created.

`Zend_Pdf` class also provides two static methods to load existing PDF. These are `Zend_Pdf::load()` and `Zend_Pdf::parse()` methods. Both of them return Zend_Pdf object as a result or throw an exception if error occurs.

[1] PDF V1.4 (Acrobat 5) documents are supported for loading now.
[2] JPG, PNG [Up to 8bit per channel+Alpha] and TIFF images are supported.

Example 35.1. Create new or load existing PDF document.

```
...
// Create new PDF document.
$pdf1 = new Zend_Pdf();

// Load PDF document from a file.
$pdf2 = Zend_Pdf::load($fileName);

// Load PDF document from a string.
$pdf3 = Zend_Pdf::parse($pdfString);
...
```

PDF file format supports incremental document update. Thus each time when document is updated, then new revision of the document is created. Zend_Pdf module supports retrieving of specified revision.

Revision can be specified as a second parameter for `Zend_Pdf::load()` and `Zend_Pdf::parse()` methods or requested by `Zend_Pdf::rollback()` [3] call.

Example 35.2. Requesting specified revision of the PDF document.

```
...
// Load PDF previouse revision of the document.
$pdf1 = Zend_Pdf::load($fileName, 1);

// Load PDF previouse revision of the document.
$pdf2 = Zend_Pdf::parse($pdfString, 1);

// Load first revision of the document.
$pdf3 = Zend_Pdf::load($fileName);
$revisions = $pdf3->revisions();
$pdf3->rollback($revisions - 1);
...
```

Save changes to the PDF document.

There are two methods, which provide saving changes to the PDF document. These are `Zend_Pdf::save()` and `Zend_Pdf::render()` methods.

`Zend_Pdf::save($filename, $updateOnly = false)` saves the PDF document to a file. If $updateOnly is true, then only the new PDF file segment is appended to a file. Otherwise, the file is overwritten.

`Zend_Pdf::render($newSegmentOnly = false)` returns the PDF document as a string. If $newSegmentOnly is true, then only the new PDF file segment is returned.

[3] `Zend_Pdf::rollback()` method must be invoked before any changes, applied to the document. Otherwise behavior is undefined.

Example 35.3. Save PDF document.

```
...
// Load PDF document.
$pdf = Zend_Pdf::load($fileName);
...
// Update document
$pdf->save($fileName, true);
// Save document as a new file
$pdf->save($newFileName);

// Return PDF document as a string.
$pdfString = $pdf->render();

...
```

Document pages.

Page creation.

PDF document page abstraction is represented by `Zend_Pdf_Page` class.

PDF pages either are loaded from existing PDF, or created.

New page can be obtained by creating new `Zend_Pdf_Page` object or calling `Zend_Pdf::newPage()` method, which returns `Zend_Pdf_Page` object. The difference is that `Zend_Pdf::newPage()` method creates a page, already attached to the document. In difference from unattached pages it can't be used with several PDF documents, but has a little bit better performance. [4]. It's your choice, which approach should be used.

`Zend_Pdf::newPage()` method and `Zend_Pdf_Page` constructors take the same set of parameters specifying page size. It either the size of page ($x, $y) in a points (1/72 inch), or predefined constant, which is treated as a page type:

- Zend_Pdf_Page::SIZE_A4
- Zend_Pdf_Page::SIZE_A4_LANDSCAPE
- Zend_Pdf_Page::SIZE_LETTER
- Zend_Pdf_Page::SIZE_LETTER_LANDSCAPE

Document pages are stored in `$pages` public member of `Zend_Pdf` class. It's an array of `Zend_Pdf_Page` objects. It completely defines set and order of document pages and can be manipulated as a common array:

[4] It's a limitation of V1.0 version of Zend_Pdf module. It will be eliminated in future versions. But unattached pages will always give better (more optimal) result for sharing pages between documents.

Example 35.4. PDF document pages management.

```
...
// Reverse page order
$pdf->pages = array_reverse($pdf->pages);
...
// Add new page
$pdf->pages[] = new Zend_Pdf_Page(Zend_Pdf_Page::SIZE_A4);
// Add new page
$pdf->pages[] = $pdf->newPage(Zend_Pdf_Page::SIZE_A4);

// Remove specified page.
unset($pdf->pages[$id]);

...
```

Page cloning.

Existing PDF page can be cloned by creating new `Zend_Pdf_Page` object with existing page as a parameter:

Example 35.5. Cloning existing page.

```
...
// Store template page in a separate variable
$template = $pdf->pages[$templatePageIndex];
...
// Add new page
$page1 = new Zend_Pdf_Page($template);
$pdf->pages[] = $page1;
...

// Add another page
$page2 = new Zend_Pdf_Page($template);
$pdf->pages[] = $page2;
...

// Remove source template page from the documents.
unset($pdf->pages[$templatePageIndex]);

...
```

It's useful if you need several pages to be created using one template.

Caution

Important! Cloned page shares some PDF resources with a template page, so it can be used only withing the same document as a template page. Modified document can be saved as new one.

Drawing

Geometry

PDF uses the same geometry as PostScript. It starts from bottom-left corner of page and by default is measured in points (1/72 of an inch).

Page size can be retrieved from a page object:

```
$width  = $pdfPage->getWidth();
$height = $pdfPage->getHeight();
```

Colors

PDF has a powerful capabilities for colors representation. Zend_Pdf module supports Gray Scale, RGB and CMYK color spaces. Any of them can be used in any place, where `Zend_Pdf_Color` object is required. `Zend_Pdf_Color_GrayScale`, `Zend_Pdf_Color_Rgb` and `Zend_Pdf_Color_Cmyk` classes provide this functionality:

```
// $grayLevel (float number). 0.0 (black) - 1.0 (white)
$color1 = new Zend_Pdf_Color_GrayScale($grayLevel);

// $r, $g, $b (float numbers). 0.0 (min intensity) - 1.0 (max intensity)
$color2 = new Zend_Pdf_Color_Rgb($r, $g, $b);

// $c, $m, $y, $k (float numbers). 0.0 (min intensity) - 1.0 (max intensity)
$color3 = new Zend_Pdf_Color_Cmyk($c, $m, $y, $k);
```

HTML style colors are also provided with `Zend_Pdf_Color_Html` class:

```
$color1 = new Zend_Pdf_Color_Html('#3366FF');
$color2 = new Zend_Pdf_Color_Html('silver');
$color3 = new Zend_Pdf_Color_Html('forestgreen');
```

Shape Drawing

All drawing operations can be done in a context of PDF page.

`Zend_Pdf_Page` class provides a set of drawing primitives:

```
/**
 * Draw a line from x1,y1 to x2,y2.
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 */
public function drawLine($x1, $y1, $x2, $y2);
```

```
/**
 * Draw a rectangle.
 *
 * Fill types:
 * Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE - fill rectangle
 *                                             and stroke (default)
 * Zend_Pdf_Page::SHAPE_DRAW_STROKE          - stroke rectangle
 * Zend_Pdf_Page::SHAPE_DRAW_FILL            - fill rectangle
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @param integer $fillType
 */
public function drawRectangle($x1, $y1, $x2, $y2,
                    $fillType = Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE);
```

```
/**
 * Draw a polygon.
 *
 * If $fillType is Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE or
 * Zend_Pdf_Page::SHAPE_DRAW_FILL, then polygon is automatically closed.
 * See detailed description of these methods in a PDF documentation
 * (section 4.4.2 Path painting Operators, Filling)
 *
 * @param array $x  - array of float (the X co-ordinates of the vertices)
 * @param array $y  - array of float (the Y co-ordinates of the vertices)
 * @param integer $fillType
 * @param integer $fillMethod
 */
public function drawPolygon($x, $y,
                            $fillType =
                                Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE,
                            $fillMethod =
                                Zend_Pdf_Page::FILL_METHOD_NON_ZERO_WINDING);



/**
 * Draw a circle centered on x, y with a radius of radius.
 *
 * Angles are specified in radians
 *
 * Method signatures:
 * drawCircle($x, $y, $radius);
 * drawCircle($x, $y, $radius, $fillType);
 * drawCircle($x, $y, $radius, $startAngle, $endAngle);
 * drawCircle($x, $y, $radius, $startAngle, $endAngle, $fillType);
 *
 *
 * It's not a really circle, because PDF supports only cubic Bezier
 * curves. But very good approximation.
 * It differs from a real circle on a maximum 0.00026 radiuses (at PI/8,
 * 3*PI/8, 5*PI/8, 7*PI/8, 9*PI/8, 11*PI/8, 13*PI/8 and 15*PI/8 angles).
 * At 0, PI/4, PI/2, 3*PI/4, PI, 5*PI/4, 3*PI/2 and 7*PI/4 it's exactly
 * a tangent to a circle.
 *
 * @param float $x
 * @param float $y
 * @param float $radius
 * @param mixed $param4
 * @param mixed $param5
 * @param mixed $param6
 */
public function  drawCircle($x,
                            $y,
                            $radius,
                            $param4 = null,
                            $param5 = null,
                            $param6 = null);




/**
 * Draw an ellipse inside the specified rectangle.
 *
 * Method signatures:
 * drawEllipse($x1, $y1, $x2, $y2);
 * drawEllipse($x1, $y1, $x2, $y2, $fillType);
 * drawEllipse($x1, $y1, $x2, $y2, $startAngle, $endAngle);
 * drawEllipse($x1, $y1, $x2, $y2, $startAngle, $endAngle, $fillType);
 *
 * Angles are specified in radians
 *
 * @param float $x1
 * @param float $y1
```

```
 * @param float $x2
 * @param float $y2
 * @param mixed $param5
 * @param mixed $param6
 * @param mixed $param7
 */
public function drawEllipse($x1, $y1, $x2, $y2, $param5 = null, $param6 = null, $p
```

Text Drawing

Text drawing operations also exist in the context of a PDF page. You can draw a single line of text at any position on the page by supplying the x and y coordinates of the baseline. Current font and current font size are used for text drawing operations (see detailed description below).

```
/**
 * Draw a line of text at the specified position.
 *
 * @param string $text
 * @param float $x
 * @param float $y
 * @param string $charEncoding (optional) Character encoding of source
 *               text.Defaults to current locale.
 * @throws Zend_Pdf_Exception
 */
public function drawText($text, $x, $y, $charEncoding = '');
```

Example 35.6. Draw a string on the page

```
...
$pdfPage->drawText('Hello world!', 72, 720);
...
```

By default, text strings are interpreted using the character encoding method of the current locale. If you have a string that uses a different encoding method (such as a UTF-8 string read from a file on disk, or a MacRoman string obtained from a legacy database), you can indicate the character encoding at draw time and Zend_Pdf will handle the conversion for you. You can supply source strings in any encoding method supported by PHP's `iconv()` [http://www.php.net/manual/function.iconv.php] function:

Example 35.7. Draw a UTF-8-encoded string on the page

```
...
// Read a UTF-8-encoded string from disk
$unicodeString = fread($fp, 1024);

// Draw the string on the page
$pdfPage->drawText($unicodeString, 72, 720, 'UTF-8');
...
```

Using fonts

`Zend_Pdf_Page::drawText()` uses the page's current font and font size, which is set with the `Zend_Pdf_Page::setFont()` method:

```
/**
 * Set current font.
 *
 * @param Zend_Pdf_Resource_Font $font
 * @param float $fontSize
 */
public function setFont(Zend_Pdf_Resource_Font $font, $fontSize);
```

PDF documents support PostScript Type 1 and TrueType fonts, as well as two specialized PDF types, Type 3 and composite fonts. There are also 14 standard Type 1 fonts built-in to every PDF viewer: Courier (4 styles), Helvetica (4 styles), Times (4 styles), Symbol, and Zapf Dingbats.

Zend_Pdf currently supports the standard 14 PDF fonts as well as your own custom TrueType fonts. Font objects are obtained via one of two factory methods: `Zend_Pdf_Font::fontWithName($fontName)` for the standard 14 PDF fonts or `Zend_Pdf_Font::fontWithPath($filePath)` for custom fonts.

Example 35.8. Create a standard font

```
...
// Create new font
$font = Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_HELVETICA);

// Apply font
$pdfPage->setFont($font, 36);
...
```

Constants for the standard 14 PDF font names are defined in the `Zend_Pdf_Font` class:

- Zend_Pdf_Font::FONT_COURIER

- Zend_Pdf_Font::FONT_COURIER_BOLD
- Zend_Pdf_Font::FONT_COURIER_ITALIC
- Zend_Pdf_Font::FONT_COURIER_BOLD_ITALIC
- Zend_Pdf_Font::FONT_TIMES
- Zend_Pdf_Font::FONT_TIMES_BOLD
- Zend_Pdf_Font::FONT_TIMES_ITALIC
- Zend_Pdf_Font::FONT_TIMES_BOLD_ITALIC
- Zend_Pdf_Font::FONT_HELVETICA
- Zend_Pdf_Font::FONT_HELVETICA_BOLD
- Zend_Pdf_Font::FONT_HELVETICA_ITALIC
- Zend_Pdf_Font::FONT_HELVETICA_BOLD_ITALIC
- Zend_Pdf_Font::FONT_SYMBOL
- Zend_Pdf_Font::FONT_ZAPFDINGBATS

You can also use any individual TrueType font (which usually has a '.ttf' extension) or an OpenType font ('.otf' extension) if it contains TrueType outlines. Currently unsupported, but planned for a future release are Mac OS X .dfont files and Microsoft TrueType Collection ('.ttc' extension) files.

To use a TrueType font, you must provide the full file path to the font program. If the font cannot be read for some reason, or if it is not a TrueType font, the factory method will throw an exception:

Example 35.9. Create a TrueType font

```
...
// Create new font
$goodDogCoolFont = Zend_Pdf_Font::fontWithPath('/path/to/GOODDC__.TTF');

// Apply font
$pdfPage->setFont($goodDogCoolFont, 36);
...
```

By default, custom fonts will be embedded in the resulting PDF document. This allows recipients to view the page as intended, even if they don't have the proper fonts installed on their system. If you are concerned about file size, you can request that the font program not be embedded by passing a 'do not embed' option to the factory method:

Example 35.10. Create a TrueType font, but do not embed it in the PDF document.

```
...
// Create new font
$goodDogCoolFont = Zend_Pdf_Font::fontWithPath('/path/to/GOODDC__.TTF',
                                               Zend_Pdf_Font::EMBED_DONT_EMBED);

// Apply font
$pdfPage->setFont($goodDogCoolFont, 36);
...
```

If the font program is not embedded but the recipient of the PDF file has the font installed on their system, they will see the document as intended. If they do not have the correct font installed, the PDF viewer application will do its best to synthesize a replacement.

Some fonts have very specific licensing rules which prevent them from being embedded in PDF documents. So you are not caught off-guard by this, if you try to use a font that cannot be embedded, the factory method will throw an exception.

You can still use these fonts, but you must either pass the do not embed flag as described above, or you can simply suppress the exception:

Example 35.11. Do not throw an exception for fonts that cannot be embedded.

```
...
$font = Zend_Pdf_Font::fontWithPath(
           '/path/to/unEmbeddableFont.ttf',
           Zend_Pdf_Font::EMBED_SUPPRESS_EMBED_EXCEPTION
        );
...
```

This suppression technique is preferred if you allow an end-user to choose their own fonts. Fonts which can be embedded in the PDF document will be; those that cannot, won't.

Font programs can be rather large, some reaching into the tens of megabytes. By default, all embedded fonts are compressed using the Flate compression scheme, resulting in a space savings of 50% on average. If, for some reason, you do not want to compress the font program, you can disable it with an option:

Example 35.12. Do not compress an embedded font.

```
...
$font = Zend_Pdf_Font::fontWithPath('/path/to/someReallyBigFont.ttf',
                                    Zend_Pdf_Font::EMBED_DONT_COMPRESS);
...
```

Finally, when necessary, you can combine the embedding options by using the bitwise OR operator:

Example 35.13. Combining font embedding options.

```
...
$font = Zend_Pdf_Font::fontWithPath(
            $someUserSelectedFontPath,
            (Zend_Pdf_Font::EMBED_SUPPRESS_EMBED_EXCEPTION |
            Zend_Pdf_Font::EMBED_DONT_COMPRESS));
...
```

Starting in 1.5, Extracting fonts.

`Zend_Pdf` module provides a possibility to extract fonts from loaded documents.

It may be useful for incremental document updates. Without this functionality you have to attach and possibly embed font into a document each time you want to update it.

`Zend_Pdf` and `Zend_Pdf_Page` objects provide special methods to extract all fonts mentioned within a document or a page:

Example 35.14. Extracting fonts from a loaded document.

```
...
$pdf = Zend_Pdf::load($documentPath);
...
// Get all document fonts
$fontList = $pdf->extractFonts();
$pdf->pages[] = ($page = $pdf->newPage(Zend_Pdf_Page::SIZE_A4));
$yPosition = 700;
foreach ($fontList as $font) {
    $page->setFont($font, 15);
    $page->drawText(
       $font->getFontName(Zend_Pdf_Font::NAME_POSTSCRIPT, 'en', 'UTF-8') .
       ':  The quick brown fox jumps over the lazy dog', 100, $yPosition, 'UTF-8')
    $yPosition -= 30;
}
...
// Get fonts referenced within the first document page
$firstPage = reset($pdf->pages);
$firstPageFonts = $firstPage->extractFonts();
...
```

Example 35.15. Extracting font from a loaded document by specifying font name.

```
...
$pdf = new Zend_Pdf();
...
$pdf->pages[] = ($page = $pdf->newPage(Zend_Pdf_Page::SIZE_A4));

$font = Zend_Pdf_Font::fontWithPath($fontPath);
$page->setFont($font, $fontSize);
$page->drawText($text, $x, $y);
...
// This font name should be stored somewhere...
$fontName = $font->getFontName(Zend_Pdf_Font::NAME_POSTSCRIPT,
                               'en',
                               'UTF-8');
...
$pdf->save($docPath);
...
```

```
...
$pdf = Zend_Pdf::load($docPath);
...
$pdf->pages[] = ($page = $pdf->newPage(Zend_Pdf_Page::SIZE_A4));

/* $srcPage->extractFont($fontName) can also be used here */
$font = $pdf->extractFont($fontName);

$page->setFont($font, $fontSize);
$page->drawText($text, $x, $y);
...
$pdf->save($docPath, true /* incremental update mode */);
...
```

Extracted fonts can be used in the place of any other font with the following limitations:

- Extracted font can be used only in the context of the document from which it was extracted.
- Possibly embedded font program is actually not extracted. So extracted font can't provide correct font metrics and original font has to be used for text width calculations:

```
...
$font = $pdf->extractFont($fontName);
$originalFont = Zend_Pdf_Font::fontWithPath($fontPath);
```

```
    $page->setFont($font /* use extracted font for drawing */, $fontSize);
    $xPosition = $x;
    for ($charIndex = 0; $charIndex < strlen($text); $charIndex++) {
        $page->drawText($text[$charIndex], xPosition, $y);

        // Use original font for text width calculation
        $width = $originalFont->widthForGlyph(
                     $originalFont->glyphNumberForCharacter($text[$charIndex])
                 );
        $xPosition += $width/$originalFont->getUnitsPerEm()*$fontSize;
    }
    ...
```

Image Drawing

`Zend_Pdf_Page` class provides drawImage() method to draw image:

```
/**
 * Draw an image at the specified position on the page.
 *
 * @param Zend_Pdf_Resource_Image $image
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 */
public function drawImage(Zend_Pdf_Resource_Image $image, $x1, $y1, $x2, $y2);
```

Image objects should be created with `Zend_Pdf_Image::imageWithPath($filePath)` method (JPG, PNG and TIFF images are supported now):

Example 35.16. Image drawing

```
...
// load image
$image = Zend_Pdf_Image::imageWithPath('my_image.jpg');

$pdfPage->drawImage($image, 100, 100, 400, 300);
...
```

Important! JPEG support requires PHP GD extension to be configured. Important! PNG support requires ZLIB extension to be configured to work with Alpha channel images.

Refer to the PHP documentation for detailed information (http://www.php.net/manual/en/ref.image.php). (http://www.php.net/manual/en/ref.zlib.php).

Line drawing style

Line drawing style is defined by line width, line color and line dashing pattern. All of this parameters can be assigned by `Zend_Pdf_Page` class methods:

```
/** Set line color. */
public function setLineColor(Zend_Pdf_Color $color);

/** Set line width. */
public function setLineWidth(float $width);

/**
 * Set line dashing pattern.
 *
 * Pattern is an array of floats:
 *     array(on_length, off_length, on_length, off_length, ...)
 * Phase is shift from the beginning of line.
 *
 * @param array $pattern
 * @param array $phase
 */
public function setLineDashingPattern($pattern, $phase = 0);
```

Fill style

`Zend_Pdf_Page::drawRectangle()`, `Zend_Pdf_Page::drawPolygon()`, `Zend_Pdf_Page::drawCircle()` and `Zend_Pdf_Page::drawEllipse()` methods take `$fillType` argument as an optional parameter. It can be:

- Zend_Pdf_Page::SHAPE_DRAW_STROKE - stroke shape
- Zend_Pdf_Page::SHAPE_DRAW_FILL - only fill shape
- Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE - fill and stroke (default behavior)

`Zend_Pdf_Page::drawPolygon()` methods also takes an additional parameter `$fillMethod`:

- Zend_Pdf_Page::FILL_METHOD_NON_ZERO_WINDING (default behavior)

  PDF reference describes this rule as follows:

  > The nonzero winding number rule determines whether a given point is inside a path by conceptually drawing a ray from that point to infinity in any direction and then examining the places where a segment of the path crosses the ray. Starting with a count of 0, the rule adds 1 each time a path segment crosses the ray from left to right and subtracts 1 each time a segment crosses from right to left. After counting all the crossings, if the result is 0 then the point is outside the path; otherwise it is inside. Note: The method just described does not specify what to do if a path segment coincides with or is tangent to the chosen ray. Since the direction of the ray is arbitrary, the rule simply chooses a ray that does not encounter such problem intersections. For simple convex paths, the nonzero winding number rule defines the inside and outside as one

> would intuitively expect. The more interesting cases are those involving complex or self-intersecting paths like the ones shown in Figure 4.10 (in a PDF Reference). For a path consisting of a five-pointed star, drawn with five connected straight line segments intersecting each other, the rule considers the inside to be the entire area enclosed by the star, including the pentagon in the center. For a path composed of two concentric circles, the areas enclosed by both circles are considered to be inside, provided that both are drawn in the same direction. If the circles are drawn in opposite directions, only the "doughnut" shape between them is inside, according to the rule; the "doughnut hole" is outside.

- Zend_Pdf_Page::FILL_METHOD_EVEN_ODD

  PDF reference describes this rule as follows:

  > An alternative to the nonzero winding number rule is the even-odd rule. This rule determines the "insideness" of a point by drawing a ray from that point in any direction and simply counting the number of path segments that cross the ray, regardless of direction. If this number is odd, the point is inside; if even, the point is outside. This yields the same results as the nonzero winding number rule for paths with simple shapes, but produces different results for more complex shapes. Figure 4.11 (in a PDF Reference) shows the effects of applying the even-odd rule to complex paths. For the five-pointed star, the rule considers the triangular points to be inside the path, but not the pentagon in the center. For the two concentric circles, only the "doughnut" shape between the two circles is considered inside, regardless of the directions in which the circles are drawn.

Rotations

PDF page can be rotated before applying any draw operation. It can be done by `Zend_Pdf_Page::rotate()` method:

```
/**
 * Rotate the page around ($x, $y) point by specified angle (in radians).
 *
 * @param float $angle
 */
public function rotate($x, $y, $angle);
```

Save/restore graphics state

At any time page graphics state (current font, font size, line color, fill color, line style, page rotation, clip area) can be saved and then restored. Save operation puts data to a graphics state stack, restore operation retrieves it from there.

There are two methods in `Zend_Pdf_Page` class for these operations:

```
/**
 * Save the graphics state of this page.
 * This takes a snapshot of the currently applied style, position,
 * clipping area and any rotation/translation/scaling that has been
 * applied.
 */
public function saveGS();

/**
 * Restore the graphics state that was saved with the last call to
 * saveGS().
 */
public function restoreGS();
```

Clipping draw area

PDF and Zend_Pdf module support clipping of draw area. Current clip area limits the regions of the page affected by painting operators. It's a whole page initially.

`Zend_Pdf_Page` class provides a set of methods for clipping operations.

```
/**
 * Intersect current clipping area with a rectangle.
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 */
public function clipRectangle($x1, $y1, $x2, $y2);
```

```
/**
 * Intersect current clipping area with a polygon.
 *
 * @param array $x  - array of float (the X co-ordinates of the vertices)
 * @param array $y  - array of float (the Y co-ordinates of the vertices)
 * @param integer $fillMethod
 */
public function clipPolygon($x,
                            $y,
                            $fillMethod =
                                Zend_Pdf_Page::FILL_METHOD_NON_ZERO_WINDING);
```

```
/**
 * Intersect current clipping area with a circle.
 *
 * @param float $x
 * @param float $y
 * @param float $radius
 * @param float $startAngle
 * @param float $endAngle
 */
public function clipCircle($x,
                           $y,
                           $radius,
                           $startAngle = null,
                           $endAngle = null);



/**
 * Intersect current clipping area with an ellipse.
 *
 * Method signatures:
 * drawEllipse($x1, $y1, $x2, $y2);
 * drawEllipse($x1, $y1, $x2, $y2, $startAngle, $endAngle);
 *
 * @todo process special cases with $x2-$x1 == 0 or $y2-$y1 == 0
 *
 * @param float $x1
 * @param float $y1
 * @param float $x2
 * @param float $y2
 * @param float $startAngle
 * @param float $endAngle
 */
public function clipEllipse($x1,
                            $y1,
                            $x2,
                            $y2,
                            $startAngle = null,
                            $endAngle = null);
```

Styles

`Zend_Pdf_Style` class provides styles functionality.

Styles can be used to store a set of graphic state parameters and apply it to a PDF page by one operation:

```
/**
 * Set the style to use for future drawing operations on this page
 *
 * @param Zend_Pdf_Style $style
 */
public function setStyle(Zend_Pdf_Style $style);

/**
```

```
 * Return the style, applied to the page.
 *
 * @return Zend_Pdf_Style|null
 */
public function getStyle();
```

`Zend_Pdf_Style` class provides a set of methods to set or get different graphics state parameters:

```
/**
 * Set line color.
 *
 * @param Zend_Pdf_Color $color
 */
public function setLineColor(Zend_Pdf_Color $color);



/**
 * Get line color.
 *
 * @return Zend_Pdf_Color|null
 */
public function getLineColor();



/**
 * Set line width.
 *
 * @param float $width
 */
public function setLineWidth($width);



/**
 * Get line width.
 *
 * @return float
 */
public function getLineWidth();



/**
 * Set line dashing pattern
 *
 * @param array $pattern
```

```
     * @param float $phase
     */
    public function setLineDashingPattern($pattern, $phase = 0);




    /**
     * Get line dashing pattern
     *
     * @return array
     */
    public function getLineDashingPattern();




    /**
     * Get line dashing phase
     *
     * @return float
     */
    public function getLineDashingPhase();




    /**
     * Set fill color.
     *
     * @param Zend_Pdf_Color $color
     */
    public function setFillColor(Zend_Pdf_Color $color);




    /**
     * Get fill color.
     *
     * @return Zend_Pdf_Color|null
     */
    public function getFillColor();




    /**
     * Set current font.
     *
     * @param Zend_Pdf_Resource_Font $font
     * @param float $fontSize
     */
    public function setFont(Zend_Pdf_Resource_Font $font, $fontSize);
```

```
/**
 * Modify current font size
 *
 * @param float $fontSize
 */
public function setFontSize($fontSize);
```

```
/**
 * Get current font.
 *
 * @return Zend_Pdf_Resource_Font $font
 */
public function getFont();
```

```
/**
 * Get current font size
 *
 * @return float $fontSize
 */
public function getFontSize();
```

Transparency

`Zend_Pdf` module supports transparency handling.

Transparency may be set using `Zend_Pdf_Page::setAlpha()` method:

```
/**
 * Set the transparency
 *
 * $alpha == 0  - transparent
 * $alpha == 1  - opaque
 *
 * Transparency modes, supported by PDF:
 * Normal (default), Multiply, Screen, Overlay, Darken, Lighten,
 * ColorDodge, ColorBurn, HardLight, SoftLight, Difference, Exclusion
 *
 * @param float $alpha
 * @param string $mode
 * @throws Zend_Pdf_Exception
 */
public function setAlpha($alpha, $mode = 'Normal');
```

Document Info and Metadata.

A PDF document may include general information such as the document’s title, author, and creation and modification dates.

Historicaly this information is stored using special Info structure. This structure is available for read and writing as an associative array using `properties` public property of Zend_Pdf objects:

```
$pdf = Zend_Pdf::load($pdfPath);

echo $pdf->properties['Title'] . "\n";
echo $pdf->properties['Author'] . "\n";

$pdf->properties['Title'] = 'New Title.';
$pdf->save($pdfPath);
```

The following keys are defined by PDF v1.4 (Acrobat 5) standard:

- *Title* - string, optional, the document’s title.
- *Author* - string, optional, the name of the person who created the document.
- *Subject* - string, optional, the subject of the document.
- *Keywords* - string, optional, keywords associated with the document.
- *Creator* - string, optional, if the document was converted to PDF from another format, the name of the application (for example, Adobe FrameMaker®) that created the original document from which it was converted.
- *Producer* - string, optional, if the document was converted to PDF from another format, the name of the application (for example, Acrobat Distiller) that converted it to PDF..
- *CreationDate* - string, optional, the date and time the document was created, in the following form: "D:YYYYMMDDHHmmSSOHH'mm'", where:
  - *YYYY* is the year.
  - *MM* is the month.
  - *DD* is the day (01–31).
  - *HH* is the hour (00–23).
  - *mm*is the minute (00–59).
  - *SS* is the second (00–59).

  - *O* is the relationship of local time to Universal Time (UT), denoted by one of the characters +, −, or Z (see below).
  - *HH* followed by ' is the absolute value of the offset from UT in hours (00–23).
  - *mm* followed by ' is the absolute value of the offset from UT in minutes (00–59).

  The apostrophe character (') after HH and mm is part of the syntax. All fields after the year are optional. (The prefix D:, although also optional, is strongly recommended.) The default values for MM and DD are both 01; all other numerical fields default to zero values. A plus sign (+) as the value of the O field signifies that local time is later than UT, a minus sign (−) that local time is earlier than UT, and the letter Z that local time is equal to UT. If no UT information is specified, the relationship of the specified time to UT is considered to be unknown. Whether or not the time zone is known, the rest of the date should be specified in local time.

  For example, December 23, 1998, at 7:52 PM, U.S. Pacific Standard Time, is represented by the string "D:199812231952−08'00'".

- *ModDate* - string, optional, the date and time the document was most recently modified, in the same form as *CreationDate*.
- *Trapped* - boolean, optional, indicates whether the document has been modified to include trapping information.
  - *true* - The document has been fully trapped; no further trapping is needed.
  - *false* - The document has not yet been trapped; any desired trapping must still be done.
  - *null* - Either it is unknown whether the document has been trapped or it has been partly but not yet fully trapped; some additional trapping may still be needed.

Since PDF v 1.6 metadata can be stored in the special XML document attached to the PDF (XMP - Extensible Metadata Platform [http://www.adobe.com/products/xmp/]).

This XML document can be retrieved and attached to the PDF with `Zend_Pdf::getMetadata()` and `Zend_Pdf::setMetadata($metadata)` methods:

```
$pdf = Zend_Pdf::load($pdfPath);
$metadata = $pdf->getMetadata();
$metadataDOM = new DOMDocument();
$metadataDOM->loadXML($metadata);

$xpath = new DOMXPath($metadataDOM);
$pdfPreffixNamespaceURI = $xpath->query('/rdf:RDF/rdf:Description')->item(0)->look
$xpath->registerNamespace('pdf', $pdfPreffixNamespaceURI);

$titleNode = $xpath->query('/rdf:RDF/rdf:Description/pdf:Title')->item(0);
$title = $titleNode->nodeValue;
...

$titleNode->nodeValue = 'New title';
$pdf->setMetadata($metadataDOM->saveXML());
$pdf->save($pdfPath);
```

Common document properties are duplicated in the Info structure and Metadata document (if presented). It's user application responsibility now to keep them syncronized.

Zend_Pdf module usage example

This section provides an example of module usage.

This example can be found in a `demos/Zend/Pdf/demo.php` file.

There are also `test.pdf` file, which can be used with this demo for test purposes.

Example 35.17. Zend_Pdf module usage demo

```
/**
 * @package Zend_Pdf
 * @subpackage demo
 */

if (!isset($argv[1])) {
    echo "USAGE: php demo.php <pdf_file> [<output_pdf_file>]\n";
    exit;
}

try {
    $pdf = Zend_Pdf::load($argv[1]);
} catch (Zend_Pdf_Exception $e) {
    if ($e->getMessage() == 'Can not open \'' . $argv[1] .
                            '\' file for reading.') {
        // Create new PDF if file doesn't exist
        $pdf = new Zend_Pdf();

        if (!isset($argv[2])) {
            // force complete file rewriting (instead of updating)
            $argv[2] = $argv[1];
        }
    } else {
        // Throw an exception if it's not the "Can't open file
        // exception
        throw $e;
    }
}

//--------------------------------------------------------------------
// Reverse page order
$pdf->pages = array_reverse($pdf->pages);

// Create new Style
$style = new Zend_Pdf_Style();
$style->setFillColor(new Zend_Pdf_Color_Rgb(0, 0, 0.9));
$style->setLineColor(new Zend_Pdf_Color_GrayScale(0.2));
$style->setLineWidth(3);
$style->setLineDashingPattern(array(3, 2, 3, 4), 1.6);
$style->setFont(Zend_Pdf_Font::fontWithName(
                    Zend_Pdf_Font::FONT_HELVETICA_BOLD), 32
               );

try {
    // Create new image object
    $stampImage = Zend_Pdf_Image::imageWithPath(dirname(__FILE__) .
                                                '/stamp.jpg');
} catch (Zend_Pdf_Exception $e) {
    // Example of operating with image loading exceptions.
    if ($e->getMessage() != 'Image extension is not installed.' &&
        $e->getMessage() != 'JPG support is not configured properly.') {
        throw $e;
    }
    $stampImage = null;
}

// Mark page as modified
foreach ($pdf->pages as $page){
    $page->saveGS();
    $page->setAlpha(0.25);
    $page->setStyle($style);
    $page->rotate(0, 0, M_PI_2/3);

    $page->saveGS();
    $page->clipCircle(550, -10, 50);
    if ($stampImage != null) {
        $page->drawImage($stampImage, 500, -60, 600, 40);
    }
    $page->restoreGS();

    $page->drawText('Modified by Zend Framework!', 150, 0);
    $page->restoreGS();
}

// Add new page generated by Zend_Pdf object
// (page is attached to the specified document)
$pdf->pages[] = ($page1 = $pdf->newPage('A4'));

// Add new page generated by Zend_Pdf_Page object
// (page is not attached to the document)
$pdf->pages[] =
    ($page2 = new Zend_Pdf_Page(Zend_Pdf_Page::SIZE_LETTER_LANDSCAPE));

// Create new font
$font = Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_HELVETICA);

// Apply font and draw text
$page1->setFont($font, 36);
$page1->drawText('Helvetica 36 text string', 60, 500);

// Use font object for another page
$page2->setFont($font, 24);
$page2->drawText('Helvetica 24 text string', 60, 500);

// Use another font
$page2->setFont(
    Zend_Pdf_Font::fontWithName(Zend_Pdf_Font::FONT_TIMES_ROMAN), 32
);

$page2->drawText('Times-Roman 32 text string', 60, 450);

// Draw rectangle
$page2->setFillColor(new Zend_Pdf_Color_GrayScale(0.8));
$page2->setLineColor(new Zend_Pdf_Color_GrayScale(0.2));
```

```
$page2->setLineDashingPattern(array(3, 2, 3, 4), 1.6);
$page2->drawRectangle(60, 400, 400, 350);

// Draw circle
$page2->setLineDashingPattern(Zend_Pdf_Page::LINE_DASHING_SOLID);
$page2->setFillColor(new Zend_Pdf_Color_Rgb(1, 0, 0));
$page2->drawCircle(85, 375, 25);

// Draw sectors
$page2->drawCircle(200, 375, 25, 2*M_PI/3, -M_PI/6);
$page2->setFillColor(new Zend_Pdf_Color_Cmyk(1, 0, 0, 0));
$page2->drawCircle(200, 375, 25, M_PI/6, 2*M_PI/3);
$page2->setFillColor(new Zend_Pdf_Color_Rgb(1, 1, 0));
$page2->drawCircle(200, 375, 25, -M_PI/6, M_PI/6);

// Draw ellipse
$page2->setFillColor(new Zend_Pdf_Color_Rgb(1, 0, 0));
$page2->drawEllipse(250, 400, 400, 350);
$page2->setFillColor(new Zend_Pdf_Color_Cmyk(1, 0, 0, 0));
$page2->drawEllipse(250, 400, 400, 350, M_PI/6, 2*M_PI/3);
$page2->setFillColor(new Zend_Pdf_Color_Rgb(1, 1, 0));
$page2->drawEllipse(250, 400, 400, 350, -M_PI/6, M_PI/6);

// Draw and fill polygon
$page2->setFillColor(new Zend_Pdf_Color_Rgb(1, 0, 1));
$x = array();
$y = array();
for ($count = 0; $count < 8; $count++) {
    $x[] = 140 + 25*cos(3*M_PI_4*$count);
    $y[] = 375 + 25*sin(3*M_PI_4*$count);
}
$page2->drawPolygon($x, $y,
                    Zend_Pdf_Page::SHAPE_DRAW_FILL_AND_STROKE,
                    Zend_Pdf_Page::FILL_METHOD_EVEN_ODD);

// Draw line
$page2->setLineWidth(0.5);
$page2->drawLine(60, 375, 400, 375);
//-----------------------------------------------------------------

if (isset($argv[2])) {
    $pdf->save($argv[2]);
} else {
    $pdf->save($argv[1], true /* update */);
}
```

Chapter 36. Zend_Registry

Using the Registry

The registry is a container for storing objects and values in the application space. By storing the value in the registry, the same object is always available throughout your application. This mechanism is an alternative to using global storage.

The typical usage of the registry is through static methods in the Zend_Registry class. Alternatively, the class is an array object, so you can access elements stored within it with a convenient array-like interface.

Setting Values in the Registry

To store an entry in the registry, use the static method `set()`.

Example 36.1. Example of set() method

```
Zend_Registry::set('index', $value);
```

The value can be an object, an array, or a scalar. You can change the value stored in a specific entry of the registry by using `set()` to set it to a new value.

The index can be a scalar, either string or integer, like an ordinary array.

Getting Values from the Registry

To retrieve an entry from the registry, use the static method `get()`.

Example 36.2. Example of get() method

```
$value = Zend_Registry::get('index');
```

The `getInstance()` method returns the static registry object.

A registry object is iterable.

Example 36.3. Example of iterating over the registry

```
$registry = Zend_Registry::getInstance();

foreach ($registry as $index => $value) {
    echo "Registry index $index contains:\n";
    var_dump($value);
}
```

Constructing a Registry Object

In addition to accessing the static registry through static methods, you can create an instance directly and use it as an object.

The registry instance you access through the static methods is simply one such instance, and it is for convenience that it is stored statically, so you can access it from anywhere in your appliation.

Use a traditional `new` constructor to create an instance of the registry. This gives you the opportunity to initialize the entries in the registry as an associatve array.

Example 36.4. Example of constructing a registry

```
$registry = new Zend_Registry(array('index' => $value));
```

After constructing this instance, you can use it using array-object methods, or you can set this instance to become the static instance using the static method `setInstance()`.

Example 36.5. Example of initializing the static registry

```
$registry = new Zend_Registry(array('index' => $value));

Zend_Registry::setInstance($registry);
```

The `setInstance()` method throws a Zend_Exception if the static registry has already been initialized by its first access.

Accessing the Registry as an Array

If you have several values to get or set, you may find it convenient to access the registry with array notation.

Example 36.6. Example of array access

```
$registry = Zend_Registry::getInstance();

$registry['index'] = $value;

var_dump( $registry['index'] );
```

Accessing the Registry as an Object

You may also find it convenient to access the registry in an object-oriented fashion, using index names as object properties. To do this, you need to specifically construct the registry object using the `ArrayObject::ARRAY_AS_PROPS` option, and initialize the static instance. You must do this before the static registry has been accessed for the first time. **Beware** of using this option, since some versions of PHP have bugs when using the registry with this option.

Example 36.7. Example of object access

```
// in your application bootstrap:
$registry = new Zend_Registry(array(), ArrayObject::ARRAY_AS_PROPS)
Zend_Registry::setInstance($registry);
$registry->tree = 'apple';

.
.
.

// in a different function, elsewhere in your application:
$registry = Zend_Registry::getInstance();

echo $registry->tree; // echo's "apple"

$registry->index = $value;

var_dump($registry->index);
```

Querying if an index exists

To find out if a particular index in the registry has a value, use the static method `isRegistered()`.

Example 36.8. Example of isRegistered() method

```
if (Zend_Registry::isRegistered($index)) {
    $value = Zend_Registry::get($index);
}
```

To find out if a particular index in a registry array-object has a value, use `isset()` like you would with an ordinary array.

Example 36.9. Example of isset() method

```
$registry = Zend_Registry::getInstance();

// using array-access syntax
if (isset($registry['index'])) {
    var_dump( $registry['index'] );
}

// using object-access syntax, if enabled
if (isset($registry->index)) {
    var_dump( $registry->index );
}
```

Extending the Registry

The static registry is an instance of the class Zend_Registry. If you want to add functionality to the registry, you can create a class that extends Zend_Registry, and then you can specify this class as the class to use for the static registry. Use the static method `setClassName()` to specify the class. The class must extend Zend_Registry.

Example 36.10. Example of specifying the static registry's class name

```
Zend_Registry::setClassName('My_Registry');

Zend_Registry::set('index', $value);
```

The registry throws a Zend_Exception if you try to set the classname after the registry has been accessed for the first time. It is recommended that you specify the classname for your static registry in your application bootstrap.

Unsetting the Static Registry

Although it is not normally necessary, you can unset the static instance of the registry. Use the static method `_unsetInstance()`.

Data loss risk

When you use `_unsetInstance()`, all data in the static registry are discarded and cannot be recovered.

You might use this method, for example, if you want to use `setInstance()` or `setClassName()` after the static registry object has been initialized. Unsetting the static instance allows you to use these methods.

Example 36.11. Example of _unsetInstance() method

```
Zend_Registry::set('index', $value);

Zend_Registry::_unsetInstance();

// change the class
Zend_Registry::setClassName('My_Registry');

Zend_Registry::set('index', $value);
```

Chapter 37. Zend_Rest

Introduction

REST Web Services use service-specific XML formats. These ad-hoc standards mean that the manner for accessing a REST web service is different for each service. REST web services typically use URL parameters (GET data) or path information for requesting data and POST data for sending data.

The Zend Framework provides both Client and Server capabilities, which, when used together allow for a much more "local" interface experience via virtual object property access. The Server component features automatic exposition of functions and classes using a meaningful and simple XML format. When accessing these services using the Client, it is possible to easily retrieve the return data from the remote call. Should you wish to use the client with a non-Zend_Rest_Server based service, it will still provide easier data access.

Zend_Rest_Client

Introduction

Using the `Zend_Rest_Client` is very similar to using `SoapClient` objects (SOAP web service extension [http://www.php.net/soap]). You can simply call the REST service procedures as `Zend_Rest_Client` methods. Specify the service's full address in the `Zend_Rest_Client` constructor.

Example 37.1. A basic REST request

```
/**
 * Connect to framework.zend.com server and retrieve a greeting
 */
$client = new Zend_Rest_Client('http://framework.zend.com/rest');

echo $client->sayHello('Davey', 'Day')->get(); // "Hello Davey, Good Day"
```

Differences in calling

`Zend_Rest_Client` attempts to make remote methods look as much like native methods as possible, the only difference being that you must follow the method call with one of either `get()`, `post()`, `put()` or `delete()`. This call may be made via method chaining or in separate method calls:

```
$client->sayHello('Davey', 'Day');
echo $client->get();
```

Responses

All requests made using `Zend_Rest_Client` return a `Zend_Rest_Client_Response` object. This object has many properties that make it easier to access the results.

When the service is based on `Zend_Rest_Server`, `Zend_Rest_Client` can make several assumptions about the response, including response status (success or failure) and return type.

Example 37.2. Response Status

```
$result = $client->sayHello('Davey', 'Day')->get();

if ($result->isSuccess()) {
    echo $result; // "Hello Davey, Good Day"
}
```

In the example above, you can see that we use the request result as an object, to call `isSuccess()`, and then because of `__toString()`, we can simply `echo` the object to get the result. `Zend_Rest_Client_Response` will allow you to echo any scalar value. For complex types, you can use either array or object notation.

If however, you wish to query a service not using `Zend_Rest_Server` the `Zend_Rest_Client_Response` object will behave more like a `SimpleXMLElement`. However, to make things easier, it will automatically query the XML using XPath if the property is not a direct descendant of the document root element. Additionally, if you access a property as a method, you will receive the PHP value for the object, or an array of PHP value results.

Example 37.3. Using Technorati's Rest Service

```
$technorati = new Zend_Rest_Client('http://api.technorati.com/bloginfo');
$technorati->key($key);
$technorati->url('http://pixelated-dreams.com');
$result = $technorati->get();
echo $result->firstname() .' '. $result->lastname();
```

Example 37.4. Example Technorati Response

```
<?xml version="1.0" encoding="utf-8"?>
<!-- generator="Technorati API version 1.0 /bloginfo" -->
<!DOCTYPE tapi PUBLIC "-//Technorati, Inc.//DTD TAPI 0.02//EN"
                      "http://api.technorati.com/dtd/tapi-002.xml">
<tapi version="1.0">
    <document>
        <result>
            <url>http://pixelated-dreams.com</url>
            <weblog>
                <name>Pixelated Dreams</name>
                <url>http://pixelated-dreams.com</url>
                <author>
                    <username>DShafik</username>
                    <firstname>Davey</firstname>
                    <lastname>Shafik</lastname>
                </author>
                <rssurl>
                    http://pixelated-dreams.com/feeds/index.rss2
                </rssurl>
                <atomurl>
                    http://pixelated-dreams.com/feeds/atom.xml
                </atomurl>
                <inboundblogs>44</inboundblogs>
                <inboundlinks>218</inboundlinks>
                <lastupdate>2006-04-26 04:36:36 GMT</lastupdate>
                <rank>60635</rank>
            </weblog>
            <inboundblogs>44</inboundblogs>
            <inboundlinks>218</inboundlinks>
        </result>
    </document>
</tapi>
```

Here we are accessing the `firstname` and `lastname` properties. Even though these are not top-level elements, they are automatically returned when accessed by name.

Multiple items

If multiple items are found when accessing a value by name, an array of SimpleXMLElements will be returned; accessing via method notation will return an array of PHP values.

Request Arguments

Unless you are making a request to a `Zend_Rest_Server` based service, chances are you will need to send multiple arguments with your request. This is done by calling a method with the name of the argument, passing in the value as the first (and only) argument. Each of these method calls returns the object itself, allowing for chaining, or "fluent" usage. The first call, or the first argument if you pass in more than one argument, is always assumed to be the method when calling a `Zend_Rest_Server` service.

Example 37.5. Setting Request Arguments

```
$client = new Zend_Rest_Client('http://example.org/rest');

$client->arg('value1');
$client->arg2('value2');
$client->get();

// or

$client->arg('value1')->arg2('value2')->get();
```

Both of the methods in the example above, will result in the following get args: `?method=arg&arg1=value1&arg=value1&arg2=value2`

You will notice that the first call of `$client->arg('value1');` resulted in both `method=arg&arg1=value1` and `arg=value1`; this is so that `Zend_Rest_Server` can understand the request properly, rather than requiring pre-existing knowledge of the service.

Strictness of Zend_Rest_Client

Any REST service that is strict about the arguments it receives will likely fail using `Zend_Rest_Client`, because of the behavior described above. This is not a common practice and should not cause problems.

Zend_Rest_Server

Introduction

Zend_Rest_Server is intended as a fully-featured REST server.

REST Server Usage

Example 37.6. Basic Zend_Rest_Server Usage - Classes

```
$server = new Zend_Rest_Server();
$server->setClass('My_Service_Class');
$server->handle();
```

Example 37.7. Basic Zend_Rest_Server Usage - Functions

```
/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return string
 */
function sayHello($who, $when)
{
    return "Hello $who, Good $when";
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello');
$server->handle();
```

Calling a Zend_Rest_Server Service

To call a `Zend_Rest_Server` service, you must supply a GET/POST `method` argument with a value that is the method you wish to call. You can then follow that up with any number of arguments using either the name of the argument (i.e. "who") or using `arg` following by the numeric position of the argument (i.e. "arg1").

Numeric index

Numeric arguments use a 1-based index.

To call `sayHello` from the example above, you can use either:

```
?method=sayHello&who=Davey&when=Day
```

or:

```
?method=sayHello&arg1=Davey&arg2=Day
```

Sending A Custom Status

When returning values, to return a custom status, you may return an array with a `status` key.

Example 37.8. Returning Custom Status

```
/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return array
 */
function sayHello($who, $when)
{
    return array('msg' => "An Error Occurred", 'status' => false);
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello');
$server->handle();
```

Returning Custom XML Responses

If you wish to return custom XML, simply return a `DOMDocument`, `DOMElement` or `SimpleXMLElement` object.

Example 37.9. Return Custom XML

```
/**
 * Say Hello
 *
 * @param string $who
 * @param string $when
 * @return SimpleXMLElement
 */
function sayHello($who, $when)
{
    $xml ='<?xml version="1.0" encoding="ISO-8859-1"?>
<mysite>
    <value>Hey $who! Hope you're having a good $when</value>
    <code>200</code>
</mysite>';

    $xml = simplexml_load_string($xml);
    return $xml;
}

$server = new Zend_Rest_Server();
$server->addFunction('sayHello');

$server->handle();
```

The response from the service will be returned without modification to the client.

Chapter 38. Zend_Search_Lucene

Overview

Introduction

Zend_Search_Lucene is a general purpose text search engine written entirely in PHP 5. Since it stores its index on the filesystem and does not require a database server, it can add search capabilities to almost any PHP-driven website. Zend_Search_Lucene supports the following features:

- Ranked searching - best results returned first
- Many powerful query types: phrase queries, boolean queries, wildcard queries, proximity queries, range queries and many others.
- Search by specific field (e.g., title, author, contents)

Zend_Search_Lucene was derived from the Apache Lucene project. The currently (starting from ZF 1.6) supported Lucene index format versions are 1.4 - 2.3. For more information on Lucene, visit http://lucene.apache.org/java/docs/.

> Previous Zend_Search_Lucene implementations support the Lucene 1.4 (1.9) - 2.1 index formats.
>
> Starting from ZF 1.5 any index created using pre-2.1 index format is automatically upgraded to Lucene 2.1 format after the Zend_Search_Lucene update and will not be compatible with Zend_Search_Lucene implementations included into ZF 1.0.x.

Document and Field Objects

Zend_Search_Lucene operates with documents as atomic objects for indexing. A document is divided into named fields, and fields have content that can be searched.

A document is represented by the Zend_Search_Lucene_Document class, and this objects of this class contain instances of Zend_Search_Lucene_Field that represent the fields on the document.

It is important to note that any information can be added to the index. Application-specific information or metadata can be stored in the document fields, and later retrieved with the document during search.

It is the responsibility of your application to control the indexer. This means that data can be indexed from any source that is accessible by your application. For example, this could be the filesystem, a database, an HTML form, etc.

`Zend_Search_Lucene_Field` class provides several static methods to create fields with different characteristics:

```
$doc = new Zend_Search_Lucene_Document();

// Field is not tokenized, but is indexed and stored within the index.
// Stored fields can be retrived from the index.
$doc->addField(Zend_Search_Lucene_Field::Keyword('doctype',
                                                  'autogenerated'));
```

```
// Field is not tokenized nor indexed, but is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('created',
                                                   time()));

// Binary String valued Field that is not tokenized nor indexed,
// but is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::Binary('icon',
                                                $iconData));

// Field is tokenized and indexed, and is stored in the index.
$doc->addField(Zend_Search_Lucene_Field::Text('annotation',
                                              'Document annotation text'));

// Field is tokenized and indexed, but is not stored in the index.
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  'My document content'));
```

Each of these methods (excluding the `Zend_Search_Lucene_Field::Binary()` method) has an optional `$encoding` parameter for specifying input data encoding.

Encoding may differ for different documents as well as for different fields within one document:

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('title',
                                              $title,
                                              'iso-8859-1'));
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  $contents,
                                                  'utf-8'));
```

If encoding parameter is omitted, then the current locale is used at processing time. For example:

```
setlocale(LC_ALL, 'de_DE.iso-8859-1');
...
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents', $contents));
```

Fields are always stored and returned from the index in UTF-8 encoding. Any required conversion to UTF-8 happens automatically.

Text analyzers (see below) may also convert text to some other encodings. Actually, the default analyzer converts text to 'ASCII//TRANSLIT' encoding. Be careful, however; this translation may depend on current locale.

Fields' names are defined at your discretion in the `addField()` method.

Java Lucene uses the 'contents' field as a default field to search. Zend_Search_Lucene searches through all fields by default, but the behavior is configurable. See the "Default search field" chapter for details.

Understanding Field Types

- `Keyword` fields are stored and indexed, meaning that they can be searched as well as displayed in search results. They are not split up into separate words by tokenization. Enumerated database fields usually translate well to Keyword fields in Zend_Search_Lucene.
- `UnIndexed` fields are not searchable, but they are returned with search hits. Database timestamps, primary keys, file system paths, and other external identifiers are good candidates for UnIndexed fields.
- `Binary` fields are not tokenized or indexed, but are stored for retrieval with search hits. They can be used to store any data encoded as a binary string, such as an image icon.
- `Text` fields are stored, indexed, and tokenized. Text fields are appropriate for storing information like subjects and titles that need to be searchable as well as returned with search results.
- `UnStored` fields are tokenized and indexed, but not stored in the index. Large amounts of text are best indexed using this type of field. Storing data creates a larger index on disk, so if you need to search but not redisplay the data, use an UnStored field. UnStored fields are practical when using a Zend_Search_Lucene index in combination with a relational database. You can index large data fields with UnStored fields for searching, and retrieve them from your relational database by using a separate field as an identifier.

Table 38.1. Zend_Search_Lucene_Field Types

Field Type	Stored	Indexed	Tokenized	Binary
Keyword	Yes	Yes	No	No
UnIndexed	Yes	No	No	No
Binary	Yes	No	No	Yes
Text	Yes	Yes	Yes	No
UnStored	No	Yes	Yes	No

HTML documents

Zend_Search_Lucene offers a HTML parsing feature. Documents can be created directly from a HTML file or string:

```
$doc = Zend_Search_Lucene_Document_Html::loadHTMLFile($filename);
$index->addDocument($doc);
...
$doc = Zend_Search_Lucene_Document_Html::loadHTML($htmlString);
$index->addDocument($doc);
```

`Zend_Search_Lucene_Document_Html` class uses the `DOMDocument::loadHTML()` and `DOMDocument::loadHTMLFile()` methods to parse the source HTML, so it doesn't need HTML to be well formed or to be XHTML. On the other hand, it's sensitive to the encoding specied by the "meta http-equiv" header tag.

`Zend_Search_Lucene_Document_Html` class recognizes document title, body and document header meta tags.

The 'title' field is actually the /html/head/title value. It's stored within the index, tokenized and available for search.

The 'body' field is the actual body content of the HTML file or string. It doesn't include scripts, comments or attributes.

The `loadHTML()` and `loadHTMLFile()` methods of `Zend_Search_Lucene_Document_Html` class also have second optional argument. If it's set to true, then body content is also stored within index and can be retrieved from the index. By default, the body is tokenized and indexed, but not stored.

Other document header meta tags produce additional document fields. The field 'name' is taken from 'name' attribute, and the 'content' attribute populates the field 'value'. Both are tokenized, indexed and stored, so documents may be searched by their meta tags (for example, by keywords).

Parsed documents may be augmented by the programmer with any other field:

```
$doc = Zend_Search_Lucene_Document_Html::loadHTML($htmlString);
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('created',
                                                   time()));
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('updated',
                                                   time()));
$doc->addField(Zend_Search_Lucene_Field::Text('annotation',
                                              'Document annotation text'));
$index->addDocument($doc);
```

Document links are not included in the generated document, but may be retrieved with the `Zend_Search_Lucene_Document_Html::getLinks()` and `Zend_Search_Lucene_Document_Html::getHeaderLinks()` methods:

```
$doc = Zend_Search_Lucene_Document_Html::loadHTML($htmlString);
$linksArray = $doc->getLinks();
$headerLinksArray = $doc->getHeaderLinks();
```

Starting from ZF 1.6 it's also possible to exclude links with `rel` attribute set to `'nofollow'`. Use `Zend_Search_Lucene_Document_Html::setExcludeNoFollowLinks($true)` to turn on this option.

`Zend_Search_Lucene_Document_Html::getExcludeNoFollowLinks()` method returns current state of "Exclude nofollow links" flag.

Building Indexes

Creating a New Index

Index creation and updating capabilities are implemented within the Zend_Search_Lucene component, as well as the Java Lucene project. You can use either of these options to create indexes that Zend_Search_Lucene can search.

The PHP code listing below provides an example of how to index a file using Zend_Search_Lucene indexing API:

```
// Create index
$index = Zend_Search_Lucene::create('/data/my-index');

$doc = new Zend_Search_Lucene_Document();

// Store document URL to identify it in the search results
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));

// Index document contents
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents', $docContent));

// Add document to the index
$index->addDocument($doc);
```

Newly added documents are immediately searchable in the index.

Updating Index

The same procedure is used to update an existing index. The only difference is that the open() method is called instead of the create() method:

```
// Open existing index
$index = Zend_Search_Lucene::open('/data/my-index');

$doc = new Zend_Search_Lucene_Document();
// Store document URL to identify it in search result.
$doc->addField(Zend_Search_Lucene_Field::Text('url', $docUrl));
// Index document content
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  $docContent));

// Add document to the index.
$index->addDocument($doc);
```

Updating Documents

The Lucene index file format doesn't support document updating. Documents should be removed and re-added to the index to effectively update them.

`Zend_Search_Lucene::delete()` method operates with an internal index document id. It can be retrieved from a query hit by 'id' property:

```
$removePath = ...;
$hits = $index->find('path:' . $removePath);
foreach ($hits as $hit) {
    $index->delete($hit->id);
}
```

Retrieving Index Size

There are two methods to retrieve the size of an index in Zend_Search_Lucene.

`Zend_Search_Lucene::maxDoc()` returns one greater than the largest possible document number. It's actually the overall number of the documents in the index including deleted documents, so it has a synonym: `Zend_Search_Lucene::count()`.

`Zend_Search_Lucene::numDocs()` returns the total number of non-deleted documents.

```
$indexSize = $index->count();
$documents = $index->numDocs();
```

`Zend_Search_Lucene::isDeleted($id)` method may be used to check if a document is deleted.

```
for ($count = 0; $count < $index->maxDoc(); $count++) {
    if ($index->isDeleted($count)) {
        echo "Document #$id is deleted.\n";
    }
}
```

Index optimization removes deleted documents and squeezes documents' IDs in to a smaller range. A document's internal id may therefore change during index optimization.

Index optimization

A Lucene index consists of many segments. Each segment is a completely independent set of data.

Lucene index segment files can't be updated by design. A segment update needs full segment reorganization. See Lucene index file formats for details (http://lucene.apache.org/java/docs/fileformats.html) [1]. New documents are added to the index by creating new segment.

Increasing number of segments reduces quality of the index, but index optimization restores it. Optimization essentially merges several segments into a new one. This process also doesn't update segments. It generates one new large segment and updates segment list ('segments' file).

Full index optimization can be trigger by calling the `Zend_Search_Lucene::optimize()` method. It merges all index segments into one new segment:

```
// Open existing index
$index = Zend_Search_Lucene::open('/data/my-index');

// Optimize index.
$index->optimize();
```

Automatic index optimization is performed to keep indexes in a consistent state.

Automatic optimization is an iterative process managed by several index options. It merges very small segments into larger ones, then merges these larger segments into even larger segments and so on.

MaxBufferedDocs auto-optimization option

MaxBufferedDocs is a minimal number of documents required before the buffered in-memory documents are written into a new segment.

MaxBufferedDocs can be retrieved or set by `$index->getMaxBufferedDocs()` or `$index->setMaxBufferedDocs($maxBufferedDocs)` calls.

Default value is 10.

MaxMergeDocs auto-optimization option

MaxMergeDocs is a largest number of documents ever merged by addDocument(). Small values (e.g., less than 10.000) are best for interactive indexing, as this limits the length of pauses while indexing to a few seconds. Larger values are best for batched indexing and speedier searches.

MaxMergeDocs can be retrieved or set by `$index->getMaxMergeDocs()` or `$index->setMaxMergeDocs($maxMergeDocs)` calls.

Default value is PHP_INT_MAX.

MergeFactor auto-optimization option

MergeFactor determines how often segment indices are merged by addDocument(). With smaller values, less RAM is used while indexing, and searches on unoptimized indices are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indices are slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indices that are interactively maintained.

[1]The currently supported Lucene index file format is version 2.3 (starting from ZF 1.6).

MergeFactor is a good estimation for average number of segments merged by one auto-optimization pass. Too large values produce large number of segments while they are not merged into new one. It may be a cause of "failed to open stream: Too many open files" error message. This limitation is system dependent.

MergeFactor can be retrieved or set by `$index->getMergeFactor()` or `$index->setMergeFactor($mergeFactor)` calls.

Default value is 10.

Lucene Java and Luke (Lucene Index Toolbox - http://www.getopt.org/luke/) can also be used to optimize an index. Latest Luke release (v0.8) is based on Lucene v2.3 and compatible with current implementation of Zend_Search_Lucene component (ZF 1.6). Earlier versions of Zend_Search_Lucene implementations need another versions of Java Lucene tools to be compatible:

- ZF 1.5 - Java Lucene 2.1 (Luke tool v0.7.1 - http://www.getopt.org/luke/luke-0.7.1/)
- ZF 1.0 - Java Lucene 1.4 - 2.1 (Luke tool v0.6 - http://www.getopt.org/luke/luke-0.6/)

Permissions

By default, index files are available for reading and writing by everyone.

It's possible to override this with the `Zend_Search_Lucene_Storage_Directory_Filesystem::setDefaultFilePermissions()` method:

```
// Get current default file permissions
$currentPermissions =
    Zend_Search_Lucene_Storage_Directory_Filesystem::getDefaultFilePermissions();

// Give read-writing permissions only for current user and group
Zend_Search_Lucene_Storage_Directory_Filesystem::setDefaultFilePermissions(0660);
```

Limitations

Index size

Index size is limited by 2GB for 32-bit platforms.

Use 64-bit platforms for larger indices.

Supported Filesystems

Zend_Search_Lucene uses `flock()` to provide concurrent searching, index updating and optimization.

According to the PHP documentation [http://www.php.net/manual/en/function.flock.php], "`flock()` will not work on NFS and many other networked file systems".

Do not use networked file systems with Zend_Search_Lucene.

Searching an Index

Building Queries

There are two ways to search the index. The first method uses query parser to construct a query from a string. The second is to programmatically create your own queries through the Zend_Search_Lucene API.

Before choosing to use the provided query parser, please consider the following:

1. If you are programmatically creating a query string and then parsing it with the query parser then you should consider building your queries directly with the query API. Generally speaking, the query parser is designed for human-entered text, not for program-generated text.

2. Untokenized fields are best added directly to queries and not through the query parser. If a field's values are generated programmatically by the application, then the query clauses for this field should also be constructed programmatically. An analyzer, which the query parser uses, is designed to convert human-entered text to terms. Program-generated values, like dates, keywords, etc., should be added with the query API.

3. In a query form, fields that are general text should use the query parser. All others, such as date ranges, keywords, etc., are better added directly through the query API. A field with a limited set of values that can be specified with a pull-down menu should not be added to a query string that is subsequently parsed but instead should be added as a TermQuery clause.

4. Boolean queries allow the programmer to logically combine two or more queries into new one. Thus it's the best way to add additional criteria to a search defined by a query string.

Both ways use the same API method to search through the index:

```
$index = Zend_Search_Lucene::open('/data/my_index');

$index->find($query);
```

The `Zend_Search_Lucene::find()` method determines the input type automatically and uses the query parser to construct an appropriate Zend_Search_Lucene_Search_Query object from an input of type string.

It is important to note that the query parser uses the standard analyzer to tokenize separate parts of query string. Thus all transformations which are applied to indexed text are also applied to query strings.

The standard analyzer may transform the query string to lower case for case-insensitivity, remove stop-words, and stem among other transformations.

The API method doesn't transform or filter input terms in any way. It's therefore more suitable for computer generated or untokenized fields.

Query Parsing

`Zend_Search_Lucene_Search_QueryParser::parse()` method may be used to parse query strings into query objects.

This query object may be used in query construction API methods to combine user entered queries with programmatically generated queries.

Actually, in some cases it's the only way to search for values within untokenized fields:

```
$userQuery = Zend_Search_Lucene_Search_QueryParser::parse($queryStr);

$pathTerm  = new Zend_Search_Lucene_Index_Term(
                     '/data/doc_dir/' . $filename, 'path'
                 );
$pathQuery = new Zend_Search_Lucene_Search_Query_Term($pathTerm);

$query = new Zend_Search_Lucene_Search_Query_Boolean();
$query->addSubquery($userQuery, true /* required */);
$query->addSubquery($pathQuery, true /* required */);

$hits = $index->find($query);
```

`Zend_Search_Lucene_Search_QueryParser::parse()` method also takes an optional encoding parameter, which can specify query string encoding:

```
$userQuery = Zend_Search_Lucene_Search_QueryParser::parse($queryStr,
                                                            'iso-8859-5');
```

If the encoding parameter is omitted, then current locale is used.

It's also possible to specify the default query string encoding with `Zend_Search_Lucene_Search_QueryParser::setDefaultEncoding()` method:

```
Zend_Search_Lucene_Search_QueryParser::setDefaultEncoding('iso-8859-5');
...
$userQuery = Zend_Search_Lucene_Search_QueryParser::parse($queryStr);
```

`Zend_Search_Lucene_Search_QueryParser::getDefaultEncoding()` returns the current default query string encoding (the empty string means "current locale").

Search Results

The search result is an array of Zend_Search_Lucene_Search_QueryHit objects. Each of these has two properties: `$hit->document` is a document number within the index and `$hit->score` is a score of the hit in a search result. The results are ordered by score (descending from highest score).

The Zend_Search_Lucene_Search_QueryHit object also exposes each field of the Zend_Search_Lucene_Document found in the search as a property of the hit. In the following example, a hit is returned with two fields from the corresponding document: title and author.

```
$index = Zend_Search_Lucene::open('/data/my_index');

$hits = $index->find($query);

foreach ($hits as $hit) {
    echo $hit->score;
    echo $hit->title;
    echo $hit->author;
}
```

Stored fields are always returned in UTF-8 encoding.

Optionally, the original Zend_Search_Lucene_Document object can be returned from the Zend_Search_Lucene_Search_QueryHit. You can retrieve stored parts of the document by using the `getDocument()` method of the index object and then get them by `getFieldValue()` method:

```
$index = Zend_Search_Lucene::open('/data/my_index');

$hits = $index->find($query);
foreach ($hits as $hit) {
    // return Zend_Search_Lucene_Document object for this hit
    echo $document = $hit->getDocument();

    // return a Zend_Search_Lucene_Field object
    // from the Zend_Search_Lucene_Document
    echo $document->getField('title');

    // return the string value of the Zend_Search_Lucene_Field object
    echo $document->getFieldValue('title');

    // same as getFieldValue()
    echo $document->title;
}
```

The fields available from the Zend_Search_Lucene_Document object are determined at the time of indexing. The document fields are either indexed, or index and stored, in the document by the indexing application (e.g. LuceneIndexCreation.jar).

Note that the document identity ('path' in our example) is also stored in the index and must be retrieved from it.

Limiting the Result Set

The most computationally expensive part of searching is score calculation. It may take several seconds for large result sets (tens of thousands of hits).

Zend_Search_Lucene gives the possibility to limit result set size with `getResultSetLimit()` and `setResultSetLimit()` methods:

```
$currentResultSetLimit = Zend_Search_Lucene::getResultSetLimit();

Zend_Search_Lucene::setResultSetLimit($newLimit);
```

The default value of 0 means 'no limit'.

It doesn't give the 'best N' results, but only the 'first N'[2].

Results Scoring

Zend_Search_Lucene uses the same scoring algorithms as Java Lucene. All hits in the search result are ordered by score by default. Hits with greater score come first, and documents having higher scores should match the query more precisely than documents having lower scores.

Roughly speaking, search hits that contain the searched term or phrase more frequently will have a higher score.

A hit's score can be retrieved by accessing the `score` property of the hit:

```
$hits = $index->find($query);

foreach ($hits as $hit) {
    echo $hit->id;
    echo $hit->score;
}
```

The Zend_Search_Lucene_Search_Similarity class is used to calculate the score for each hit. See Extensibility. Scoring Algorithms section for details.

Search Result Sorting

By default, the search results are ordered by score. The programmer can change this behavior by setting a sort field (or a list of fields), sort type and sort order parameters.

`$index->find()` call may take several optional parameters:

```
$index->find($query [, $sortField [, $sortType [, $sortOrder]]]
                    [, $sortField2 [, $sortType [, $sortOrder]]]
             ...);
```

A name of stored field by which to sort result should be passed as the `$sortField` parameter.

[2]Returned hits are still ordered by score or by the the specified order, if given.

`$sortType` may be omitted or take the following enumerated values: `SORT_REGULAR` (compare items normally- default value), `SORT_NUMERIC` (compare items numerically), `SORT_STRING` (compare items as strings).

`$sortOrder` may be omitted or take the following enumerated values: `SORT_ASC` (sort in ascending order- default value), `SORT_DESC` (sort in descending order).

Examples:

```
$index->find($query, 'quantity', SORT_NUMERIC, SORT_DESC);
```

```
$index->find($query, 'fname', SORT_STRING, 'lname', SORT_STRING);
```

```
$index->find($query, 'name', SORT_STRING, 'quantity', SORT_NUMERIC, SORT_DESC);
```

Please use caution when using a non-default search order; the query needs to retrieve documents completely from an index, which may dramatically reduce search performance.

Search Results Highlighting

`Zend_Search_Lucene_Search_Query::highlightMatches()` method allows the developer to highlight HTML document terms in the context of a search query:

```
$query = Zend_Search_Lucene_Search_QueryParser::parse($queryStr);
$hits = $index->find($query);
...
$highlightedHTML = $query->highlightMatches($sourceHTML);
```

`highlightMatches()` method utilizes `Zend_Search_Lucene_Document_Html` class (see HTML documents section for details) for HTML processing, so it has the same requirements for HTML source.

Query Language

Java Lucene and Zend_Search_Lucene provide quite powerful query languages.

These languages are mostly the same with some minor differences, which are mentioned below.

Full Java Lucene query language syntax documentation can be found here [http://lucene.apache.org/java/docs/queryparsersyntax.html].

Terms

A query is broken up into terms and operators. There are three types of terms: Single Terms, Phrases, and Subqueries.

A Single Term is a single word such as "test" or "hello".

A Phrase is a group of words surrounded by double quotes such as "hello dolly".

A Subquery is a query surrounded by parentheses such as "(hello dolly)".

Multiple terms can be combined together with boolean operators to form complex queries (see below).

Fields

Lucene supports fields of data. When performing a search you can either specify a field, or use the default field. The field names depend on indexed data and default field is defined by current settings.

The first and most significant difference from Java Lucene is that terms are searched through *all fields* by default.

There are two static methods in the Zend_Search_Lucene class which allow the developer to configure these settings:

```
$defaultSearchField = Zend_Search_Lucene::getDefaultSearchField();
...
Zend_Search_Lucene::setDefaultSearchField('contents');
```

The `null` value indicated that the search is performed across all fields. It's the default setting.

You can search specific fields by typing the field name followed by a colon ":" followed by the term you are looking for.

As an example, let's assume a Lucene index contains two fields- title and text- with text as the default field. If you want to find the document entitled "The Right Way" which contains the text "don't go this way", you can enter:

```
title:"The Right Way" AND text:go
```

or

```
title:"Do it right" AND go
```

Because "text" is the default field, the field indicator is not required.

Note: The field is only valid for the term, phrase or subquery that it directly precedes, so the query

```
title:Do it right
```

Will only find "Do" in the title field. It will find "it" and "right" in the default field (if the default field is set) or in all indexed fields (if the default field is set to `null`).

Wildcards

Lucene supports single and multiple character wildcard searches within single terms (but not within phrase queries).

To perform a single character wildcard search use the "?" symbol.

To perform a multiple character wildcard search use the "*" symbol.

The single character wildcard search looks for string that match the term with the "?" replaced by any single character. For example, to search for "text" or "test" you can use the search:

```
te?t
```

Multiple character wildcard searches look for 0 or more characters when matching strings against terms. For example, to search for test, tests or tester, you can use the search:

```
test*
```

You can use "?", "*" or both at any place of the term:

```
*wr?t*
```

It searches for "write", "wrote", "written", "rewrite", "rewrote" and so on.

Term Modifiers

Lucene supports modifying query terms to provide a wide range of searching options.

"~" modifier can be used to specify proximity search for phrases or fuzzy search for individual terms.

Range Searches

Range queries allow the developer or user to match documents whose field(s) values are between the lower and upper bound specified by the range query. Range Queries can be inclusive or exclusive of the upper and lower bounds. Sorting is performed lexicographically.

```
mod_date:[20020101 TO 20030101]
```

This will find documents whose mod_date fields have values between 20020101 and 20030101, inclusive. Note that Range Queries are not reserved for date fields. You could also use range queries with non-date fields:

```
title:{Aida TO Carmen}
```

This will find all documents whose titles would be sorted between Aida and Carmen, but not including Aida and Carmen.

Inclusive range queries are denoted by square brackets. Exclusive range queries are denoted by curly brackets.

If field is not specified then Zend_Search_Lucene searches for specified interval through all fields by default.

```
{Aida TO Carmen}
```

Fuzzy Searches

Zend_Search_Lucene as well as Java Lucene supports fuzzy searches based on the Levenshtein Distance, or Edit Distance algorithm. To do a fuzzy search use the tilde, "~", symbol at the end of a Single word Term. For example to search for a term similar in spelling to "roam" use the fuzzy search:

```
roam~
```

This search will find terms like foam and roams. Additional (optional) parameter can specify the required similarity. The value is between 0 and 1, with a value closer to 1 only terms with a higher similarity will be matched. For example:

```
roam~0.8
```

The default that is used if the parameter is not given is 0.5.

Proximity Searches

Lucene supports finding words from a phrase that are within a specified word distance in a string. To do a proximity search use the tilde, "~", symbol at the end of the phrase. For example to search for a "Zend" and "Framework" within 10 words of each other in a document use the search:

```
"Zend Framework"~10
```

Boosting a Term

Java Lucene and Zend_Search_Lucene provide the relevance level of matching documents based on the terms found. To boost the relevance of a term use the caret, "^", symbol with a boost factor (a number) at the end of the term you are searching. The higher the boost factor, the more relevant the term will be.

Boosting allows you to control the relevance of a document by boosting individual terms. For example, if you are searching for

```
PHP framework
```

and you want the term "PHP" to be more relevant boost it using the ^ symbol along with the boost factor next to the term. You would type:

```
PHP^4 framework
```

This will make documents with the term PHP appear more relevant. You can also boost phrase terms and subqueries as in the example:

```
"PHP framework"^4 "Zend Framework"
```

By default, the boost factor is 1. Although the boost factor must be positive, it may be less than 1 (e.g. 0.2).

Boolean Operators

Boolean operators allow terms to be combined through logic operators. Lucene supports AND, "+", OR, NOT and "-" as Boolean operators. Java Lucene requires boolean operators to be ALL CAPS. Zend_Search_Lucene does not.

AND, OR, and NOT operators and "+", "-" defines two different styles to construct boolean queries. Unlike Java Lucene, Zend_Search_Lucene doesn't allow these two styles to be mixed.

If the AND/OR/NOT style is used, then an AND or OR operator must be present between all query terms. Each term may also be preceded by NOT operator. The AND operator has higher precedence than the OR operator. This differs from Java Lucene behavior.

AND

The AND operator means that all terms in the "AND group" must match some part of the searched field(s).

To search for documents that contain "PHP framework" and "Zend Framework" use the query:

```
"PHP framework" AND "Zend Framework"
```

OR

The OR operator divides the query into several optional terms.

To search for documents that contain "PHP framework" or "Zend Framework" use the query:

```
"PHP framework" OR "Zend Framework"
```

NOT

The NOT operator excludes documents that contain the term after NOT. But an "AND group" which contains only terms with the NOT operator gives an empty result set instead of a full set of indexed documents.

To search for documents that contain "PHP framework" but not "Zend Framework" use the query:

```
"PHP framework" AND NOT "Zend Framework"
```

&&, ||, and ! operators

&&, ||, and ! may be used instead of AND, OR, and NOT notation.

+

The "+" or required operator stipulates that the term after the "+" symbol must match the document.

To search for documents that must contain "Zend" and may contain "Framework" use the query:

```
+Zend Framework
```

-

The "-" or prohibit operator excludes documents that match the term after the "-" symbol.

To search for documents that contain "PHP framework" but not "Zend Framework" use the query:

```
"PHP framework" -"Zend Framework"
```

No Operator

If no operator is used, then the search behavior is defined by the "default boolean operator".

This is set to `OR` by default.

That implies each term is optional by default. It may or may not be present within document, but documents with this term will recieve a higher score.

To search for documents that requires "PHP framework" and may contain "Zend Framework" use the query:

```
+"PHP framework" "Zend Framework"
```

The default boolean operator may be set or retrieved with the `Zend_Search_Lucene_Search_QueryParser::setDefaultOperator($operator)` and `Zend_Search_Lucene_Search_QueryParser::getDefaultOperator()` methods, respectively.

These methods operate with the `Zend_Search_Lucene_Search_QueryParser::B_AND` and `Zend_Search_Lucene_Search_QueryParser::B_OR` constants.

Grouping

Java Lucene and Zend_Search_Lucene support using parentheses to group clauses to form sub queries. This can be useful if you want to control the precedence of boolean logic operators for a query or mix different boolean query styles:

```
+(framework OR library) +php
```

Zend_Search_Lucene supports subqueries nested to any level.

Field Grouping

Lucene also supports using parentheses to group multiple clauses to a single field.

To search for a title that contains both the word "return" and the phrase "pink panther" use the query:

```
title:(+return +"pink panther")
```

Escaping Special Characters

Lucene supports escaping special characters that are used in query syntax. The current list of special characters is:

+ - && || ! ( ) { } [ ] ^ " ~ * ? : \

+ and - inside single terms are automatically treated as common characters.

For other instances of these characters use the \ before each special character you'd like to escape. For example to search for (1+1):2 use the query:

```
\(1\+1\)\:2
```

Query Construction API

In addition to parsing a string query automatically it's also possible to construct them with the query API.

User queries can be combined with queries created through the query API. Simply use the query parser to construct a query from a string:

```
$query = Zend_Search_Lucene_Search_QueryParser::parse($queryString);
```

Query Parser Exceptions

The query parser may generate two types of exceptions:

- `Zend_Search_Lucene_Exception` is thrown if something goes wrong in the query parser itself.
- `Zend_Search_Lucene_Search_QueryParserException` is thrown when there is an error in the query syntax.

It's a good idea to catch Zend_Search_Lucene_Search_QueryParserExceptions and handle it appropriately:

```
try {
    $query = Zend_Search_Lucene_Search_QueryParser::parse($queryString);
} catch (Zend_Search_Lucene_Search_QueryParserException $e) {
    echo "Query syntax error: " . $e->getMessage() . "\n";
}
```

The same technique should be used for the find() method of a Zend_Search_Lucene object.

Starting in 1.5, query parsing exceptions are suppressed by default. If query doesn't conform query language, then it's tokenized using current default analyzer and all tokenized terms are used for searching. Use `Zend_Search_Lucene_Search_QueryParser::dontSuppressQueryParsingExceptions()` method to turn exceptions on. `Zend_Search_Lucene_Search_QueryParser::suppressQueryParsingExceptions()` and `Zend_Search_Lucene_Search_QueryParser::queryParsingExceptionsSuppressed()` methods are also intended to manage exceptions handling behavior.

Term Query

Term queries can be used for searching with a single term.

Query string:

```
word1
```

or

Query construction by API:

```
$term  = new Zend_Search_Lucene_Index_Term('word1', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Term($term);
$hits  = $index->find($query);
```

The term field is optional. Zend_Search_Lucene searches through all indexed fields in each document if the field is not specified:

```
// Search for 'word1' in all indexed fields
$term  = new Zend_Search_Lucene_Index_Term('word1');
$query = new Zend_Search_Lucene_Search_Query_Term($term);
$hits  = $index->find($query);
```

Multi-Term Query

Multi-term queries can be used for searching with a set of terms.

Each term in a set can be defined as *required*, *prohibited*, or *neither*.

- *required* means that documents not matching this term will not match the query;
- *prohibited* means that documents matching this term will not match the query;
- *neither*, in which case matched documents are neither prohibited from, nor required to, match the term. A document must match at least 1 term, however, to match the query.

If optional terms are added to a query with required terms, both queries will have the same result set but the optional terms may affect the score of the matched documents.

Both search methods can be used for multi-term queries.

Query string:

```
+word1 author:word2 -word3
```

- '+' is used to define a required term.
- '-' is used to define a prohibited term.
- 'field:' prefix is used to indicate a document field for a search. If it's omitted, then all fields are searched.

or

Query construction by API:

```
$query = new Zend_Search_Lucene_Search_Query_MultiTerm();

$query->addTerm(new Zend_Search_Lucene_Index_Term('word1'), true);
$query->addTerm(new Zend_Search_Lucene_Index_Term('word2', 'author'),
                null);
$query->addTerm(new Zend_Search_Lucene_Index_Term('word3'), false);

$hits  = $index->find($query);
```

It's also possible to specify terms list within MultiTerm query constructor:

```
$terms = array(new Zend_Search_Lucene_Index_Term('word1'),
               new Zend_Search_Lucene_Index_Term('word2', 'author'),
               new Zend_Search_Lucene_Index_Term('word3'));
$signs = array(true, null, false);

$query = new Zend_Search_Lucene_Search_Query_MultiTerm($terms, $signs);

$hits  = $index->find($query);
```

The `$signs` array contains information about the term type:

- `true` is used to define required term.
- `false` is used to define prohibited term.
- `null` is used to define a term that is neither required nor prohibited.

Boolean Query

Boolean queries allow to construct query using other queries and boolean operators.

Each subquery in a set can be defined as *required*, *prohibited*, or *optional*.

- *required* means that documents not matching this subquery will not match the query;
- *prohibited* means that documents matching this subquery will not match the query;
- *optional*, in which case matched documents are neither prohibited from, nor required to, match the subquery. A document must match at least 1 subquery, however, to match the query.

If optional subqueries are added to a query with required suqueries, both queries will have the same result set but the optional suqueries may affect the score of the matched documents.

Both search methods can be used for boolean queries.

Query string:

```
+(word1 word2 word3) author:(word4 word5) -word6
```

- '+' is used to define a required subquery.
- '-' is used to define a prohibited subquery.
- 'field:' prefix is used to indicate a document field for a search. If it's omitted, then all fields are searched.

or

Query construction by API:

```
$query = new Zend_Search_Lucene_Search_Query_Boolean();

$subquery1 = new Zend_Search_Lucene_Search_Query_MultiTerm();
$subquery1->addTerm(new Zend_Search_Lucene_Index_Term('word1'));
$subquery1->addTerm(new Zend_Search_Lucene_Index_Term('word2'));
$subquery1->addTerm(new Zend_Search_Lucene_Index_Term('word3'));

$subquery2 = new Zend_Search_Lucene_Search_Query_MultiTerm();
$subquery2->addTerm(new Zend_Search_Lucene_Index_Term('word4', 'author'));
$subquery2->addTerm(new Zend_Search_Lucene_Index_Term('word5', 'author'));

$term6 = new Zend_Search_Lucene_Index_Term('word6');
$subquery3 = new Zend_Search_Lucene_Search_Query_Term($term6);

$query->addSubquery($subquery1, true  /* required */);
$query->addSubquery($subquery2, null  /* optional */);
$query->addSubquery($subquery3, false /* prohibited */);

$hits  = $index->find($query);
```

It's also possible to specify subqueries list within Boolean query constructor:

```
...
$subqueries = array($subquery1, $subquery2, $subquery3);
$signs = array(true, null, false);

$query = new Zend_Search_Lucene_Search_Query_Boolean($subqueries, $signs);

$hits  = $index->find($query);
```

The `$signs` array contains information about the subquery type:

- `true` is used to define required subquery.
- `false` is used to define prohibited subquery.
- `null` is used to define a subquery that is neither required nor prohibited.

Each query which uses boolean operators can be rewritten using signs notation and constructed using API. For example:

```
word1 AND (word2 AND word3 AND NOT word4) OR word5
```

is equivalent to

```
(+(word1) +(+word2 +word3 -word4)) (word5)
```

Wildcard Query

Wildcard queries can be used to search for documents containing strings matching specified patterns.

The '?' symbol is used as a single character wildcard.

The '*' symbol is used as a multiple character wildcard.

Query string:

```
field1:test*
```

or

Query construction by API:

```
$pattern = new Zend_Search_Lucene_Index_Term('test*', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Wildcard($pattern);
$hits  = $index->find($query);
```

The term field is optional. Zend_Search_Lucene searches through all fields on each document if a field is not specified:

```
$pattern = new Zend_Search_Lucene_Index_Term('test*');
$query = new Zend_Search_Lucene_Search_Query_Wildcard($pattern);
$hits  = $index->find($query);
```

Fuzzy Query

Fuzzy queries can be used to search for documents containing strings matching terms similar to specified term.

Query string:

```
field1:test~
```

This query matches documents containing 'test' 'text' 'best' words and others.

or

Query construction by API:

```
$term = new Zend_Search_Lucene_Index_Term('test', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Fuzzy($term);
$hits  = $index->find($query);
```

Optional similarity can be specified after "~" sign.

Query string:

```
field1:test~0.4
```

or

Query construction by API:

```
$term = new Zend_Search_Lucene_Index_Term('test', 'field1');
$query = new Zend_Search_Lucene_Search_Query_Fuzzy($term, 0.4);
$hits  = $index->find($query);
```

The term field is optional. Zend_Search_Lucene searches through all fields on each document if a field is not specified:

```
$term = new Zend_Search_Lucene_Index_Term('test');
$query = new Zend_Search_Lucene_Search_Query_Fuzzy($term);
$hits  = $index->find($query);
```

Phrase Query

Phrase Queries can be used to search for a phrase within documents.

Phrase Queries are very flexible and allow the user or developer to search for exact phrases as well as 'sloppy' phrases.

Phrases can also contain gaps or terms in the same places; they can be generated by the analyzer for different purposes. For example, a term can be duplicated to increase the term its weight, or several synonyms can be placed into a single position.

```
$query1 = new Zend_Search_Lucene_Search_Query_Phrase();

// Add 'word1' at 0 relative position.
$query1->addTerm(new Zend_Search_Lucene_Index_Term('word1'));

// Add 'word2' at 1 relative position.
$query1->addTerm(new Zend_Search_Lucene_Index_Term('word2'));

// Add 'word3' at 3 relative position.
$query1->addTerm(new Zend_Search_Lucene_Index_Term('word3'), 3);

...

$query2 = new Zend_Search_Lucene_Search_Query_Phrase(
                array('word1', 'word2', 'word3'), array(0,1,3));

...

// Query without a gap.
$query3 = new Zend_Search_Lucene_Search_Query_Phrase(
                array('word1', 'word2', 'word3'));

...

$query4 = new Zend_Search_Lucene_Search_Query_Phrase(
                array('word1', 'word2'), array(0,1), 'annotation');
```

A phrase query can be constructed in one step with a class constructor or step by step with `Zend_Search_Lucene_Search_Query_Phrase::addTerm()` method calls.

Zend_Search_Lucene_Search_Query_Phrase class constructor takes three optional arguments:

```
Zend_Search_Lucene_Search_Query_Phrase(
    [array $terms[, array $offsets[, string $field]]]
);
```

The `$terms` parameter is an array of strings that contains a set of phrase terms. If it's omitted or equal to null, then an empty query is constructed.

The `$offsets` parameter is an array of integers that contains offsets of terms in a phrase. If it's omitted or equal to null, then the terms' positions are assumed to be sequential with no gaps.

The `$field` parameter is a string that indicates the document field to search. If it's omitted or equal to null, then the default field is searched.

Thus:

```
$query =
    new Zend_Search_Lucene_Search_Query_Phrase(array('zend', 'framework'));
```

will search for the phrase 'zend framework' in all fields.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase(
                 array('zend', 'download'), array(0, 2)
             );
```

will search for the phrase 'zend ????? download' and match 'zend platform download', 'zend studio download', 'zend core download', 'zend framework download', and so on.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase(
                 array('zend', 'framework'), null, 'title'
             );
```

will search for the phrase 'zend framework' in the 'title' field.

`Zend_Search_Lucene_Search_Query_Phrase::addTerm()` takes two arguments, a required `Zend_Search_Lucene_Index_Term` object and an optional position:

```
Zend_Search_Lucene_Search_Query_Phrase::addTerm(
    Zend_Search_Lucene_Index_Term $term[, integer $position]
);
```

The `$term` parameter describes the next term in the phrase. It must indicate the same field as previous terms, or an exception will be thrown.

The `$position` parameter indicates the term position in the phrase.

Thus:

```
$query = new Zend_Search_Lucene_Search_Query_Phrase();
$query->addTerm(new Zend_Search_Lucene_Index_Term('zend'));
$query->addTerm(new Zend_Search_Lucene_Index_Term('framework'));
```

will search for the phrase 'zend framework'.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase();
$query->addTerm(new Zend_Search_Lucene_Index_Term('zend'), 0);
$query->addTerm(new Zend_Search_Lucene_Index_Term('framework'), 2);
```

will search for the phrase 'zend ????? download' and match 'zend platform download', 'zend studio download', 'zend core download', 'zend framework download', and so on.

```
$query = new Zend_Search_Lucene_Search_Query_Phrase();
$query->addTerm(new Zend_Search_Lucene_Index_Term('zend', 'title'));
$query->addTerm(new Zend_Search_Lucene_Index_Term('framework', 'title'));
```

will search for the phrase 'zend framework' in the 'title' field.

The slop factor sets the number of other words permitted between specified words in the query phrase. If set to zero, then the corresponding query is an exact phrase search. For larger values this works like the WITHIN or NEAR operators.

The slop factor is in fact an edit distance, where the edits correspond to moving terms in the query phrase. For example, to switch the order of two words requires two moves (the first move places the words atop one another), so to permit re-orderings of phrases, the slop factor must be at least two.

More exact matches are scored higher than sloppier matches; thus, search results are sorted by exactness. The slop is zero by default, requiring exact matches.

The slop factor can be assigned after query creation:

```
// Query without a gap.
$query =
    new Zend_Search_Lucene_Search_Query_Phrase(array('word1', 'word2'));

// Search for 'word1 word2', 'word1 ... word2'
$query->setSlop(1);
$hits1 = $index->find($query);

// Search for 'word1 word2', 'word1 ... word2',
// 'word1 ... ... word2', 'word2 word1'
$query->setSlop(2);
$hits2 = $index->find($query);
```

Range Query

Range queries are intended for searching terms within specified interval.

Query string:

```
mod_date:[20020101 TO 20030101]
title:{Aida TO Carmen}
```

or

Query construction by API:

```
$from = new Zend_Search_Lucene_Index_Term('20020101', 'mod_date');
$to   = new Zend_Search_Lucene_Index_Term('20030101', 'mod_date');
$query = new Zend_Search_Lucene_Search_Query_Range(
                 $from, $to, true // inclusive
             );
$hits  = $index->find($query);
```

Term fields are optional. Zend_Search_Lucene searches through all fields if the field is not specified:

```
$from = new Zend_Search_Lucene_Index_Term('Aida');
$to   = new Zend_Search_Lucene_Index_Term('Carmen');
$query = new Zend_Search_Lucene_Search_Query_Range(
                 $from, $to, false // non-inclusive
             );
$hits  = $index->find($query);
```

Either (but not both) of the boundary terms may be set to null. Zend_Search_Lucene searches from the beginning or up to the end of the dictionary for the specified field(s) in this case:

```
// searches for ['20020101' TO ...]
$from = new Zend_Search_Lucene_Index_Term('20020101', 'mod_date');
$query = new Zend_Search_Lucene_Search_Query_Range(
                 $from, null, true // inclusive
             );
$hits  = $index->find($query);
```

Character Set

UTF-8 and single-byte character set support

Zend_Search_Lucene works with the UTF-8 charset internally. Index files store unicode data in Java's "modified UTF-8 encoding". Zend_Search_Lucene core completely supports this encoding with one exception. [3]

Actual input data encoding may be specified through Zend_Search_Lucene API. Data will be automatically converted into UTF-8 encoding.

Default text analyzer

However, the default text analyzer (which is also used within query parser) uses ctype_alpha() for tokenizing text and queries.

ctype_alpha() is not UTF-8 compatible, so the analyzer converts text to 'ASCII//TRANSLIT' encoding before indexing. The same processing is transparently performed during query parsing. [4]

> Default analyzer doesn't treats numbers as parts of terms. Use corresponding 'Num' analyzer if you don't want words to be broken by numbers.

UTF-8 compatible text analyzers

Zend_Search_Lucene also contains a set of UTF-8 compatible analyzers: `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8`, `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8Num`, `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8_CaseInsensitive`, `Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8Num_CaseInsensitive`.

Any of this analyzers can be enabled with the code like this:

```
Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8());
```

> UTF-8 compatible analyzers were improved in ZF 1.5. Early versions of analyzers assumed all non-ascii characters are letters. New analyzers implementation has more accurate behavior.
>
> This may need you to re-build index to have data and search queries tokenized in the same way, otherwise search engine may return wrong result sets.

All of these analyzers need PCRE (Perl-compatible regular expressions) library to be compiled with UTF-8 support turned on. PCRE UTF-8 support is turned on for the PCRE library sources bundled with PHP

[3] Zend_Search_Lucene supports only Basic Multilingual Plane (BMP) characters (from 0x0000 to 0xFFFF) and doesn't support "supplementary characters" (characters whose code points are greater than 0xFFFF)

Java 2 represents these characters as a pair of char (16-bit) values, the first from the high-surrogates range (0xD800-0xDBFF), the second from the low-surrogates range (0xDC00-0xDFFF). Then they are encoded as usual UTF-8 characters in six bytes. Standard UTF-8 representation uses four bytes for supplementary characters.

[4] Conversion to 'ASCII//TRANSLIT' may depend on current locale and OS.

source code distribution, but if shared library is used instead of bundled with PHP sources, then UTF-8 support state may depend on you operating system.

Use the following code to check, if PCRE UTF-8 support is enabled:

```
if (@preg_match('/\pL/u', 'a') == 1) {
    echo "PCRE unicode support is turned on.\n";
} else {
    echo "PCRE unicode support is turned off.\n";
}
```

Case insensitive versions of UTF-8 compatible analyzers also need mbstring [http://www.php.net/manual/en/ref.mbstring.php] extension to be enabled.

If you don't want mbstring extension to be turned on, but need case insensitive search, you may use the following approach: normalize source data before indexing and query string before searching by converting them to lowercase:

```
// Indexing
setlocale(LC_CTYPE, 'de_DE.iso-8859-1');

...

Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8());

...

$doc = new Zend_Search_Lucene_Document();

$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  strtolower($contents)));

// Title field for search through (indexed, unstored)
$doc->addField(Zend_Search_Lucene_Field::UnStored('title',
                                                  strtolower($title)));

// Title field for retrieving (unindexed, stored)
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('_title', $title));
```

```
// Searching
setlocale(LC_CTYPE, 'de_DE.iso-8859-1');

...

Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Utf8());

...

$hits = $index->find(strtolower($query));
```

Extensibility

Text Analysis

The Zend_Search_Lucene_Analysis_Analyzer class is used by the indexer to tokenize document text fields.

The Zend_Search_Lucene_Analysis_Analyzer::getDefault() and Zend_Search_Lucene_Analysis_Analyzer::setDefault() methods are used to get and set the default analyzer.

You can assign your own text analyzer or choose it from the set of predefined analyzers: Zend_Search_Lucene_Analysis_Analyzer_Common_Text and Zend_Search_Lucene_Analysis_Analyzer_Common_Text_CaseInsensitive (default). Both of them interpret tokens as sequences of letters. Zend_Search_Lucene_Analysis_Analyzer_Common_Text_CaseInsensitive converts all tokens to lower case.

To switch between analyzers:

```
Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new Zend_Search_Lucene_Analysis_Analyzer_Common_Text());
...
$index->addDocument($doc);
```

The Zend_Search_Lucene_Analysis_Analyzer_Common class is designed to be an ancestor of all user defined analyzers. User should only define the reset() and nextToken() methods, which takes its string from the $_input member and returns tokens one by one (a null value indicates the end of the stream).

The nextToken() method should call the normalize() method on each token. This will allow you to use token filters with your analyzer.

Here is an example of a custom analyzer, which accepts words with digits as terms:

Example 38.1. Custom text Analyzer.

```
/**
 * Here is a custom text analyser, which treats words with digits as
 * one term
 */

class My_Analyzer extends Zend_Search_Lucene_Analysis_Analyzer_Common
{
    private $_position;

    /**
     * Reset token stream
     */
    public function reset()
    {
        $this->_position = 0;
    }

    /**
     * Tokenization stream API
     * Get next token
     * Returns null at the end of stream
     *
     * @return Zend_Search_Lucene_Analysis_Token|null
     */
    public function nextToken()
    {
        if ($this->_input === null) {
            return null;
        }

        while ($this->_position < strlen($this->_input)) {
            // skip white space
            while ($this->_position < strlen($this->_input) &&
                   !ctype_alnum( $this->_input[$this->_position] )) {
                $this->_position++;
            }

            $termStartPosition = $this->_position;

            // read token
            while ($this->_position < strlen($this->_input) &&
                   ctype_alnum( $this->_input[$this->_position] )) {
                $this->_position++;
            }

            // Empty token, end of stream.
            if ($this->_position == $termStartPosition) {
                return null;
            }
            $token = new Zend_Search_Lucene_Analysis_Token(
                                      substr($this->_input,
                                             $termStartPosition,
                                             $this->_position -
                                             $termStartPosition),
                                      $termStartPosition,
                                      $this->_position);
            $token = $this->normalize($token);
            if ($token !== null) {
                return $token;
            }
            // Continue if token is skipped
        }

        return null;
    }
}

Zend_Search_Lucene_Analysis_Analyzer::setDefault(
    new My_Analyzer());
```

Tokens Filtering

The `Zend_Search_Lucene_Analysis_Analyzer_Common` analyzer also offers a token filtering mechanism.

The `Zend_Search_Lucene_Analysis_TokenFilter` class provides an abstract interface for such filters. Your own filters should extend this class either directly or indirectly.

Any custom filter must implement the `normalize()` method which may transform input token or signal that the current token should be skipped.

There are three filters already defined in the analysis subpackage:

- `Zend_Search_Lucene_Analysis_TokenFilter_LowerCase`
- `Zend_Search_Lucene_Analysis_TokenFilter_ShortWords`
- `Zend_Search_Lucene_Analysis_TokenFilter_StopWords`

The `LowerCase` filter is already used for `Zend_Search_Lucene_Analysis_Analyzer_Common_Text_CaseInsensitive` analyzer by default.

The `ShortWords` and `StopWords` filters may be used with pre-defined or custom analyzers like this:

```
$stopWords = array('a', 'an', 'at', 'the', 'and', 'or', 'is', 'am');
$stopWordsFilter =
    new Zend_Search_Lucene_Analysis_TokenFilter_StopWords($stopWords);

$analyzer =
    new Zend_Search_Lucene_Analysis_Analyzer_Common_TextNum_CaseInsensitive();
$analyzer->addFilter($stopWordsFilter);
```

```
Zend_Search_Lucene_Analysis_Analyzer::setDefault($analyzer);
```

```
$shortWordsFilter = new Zend_Search_Lucene_Analysis_TokenFilter_ShortWords();

$analyzer =
    new Zend_Search_Lucene_Analysis_Analyzer_Common_TextNum_CaseInsensitive();
$analyzer->addFilter($shortWordsFilter);

Zend_Search_Lucene_Analysis_Analyzer::setDefault($analyzer);
```

The `Zend_Search_Lucene_Analysis_TokenFilter_StopWords` constructor takes an array of stop-words as an input. But stop-words may be also loaded from a file:

```
$stopWordsFilter = new Zend_Search_Lucene_Analysis_TokenFilter_StopWords();
$stopWordsFilter->loadFromFile($my_stopwords_file);

$analyzer =
   new Zend_Search_Lucene_Analysis_Analyzer_Common_TextNum_CaseInsensitive();
$analyzer->addFilter($stopWordsFilter);

Zend_Search_Lucene_Analysis_Analyzer::setDefault($analyzer);
```

This file should be a common text file with one word in each line. The '#' character marks a line as a comment.

The `Zend_Search_Lucene_Analysis_TokenFilter_ShortWords` constructor has one optional argument. This is the word length limit, set by default to 2.

Scoring Algorithms

The score of a document `d` for a query `q` is defined as follows:

```
score(q,d) = sum( tf(t in d) * idf(t) * getBoost(t.field in d) *
lengthNorm(t.field in d) ) * coord(q,d) * queryNorm(q)
```

tf(t in d) - `Zend_Search_Lucene_Search_Similarity::tf($freq)` - a score factor based on the frequency of a term or phrase in a document.

idf(t) - `Zend_Search_Lucene_Search_Similarity::tf($term, $reader)` - a score factor for a simple term with the specified index.

getBoost(t.field in d) - the boost factor for the term field.

lengthNorm($term) - the normalization value for a field given the total number of terms contained in a field. This value is stored within the index. These values, together with field boosts, are stored in an index and multiplied into scores for hits on each field by the search code.

Matches in longer fields are less precise, so implementations of this method usually return smaller values when numTokens is large, and larger values when numTokens is small.

coord(q,d) - `Zend_Search_Lucene_Search_Similarity::coord($overlap, $maxOverlap)` - a score factor based on the fraction of all query terms that a document contains.

The presence of a large portion of the query terms indicates a better match with the query, so implementations of this method usually return larger values when the ratio between these parameters is large and smaller values when the ratio between them is small.

queryNorm(q) - the normalization value for a query given the sum of the squared weights of each of the query terms. This value is then multiplied into the weight of each query term.

This does not affect ranking, but rather just attempts to make scores from different queries comparable.

The scoring algorithm can be customized by defining your own Similarity class. To do this extend the Zend_Search_Lucene_Search_Similarity class as defined below, then use the `Zend_Search_Lucene_Search_Similarity::setDefault($similarity);` method to set it as default.

```
class MySimilarity extends Zend_Search_Lucene_Search_Similarity {
    public function lengthNorm($fieldName, $numTerms) {
        return 1.0/sqrt($numTerms);
    }

    public function queryNorm($sumOfSquaredWeights) {
        return 1.0/sqrt($sumOfSquaredWeights);
    }

    public function tf($freq) {
        return sqrt($freq);
    }

    /**
     * It's not used now. Computes the amount of a sloppy phrase match,
     * based on an edit distance.
     */
    public function sloppyFreq($distance) {
        return 1.0;
    }

    public function idfFreq($docFreq, $numDocs) {
        return log($numDocs/(float)($docFreq+1)) + 1.0;
    }

    public function coord($overlap, $maxOverlap) {
        return $overlap/(float)$maxOverlap;
    }
}

$mySimilarity = new MySimilarity();
Zend_Search_Lucene_Search_Similarity::setDefault($mySimilarity);
```

Storage Containers

The abstract class `Zend_Search_Lucene_Storage_Directory` defines directory functionality.

The `Zend_Search_Lucene` constructor uses either a string or a `Zend_Search_Lucene_Storage_Directory` object as an input.

The `Zend_Search_Lucene_Storage_Directory_Filesystem` class implements directory functionality for a file system.

If a string is used as an input for the `Zend_Search_Lucene` constructor, then the index reader (`Zend_Search_Lucene` object) treats it as a file system path and instantiates the `Zend_Search_Lucene_Storage_Directory_Filesystem` object.

You can define your own directory implementation by extending the `Zend_Search_Lucene_Storage_Directory` class.

`Zend_Search_Lucene_Storage_Directory` methods:

```
abstract class Zend_Search_Lucene_Storage_Directory {
/**
 * Closes the store.
 *
 * @return void
 */
abstract function close();


/**
 * Creates a new, empty file in the directory with the given $filename.
 *
 * @param string $name
 * @return void
 */
abstract function createFile($filename);


/**
 * Removes an existing $filename in the directory.
 *
 * @param string $filename
 * @return void
 */
abstract function deleteFile($filename);


/**
 * Returns true if a file with the given $filename exists.
 *
 * @param string $filename
 * @return boolean
 */
abstract function fileExists($filename);
/**
 * Returns the length of a $filename in the directory.
 *
 * @param string $filename
 * @return integer
 */
abstract function fileLength($filename);


/**
 * Returns the UNIX timestamp $filename was last modified.
 *
 * @param string $filename
 * @return integer
 */
abstract function fileModified($filename);


/**
 * Renames an existing file in the directory.
 *
 * @param string $from
 * @param string $to
 * @return void
 */
abstract function renameFile($from, $to);


/**
 * Sets the modified time of $filename to now.
 *
 * @param string $filename
 * @return void
 */
abstract function touchFile($filename);


/**
 * Returns a Zend_Search_Lucene_Storage_File object for a given
 * $filename in the directory.
 *
 * @param string $filename
 * @return Zend_Search_Lucene_Storage_File
 */
abstract function getFileObject($filename);

}
```

The `getFileObject($filename)` method of a `Zend_Search_Lucene_Storage_Directory` instance returns a `Zend_Search_Lucene_Storage_File` object.

The `Zend_Search_Lucene_Storage_File` abstract class implements file abstraction and index file reading primitives.

You must also extend `Zend_Search_Lucene_Storage_File` for your directory implementation.

Only two methods of `Zend_Search_Lucene_Storage_File` must be overridden in your implementation:

```
class MyFile extends Zend_Search_Lucene_Storage_File {
    /**
     * Sets the file position indicator and advances the file pointer.
     * The new position, measured in bytes from the beginning of the file,
     * is obtained by adding offset to the position specified by whence,
     * whose values are defined as follows:
     * SEEK_SET - Set position equal to offset bytes.
     * SEEK_CUR - Set position to current location plus offset.
     * SEEK_END - Set position to end-of-file plus offset. (To move to
     * a position before the end-of-file, you need to pass a negative value
     * in offset.)
     * Upon success, returns 0; otherwise, returns -1
     *
     * @param integer $offset
     * @param integer $whence
     * @return integer
     */
    public function seek($offset, $whence=SEEK_SET) {
        ...
    }

    /**
     * Read a $length bytes from the file and advance the file pointer.
     *
     * @param integer $length
     * @return string
     */
    protected function _fread($length=1) {
        ...
    }
}
```

Interoperating with Java Lucene

File Formats

Zend_Search_Lucene index file formats are binary compatible with Java Lucene version 1.4 and greater.

A detailed description of this format is available here: http://lucene.apache.org/java/docs/fileformats.html [5].

[5]The currently supported Lucene index file format version is 2.3 (starting from ZF 1.6).

Index Directory

After index creation, the index directory will contain several files:

- The `segments` file is a list of index segments.
- The `*.cfs` files contain index segments. Note! An optimized index always has only one segment.
- The `deletable` file is a list of files that are no longer used by the index, but which could not be deleted.

Java Source Code

The Java program listing below provides an example of how to index a file using Java Lucene:

```
/**
* Index creation:
*/
import org.apache.lucene.index.IndexWriter;
import org.apache.lucene.document.*;

import java.io.*

...

IndexWriter indexWriter = new IndexWriter("/data/my_index",
                                          new SimpleAnalyzer(), true);

...

String filename = "/path/to/file-to-index.txt"
File f = new File(filename);

Document doc = new Document();
doc.add(Field.Text("path", filename));
doc.add(Field.Keyword("modified",DateField.timeToString(f.lastModified())));
doc.add(Field.Text("author", "unknown"));
FileInputStream is = new FileInputStream(f);
Reader reader = new BufferedReader(new InputStreamReader(is));
doc.add(Field.Text("contents", reader));

indexWriter.addDocument(doc);
```

Advanced

Starting from 1.6, handling index format transformations.

Zend_Search_Lucene component works with Java Lucene 1.4-1.9, 2.1 and 2.3 index formats.

Current index format may be requested using `$index->getFormatVersion()` call. It returns one of the following values:

- `Zend_Search_Lucene::FORMAT_PRE_2_1` for Java Lucene 1.4-1.9 index format.
- `Zend_Search_Lucene::FORMAT_2_1` for Java Lucene 2.1 index format (also used for Lucene 2.2).
- `Zend_Search_Lucene::FORMAT_2_3` for Java Lucene 2.3 index format.

Index modifications are performed **only** if any index update is done. That happens if a new document is added to an index or index optimization is started manually by `$index->optimize()` call.

In a such case Zend_Search_Lucene may convert index to the higher format version. That **always** happens for the indices in `Zend_Search_Lucene::FORMAT_PRE_2_1` format, which are automatically converted to 2.1 format.

You may manage conversion process and assign target index format by `$index->setFormatVersion()` which takes `Zend_Search_Lucene::FORMAT_2_1` or `Zend_Search_Lucene::FORMAT_2_3` constant as a parameter:

- `Zend_Search_Lucene::FORMAT_2_1` actually does nothing since pre-2.1 indices are automatically converted to 2.1 format.
- `Zend_Search_Lucene::FORMAT_2_3` forces conversion to the 2.3 format.

Backward conversions are not supported.

Important!

Once index is converted to upper version it can't be converted back. So make a backup of your index when you plan migration to upper version, but want to have possibility to go back.

Using the index as static property

The `Zend_Search_Lucene` object uses the destructor method to commit changes and clean up resources.

It stores added documents in memory and dumps new index segment to disk depending on `MaxBufferedDocs` parameter.

If `MaxBufferedDocs` limit is not reached then there are some "unsaved" documents which are saved as a new segment in the object's destructor method. The index auto-optimization procedure is invoked if necessary depending on the values of the `MaxBufferedDocs`, `MaxMergeDocs` and `MergeFactor` parameters.

Static object properties (see below) are destroyed *after* the last line of the executed script.

```
class Searcher {
    private static $_index;

    public static function initIndex() {
        self::$_index = Zend_Search_Lucene::open('path/to/index');
    }
}

Searcher::initIndex();
```

All the same, the destructor for static properties is correctly invoked at this point in the program's execution.

One potential problem is exception handling. Exceptions thrown by destructors of static objects don't have context, because the destructor is executed after the script has already completed.

You might see a "Fatal error: Exception thrown without a stack frame in Unknown on line 0" error message instead of exception description in such cases.

Zend_Search_Lucene provides a workaround to this problem with the `commit()` method. It saves all unsaved changes and frees memory used for storing new segments. You are free to use the commit operation any time- or even several times- during script execution. You can still use the `Zend_Search_Lucene` object for searching, adding or deleting document after the commit operation. But the `commit()` call guarantees that if there are no document added or deleted after the call to `commit()`, then the `Zend_Search_Lucene` destructor has nothing to do and will not throw exception:

```
class Searcher {
    private static $_index;

    public static function initIndex() {
        self::$_index = Zend_Search_Lucene::open('path/to/index');
    }

    ...

    public static function commit() {
        self::$_index->commit();
    }
}

Searcher::initIndex();

...

// Script shutdown routine
...
Searcher::commit();
...
```

Best Practices

Field names

There are no limitations for field names in Zend_Search_Lucene.

Nevertheless it's a good idea not to use '*id*' and '*score*' names to avoid ambiguity in `QueryHit` properties names.

The `Zend_Search_Lucene_Search_QueryHit` `id` and `score` properties always refer to internal Lucene document id and hit score. If the indexed document has the same stored fields, you have to use the `getDocument()` method to access them:

```
$hits = $index->find($query);

foreach ($hits as $hit) {
    // Get 'title' document field
    $title = $hit->title;

    // Get 'contents' document field
    $contents = $hit->contents;


    // Get internal Lucene document id
    $id = $hit->id;

    // Get query hit score
    $score = $hit->score;


    // Get 'id' document field
    $docId = $hit->getDocument()->id;

    // Get 'score' document field
    $docId = $hit->getDocument()->score;

    // Another way to get 'title' document field
    $title = $hit->getDocument()->title;
}
```

Indexing performance

Indexing performance is a compromise between used resources, indexing time and index quality.

Index quality is completely determined by number of index segments.

Each index segment is entirely independent portion of data. So indexes containing more segments need more memory and time for searching.

Index optimization is a process of merging several segments into a new one. A fully optimized index contains only one segment.

Full index optimization may be performed with the `optimize()` method:

```
$index = Zend_Search_Lucene::open($indexPath);

$index->optimize();
```

Index optimization works with data streams and doesn't take a lot of memory but does require processor resources and time.

Lucene index segments are not updatable by their nature (the update operation requires the segment file to be completely rewritten). So adding new document(s) to an index always generates a new segment. This, in turn, decreases index quality.

An index auto-optimization process is performed after each segment generation and consists of merging partial segments.

There are three options to control the behavior of auto-optimization (see Index optimization section):

- *MaxBufferedDocs* is the number of documents that can be buffered in memory before a new segment is generated and written to the hard drive.
- *MaxMergeDocs* is the maximum number of documents merged by auto-optimization process into a new segment.
- *MergeFactor* determines how often auto-optimization is performed.

Note

All these options are Zend_Search_Lucene object properties- not index properties. They affect only current `Zend_Search_Lucene` object behavior and may vary for different scripts.

MaxBufferedDocs doesn't have any effect if you index only one document per script execution. On the other hand, it's very important for batch indexing. Greater values increase indexing performance, but also require more memory.

There is simply no way to calculate the best value for the *MaxBufferedDocs* parameter because it depends on average document size, the analyzer in use and allowed memory.

A good way to find the right value is to perform several tests with the largest document you expect to be added to the index [6]. It's a best practice not to use more than a half of the allowed memory.

MaxMergeDocs limits the segment size (in terms of documents). It therefore also limits auto-optimization time by guaranteeing that the `addDocument()` method is not executed more than a certain number of times. This is very important for interactive applications.

Lowering the *MaxMergeDocs* parameter also may improve batch indexing performance. Index auto-optimization is an iterative process and is performed from bottom up. Small segments are merged into larger segment, which are in turn merged into even larger segments and so on. Full index optimization is achieved when only one large segment file remains.

Small segments generally decrease index quality. Many small segments may also trigger the "Too many open files" error determined by OS limitations [7].

in general, background index optimization should be performed for interactive indexing mode and *MaxMergeDocs* shouldn't be too low for batch indexing.

MergeFactor affects auto-optimization frequency. Lower values increase the quality of unoptimized indexes. Larger values increase indexing performance, but also increase the number of merged segments. This again may trigger the "Too many open files" error.

[6]`memory_get_usage()` and `memory_get_peak_usage()` may be used to control memory usage.
[7]Zend_Search_Lucene keeps each segment file opened to improve search performance.

MergeFactor groups index segments by their size:

1. Not greater than *MaxBufferedDocs*.
2. Greater than *MaxBufferedDocs*, but not greater than *MaxBufferedDocs***MergeFactor*.
3. Greater than *MaxBufferedDocs***MergeFactor*, but not greater than *MaxBufferedDocs***MergeFactor***MergeFactor*.
4. ...

Zend_Search_Lucene checks during each `addDocument()` call to see if merging any segments may move the newly created segment into the next group. If yes, then merging is performed.

So an index with N groups may contain *MaxBufferedDocs* + (N-1)**MergeFactor* segments and contains at least *MaxBufferedDocs**$MergeFactor^{(N-1)}$ documents.

This gives good approximation for the number of segments in the index:

NumberOfSegments <= *MaxBufferedDocs* + *MergeFactor**$\log_{MergeFactor}$ (*NumberOfDocuments*/*MaxBufferedDocs*)

MaxBufferedDocs is determined by allowed memory. This allows for the appropriate merge factor to get a reasonable number of segments.

Tuning the *MergeFactor* parameter is more effective for batch indexing performance than *MaxMergeDocs*. But it's also more course-grained. So use the estimation above for tuning *MergeFactor*, then play with *MaxMergeDocs* to get best batch indexing performance.

Index during Shut Down

The `Zend_Search_Lucene` instance performs some work at exit time if any documents were added to the index but not written to a new segment.

It also may trigger an auto-optimization process.

The index object is automatically closed when it, and all returned QueryHit objects, go out of scope.

If index object is stored in global variable than it's closed only at the end of script execution[8].

PHP exception processing is also shut down at this moment.

It doesn't prevent normal index shutdown process, but may prevent accurate error diagnostic if any error occurs during shutdown.

There are two ways with which you may avoid this problem.

The first is to force going out of scope:

```
$index = Zend_Search_Lucene::open($indexPath);

...
```

[8]This also may occur if the index or QueryHit instances are referred to in some cyclical data structures, because PHP garbage collects objects with cyclic references only at the end of script execution.

```
unset($index);
```

And the second is to perform a commit operation before the end of script execution:

```
$index = Zend_Search_Lucene::open($indexPath);

$index->commit();
```

This possibility is also described in the "Advanced. Using index as static property" section.

Retrieving documents by unique id

It's a common practice to store some unique document id in the index. Examples include url, path, or database id.

`Zend_Search_Lucene` provides a `termDocs()` method for retrieving documents containing specified terms.

This is more efficient than using the `find()` method:

```
// Retrieving documents with find() method using a query string
$query = $idFieldName . ':' . $docId;
$hits  = $index->find($query);
foreach ($hits as $hit) {
    $title    = $hit->title;
    $contents = $hit->contents;
    ...
}
...

// Retrieving documents with find() method using the query API
$term = new Zend_Search_Lucene_Index_Term($docId, idFieldName);
$query = new Zend_Search_Lucene_Search_Query_Term($term);
$hits  = $index->find($query);
foreach ($hits as $hit) {
    $title    = $hit->title;
    $contents = $hit->contents;
    ...
}

...

// Retrieving documents with termDocs() method
$term = new Zend_Search_Lucene_Index_Term($docId, idFieldName);
$docIds  = $index->termDocs($term);
foreach ($docIds as $id) {
    $doc = $index->getDocument($id);
    $title    = $doc->title;
    $contents = $doc->contents;
    ...
}
```

Memory Usage

Zend_Search_Lucene is a relatively memory-intensive module.

It uses memory to cache some information and optimize searching and indexing performance.

The memory required differs for different modes.

The terms dictionary index is loaded during the search. It's actually each 128th [9] term of the full dictionary.

Thus memory usage is increased if you have a high number of unique terms. This may happen if you use untokenized phrases as a field values or index a large volume of non-text information.

An unoptimized index consists of several segments. It also increases memory usage. Segments are independent, so each segment contains its own terms dictionary and terms dictionary index. If an index consists of *N* segments it may increase memory usage by *N* times in worst case. Perform index optimization to merge all segments into one to avoid such memory consumption.

Indexing uses the same memory as searching plus memory for buffering documents. The amount of memory used may be managed with *MaxBufferedDocs* parameter.

Index optimization (full or partial) uses stream-style data processing and doesn't require a lot of memory.

Encoding

Zend_Search_Lucene works with UTF-8 strings internally. So all strings returned by Zend_Search_Lucene are UTF-8 encoded.

You shouldn't be concerned with encoding if you work with pure ASCII data, but you should be careful if this is not the case.

Wrong encoding may cause error notices at the encoding conversion time or loss of data.

Zend_Search_Lucene offers a wide range of encoding possibilities for indexed documents and parsed queries.

Encoding may be explicitly specified as an optional parameter of field creation methods:

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::Text('title',
                                              $title,
                                              'iso-8859-1'));
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
                                                  $contents,
```

[9] The Lucene file format allows you to configure this number, but Zend_Search_Lucene doesn't expose this in its API. Nevertheless you still have the ability to configure this value if the index is prepared with another Lucene implementation.

```
                                                         'utf-8'));
```

This is the best way to avoid ambiguity in the encoding used.

If optional encoding parameter is omitted, then the current locale is used. The current locale may contain character encoding data in addition to the language specification:

```
setlocale(LC_ALL, 'fr_FR');
...

setlocale(LC_ALL, 'de_DE.iso-8859-1');
...

setlocale(LC_ALL, 'ru_RU.UTF-8');
...
```

The same approach is used to set query string encoding.

If encoding is not specified, then the current locale is used to determine the encoding.

Encoding may be passed as an optional parameter, if the query is parsed explicitly before search:

```
$query =
    Zend_Search_Lucene_Search_QueryParser::parse($queryStr, 'iso-8859-5');
$hits = $index->find($query);
...
```

The default encoding may also be specified with `setDefaultEncoding()` method:

```
Zend_Search_Lucene_Search_QueryParser::setDefaultEncoding('iso-8859-1');
$hits = $index->find($queryStr);
...
```

The empty string implies 'current locale'.

If the correct encoding is specified it can be correctly processed by analyzer. The actual behavior depends on which analyzer is used. See the Character Set documentation section for details.

Index maintenance

It should be clear that Zend_Search_Lucene as well as any other Lucene implementation does not comprise a "database".

Indexes should not be used for data storage. They do not provide partial backup/restore functionality, journaling, logging, transactions and many other feautures associated with database management systems.

Nevertheless, Zend_Search_Lucene attempts to keep indexes in a consistent state at all times.

Index backup and restoration should be performed by copying the contents of the index folder.

If index corruption occures for any reason, the corrupted index should be restored or completely rebuilt.

So it's a good idea to backup large indexes and store changelogs to perform manual restoration and roll-forward operations if necessary. This practice dramatically reduces index restoration time.

Chapter 39. Zend_Server

Introduction

The `Zend_Server` family of classes provides functionality for the various server classes, including `Zend_XmlRpc_Server`, `Zend_Rest_Server`, `Zend_Json_Server` and `Zend_Soap_Wsdl`. `Zend_Server_Interface` 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_Interface`'s `getFunctions()` and `loadFunctions()` methods.

Zend_Server_Reflection

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, and extends it to provide 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/method descriptions.

Typically, this functionality will only be used by developers of server classes for the framework.

Usage

Basic usage is simple:

```
$class    = Zend_Server_Reflection::reflectClass('My_Class');
$function = Zend_Server_Reflection::reflectFunction('my_function');

// Get prototypes
$prototypes = $reflection->getPrototypes();

// Loop through each prototype for the function
foreach ($prototypes as $prototype) {

    // Get prototype return type
    echo "Return type: ", $prototype->getReturnType(), "\n";

    // Get prototype parameters
    $parameters = $prototype->getParameters();

    echo "Parameters: \n";
    foreach ($parameters as $parameter) {
        // Get parameter type
        echo "    ", $parameter->getType(), "\n";
    }
}
```

```
// Get namespace for a class, function, or method.
// Namespaces may be set at instantiation time (second argument), or using
// setNamespace()
$reflection->getNamespace();
```

`reflectFunction()` returns a `Zend_Server_Reflection_Function` object; `reflectClass` returns a `Zend_Server_Reflection_Class` object. Please refer to the API documentation to know what methods are available to each.

Chapter 40. Zend_Service

Introduction

`Zend_Service` is an abstract class which serves as a foundation for web service implementations, such as SOAP or REST.

If you need support for generic, XML-based REST services, you may want to look at `Zend_Rest_Client`.

In addition to being able to extend the `Zend_Service` and use `Zend_Rest_Client` for REST-based web services, Zend also provides support for popular web services. See the following sections for specific information on each supported web service.

- Akismet
- Amazon
- Audioscrobbler
- Del.icio.us
- Flickr
- Simpy
- SlideShare
- StrikeIron
- Yahoo!

Additional services are coming in the future.

Zend_Service_Akismet

Introduction

`Zend_Service_Akismet` provides a client for the Akismet API [http://akismet.com/development/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). Originally intended to help categorize and identify spam for Wordpress, it can be used for any type of data.

Akismet requires an API key for usage. You may get one for signing up for a WordPress.com [http://wordpress.com/] account. You do not need to activate a blog; simply acquiring the account will provide you with the API key.

Additionally, Akismet requires that all requests contain a URL to the resource for which data is being filtered, and, 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()` accessor, or overridden by specifying a 'blog' key in the various method calls.

Verify an API key

`Zend_Service_Akismet::verifyKey($key)` is used to verify that an Akismet API key is valid. In most cases, you will not need to check, but if you need a sanity check, or to determine if a newly acquired key is active, you may do so with this method.

```
// Instantiate with the API key and a URL to the application or
// resource being used
$akismet = new Zend_Service_Akismet($apiKey,
                                    'http://framework.zend.com/wiki/');
if ($akismet->verifyKey($apiKey) {
    echo "Key is valid.\n";
} else {
    echo "Key is not valid\n";
}
```

If called with no arguments, `verifyKey()` uses the API key provided to the constructor.

`verifyKey()` implements Akismet's `verify-key` REST method.

Check for spam

`Zend_Service_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 specifically specified in the API include 'comment', 'trackback', 'pingback', and an empty string (''), but it may be any value.
- `comment_author`, name of the person submitting the data.
- `comment_author_email`, email of the person submitting the data.
- `comment_author_url`, URL or home page of the person submitting the data.
- `comment_content`, 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, and throw an exception if the API key is invalid.

Example 40.1. isSpam() Usage

```
$data = array(
    'user_ip'              => '111.222.111.222',
    'user_agent'           => 'Mozilla/5.0 ' . (Windows; U; Windows NT ' .
                              '5.2; en-GB; rv:1.8.1) Gecko/20061010 ' .
                              'Firefox/2.0',
    'comment_type'         => 'contact',
    'comment_author'       => 'John Doe',
    'comment_author_email' => 'nospam@myhaus.net',
    'comment_content'      => "I'm not a spammer, honest!"
);
if ($akismet->isSpam($data)) {
    echo "Sorry, but we think you're a spammer.";
} else {
    echo "Welcome to our site!";
}
```

`isSpam()` implements the `comment-check` Akismet API method.

Submitting known spam

Occasionally spam data will get through the filter. If in your review of incoming data you discover spam that you feel should have been caught, you can submit it to Akismet to help improve their filter.

`Zend_Service_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.

Example 40.2. submitSpam() Usage

```
$data = array(
    'user_ip'              => '111.222.111.222',
    'user_agent'           => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
                              'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
    'comment_type'         => 'contact',
    'comment_author'       => 'John Doe',
    'comment_author_email' => 'nospam@myhaus.net',
    'comment_content'      => "I'm not a spammer, honest!"
);
$akismet->submitSpam($data));
```

`submitSpam()` implements the `submit-spam` Akismet API method.

Submitting false positives (ham)

Occasionally data will 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).

`Zend_Service_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.

Example 40.3. submitHam() Usage

```
$data = array(
    'user_ip'              => '111.222.111.222',
    'user_agent'           => 'Mozilla/5.0 (Windows; U; Windows NT 5.2;' .
                              'en-GB; rv:1.8.1) Gecko/20061010 Firefox/2.0',
    'comment_type'         => 'contact',
    'comment_author'       => 'John Doe',
    'comment_author_email' => 'nospam@myhaus.net',
    'comment_content'      => "I'm not a spammer, honest!"
);
$akismet->submitHam($data));
```

`submitHam()` implements the `submit-ham` Akismet API method.

Zend-specific Accessor Methods

While the Akismet API only specifies four methods, `Zend_Service_Akismet` has several additional accessors that may be used for 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_Framework::VERSION` constant.

Zend_Service_Amazon

Introduction

`Zend_Service_Amazon` is a simple API for using Amazon web services. `Zend_Service_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.

`Zend_Service_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 `Zend_Service_Amazon`, you should already have an Amazon developer API key. To get a key and for more information, please visit the Amazon Web Services [http://www.amazon.com/gp/aws/landing.html] web site.

Attention

Your Amazon developer API key is linked to your Amazon identity, so take appropriate measures to keep your API key private.

Example 40.4. Search Amazon Using the Traditional API

In this example, we search for PHP books at Amazon and loop through the results, printing them.

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY');
$results = $amazon->itemSearch(array('SearchIndex' => 'Books',
                                     'Keywords' => 'php'));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

Example 40.5. Search Amazon Using the Query API

Here, we also search for PHP books at Amazon, but we instead use the Query API, which resembles the Fluent Interface design pattern.

```
$query = new Zend_Service_Amazon_Query('AMAZON_API_KEY');
$query->category('Books')->Keywords('PHP');
$results = $query->search();
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

Country Codes

By default, `Zend_Service_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:

Example 40.6. Choosing an Amazon Web Service Country

```
// Connect to Amazon in Japan
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY', 'JP');
```

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.

Example 40.7. Looking up a Specific Amazon Item by ASIN

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY');
$item = $amazon->itemLookup('B0000A432X');
```

The `itemLookup()` method also accepts an optional second parameter for handling search options. For full details, including a list of available options, please see the relevant Amazon documentation [http://www.amazon.com/gp/aws/sdk/main.html/103-9285448-4703844?s=AWSEcommerceService&v=2005-10-05&p=ApiReference/ItemLookupOperation].

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:

Example 40.8. Performing Amazon Item Searches

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY');
$results = $amazon->itemSearch(array('SearchIndex' => 'Books',
                                     'Keywords' => 'php'));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

Example 40.9. Using the `ResponseGroup` Option

The `ResponseGroup` option is used to control the specific information that will be returned in the response.

```
$amazon = new Zend_Service_Amazon('AMAZON_API_KEY');
$results = $amazon->itemSearch(array(
    'SearchIndex'   => 'Books',
    'Keywords'      => 'php',
    'ResponseGroup' => 'Small,ItemAttributes,Images,SalesRank,Reviews,' .
                       'EditorialReview,Similarities,ListmaniaLists'
    ));
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

The `itemSearch()` method accepts a single array parameter for handling search options. For full details, including a list of available options, please see the relevant Amazon documentation [http://www.amazon.com/gp/aws/sdk/main.html/103-9285448-4703844?s=AWSEcommerceService&v=2005-10-05&p=ApiReference/ItemSearchOperation]

Tip

The `Zend_Service_Amazon_Query` class is an easy to use wrapper around this method.

Using the Alternative Query API

Introduction

`Zend_Service_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 `Zend_Service_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:

Example 40.10. Search Amazon Using the Alternative Query API

In this example, the alternative query API is used as a fluent interface to specify options and their respective values:

```
$query = new Zend_Service_Amazon_Query('MY_API_KEY');
$query->Category('Books')->Keywords('PHP');
$results = $query->search();
foreach ($results as $result) {
    echo $result->Title . '<br />';
}
```

This sets the option `Category` to "Books" and `Keywords` to "PHP".

For more information on the available options, please refer to the relevant Amazon documentation [http://www.amazon.com/gp/aws/sdk/main.html/102-9041115-9057709?s=AWSEcommerceService&v=2005-10-05&p=ApiReference/ItemSearchOperation].

Zend_Service_Amazon Classes

The following classes are all returned by `Zend_Service_Amazon::itemLookup()` and `Zend_Service_Amazon::itemSearch()`:

- `Zend_Service_Amazon_Item`
- `Zend_Service_Amazon_Image`
- `Zend_Service_Amazon_ResultSet`
- `Zend_Service_Amazon_OfferSet`
- `Zend_Service_Amazon_Offer`

- `Zend_Service_Amazon_SimilarProduct`
- `Zend_Service_Amazon_Accessories`
- `Zend_Service_Amazon_CustomerReview`
- `Zend_Service_Amazon_EditorialReview`
- `Zend_Service_Amazon_ListMania`

Zend_Service_Amazon_Item

`Zend_Service_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.

Zend_Service_Amazon_Item::asXML()

```
string asXML();
```

Return the original XML for the item

Properties

`Zend_Service_Amazon_Item` has a number of properties directly related to their standard Amazon API counterparts.

Table 40.1. Zend_Service_Amazon_Item Properties

Name	Type	Description
ASIN	string	Amazon Item ID
DetailPageURL	string	URL to the Items Details Page
SalesRank	int	Sales Rank for the Item
SmallImage	Zend_Service_Amazon_Image	Small Image of the Item
MediumImage	Zend_Service_Amazon_Image	Medium Image of the Item
LargeImage	Zend_Service_Amazon_Image	Large Image of the Item
Subjects	array	Item Subjects
Offers	`Zend_Service_Amazon_OfferSet`	Offer Summary and Offers for the Item
CustomerReviews	array	Customer reviews represented as an array of `Zend_Service_Amazon_CustomerReview` objects
EditorialReviews	array	Editorial reviews represented as an array of `Zend_Service_Amazon_EditorialReview` objects
SimilarProducts	array	Similar Products represented as an array of `Zend_Service_Amazon_SimilarProduct` objects
Accessories	array	Accessories for the item represented as an array of `Zend_Service_Amazon_Accessories` 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 `Zend_Service_Amazon_ListmainList` objects
PromotionalTag	string	Item Promotional Tag

Back to Class List

Zend_Service_Amazon_Image

`Zend_Service_Amazon_Image` represents a remote Image for a product.

Properties

Table 40.2. Zend_Service_Amazon_Image Properties

Name	Type	Description
Url	Zend_Uri	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

Zend_Service_Amazon_ResultSet

`Zend_Service_Amazon_ResultSet` objects are returned by Zend_Service_Amazon::itemSearch() and allow you to easily handle the multiple results returned.

SeekableIterator

Implements the `SeekableIterator` for easy iteration (e.g. using `foreach`), as well as direct access to a specific result using `seek()`.

Zend_Service_Amazon_ResultSet::totalResults()

```
int totalResults();
```

Returns the total number of results returned by the search

Back to Class List

Zend_Service_Amazon_OfferSet

Each result returned by Zend_Service_Amazon::itemSearch() and Zend_Service_Amazon::itemLookup() contains a `Zend_Service_Amazon_OfferSet` object through which pricing information for the item can be retrieved.

Properties

Table 40.3. Zend_Service_Amazon_OfferSet 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 `Zend_Service_Amazon_Offer` objects.

Back to Class List

Zend_Service_Amazon_Offer

Each offer for an item is returned as an `Zend_Service_Amazon_Offer` object.

Zend_Service_Amazon_Offer Properties

Table 40.4. Properties

Name	Type	Description
MerchantId	string	Merchants Amazon ID
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
IsEligibleForSuperSaverShipping	boolean	Whether the item is eligible for Super Saver Shipping or not

Back to Class List

Zend_Service_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 `Zend_Service_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 40.5. Zend_Service_Amazon_SimilarProduct Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

Back to Class List

Zend_Service_Amazon_Accessories

Accessories for the returned item are represented as `Zend_Service_Amazon_Accessories` objects

Properties

Table 40.6. Zend_Service_Amazon_Accessories Properties

Name	Type	Description
ASIN	string	Products Amazon Unique ID (ASIN)
Title	string	Products Title

Back to Class List

Zend_Service_Amazon_CustomerReview

Each Customer Review is returned as a `Zend_Service_Amazon_CustomerReview` object.

Properties

Table 40.7. Zend_Service_Amazon_CustomerReview 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

Zend_Service_Amazon_EditorialReview

Each items Editorial Reviews are returned as a `Zend_Service_Amazon_EditorialReview` object

Properties

Table 40.8. Zend_Service_Amazon_EditorialReview Properties

Name	Type	Description
Source	string	Source of the Editorial Review
Content	string	Review Content

Back to Class List

Zend_Service_Amazon_Listmania

Each results List Mania List items are returned as `Zend_Service_Amazon_Listmania` objects.

Properties

Table 40.9. Zend_Service_Amazon_Listmania Properties

Name	Type	Description
ListId	string	List ID
ListName	string	List Name

Back to Class List

Zend_Service_Audioscrobbler

Introduction to Searching Audioscrobbler

`Zend_Service_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 `Zend_Service_Audioscrobbler` class begin with one of these terms. The syntax and namespaces of the Audioscrobbler Web Service are mirrored in `Zend_Service_Audioscrobbler`. For more information about the Audioscrobbler REST Web Service, please visit the Audioscrobbler Web Service site [http://www.audioscrobbler.net/data/webservices/].

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. `Zend_Service_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.

Example 40.11. Retrieving User Profile Information

In this example, we use the `setUser()` and `userGetProfileInformation()` methods to retrieve a specific user's profile information:

```
$as = new Zend_Service_Audioscrobbler();
$as->setUser('BigDaddy71'); // Set the user whose profile information we want to r
$profileInfo = $as->userGetProfileInformation(); // Retrieve BigDaddy71's profile
print "Information for $profileInfo->realname can be found at $profileInfo->url";
```

Example 40.12. Retrieving a User's Weekly Artist Chart

```
$as = new Zend_Service_Audioscrobbler();
$as->setUser('lo_fye'); // Set the user whose profile weekly artist chart we want
$weeks = $as->userGetWeeklyChartList(); // Retrieves a list of previous weeks for
if (count($weeks) < 1) {
    echo 'No data available';
}
sort($weeks); // Order the list of weeks

$as->setFromDate($weeks[0]); // Set the starting date
$as->setToDate($weeks[0]); // Set the ending date

$previousWeeklyArtists = $as->userGetPreviousWeeklyArtistChart();

echo 'Artist Chart For Week Of ' . date('Y-m-d h:i:s', $as->from_date) . '<br />';

foreach ($previousWeeklyArtists as $artist) {
    // Display the artists' names with links to their profiles
    print '<a href="' . $artist->url . '">' . $artist->name . '</a><br />';
}
```

Artists

`Zend_Service_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.

Example 40.13. Retrieving Related Artists

```
$as = new Zend_Service_Audioscrobbler();
$as->setArtist('LCD Soundsystem'); // Set the artist for whom you would like to re
$relatedArtists = $as->artistGetRelatedArtists(); // Retrieve the related artists
foreach ($relatedArtists as $artist) {
    print '<a href="' . $artist->url . '">' . $artist->name . '</a><br />'; // Dis
}
```

Tracks

`Zend_Service_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

`Zend_Service_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

`Zend_Service_Audioscrobbler` provides several methods for retrieving data specific to a single group, specified via the `setGroup()` method:

- `groupGetRecentJournals()`: Returns a SimpleXML object containing a list of recent journal posts by Users in the current Group.
- `groupGetWeeklyChart()`: Returns a SimpleXML object containing a list of weeks for which there exist Weekly Charts for the current Group.
- `groupGetRecentWeeklyArtistChart()`: Returns a SimpleXML object containing the most recent Weekly Artist Chart for the current Group.

- `groupGetRecentWeeklyAlbumChart()`: Returns a SimpleXML object containing the most recent Weekly Album Chart for the current Group.
- `groupGetRecentWeeklyTrackChart()`: Returns a SimpleXML object containing the most recent Weekly Track Chart for the current Group.
- `groupGetPreviousWeeklyArtistChart($fromDate, $toDate)`: Requires `setFromDate()` and `setToDate()`. Returns a SimpleXML object containing the Weekly Artist Chart from the current fromDate to the current toDate for the current Group.
- `groupGetPreviousWeeklyAlbumChart($fromDate, $toDate)`: Requires `setFromDate()` and `setToDate()`. Returns a SimpleXML object containing the Weekly Album Chart from the current fromDate to the current toDate for the current Group.
- `groupGetPreviousWeeklyTrackChart($fromDate, $toDate)`: Returns a SimpleXML object containing the Weekly Track Chart from the current fromDate to the current toDate for the current Group.

Forums

`Zend_Service_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.

Zend_Service_Delicious

Introduction

`Zend_Service_Delicious` is simple API for using del.icio.us [http://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.

Example 40.14. Get all posts

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

foreach ($posts as $post) {
    echo "--\n";
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

Retrieving posts

Zend_Service_Delicious provides three methods for retrieving posts: getPosts(), getRecentPosts() and getAllPosts(). All of these methods return an instance of Zend_Service_Delicious_PostList, which holds all retrieved posts.

```
/**
 * Get posts matching the arguments. If no date or url is given,
 * most recent date will be used.
 *
 * @param string $tag Optional filtering by tag
 * @param Zend_Date $dt Optional filtering by date
 * @param string $url Optional filtering by url
 * @return Zend_Service_Delicious_PostList
 */
public function getPosts($tag = null, $dt = null, $url = null);

/**
 * Get recent posts
 *
 * @param string $tag   Optional filtering by tag
 * @param string $count Maximal number of posts to be returned
 *                      (default 15)
 * @return Zend_Service_Delicious_PostList
 */
public function getRecentPosts($tag = null, $count = 15);

/**
 * Get all posts
 *
 * @param string $tag Optional filtering by tag
 * @return Zend_Service_Delicious_PostList
 */
public function getAllPosts($tag = null);
```

Zend_Service_Delicious_PostList

Instances of this class are returned by the getPosts(), getAllPosts(), getRecentPosts(), and getUserPosts() methods of Zend_Service_Delicious.

For easier data access this class implements the Countable, Iterator, and ArrayAccess interfaces.

Example 40.15. Accessing post lists

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// count posts
echo count($posts);

// iterate over posts
foreach ($posts as $post) {
    echo "--\n";
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}

// get post using array access
echo $posts[0]->getTitle();
```

Note

The `ArrayAccess::offsetSet()` and `ArrayAccess::offsetUnset()` methods throw exceptions in this implementation. Thus, code like `unset($posts[0]);` and `$posts[0] = 'A';` will throw exceptions because these properties are read-only.

Post list objects have two built-in filtering capabilities. Post lists may be filtered by tags and by URL.

Example 40.16. Filtering a Post List with Specific Tags

Posts may be filtered by specific tags using `withTags()`. As a convenience, `withTag()` is also provided for when only a single tag needs to be specified.

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// Print posts having "php" and "zend" tags
foreach ($posts->withTags(array('php', 'zend')) as $post) {
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

Example 40.17. Filtering a Post List by URL

Posts may be filtered by URL matching a specified regular expression using the `withUrl()` method:

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getAllPosts();

// Print posts having "help" in the URL
foreach ($posts->withUrl('/help/') as $post) {
    echo "Title: {$post->getTitle()}\n";
    echo "Url: {$post->getUrl()}\n";
}
```

Editing posts

Example 40.18. Post editing

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getPosts();

// set title
$posts[0]->setTitle('New title');
// save changes
$posts[0]->save();
```

Example 40.19. Method call chaining

Every setter method returns the post object so that you can chain method calls using a fluent interface.

```
$delicious = new Zend_Service_Delicious('username', 'password');
$posts = $delicious->getPosts();

$posts[0]->setTitle('New title')
         ->setNotes('New notes')
         ->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.

Example 40.20. Deleting posts

```
$delicious = new Zend_Service_Delicious('username', 'password');

// by specifying URL
$delicious->deletePost('http://framework.zend.com');

// or by calling the method upon a post object
$posts = $delicious->getPosts();
$posts[0]->delete();

// another way of using deletePost()
$delicious->deletePost($posts[0]->getUrl());
```

Adding new posts

To add a post you first need to call the `createNewPost()` method, which returns a `Zend_Service_Delicious_Post` object. When you edit the post, you need to save it to the del.icio.us database by calling the `save()` method.

Example 40.21. Adding a post

```
$delicious = new Zend_Service_Delicious('username', 'password');

// create a new post and save it (with method call chaining)
$delicious->createNewPost('Zend Framework', 'http://framework.zend.com')
          ->setNotes('Zend Framework Homepage')
          ->save();

// create a new post and save it  (without method call chaining)
$newPost = $delicious->createNewPost('Zend Framework',
                                     'http://framework.zend.com');
$newPost->setNotes('Zend Framework Homepage');
$newPost->save();
```

Tags

Example 40.22. Tags

```
$delicious = new Zend_Service_Delicious('username', 'password');

// get all tags
print_r($delicious->getTags());

// rename tag ZF to zendFramework
$delicious->renameTag('ZF', 'zendFramework');
```

Bundles

Example 40.23. Bundles

```
$delicious = new Zend_Service_Delicious('username', 'password');

// get all bundles
print_r($delicious->getBundles());

// delete bundle someBundle
$delicious->deleteBundle('someBundle');

// add bundle
$delicious->addBundle('newBundle', array('tag1', 'tag2'));
```

Public data

The del.icio.us web API allows access to the public data of all users.

Table 40.10. 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	Zend_Service_Delicious_PostList
`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 `Zend_Service_Delicious` object.

Example 40.24. Retrieving public data

```
// username and password are not required
$delicious = new Zend_Service_Delicious();

// get fans of user someUser
print_r($delicious->getUserFans('someUser'));

// get network of user someUser
print_r($delicious->getUserNetwork('someUser'));

// get tags of user someUser
print_r($delicious->getUserTags('someUser'));
```

Public posts

When retrieving public posts with the `getUserPosts()` method, a `Zend_Service_Delicious_PostList` object is returned, and it contains `Zend_Service_Delicious_SimplePost` objects, which contain basic information about the posts, including URL, title, notes, and tags.

Table 40.11. Methods of the `Zend_Service_Delicious_SimplePost` class

Name	Description	Return type
`getNotes()`	Returns notes of a post	String
`getTags()`	Returns tags of a post	Array
`getTitle()`	Returns title of a post	String
`getUrl()`	Returns URL of a post	String

HTTP client

`Zend_Service_Delicious` uses `Zend_Rest_Client` for making HTTP requests to the del.icio.us web service. To change which HTTP client `Zend_Service_Delicious` uses, you need to change the HTTP client of `Zend_Rest_Client`.

Example 40.25. Changing the HTTP client of `Zend_Rest_Client`

```
$myHttpClient = new My_Http_Client();
Zend_Rest_Client::setHttpClient($myHttpClient);
```

When you are making more than one request with `Zend_Service_Delicious` to speed your requests, it's better to configure your HTTP client to keep connections alive.

Example 40.26. Configuring your HTTP client to keep connections alive

```
Zend_Rest_Client::getHttpClient()->setConfig(array(
        'keepalive' => true
));
```

Note

When a `Zend_Service_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).

Zend_Service_Flickr

Introduction to Searching Flickr

`Zend_Service_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 [http://www.flickr.com/services/api/].

In the following example, we use the `tagSearch()` method to search for photos having "php" in the tags.

Example 40.27. Simple Flickr Photo Search

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

$results = $flickr->tagSearch("php");

foreach ($results as $result) {
    echo $result->title . '<br />';
}
```

Optional parameter

`tagSearch()` accepts an optional second parameter as an array of options.

Finding Flickr Users' Photos and Information

`Zend_Service_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 `Zend_Service_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.

Example 40.28. Finding a Flickr User's Public Photos by E-Mail Address

In this example, we have a Flickr user's e-mail address, and we search for the user's public photos by using the `userSearch()` method:

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

$results = $flickr->userSearch($userEmail);

foreach ($results as $result) {
    echo $result->title . '<br />';
}
```

Finding photos From a Group Pool

`Zend_Service_Flickr` allows to retrieve a group's pool photos based on the group ID. Use the `groupPoolGetPhotos()` method:

Example 40.29. Retrieving a Group's Pool Photos by Group ID

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

    $results = $flickr->groupPoolGetPhotos($groupId);

    foreach ($results as $result) {
        echo $result->title . '<br />';
    }
```

Optional parameter

`groupPoolGetPhotos()` accepts an optional second parameter as an array of options.

Retrieving Flickr Image Details

`Zend_Service_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:

Example 40.30. Retrieving Flickr Image Details

Once you have a Flickr image ID, it is a simple matter to fetch information about the image:

```
$flickr = new Zend_Service_Flickr('MY_API_KEY');

$image = $flickr->getImageDetails($imageId);

echo "Image ID $imageId is $image->width x $image->height pixels.<br />\n";
echo "<a href=\"$image->clickUri\">Click for Image</a>\n";
```

Zend_Service_Flickr Result Classes

The following classes are all returned by `tagSearch()` and `userSearch()`:

- `Zend_Service_Flickr_ResultSet`
- `Zend_Service_Flickr_Result`
- `Zend_Service_Flickr_Image`

Zend_Service_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 40.12. Zend_Service_Flickr_ResultSet Properties

Name	Type	Description
totalResultsAvailable	int	Total Number of Results available
totalResultsReturned	int	Total Number of Results returned
firstResultPosition	int	The offset in the total result set of this result set

Zend_Service_Flickr_ResultSet::totalResults()

```
int totalResults();
```

Returns the total number of results in this result set.

Back to Class List

Zend_Service_Flickr_Result

A single Image result from a Flickr query

Properties

Table 40.13. Zend_Service_Flickr_Result 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	Zend_Service_Flickr_Image	A 75x75 thumbnail of the image.
Thumbnail	Zend_Service_Flickr_Image	A 100 pixel thumbnail of the image.
Small	Zend_Service_Flickr_Image	A 240 pixel version of the image.
Medium	Zend_Service_Flickr_Image	A 500 pixel version of the image.
Large	Zend_Service_Flickr_Image	A 640 pixel version of the image.
Original	Zend_Service_Flickr_Image	The original image.

Back to Class List

Zend_Service_Flickr_Image

Represents an Image returned by a Flickr search.

Properties

Table 40.14. Zend_Service_Flickr_Image 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

Zend_Service_Nirvanix

Introduction

Nirvanix provides an Internet Media File System (IMFS), an Internet storage service that allows applications to upload, store and organize files and subsequently access them using a standard Web Services interface. An IMFS is distributed clustered file system, accessed over the Internet, and optimized for dealing with media files (audio, video, etc). The goal of an IMFS is to provide massive scalability to deal with the challenges of media storage growth, with guaranteed access and availability regardless of time and location. Finally, an IMFS gives applications the ability to access data securely, without the large fixed costs associated with acquiring and maintaining physical storage assets.

Registering with Nirvanix

Before you can get started with `Zend_Service_Nirvanix`, you must first register for an account. Please see the Getting Started [http://www.nirvanix.com/gettingStarted.aspx] page on the Nirvanix website for more information.

After registering, you will receive a Username, Password, and Application Key. All three are required to use `Zend_Service_Nirvanix`.

API Documentation

Access to the Nirvanix IMFS is available through both SOAP and a faster REST service. `Zend_Service_Nirvanix` provides a relatively thin PHP 5 wrapper around the REST service.

`Zend_Service_Nirvanix` aims to make using the Nirvanix REST service easier but understanding the service itself is still essential to be successful with Nirvanix.

The Nirvanix API Documentation [http://developer.nirvanix.com/sitefiles/1000/API.html] provides an overview as well as detailed information using the service. Please familiarize yourself with this document and refer back to it as you use `Zend_Service_Nirvanix`.

Features

Nirvanix's REST service can be used effectively with PHP using the SimpleXML [http://www.php.net/simplexml] extension and `Zend_Http_Client` alone. However, using it this way is somewhat inconvenient due to repetitive operations like passing the session token on every request and repeatedly checking the response body for error codes.

`Zend_Service_Nirvanix` provides the following functionality:

- A single point for configuring your Nirvanix authentication credentials that can be used across the Nirvanix namespaces.
- A proxy object that is more convenient to use than an HTTP client alone, mostly removing the need to manually construct HTTP POST requests to access the REST service.
- A response wrapper that parses each response body and throws an exception if an error occurred, alleviating the need to repeatedly check the success of many commands.
- Additional convenience methods for some of the more common operations.

Getting Started

Once you have registered with Nirvanix, you're ready to store your first file on the IMFS. The most common operations that you will need to do on the IMFS are creating a new file, downloading an existing file, and deleting a file. `Zend_Service_Nirvanix` provides convenience methods for these three operations.

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new Zend_Service_Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$imfs->putContents('/foo.txt', 'contents to store');

echo $imfs->getContents('/foo.txt');

$imfs->unlink('/foo.txt');
```

The first step to using `Zend_Service_Nirvanix` is always to authenticate against the service. This is done by passing your credentials to the `Zend_Service_Nirvanix` constructor above. The associative array is passed directly to Nirvanix as POST parameters.

Nirvanix divides its web services into namespaces [http://developer.nirvanix.com/sitefiles/1000/API.html#_Toc175999879]. Each namespace encapsulates a group of related operations. After getting an instance of `Zend_Service_Nirvanix`, call the `getService()` method to create a proxy for the namespace you want to use. Above, a proxy for the `IMFS` namespace is created.

After you have a proxy for the namespace you want to use, call methods on it. The proxy will allow you to use any command available on the REST API. The proxy may also make convenience methods available, which wrap web service commands. The example above shows using the IMFS convenience methods to create a new file, retrieve and display that file, and finally delete the file.

Understanding the Proxy

In the previous example, we used the `getService()` method to return a proxy object to the `IMFS` namespace. The proxy object allows you to use the Nirvanix REST service in a way that's closer to making a normal PHP method call, as opposed to constructing your own HTTP request objects.

A proxy object may provide convenience methods. These are methods that the `Zend_Service_Nirvanix` provides to simplify the use of the Nirvanix web services. In the previous example, the methods `putContents()`, `getContents()`, and `unlink()` do not have direct equivalents in the REST API. They are convenience methods provided by `Zend_Service_Nirvanix` that abstract more complicated operations on the REST API.

For all other method calls to the proxy object, the proxy will dynamically convert the method call to the equivalent HTTP POST request to the REST API. It does this by using the method name as the API command, and an associative array in the first argument as the POST parameters.

Let's say you want to call the REST API method RenameFile [http://developer.nirvanix.com/sitefiles/1000/API.html#_Toc175999923], which does not have a convenience method in `Zend_Service_Nirvanix`:

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new Zend_Service_Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$result = $imfs->renameFile(array('filePath' => '/path/to/foo.txt',
                                  'newFileName' => 'bar.txt'));
```

Above, a proxy for the `IMFS` namespace is created. A method, `renameFile()`, is then called on the proxy. This method does not exist as a convenience method in the PHP code, so it is trapped by `__call()` and converted into a POST request to the REST API where the associative array is used as the POST parameters.

Notice in the Nirvanix API documentation that `sessionToken` is required for this method but we did not give it to the proxy object. It is added automatically for your convenience.

The result of this operation will either be a `Zend_Service_Nirvanix_Response` object wrapping the XML returned by Nirvanix, or a `Zend_Service_Nirvanix_Exception` if an error occurred.

Examining Results

The Nirvanix REST API always returns its results in XML. `Zend_Service_Nirvanix` parses this XML with the `SimpleXML` extension and then decorates the resulting `SimpleXMLElement` with a `Zend_Service_Nirvanix_Response` object.

The simplest way to examine a result from the service is to use the built-in PHP functions like `print_r()`:

```
<?php
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');

$nirvanix = new Zend_Service_Nirvanix($auth);
$imfs = $nirvanix->getService('IMFS');

$result = $imfs->putContents('/foo.txt', 'fourteen bytes');
print_r($result);
?>
```

```
Zend_Service_Nirvanix_Response Object
(
    [_sxml:protected] => SimpleXMLElement Object
        (
            [ResponseCode] => 0
            [FilesUploaded] => 1
            [BytesUploaded] => 14
        )
)
```

You can access any property or method of the decorated `SimpleXMLElement`. In the above example, `$result->BytesUploaded` could be used to see the number of bytes received. Should you want to access the `SimpleXMLElement` directly, just use `$result->getSxml()`.

The most common response from Nirvanix is success (`ResponseCode` of zero). It is not normally necessary to check `ResponseCode` because any non-zero result will throw a `Zend_Service_Nirvanix_Exception`. See the next section on handling errors.

Handling Errors

When using Nirvanix, it's important to anticipate errors that can be returned by the service and handle them appropriately.

All operations against the REST service result in an XML return payload that contains a `ResponseCode` element, such as the following example:

```
<Response>
   <ResponseCode>0</ResponseCode>
</Response>
```

When the `ResponseCode` is zero such as in the example above, the operation was successful. When the operation is not successful, the `ResponseCode` is non-zero and an `ErrorMessage` element should be present.

To alleviate the need to repeatedly check if the `ResponseCode` is non-zero, `Zend_Service_Nirvanix` automatically checks each response returned by Nirvanix. If the `ResponseCode` indicates an error, a `Zend_Service_Nirvanix_Exception` will be thrown.

```
$auth = array('username' => 'your-username',
              'password' => 'your-password',
              'appKey'   => 'your-app-key');
$nirvanix = new Zend_Service_Nirvanix($auth);

try {

  $imfs = $nirvanix->getService('IMFS');
  $imfs->unlink('/a-nonexistant-path');

} catch (Zend_Service_Nirvanix_Exception $e) {
  echo $e->getMessage() . "\n";
  echo $e->getCode();
}
```

In the example above, `unlink()` is a convenience method that wraps the `DeleteFiles` command on the REST API. The `filePath` parameter required by the DeleteFiles [http://developer.nirvanix.com/sitefiles/1000/API.html#_Toc175999918] command contains a path that does not exist. This will result in a `Zend_Service_Nirvanix` exception being thrown with the message "Invalid path" and code 70005.

The Nirvanix API Documentation [http://developer.nirvanix.com/sitefiles/1000/API.html] describes the errors associated with each command. Depending on your needs, you may wrap each command in a `try` block or wrap many commands in the same `try` block for convenience.

Zend_Service_ReCaptcha

Introduction

`Zend_Service_ReCaptcha` provides a client for the reCAPTCHA Web Service [http://recaptcha.net/]. 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 identifiy. 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 [http://recaptcha.net/whyrecaptcha.html] and register one or more domains with the service in order to generate public and private keys.

Simplest use

Instantiate a `Zend_Service_ReCaptcha` object, passing it your public and private keys:

```
$recaptcha = new Zend_Service_ReCaptcha($pubKey, $privKey);
```

To render the reCAPTCHA, simply call the `getHTML()` method:

```
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:

```
$result = $recaptcha->verify(
    $_POST['recaptcha_challenge_field'],
    $_POST['recaptcha_response_field']
);
```

Once you have the result, test against it to see if it is valid. The result is a `Zend_Service_ReCaptcha_Response` object, which provides an `isValid()` method.

```
if (!$result->isValid()) {
    // Failed validation
}
```

Even simpler is 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.

Zend_Service_Simpy

Introduction

`Zend_Service_Simpy` is a lightweight wrapper for the free REST API available for the Simpy social bookmarking service.

In order to use `Zend_Service_Simpy`, you should already have a Simpy account. To get an account, visit the Simpy web site [http://simpy.com]. For more information on the Simpy REST API, refer to the Simpy REST API documentation [http://www.simpy.com/doc/api/rest].

The Simpy REST API allows developers to interact with specific aspects of the service that the Simpy web site offers. The sections following will outline the use of `Zend_Service_Simpy` for each of these areas.

- Links: Create, Retrieve, Update, Delete
- Tags: Retrieve, Delete, Rename, Merge, Split
- Notes: Create, Retrieve, Update, Delete
- Watchlists: Get, Get All

Links

When querying links, results are returned in descending order by date added. Links can be searched by title, nickname, tags, note, or even the content of the web page associated with the link. Simpy offers searching by any or all of these fields with phrases, boolean operators, and wildcards. See the search syntax [http://www.simpy.com/faq#searchSyntax] and search fields [http://www.simpy.com/faq#searchFieldsLinks] sections of the Simpy FAQ for more information.

Example 40.31. Querying Links

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Search for the 10 links added most recently */
$linkQuery = new Zend_Service_Simpy_LinkQuery();
$linkQuery->setLimit(10);

/* Get and display the links */
$linkSet = $simpy->getLinks($linkQuery);
foreach ($linkSet as $link) {
    echo '<a href="';
    echo $link->getUrl();
    echo '">';
    echo $link->getTitle();
    echo '</a><br />';
}

/* Search for the 5 links added most recently with 'PHP' in
the title */
$linkQuery->setQueryString('title:PHP');
$linkQuery->setLimit(5);

/* Search for all links with 'French' in the title and
'language' in the tags */
$linkQuery->setQueryString('+title:French +tags:language');

/* Search for all links with 'French' in the title and without
'travel' in the tags */
$linkQuery->setQueryString('+title:French -tags:travel');

/* Search for all links added on 12/9/06 */
$linkQuery->setDate('2006-12-09');

/* Search for all links added after 12/9/06 (excluding that
date) */
$linkQuery->setAfterDate('2006-12-09');

/* Search for all links added before 12/9/06 (excluding that
date) */
$linkQuery->setBeforeDate('2006-12-09');

/* Search for all links added between 12/1/06 and 12/9/06
(excluding those two dates) */
$linkQuery->setBeforeDate('2006-12-01');
$linkQuery->setAfterDate('2006-12-09');
```

Links are represented uniquely by their URLs. In other words, if an attempt is made to save a link that has the same URL as an existing link, data for the existing link will be overwritten with the data specified in the save attempt.

Example 40.32. Modifying Links

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Save a link */
$simpy->saveLink(
    'Zend Framework' // Title
    'http://framework.zend.com', // URL
    Zend_Service_Simpy_Link::ACCESSTYPE_PUBLIC, // Access Type
    'zend, framework, php' // Tags
    'Zend Framework home page' // Alternative title
    'This site rocks!' // Note
);

/* Overwrite the existing link with new data */
$simpy->saveLink(
    'Zend Framework'
    'http://framework.zend.com',
    Zend_Service_Simpy_Link::ACCESSTYPE_PRIVATE, // Access Type has changed
    'php, zend, framework' // Tags have changed order
    'Zend Framework' // Alternative title has changed
    'This site REALLY rocks!' // Note has changed
);

/* Delete the link */
$simpy->deleteLink('http://framework.zend.com');

/* A really easy way to do spring cleaning on your links ;) */
$linkSet = $this->_simpy->getLinks();
foreach ($linkSet as $link) {
    $this->_simpy->deleteLink($link->getUrl());
}
```

Tags

When retrieved, tags are sorted in decreasing order (i.e. highest first) by the number of links that use the tag.

Example 40.33. Working With Tags

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Save a link with tags */
$simpy->saveLink(
    'Zend Framework' // Title
    'http://framework.zend.com', // URL
    Zend_Service_Simpy_Link::ACCESSTYPE_PUBLIC, // Access Type
    'zend, framework, php' // Tags
);

/* Get a list of all tags in use by links and notes */
$tagSet = $simpy->getTags();

/* Display each tag with the number of links using it */
foreach ($tagSet as $tag) {
    echo $tag->getTag();
    echo ' - ';
    echo $tag->getCount();
    echo '<br />';
}

/* Remove the 'zend' tag from all links using it */
$simpy->removeTag('zend');

/* Rename the 'framework' tag to 'frameworks' */
$simpy->renameTag('framework', 'frameworks');

/* Split the 'frameworks' tag into 'framework' and
'development', which will remove the 'frameworks' tag for
all links that use it and add the tags 'framework' and
'development' to all of those links */
$simpy->splitTag('frameworks', 'framework', 'development');

/* Merge the 'framework' and 'development' tags back into
'frameworks', basically doing the opposite of splitting them */
$simpy->mergeTags('framework', 'development', 'frameworks');
```

Notes

Notes can be saved, retrieved, and deleted. They are uniquely identified by a numeric ID value.

Example 40.34. Working With Notes

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Save a note */
$simpy->saveNote(
    'Test Note', // Title
    'test,note', // Tags
    'This is a test note.' // Description
);

/* Overwrite an existing note */
$simpy->saveNote(
    'Updated Test Note', // Title
    'test,note,updated', // Tags
    'This is an updated test note.', // Description
    $note->getId() // Unique identifier
);

/* Search for the 10 most recently added notes */
$noteSet = $simpy->getNotes(null, 10);

/* Display the notes */
foreach ($noteSet as $note) {
    echo '<p>';
    echo $note->getTitle();
    echo '<br />';
    echo $note->getDescription();
    echo '<br >';
    echo $note->getTags();
    echo '</p>';
}

/* Search for all notes with 'PHP' in the title */
$noteSet = $simpy->getNotes('title:PHP');

/* Search for all notes with 'PHP' in the title and
without 'framework' in the description */
$noteSet = $simpy->getNotes('+title:PHP -description:framework');

/* Delete a note */
$simpy->deleteNote($note->getId());
```

Watchlists

Watchlists cannot be created or removed using the API, only retrieved. Thus, you must set up a watchlist via the Simpy web site prior to attempting to access it using the API.

Example 40.35. Retrieving Watchlists

```
$simpy = new Zend_Service_Simpy('yourusername', 'yourpassword');

/* Get a list of all watchlists */
$watchlistSet = $simpy->getWatchlists();

/* Display data for each watchlist */
foreach ($watchlistSet as $watchlist) {
    echo $watchlist->getId();
    echo '<br />';
    echo $watchlist->getName();
    echo '<br />';
    echo $watchlist->getDescription();
    echo '<br />';
    echo $watchlist->getAddDate();
    echo '<br />';
    echo $watchlist->getNewLinks();
    echo '<br />';

    foreach ($watchlist->getUsers() as $user) {
        echo $user;
        echo '<br />';
    }

    foreach ($watchlist->getFilters() as $filter) {
        echo $filter->getName();
        echo '<br />';
        echo $filter->getQuery();
        echo '<br />';
    }
}

/* Get an individual watchlist by its identifier */
$watchlist = $simpy->getWatchlist($watchlist->getId());
$watchlist = $simpy->getWatchlist(1);
```

Introduction

The `Zend_Service_SlideShare` component is used to interact with the slideshare.net [http://www.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 `Zend_Service_SlideShare`

In order to use the Zend_Service_SlideShare component you must first create an account on the slideshare.net servers (more information can be found here [http://www.slideshare.net/developers/]) in order to receive an API key, username, password and shared secret value -- all of which are needed in order to use the `Zend_Service_SlideShare` component.

Once you have setup an account, you can begin using the `Zend_Service_SlideShare` component by creating a new instance of the `Zend_Service_SlideShare` object and providing these values as shown below:

```
// Create a new instance of the component
$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
                                  'PASSWORD');
```

The SlideShow object

All slide shows in the `Zend_Service_SlideShare` component are represented using the `Zend_Service_SlideShare_SlideShow` object (both when retrieving and uploading new slide shows). For your reference a pesudo-code version of this class is provided below.

```
class Zend_Service_SlideShare_SlideShow {

    /**
     * Retrieves the location of the slide show
     */
    public function getLocation() {
        return $this->_location;
    }

    /**
     * Gets the transcript for this slide show
     */
    public function getTranscript() {
        return $this->_transcript;
    }

    /**
     * Adds a tag to the slide show
     */
    public function addTag($tag) {
        $this->_tags[] = (string)$tag;
        return $this;
    }

    /**
     * Sets the tags for the slide show
     */
    public function setTags(Array $tags) {
        $this->_tags = $tags;
        return $this;
    }
    /**
     * Gets all of the tags associated with the slide show
     */
    public function getTags() {
        return $this->_tags;
    }

    /**
     * Sets the filename on the local filesystem of the slide show
     * (for uploading a new slide show)
     */
    public function setFilename($file) {
        $this->_slideShowFilename = (string)$file;
        return $this;
    }

    /**
     * Retrieves the filename on the local filesystem of the slide show
     * which will be uploaded
     */
    public function getFilename() {
        return $this->_slideShowFilename;
    }

    /**
     * Gets the ID for the slide show
     */
    public function getId() {
        return $this->_slideShowId;
    }

    /**
     * Retrieves the HTML embed code for the slide show
     */
    public function getEmbedCode() {
        return $this->_embedCode;
    }

    /**
     * Retrieves the Thumbnail URi for the slide show
     */
    public function getThumbnailUrl() {
        return $this->_thumbnailUrl;
    }

    /**
     * Sets the title for the Slide show
     */
    public function setTitle($title) {
        $this->_title = (string)$title;
        return $this;
    }
    /**
     * Retrieves the Slide show title
     */
    public function getTitle() {
        return $this->_title;
    }

    /**
     * Sets the description for the Slide show
     */
    public function setDescription($desc) {
        $this->_description = (string)$desc;
        return $this;
    }

    /**
     * Gets the description of the slide show
     */
    public function getDescription() {
        return $this->_description;
    }

    /**
     * Gets the numeric status of the slide show on the server
     */
    public function getStatus() {
        return $this->_status;
    }

    /**
     * Gets the textual description of the status of the slide show on
     * the server
     */
    public function getStatusDescription() {
        return $this->_statusDescription;
    }

    /**
     * Gets the permanent link of the slide show
     */
    public function getPermaLink() {
        return $this->_permalink;
    }

    /**
     * Gets the number of views the slide show has received
     */
    public function getNumViews() {
        return $this->_numViews;
    }
}
```

Note

The above pseudo-class only shows those methods which should be used by end-user developers. Other available methods are internal to the component.

When using the `Zend_Service_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 `Zend_Service_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 `Zend_Service_SlideShare` object and using the resulting `Zend_Service_SlideShare_SlideShow` object as shown.

```
// Create a new instance of the component
$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
                                  'PASSWORD');

$slideshow = $ss->getSlideShow(123456);

print "Slide Show Title: {$slideshow->getTitle()}<br/>\n";
print "Number of views: {$slideshow->getNumViews()}<br/>\n";
```

Retrieving Groups of Slide Shows

If you do not know the specific ID of a slide show you are interested in retrieving, you can retrieving groups of slide shows by using one of three methods:

- **Slide shows from a specific account**

  You can retrieve slide shows from a specific account by using the `getSlideShowsByUsername()` method and providing the username from which the slide shows should be retrieved

- **Slide shows which contain specific tags**

  You can retrieve slide shows which contain one or more specific tags by using the `getSlideShows-ByTag` 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 `getSlideShowsBy-Group` method and providng the name of the group which the slide show must belong to in order to be retrieved

Each of the above methods of retrieving multiple slide shows a similar approach is used. An example of using each method is shown below:

```
// Create a new instance of the component
$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
                                  'PASSWORD');

$starting_offset = 0;
$limit = 10;

// Retrieve the first 10 of each type
$ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
$ss_tags = $ss->getSlideShowsByTag('zend', $starting_offset, $limit);
$ss_group = $ss->getSlideShowsByGroup('mygroup', $starting_offset, $limit);

// Iterate over the slide shows
foreach($ss_user as $slideshow) {
   print "Slide Show Title: {$slideshow->getTitle}<br/>\n";
}
```

`Zend_Service_SlideShare` Caching policies

By default, `Zend_Service_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 Chapter 4, *Zend_Cache* object using the `setCacheObject` method as shown:

```
$frontendOptions = array(
                        'lifetime' => 7200,
                        'automatic_serialization' => true);
$backendOptions  = array(
                        'cache_dir' => '/webtmp/');

$cache = Zend_Cache::factory('Core',
                             'File',
                             $frontendOptions,
                             $backendOptions);

$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
                                  'PASSWORD');
$ss->setCacheObject($cache);

$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 Chapter 21, *Zend_Http*). This is useful for instance when it is desirable to set the timeout for the connection to something other then default as shown:

```
$client = new Zend_Http_Client();
$client->setConfig(array('timeout' => 5));

$ss = new Zend_Service_SlideShare('APIKEY',
                                  'SHAREDSECRET',
                                  'USERNAME',
                                  'PASSWORD');
$ss->setHttpClient($client);
$ss_user = $ss->getSlideShowsByUser('username', $starting_offset, $limit);
```

Zend_Service_StrikeIron

Zend_Service_StrikeIron provides a PHP 5 client to StrikeIron web services. See the following sections:

- the section called “Zend_Service_StrikeIron”
- the section called “Zend_Service_StrikeIron: Bundled Services”
- the section called “Zend_Service_StrikeIron: Advanced Uses”

Overview

StrikeIron [http://www.strikeiron.com] 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 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 [http://us.php.net/soap] alone. However, using StrikeIron this way does not give an ideal PHP-like interface. The Zend_Service_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 Zend_Service_StrikeIron.

The Zend_Service_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 Zend_Service_StrikeIron, you must first register [http://strikeiron.com/Register.aspx] for a StrikeIron developer account.

After registering, you will receive a StrikeIron username and password. These will be used when connecting to StrikeIron using Zend_Service_StrikeIron.

You will also need to sign up [http://www.strikeiron.com/ProductDetail.aspx?p=257] 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 [http://strikeiron.com/Register.aspx] for a StrikeIron account and signed up for the Super Data Pack [http://www.strikeiron.com/ProductDetail.aspx?p=257], you're ready to start using Zend_Service_StrikeIron.

StrikeIron consists of hundreds of different web services. Zend_Service_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 `Zend_Service_StrikeIron` provides a simple way of specifying your StrikeIron account information and other options in its constructor. It also has a factory method that will return clients for StrikeIron services:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));
```

The `getService()` method will return a client for any StrikeIron service by the name of its PHP wrapper class. In this case, the name `SalesUseTaxBasic` refers to the wrapper class `Zend_Service_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.

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Sales & Use Tax Basic service
$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

// Query tax rate for Ontario, Canada
$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
echo $rateInfo->province;
echo $rateInfo->abbreviation;
echo $rateInfo->GST;
```

In the example above, the `getService()` method is used to return a client to the Sales & Use Tax Basic service. The client object is stored in `$taxBasic`.

The `getTaxRateCanada()` method is then called on the service. An associative array is used to supply keyword parameters to the method. This is the way that all StrikeIron methods are called.

The result from `getTaxRateCanada()` is stored in `$rateInfo` and has properties like `province` and `GST`.

Many of the StrikeIron services are as simple to use as the example above. See Bundled Services for detailed information on three StrikeIron services.

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 `Zend_Service_StrikeIron_Decorator`. This is a small decorator [http://en.wikipedia.org/wiki/Decorator_pattern] 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() [http://www.php.net/print_r]:

```
<?php
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'ontario'));
print_r($rateInfo);
```

```
?>


Zend_Service_StrikeIron_Decorator Object
(
    [_name:protected] => GetTaxRateCanadaResult
    [_object:protected] => stdClass Object
        (
            [abbreviation] => ON
            [province] => ONTARIO
            [GST] => 0.06
            [PST] => 0.08
            [total] => 0.14
            [HST] => Y
        )
)
```

In the output above, we see that the decorator (`$rateInfo`) wraps an object named `GetTaxRateCanadaResult`, the result of the call to `getTaxRateCanada()`.

This means that `$rateInfo` has public properties like `abbreviation`, `province`, and `GST`. These are accessed like `$rateInfo->province`.

Tip

StrikeIron result properties sometimes start with an uppercase letter such as `Foo` or `Bar` where most PHP object properties normally start with a lowercase letter as in `foo` or `bar`. The decorator will automatically do this inflection so you may read a property `Foo` as `foo`.

If you ever need to get the original object or its name out of the decorator, use the respective methods `getDecoratedObject()` and `getDecoratedObjectName()`.

Handling Errors

The previous examples are naive, i.e. no error handling was shown. It's possible that StrikeIron will return a fault during a method call. Events like bad account credentials or an expired subscription can cause StrikeIron to raise a fault.

An exception will be thrown when such a fault occurs. You should anticipate and catch these exceptions when making method calls to the service:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

try {

  $taxBasic->getTaxRateCanada(array('province' => 'ontario'));

} catch (Zend_Service_StrikeIron_Exception $e) {
  // error handling for events like connection
  // problems or subscription errors

}
```

The exceptions thrown will always be `Zend_Service_StrikeIron_Exception`.

It's important to understand the difference between exceptions and normal failed method calls. Exceptions occur for *exceptional* conditions, such as the network going down or your subscription expiring. Failed method calls that are a common occurrence, such as `getTaxRateCanada()` not finding the `province` you supplied, will not result an in exception.

Note

Every time you make a method call to a StrikeIron service, you should check the response object for validity and also be prepared to catch an exception.

Checking Your Subscription

StrikeIron provides many different services. Some of these are free, some are available on a trial basis, and some are pay subscription only. When using StrikeIron, it's important to be aware of your subscription status for the services you are using and check it regularly.

Each StrikeIron client returned by the `getService` method has the ability to check the subscription status for that service using the `getSubscriptionInfo()` method of the client:

```
// Get a client for the Sales & Use Tax Basic service
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

$taxBasic = $strikeIron->getService(array('class => 'SalesUseTaxBasic'));

// Check remaining hits for the Sales & Use Tax Basic service
$subscription = $taxBasic->getSubscriptionInfo();
echo $subscription->remainingHits;
```

The `getSubscriptionInfo()` method will return an object that typically has a `remainingHits` property. It's important to check the status on each service that you are using. If a method call is made to StrikeIron after the remaining hits have been used up, an exception will occur.

Checking your subscription to a service does not use any remaining hits to the service. Each time any method call to the service is made, the number of hits remaining will be cached and this cached value will be returned by `getSubscriptionInfo()` without connecting to the service again. To force `getSubscriptionInfo()` to override its cache and query the subscription information again, use `getSubscriptionInfo(true)`.

Zend_Service_StrikeIron: Bundled Services

Zend_Service_StrikeIron comes with wrapper classes for three popular StrikeIron services.

ZIP Code Information

`Zend_Service_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 [http://www.strikeiron.com/ProductDetail.aspx?p=267]
- Zip Code Information Service WSDL [http://sdpws.strikeiron.com/zf1.StrikeIron/sdpZIPCodeInfo?WSDL]

The service contains a `getZipCode()` method that will retrieve information about a United States ZIP code or Canadian postal code:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Zip Code Information service
$zipInfo = $strikeIron->getService(array('class' => 'ZipCodeInfo'));

// Get the Zip information for 95014
$response = $zipInfo->getZipCode(array('ZipCode' => 95014));
$zips = $response->serviceResult;

// Display the results
if ($zips->count == 0) {
    echo 'No results found';
} else {
    // a result with one single zip code is returned as an object,
    // not an array with one element as one might expect.
    if (! is_array($zips->zipCodes)) {
        $zips->zipCodes = array($zips->zipCodes);
    }

    // print all of the possible results
    foreach ($zips->zipCodes as $z) {
        $info = $z->zipCodeInfo;

        // show all properties
        print_r($info);

        // or just the city name
        echo $info->preferredCityName;
    }
}

// Detailed status information
// http://www.strikeiron.com/exampledata/StrikeIronZipCodeInformation_v3.pdf
$status = $response->serviceStatus;
```

U.S. Address Verification

`Zend_Service_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 [http://www.strikeiron.com/ProductDetail.aspx?p=198]
- U.S. Address Verification Service WSDL [http://ws.strikeiron.com/zf1.StrikeIron/USAddressVerification4_0?WSDL]

The service contains a `verifyAddressUSA()` method that will verify an address in the United States:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Zip Code Information service
$verifier = $strikeIron->getService(array('class' => 'USAddressVerification'));

// Address to verify.  Not all fields are required but
// supply as many as possible for the best results.
$address = array('firm'           => 'Zend Technologies',
                 'addressLine1'   => '19200 Stevens Creek Blvd',
                 'addressLine2'   => '',
                 'city_state_zip' => 'Cupertino CA 95014');

// Verify the address
$result = $verifier->verifyAddressUSA($address);

// Display the results
if ($result->addressErrorNumber != 0) {
    echo $result->addressErrorNumber;
    echo $result->addressErrorMessage;
} else {
    // show all properties
    print_r($result);

    // or just the firm name
    echo $result->firm;

    // valid address?
    $valid = ($result->valid == 'VALID');
}
```

Sales & Use Tax Basic

`Zend_Service_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 [http://www.strikeiron.com/ProductDetail.aspx?p=351]
- Sales & Use Tax Basic Service WSDL [http://ws.strikeiron.com/zf1.StrikeIron/taxdatabasic4?WSDL]

The service contains two methods, `getTaxRateUSA()` and `getTaxRateCanada()`, that will retrieve sales and use tax data for the United States and Canada, respectively.

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a client for the Sales & Use Tax Basic service
$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

// Query tax rate for Ontario, Canada
$rateInfo = $taxBasic->getTaxRateCanada(array('province' => 'foo'));
print_r($rateInfo);               // show all properties
echo $rateInfo->GST;              // or just the GST (Goods & Services Tax)

// Query tax rate for Cupertino, CA USA
$rateInfo = $taxBasic->getTaxRateUS(array('zip_code' => 95014));
print_r($rateInfo);               // show all properties
echo $rateInfo->state_sales_tax;  // or just the state sales tax
```

Zend_Service_StrikeIron: Advanced Uses

This section describes the more advanced uses of Zend_Service_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:

```
$strikeIron = new Zend_Service_StrikeIron(array('username' => 'your-username',
                                                'password' => 'your-password'));

// Get a generic client to the Reverse Phone Lookup service
$phone = $strikeIron->getService(
                          array('wsdl' =>
                                'http://ws.strikeiron.com/ReversePhoneLookup?WSDL'
                      );

$result = $phone->lookup(array('Number' => '(408) 253-8800'));
echo $result->listingName;

// 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 [http://janschneider.de] from the Horde project [http://horde.org] has written a small PHP routine [http://janschneider.de/news/25/268] 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 `Zend_Service_StrikeIron_Base`) contains a `getSoapClient()` method to return the underlying instance of `SOAPClient` used to communicate with StrikeIron.

PHP's SOAPClient [http://www.php.net/manual/en/function.soap-soapclient-construct.php] has a `trace` option that causes it to remember the XML exchanged during the last transaction. Zend_Service_StrikeIron does not enable the `trace` option by default but this can easily by 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()` [http://www.php.net/manual/en/function.soap-soapclient-getlastrequest.php] and `__getLastRequest()` [http://www.php.net/manual/en/function.soap-soapclient-getlastresponse.php]:

```
$strikeIron =
    new Zend_Service_StrikeIron(array('username' => 'your-username',
                                      'password' => 'your-password',
                                      'options'  => array('trace' => true)));

// Get a client for the Sales & Use Tax Basic service
$taxBasic = $strikeIron->getService(array('class' => 'SalesUseTaxBasic'));

// Perform a method call
$taxBasic->getTaxRateCanada(array('province' => 'ontario'));

// Get SOAPClient instance and view XML
$soapClient = $taxBasic->getSoapClient();
echo $soapClient->__getLastRequest();
echo $soapClient->__getLastResponse();
```

Zend_Service_Technorati

Introduction

`Zend_Service_Technorati` provides an easy, intuitive and object-oriented interface for using the Technorati API. It provides access to all available Technorati API queries [http://technorati.com/developers/api/] and returns the original XML response as a friendly PHP object.

Technorati [http://technorati.com/] 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 [http://technorati.com/developers/api/] 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 [http://technorati.com/signup/], then visit the API Key section [http://technorati.com/developers/apikey.html].

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 `Zend_Service_Technorati`.

Making Your First Query

In order to run a query, first you need a `Zend_Service_Technorati` instance with a valid API key. Then choose one of the available query methods, and call it providing required arguments.

Example 40.36. Sending your first query

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
$resultSet = $technorati->search('PHP');
```

Each query method accepts an array of optional parameters that can be used to refine your query.

Example 40.37. Refining your query

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// filter your query including only results
// with some authority (Results from blogs with a handful of links)
$options = array('authority' => 'a4');

// search Technorati for PHP keyword
$resultSet = $technorati->search('PHP', $options);
```

A `Zend_Service_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 `Zend_Service_Technorati` object as long as you need it.

Example 40.38. Sending multiple queries with the same `Zend_Service_Technorati` instance

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
$search = $technorati->search('PHP');

// get top tags indexed by Technorati
$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 `Zend_Service_Technorati_*ResultSet` objects. A result set object is basically a collection of result objects. It extends the basic `Zend_Service_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.

Example 40.39. Consuming a result set object

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
// $resultSet is an instance of Zend_Service_Technorati_SearchResultSet
$resultSet = $technorati->search('PHP');

// loop over all result objects
foreach ($resultSet as $result) {
    // $result is an instance of Zend_Service_Technorati_SearchResult
}
```

Because `Zend_Service_Technorati_ResultSet` implements the `SeekableIterator` interface, you can seek a specific result object using its position in the result collection.

Example 40.40. Seeking a specific result set object

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// search Technorati for PHP keyword
// $resultSet is an instance of Zend_Service_Technorati_SearchResultSet
$resultSet = $technorati->search('PHP');

// $result is an instance of Zend_Service_Technorati_SearchResult
$resultSet->seek(1);
$result = $resultSet->current();
```

Note

`SeekableIterator` works as an array and counts positions starting from index 0. Fetching position number 1 means getting the second result in the collection.

The second group is represented by special standalone result objects. `Zend_Service_Technorati_GetInfoResult`, `Zend_Service_Technorati_BlogInfoResult` and `Zend_Service_Technorati_KeyInfoResult` act as wrappers for additional objects, such as `Zend_Service_Technorati_Author` and `Zend_Service_Technorati_Weblog`.

Example 40.41. Consuming a standalone result object

```
// create a new Zend_Service_Technorati
// with a valid API_KEY
$technorati = new Zend_Service_Technorati('VALID_API_KEY');

// get info about weppos author
$result = $technorati->getInfo('weppos');

$author = $result->getAuthor();
echo '<h2>Blogs authored by ' . $author->getFirstName() . " " .
          $author->getLastName() . '</h2>';
echo '<ol>';
foreach ($result->getWeblogs() as $weblog) {
    echo '<li>' . $weblog->getName() . '</li>';
}
echo "</ol>";
```

Please read the Zend_Service_Technorati Classes section for further details about response classes.

Handling Errors

Each `Zend_Service_Technorati` query method throws a `Zend_Service_Technorati_Exception` exception on failure with a meaningful error message.

There are several reasons that may cause a `Zend_Service_Technorati` query to fail. `Zend_Service_Technorati` validates all parameters for any query request. If a parameter is invalid or it contains an invalid value, a new `Zend_Service_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.

Example 40.42. Handling a Query Exception

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
try {
    $resultSet = $technorati->search('PHP');
} catch(Zend_Service_Technorati_Exception $e) {
    echo "An error occurred: " $e->getMessage();
}
```

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 `Zend_Service_Technorati` if you try to use it beyond this limit. You can get information about your API key usage using the `Zend_Service_Technorati::keyInfo()` method.

`Zend_Service_Technorati::keyInfo()` returns a `Zend_Service_Technorati_KeyInfoResult` object. For full details please see the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.43. Getting API key daily usage information

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$key = $technorati->keyInfo();

echo "API Key: " . $key->getApiKey() . "<br />";
echo "Daily Usage: " . $key->getApiQueries() . "/" .
     $key->getMaxQueries() . "<br />";
```

Available Technorati Queries

`Zend_Service_Technorati` provides support for the following queries:

- `Cosmos`
- `Search`
- `Tag`
- `DailyCounts`
- `TopTags`
- `BlogInfo`
- `BlogPostTags`
- `GetInfo`

Technorati Cosmos

Cosmos [http://technorati.com/developers/api/cosmos.html] query lets you see what blogs are linking to a given URL. It returns a `Zend_Service_Technorati_CosmosResultSet` object. For full details please see `Zend_Service_Technorati::cosmos()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.44. Cosmos Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->cosmos('http://devzone.zend.com/');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

Technorati Search

The Search [http://technorati.com/developers/api/search.html] query lets you see what blogs contain a given search string. It returns a `Zend_Service_Technorati_SearchResultSet` object. For full details please see `Zend_Service_Technorati::search()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.45. Search Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->search('zend framework');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

Technorati Tag

The Tag [http://technorati.com/developers/api/tag.html] query lets you see what posts are associated with a given tag. It returns a `Zend_Service_Technorati_TagResultSet` object. For full details please see `Zend_Service_Technorati::tag()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.46. Tag Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->tag('php');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getWeblog()->getName() . "</li>";
}
echo "</ol>";
```

Technorati DailyCounts

The DailyCounts [http://technorati.com/developers/api/dailycounts.html] query provides daily counts of posts containing the queried keyword. It returns a `Zend_Service_Technorati_DailyCountsResultSet` object. For full details please see `Zend_Service_Technorati::dailyCounts()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.47. DailyCounts Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->dailyCounts('php');

foreach ($resultSet as $result) {
    echo "<li>" . $result->getDate() .
         "(" . $result->getCount() . ")</li>";
}
echo "</ol>";
```

Technorati TopTags

The TopTags [http://technorati.com/developers/api/toptags.html] query provides information on top tags indexed by Technorati. It returns a `Zend_Service_Technorati_TagsResultSet` object. For full details please see `Zend_Service_Technorati::topTags()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.48. TopTags Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->topTags();

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getTag() . "</li>";
}
echo "</ol>";
```

Technorati BlogInfo

The BlogInfo [http://technorati.com/developers/api/bloginfo.html] query provides information on what blog, if any, is associated with a given URL. It returns a `Zend_Service_Technorati_BlogInfoResult` object. For full details please see `Zend_Service_Technorati::blogInfo()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.49. BlogInfo Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$result = $technorati->blogInfo('http://devzone.zend.com/');

echo '<h2><a href="' . (string) $result->getWeblog()->getUrl() . '">' .
     $result->getWeblog()->getName() . '</a></h2>';
```

Technorati BlogPostTags

The BlogPostTags [http://technorati.com/developers/api/blogposttags.html] query provides information on the top tags used by a specific blog. It returns a `Zend_Service_Technorati_TagsResultSet` object. For full details please see `Zend_Service_Technorati::blogPostTags()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.50. BlogPostTags Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->blogPostTags('http://devzone.zend.com/');

echo "<p>Reading " . $resultSet->totalResults() .
     " of " . $resultSet->totalResultsAvailable() .
     " available results</p>";
echo "<ol>";
foreach ($resultSet as $result) {
    echo "<li>" . $result->getTag() . "</li>";
}
echo "</ol>";
```

Technorati GetInfo

The GetInfo [http://technorati.com/developers/api/getinfo.html] query tells you things that Technorati knows about a member. It returns a `Zend_Service_Technorati_GetInfoResult` object. For full details please see `Zend_Service_Technorati::getInfo()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Example 40.51. GetInfo Query

```
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$result = $technorati->getInfo('weppos');

$author = $result->getAuthor();
echo "<h2>Blogs authored by " . $author->getFirstName() . " " .
     $author->getLastName() . "</h2>";
echo "<ol>";
foreach ($result->getWeblogs() as $weblog) {
    echo "<li>" . $weblog->getName() . "</li>";
}
echo "</ol>";
```

Technorati KeyInfo

The KeyInfo query provides information on daily usage of an API key. It returns a `Zend_Service_Technorati_KeyInfoResult` object. For full details please see `Zend_Service_Technorati::keyInfo()` in the API reference guide [http://framework.zend.com/apidoc/core/].

Zend_Service_Technorati Classes

The following classes are returned by the various Technorati queries. Each `Zend_Service_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 `Zend_Service_Technorati_Res-`

ultSet class and implement the SeekableIterator interface, allowing for easy iteration and seeking to a specific result.

- Zend_Service_Technorati_ResultSet
- Zend_Service_Technorati_CosmosResultSet
- Zend_Service_Technorati_SearchResultSet
- Zend_Service_Technorati_TagResultSet
- Zend_Service_Technorati_DailyCountsResultSet
- Zend_Service_Technorati_TagsResultSet
- Zend_Service_Technorati_Result
- Zend_Service_Technorati_CosmosResult
- Zend_Service_Technorati_SearchResult
- Zend_Service_Technorati_TagResult
- Zend_Service_Technorati_DailyCountsResult
- Zend_Service_Technorati_TagsResult
- Zend_Service_Technorati_GetInfoResult
- Zend_Service_Technorati_BlogInfoResult
- Zend_Service_Technorati_KeyInfoResult

Note

Zend_Service_Technorati_GetInfoResult, Zend_Service_Technorati_BlogInfoResult and Zend_Service_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 Zend_Service_Technorati objects, such as Zend_Service_Technorati_Author and Zend_Service_Technorati_Weblog.

The Zend_Service_Technorati library includes additional convenient classes representing specific response objects. Zend_Service_Technorati_Author represents a single Technorati account, also known as a blog author or blogger. Zend_Service_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 Zend_Service_Technorati in the API reference guide [http://framework.zend.com/apidoc/core/].

Zend_Service_Technorati_ResultSet

Zend_Service_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 Zend_Service_Technorati_Result objects.

`Zend_Service_Technorati_ResultSet` implements the PHP `SeekableIterator` interface, and you can iterate all result objects via the PHP `foreach` statement.

Example 40.52. Iterating result objects from a resultset collection

```
// run a simple query
$technorati = new Zend_Service_Technorati('VALID_API_KEY');
$resultSet = $technorati->search('php');

// $resultSet is now an instance of
// Zend_Service_Technorati_SearchResultSet
// it extends Zend_Service_Technorati_ResultSet
foreach ($resultSet as $result) {
    // do something with your
    // Zend_Service_Technorati_SearchResult object
}
```

Zend_Service_Technorati_CosmosResultSet

`Zend_Service_Technorati_CosmosResultSet` represents a Technorati Cosmos query result set.

Note

`Zend_Service_Technorati_CosmosResultSet` extends Zend_Service_Technorati_ResultSet.

Zend_Service_Technorati_SearchResultSet

`Zend_Service_Technorati_SearchResultSet` represents a Technorati Search query result set.

Note

`Zend_Service_Technorati_SearchResultSet` extends Zend_Service_Technorati_ResultSet.

Zend_Service_Technorati_TagResultSet

`Zend_Service_Technorati_TagResultSet` represents a Technorati Tag query result set.

Note

`Zend_Service_Technorati_TagResultSet` extends Zend_Service_Technorati_ResultSet.

Zend_Service_Technorati_DailyCountsResultSet

`Zend_Service_Technorati_DailyCountsResultSet` represents a Technorati DailyCounts query result set.

Note

`Zend_Service_Technorati_DailyCountsResultSet` extends Zend_Service_Technorati_ResultSet.

Zend_Service_Technorati_TagsResultSet

`Zend_Service_Technorati_TagsResultSet` represents a Technorati TopTags or BlogPostTags queries result set.

Note

`Zend_Service_Technorati_TagsResultSet` extends Zend_Service_Technorati_ResultSet.

Zend_Service_Technorati_Result

`Zend_Service_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.

Zend_Service_Technorati_CosmosResult

`Zend_Service_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 Zend_Service_Technorati_CosmosResultSet object.

Note

`Zend_Service_Technorati_CosmosResult` extends Zend_Service_Technorati_Result.

Zend_Service_Technorati_SearchResult

`Zend_Service_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 Zend_Service_Technorati_SearchResultSet object.

Note

`Zend_Service_Technorati_SearchResult` extends Zend_Service_Technorati_Result.

Zend_Service_Technorati_TagResult

`Zend_Service_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 Zend_Service_Technorati_TagResultSet object.

Note

`Zend_Service_Technorati_TagResult` extends Zend_Service_Technorati_Result.

Zend_Service_Technorati_DailyCountsResult

`Zend_Service_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 Zend_Service_Technorati_DailyCountsResultSet object.

Note

`Zend_Service_Technorati_DailyCountsResult` extends Zend_Service_Technorati_Result.

Zend_Service_Technorati_TagsResult

`Zend_Service_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 Zend_Service_Technorati_TagsResultSet object.

Note

`Zend_Service_Technorati_TagsResult` extends Zend_Service_Technorati_Result.

Zend_Service_Technorati_GetInfoResult

`Zend_Service_Technorati_GetInfoResult` represents a single Technorati GetInfo query result object.

Zend_Service_Technorati_BlogInfoResult

`Zend_Service_Technorati_BlogInfoResult` represents a single Technorati BlogInfo query result object.

Zend_Service_Technorati_KeyInfoResult

`Zend_Service_Technorati_KeyInfoResult` represents a single Technorati KeyInfo query result object. It provides information about your Technorati API Key daily usage.

Zend_Service_Yahoo

Introduction

`Zend_Service_Yahoo` is a simple API for using many of the Yahoo! REST APIs. `Zend_Service_Yahoo` allows you to search Yahoo! Web search, Yahoo! News, Yahoo! Local, Yahoo! Images. In order to use the Yahoo! REST API, you must have a Yahoo! Application ID. To obtain an Application ID, please complete and submit the Application ID Request Form [http://developer.yahoo.com/wsregapp/].

Searching the Web with Yahoo!

`Zend_Service_Yahoo` enables you to search the Web with Yahoo! using the `webSearch()` method, which accepts a string query parameter and an optional second parameter as an array of search options. For full details and an option list, please visit the Yahoo! Web Search Documentation [http://developer.yahoo.com/search/web/V1/webSearch.html]. The `webSearch()` method returns a `Zend_Service_Yahoo_WebResultSet` object.

Example 40.53. Searching the Web with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->webSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

Finding Images with Yahoo!

You can search for Images with Yahoo using `Zend_Service_Yahoo`'s `imageSearch()` method. This method accepts a string query parameter and an optional array of search options, as for the `webSearch()` method. For full details and an option list, please visit the Yahoo! Image Search Documentation [http://developer.yahoo.com/search/image/V1/imageSearch.html].

Example 40.54. Finding Images with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->imageSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

Finding videos with Yahoo!

You can search for videos with Yahoo using `Zend_Service_Yahoo`'s `videoSearch()` method. For full details and an option list, please visit the Yahoo! Video Search Documentation [http://developer.yahoo.com/search/video/V1/videoSearch.html].

Example 40.55. Finding videos with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->videoSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

Finding Local Businesses and Services with Yahoo!

You can search for local businesses and services with Yahoo! by using the `localSearch()` method. For full details, please see the Yahoo! Local Search Documentation [http://developer.yahoo.com/search/local/V1/localSearch.html].

Example 40.56. Finding Local Businesses and Services with Yahoo!

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->localSearch('Apple Computers', array('zip' => '95014'));
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

Searching Yahoo! News

Searching Yahoo! News is simple; just use the `newsSearch()` method, as in the following example. For full details, please see the Yahoo! News Search Documentation [http://developer.yahoo.com/search/news/V1/newsSearch.html].

Example 40.57. Searching Yahoo! News

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->newsSearch('PHP');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

Searching Yahoo! Site Explorer Inbound Links

Searching Yahoo! Site Explorer Inbound Links is simple; just use the `inlinkDataSearch()` method, as in the following example. For full details, please see the Yahoo! Site Explorer Inbound Links Documentation [http://developer.yahoo.com/search/siteexplorer/V1/inlinkData.html].

Example 40.58. Searching Yahoo! Site Explorer Inbound Links

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->inlinkDataSearch('http://framework.zend.com/');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

Searching Yahoo! Site Explorer's PageData

Searching Yahoo! Site Explorer's PageData is simple; just use the `pageDataSearch()` method, as in the following example. For full details, please see the Yahoo! Site Explorer PageData Documentation [http://developer.yahoo.com/search/siteexplorer/V1/pageData.html].

Example 40.59. Searching Yahoo! Site Explorer's PageData

```
$yahoo = new Zend_Service_Yahoo("YAHOO_APPLICATION_ID");
$results = $yahoo->pageDataSearch('http://framework.zend.com/');
foreach ($results as $result) {
    echo $result->Title .'<br />';
}
```

Zend_Service_Yahoo Classes

The following classes are all returned by the various Yahoo! searches. Each search type returns a type-specific result set which can be easily iterated, with each result being contained in a type result object. All result set classes implement the `SeekableIterator` interface, allowing for easy iteration and seeking to a specific result.

- `Zend_Service_Yahoo_ResultSet`
- `Zend_Service_Yahoo_WebResultSet`
- `Zend_Service_Yahoo_ImageResultSet`
- `Zend_Service_Yahoo_VideoResultSet`
- `Zend_Service_Yahoo_LocalResultSet`
- `Zend_Service_Yahoo_NewsResultSet`
- `Zend_Service_Yahoo_InlinkDataResultSet`
- `Zend_Service_Yahoo_PageDataResultSet`
- `Zend_Service_Yahoo_Result`
- `Zend_Service_Yahoo_WebResult`
- `Zend_Service_Yahoo_ImageResult`
- `Zend_Service_Yahoo_VideoResult`
- `Zend_Service_Yahoo_LocalResult`
- `Zend_Service_Yahoo_NewsResult`
- `Zend_Service_Yahoo_InlinkDataResult`
- `Zend_Service_Yahoo_PageDataResult`
- `Zend_Service_Yahoo_Image`

Zend_Service_Yahoo_ResultSet

Each of the search specific result sets is extended from this base class.

Each of the specific result sets returns a search specific Zend_Service_Yahoo_Result objects.

Zend_Service_Yahoo_ResultSet::totalResults()

```
int totalResults();
```

Returns the number of results returned for the search.

Properties

Table 40.15. Zend_Service_Yahoo_ResultSet

Name	Type	Description
totalResultsAvailable	int	Total number of results found.
totalResultsReturned	int	Number of results in the current result set
firstResultPosition	int	Position of the first result in this set relative to the total number of results.

Back to Class List

Zend_Service_Yahoo_WebResultSet

`Zend_Service_Yahoo_WebResultSet` represents a Yahoo! Web Search result set.

Note

`Zend_Service_Yahoo_WebResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

Zend_Service_Yahoo_ImageResultSet

`Zend_Service_Yahoo_ImageResultSet` represents a Yahoo! Image Search result set.

Note

`Zend_Service_Yahoo_ImageResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

Zend_Service_Yahoo_VideoResultSet

`Zend_Service_Yahoo_VideoResultSet` represents a Yahoo! Video Search result set.

Note

`Zend_Service_Yahoo_VideoResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

Zend_Service_Yahoo_LocalResultSet

`Zend_Service_Yahoo_LocalResultSet` represents a Yahoo! Local Search result set.

Table 40.16. Zend_Service_Yahoo_LocalResultSet Properties

Name	Type	Description
resultSetMapURL	string	The URL of a webpage containing a map graphic with all returned results plotted on it.

Note

`Zend_Service_Yahoo_LocalResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

Zend_Service_Yahoo_NewsResultSet

`Zend_Service_Yahoo_NewsResultSet` represents a Yahoo! News Search result set.

Note

`Zend_Service_Yahoo_NewsResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

Zend_Service_Yahoo_InlinkDataResultSet

`Zend_Service_Yahoo_InlinkDataResultSet` represents a Yahoo! Inbound Link Search result set.

Note

`Zend_Service_Yahoo_InlinkDataResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

Zend_Service_Yahoo_PageDataResultSet

`Zend_Service_Yahoo_PageDataResultSet` represents a Yahoo! PageData Search result set.

Note

`Zend_Service_Yahoo_PageDataResultSet` extends Zend_Service_Yahoo_ResultSet

Back to Class List

Zend_Service_Yahoo_Result

Each of the search specific results is extended from this base class.

Properties

Table 40.17. Zend_Service_Yahoo_Result Properties

Name	Type	Description
Title	string	Title of the Result item
Url	string	The URL of the result item
ClickUrl	string	The URL for linking to the result item

Back to Class List

Zend_Service_Yahoo_WebResult

Each Web Search result is returned as a `Zend_Service_Yahoo_WebResult` object.

Properties

Table 40.18. Zend_Service_Yahoo_WebResult Properties

Name	Type	Description
Summary	string	Result summary
MimeType	string	Result mimetype
ModificationDate	string	The last modification date of the result as a UNIX timestamp.
CacheUrl	string	Yahoo! web cache URL for the result, if it exists.
CacheSize	int	The size of the Cache entry

Back to Class List

Zend_Service_Yahoo_ImageResult

Each Image Search result is returned as a `Zend_Service_Yahoo_ImageResult` object.

Properties

Table 40.19. Zend_Service_Yahoo_ImageResult Properties

Name	Type	Description
Summary	string	Result summary
RefererUrl	string	The URL of the page which contains the image
FileSize	int	The size of the image file in bytes
FileFormat	string	The format of the image (bmp, gif, jpeg, png, etc.)
Height	int	The height of the image
Width	int	The width of the image
Thumbnail	Zend_Service_Yahoo_Image	Image thumbnail

Back to Class List

Zend_Service_Yahoo_VideoResult

Each Video Search result is returned as a `Zend_Service_Yahoo_VideoResult` object.

Properties

Table 40.20. Zend_Service_Yahoo_VideoResult Properties

Name	Type	Description
Summary	string	Result summary
RefererUrl	string	The URL of the page which contains the video
FileSize	int	The size of the video file in bytes
FileFormat	string	The format of the video (avi, flash, mpeg, msmedia, quicktime, realmedia, etc.)
Height	int	The height of the video in pixels
Width	int	The width of the video in pixels
Duration	int	The length of the video in seconds
Channels	int	Number of audio channels in the video
Streaming	boolean	Whether the video is streaming or not
Thumbnail	Zend_Service_Yahoo_Image	Image thumbnail

Back to Class List

Zend_Service_Yahoo_LocalResult

Each Local Search result is returned as a `Zend_Service_Yahoo_LocalResult` object.

Properties

Table 40.21. Zend_Service_Yahoo_LocalResult Properties

Name	Type	Description
Address	string	Street Address of the result
City	string	City in which the result resides in
State	string	State in which the result resides in
Phone	string	Phone number for the result
Rating	int	User submitted rating for the result
Distance	float	The distance to the result from your specified location
MapUrl	string	A URL of a map for the result
BusinessUrl	string	The URL for the business website, if known
BusinessClickUrl	string	The URL for linking to the business website, if known

Back to Class List

Zend_Service_Yahoo_NewsResult

Each News Search result is returned as a `Zend_Service_Yahoo_NewsResult` object.

Properties

Table 40.22. Zend_Service_Yahoo_NewsResult Properties

Name	Type	Description
Summary	string	Result summary
NewsSource	string	The company who distributed the article
NewsSourceUrl	string	The URL for the company who distributed the article
Language	string	The language the article is in
PublishDate	string	The date the article was published as a UNIX timestamp
ModificationDate	string	The date the article was last modified as a UNIX timestamp
Thumbnail	Zend_Service_Yahoo_Image	Image Thumbnail for the article, if it exists

Back to Class List

Zend_Service_Yahoo_InlinkDataResult

Each Inbound Link Search result is returned as a `Zend_Service_Yahoo_InlinkDatabResult` object.

Back to Class List

Zend_Service_Yahoo_PageDataResult

Each Page Data Search result is returned as a `Zend_Service_Yahoo_PageDatabResult` object.

Back to Class List

Zend_Service_Yahoo_Image

All images returned either by the Yahoo! Image Search or the Yahoo! News Search are represented by `Zend_Service_Yahoo_Image` objects

Properties

Table 40.23. Zend_Service_Yahoo_Image Properties

Name	Type	Description
Url	string	Image URL
Width	int	Image Width
Height	int	Image Height

Back to Class List

Chapter 41. Zend_Session

Introduction

The Zend Framework Auth team greatly appreciates your feedback and contributions on our email list: fw-auth@lists.zend.com [mailto:fw-auth@lists.zend.com]

With web applications written using PHP, a **session** represents a logical, one-to-one connection between server-side, persistent state data and a particular user agent client (e.g., web browser). `Zend_Session` helps manage and preserve session data, a logical complement of cookie data, across multiple page requests by the same client. Unlike cookie data, session data are not stored on the client side and are only shared with the client when server-side source code voluntarily makes the data available in response to a client request. For the purposes of this component and documentation, the term "session data" refers to the server-side data stored in `$_SESSION` [http://www.php.net/manual/en/reserved.variables.php#reserved.variables.session], managed by `Zend_Session`, and individually manipulated by `Zend_Session_Namespace` accessor objects. **Session namespaces** provide access to session data using classic namespaces [http://en.wikipedia.org/wiki/Namespace_%28computer_science%29] implemented logically as named groups of associative arrays, keyed by strings (similar to normal PHP arrays).

`Zend_Session_Namespace` instances are accessor objects for namespaced slices of `$_SESSION`. The `Zend_Session` component wraps the existing PHP ext/session with an administration and management interface, as well as providing an API for `Zend_Session_Namespace` to persist session namespaces. `Zend_Session_Namespace` provides a standardized, object-oriented interface for working with namespaces persisted inside PHP's standard session mechanism. Support exists for both anonymous and authenticated (e.g., "login") session namespaces. `Zend_Auth`, the authentication component of the Zend Framework, uses `Zend_Session_Namespace` to store some information associated with authenticated users. Since `Zend_Session` uses the normal PHP ext/session functions internally, all the familiar configuration options and settings apply (see http://www.php.net/session), with such bonuses as the convenience of an object-oriented interface and default behavior that provides both best practices and smooth integration with the Zend Framework. Thus, a standard PHP session identifier, whether conveyed by cookie or within URLs, maintains the association between a client and session state data.

The default ext/session save handler [http://www.php.net/manual/en/function.session-set-save-handler.php] does not maintain this association for server clusters under certain conditions because session data are stored to the filesystem of the server that responded to the request. If a request may be processed by a different server than the one where the session data are located, then the responding server has no access to the session data (if they are not available from a networked filesystem). A list of additional, appropriate save handlers will be provided, when available. Community members are encouraged to suggest and submit save handlers to the fw-auth@lists.zend.com [mailto:fw-auth@lists.zend.com] list. A Zend_Db compatible save handler has been posted to the list.

Basic Usage

`Zend_Session_Namespace` instances provide the primary API for manipulating session data in the Zend Framework. Namespaces are used to segregate all session data, although a default namespace exists for those who only want one namespace for all their session data. `Zend_Session` utilizes ext/session and its special `$_SESSION` superglobal as the storage mechanism for session state data. While `$_SESSION` is still available in PHP's global namespace, developers should refrain from directly accessing it, so that `Zend_Session` and `Zend_Session_Namespace` can most effectively and securely provide its suite of session related functionality.

Each instance of `Zend_Session_Namespace` corresponds to an entry of the `$_SESSION` superglobal array, where the namespace is used as the key.

```
$myNamespace = new Zend_Session_Namespace('myNamespace');

// $myNamespace corresponds to $_SESSION['myNamespace']
```

It is possible to use Zend_Session in conjunction with other code that uses `$_SESSION` directly. To avoid problems, however, it is highly recommended that such code only uses parts of `$_SESSION` that do not correspond to instances of `Zend_Session_Namespace`.

Tutorial Examples

If no namespace is specified when instantiating `Zend_Session_Namespace`, all data will be transparently stored in a namespace called "`Default`". `Zend_Session` is not intended to work directly on the contents of session namespace containers. Instead, we use `Zend_Session_Namespace`. The example below demonstrates use of this default namespace, showing how to count the number of client requests during a session:

Example 41.1. Counting Page Views

```
$defaultNamespace = new Zend_Session_Namespace('Default');

if (isset($defaultNamespace->numberOfPageRequests)) {
    // this will increment for each page load.
    $defaultNamespace->numberOfPageRequests++;
} else {
    $defaultNamespace->numberOfPageRequests = 1; // first time
}

echo "Page requests this session: ",
    $defaultNamespace->numberOfPageRequests;
```

When multiple modules use instances of `Zend_Session_Namespace` having different namespaces, each module obtains data encapsulation for its session data. The `Zend_Session_Namespace` constructor can be passed an optional `$namespace` argument, which allows developers to partition session data into separate namespaces. Namespacing provides an effective and popular way to secure session state data against changes due to accidental naming collisions.

Namespace names are restricted to character sequences represented as non-empty PHP strings that do not begin with an underscore ("`_`") character. Only core components included in the Zend Framework should use namespace names starting with "`Zend`".

Example 41.2. New Way: Namespaces Avoid Collisions

```
// in the Zend_Auth component
$authNamespace = new Zend_Session_Namespace('Zend_Auth');
$authNamespace->user = "myusername";

// in a web services component
$webServiceNamespace = new Zend_Session_Namespace('Some_Web_Service');
$webServiceNamespace->user = "mywebusername";
```

The example above achieves the same effect as the code below, except that the session objects above preserve encapsulation of session data within their respective namespaces.

Example 41.3. Old Way: PHP Session Access

```
$_SESSION['Zend_Auth']['user'] = "myusername";
$_SESSION['Some_Web_Service']['user'] = "mywebusername";
```

Iterating Over Session Namespaces

`Zend_Session_Namespace` provides the full IteratorAggregate interface [http://www.php.net/~helly/php/ext/spl/interfaceIteratorAggregate.html], including support for the `foreach` statement:

Example 41.4. Session Iteration

```
$aNamespace =
    new Zend_Session_Namespace('some_namespace_with_data_present');

foreach ($aNamespace as $index => $value) {
    echo "aNamespace->$index = '$value';\n";
}
```

Accessors for Session Namespaces

`Zend_Session_Namespace` implements the `__get()`, `__set()`, `__isset()`, and `__unset()` magic methods [http://www.php.net/manual/en/language.oop5.overloading.php], which should not be invoked directly, except from within a subclass. Instead, the normal operators automatically invoke these methods, such as in the following example:

Example 41.5. Accessing Session Data

```
$namespace = new Zend_Session_Namespace(); // default namespace

$namespace->foo = 100;

echo "\$namespace->foo = $namespace->foo\n";

if (!isset($namespace->bar)) {
    echo "\$namespace->bar not set\n";
}

unset($namespace->foo);
```

Advanced Usage

While the basic usage examples are a perfectly acceptable way to utilize Zend Framework sessions, there are some best practices to consider. This section discusses the finer details of session handling and illustrates more advanced usage of the Zend_Session component.

Starting a Session

If you want all requests to have a session facilitated by Zend_Session, then start the session in the bootstrap file:

Example 41.6. Starting the Global Session

```
Zend_Session::start();
```

By starting the session in the bootstrap file, you avoid the possibility that your session might be started after headers have been sent to the browser, which results in an exception, and possibly a broken page for website viewers. Various advanced features require `Zend_Session::start()` first. (More on advanced features later.)

There are four ways to start a session, when using Zend_Session. Two are wrong.

1. Wrong: Do not enable PHP's `session.auto_start` setting [http://www.php.net/manual/en/ref.session.php#ini.session.auto-start]. If you do not have the ability to disable this setting in php.ini, you are using mod_php (or equivalent), and the setting is already enabled in `php.ini`, then add the following to your `.htaccess` file (usually in your HTML document root directory):

   ```
   php_value session.auto_start 0
   ```

2. Wrong: Do not use PHP's `session_start()` [http://www.php.net/session_start] function directly. If you use `session_start()` directly, and then start using `Zend_Session_Namespace`, an exception will be thrown by `Zend_Session::start()` ("session has already been started"). If you call `session_start()` after using `Zend_Session_Namespace` or calling `Zend_Session::start()`, an error of level `E_NOTICE` will be generated, and the call will be ignored.

3. Correct: Use `Zend_Session::start()`. If you want all requests to have and use sessions, then place this function call early and unconditionally in your bootstrap code. Sessions have some overhead. If some requests need sessions, but other requests will not need to use sessions, then:

   - Unconditionally set the `strict` option to `true` using `Zend_Session::setOptions()` in your bootstrap.

   - Call `Zend_Session::start()` only for requests that need to use sessions and before any `Zend_Session_Namespace` objects are instantiated.

   - Use "`new Zend_Session_Namespace()`" normally, where needed, but make sure `Zend_Session::start()` has been called previously.

   The `strict` option prevents `new Zend_Session_Namespace()` from automatically starting the session using `Zend_Session::start()`. Thus, this option helps application developers enforce a design decision to avoid using sessions for certain requests, since it causes an exception to be thrown when `Zend_Session_Namespace` is instantiated before `Zend_Session::start()` is called. Developers should carefully consider the impact of using `Zend_Session::setOptions()`, since these options have global effect, owing to their correspondence to the underlying options for ext/session.

4. Correct: Just instantiate `Zend_Session_Namespace` whenever needed, and the underlying PHP session will be automatically started. This offers extremely simple usage that works well in most situations. However, you then become responsible for ensuring that the first `new Zend_Session_Namespace()` happens **before** any output (e.g., HTTP headers [http://www.php.net/headers_sent]) has been sent by PHP to the client, if you are using the default, cookie-based sessions (strongly recommended). See the section called “Error: Headers Already Sent” for more information.

Locking Session Namespaces

Session namespaces can be locked, to prevent further alterations to the data in that namespace. Use `lock()` to make a specific namespace read-only, `unLock()` to make a read-only namespace read-write, and `isLocked()` to test if a namespace has been previously locked. Locks are transient and do not persist from one request to the next. Locking the namespace has no effect on setter methods of objects stored in the namespace, but does prevent the use of the namespace's setter method to remove or replace objects stored directly in the namespace. Similarly, locking `Zend_Session_Namespace` instances does not prevent the use of symbol table aliases to the same data (see PHP references [http://www.php.net/references]).

Example 41.7. Locking Session Namespaces

```
$userProfileNamespace = new Zend_Session_Namespace('userProfileNamespace');

// marking session as read only locked
$userProfileNamespace->lock();

// unlocking read-only lock
if ($userProfileNamespace->isLocked()) {
    $userProfileNamespace->unLock();
}
```

Namespace Expiration

Limits can be placed on the longevity of both namespaces and individual keys in namespaces. Common use cases include passing temporary information between requests, and reducing exposure to certain security risks by removing access to potentially sensitive information some time after authentication occurred. Expiration can be based on either elapsed seconds or the number of "hops", where a hop occurs for each successive request that instantiates the namespace at least once.

Example 41.8. Expiration Examples

```
$s = new Zend_Session_Namespace('expireAll');
$s->a = 'apple';
$s->p = 'pear';
$s->o = 'orange';

$s->setExpirationSeconds(5, 'a'); // expire only the key "a" in 5 seconds

// expire entire namespace in 5 "hops"
$s->setExpirationHops(5);

$s->setExpirationSeconds(60);
// The "expireAll" namespace will be marked "expired" on
// the first request received after 60 seconds have elapsed,
// or in 5 hops, whichever happens first.
```

When working with data expiring from the session in the current request, care should be used when retrieving them. Although the data are returned by reference, modifying the data will not make expiring data persist past the current request. In order to "reset" the expiration time, fetch the data into temporary variables, use the namespace to unset them, and then set the appropriate keys again.

Session Encapsulation and Controllers

Namespaces can also be used to separate session access by controllers to protect variables from contamination. For example, an authentication controller might keep its session state data separate from all other controllers for meeting security requirements.

Example 41.9. Namespaced Sessions for Controllers with Automatic Expiration

The following code, as part of a controller that displays a test question, initiates a boolean variable to represent whether or not a submitted answer to the test question should be accepted. In this case, the application user is given 300 seconds to answer the displayed question.

```
// ...
// in the question view controller
$testSpace = new Zend_Session_Namespace('testSpace');
// expire only this variable
$testSpace->setExpirationSeconds(300, 'accept_answer');
$testSpace->accept_answer = true;
//...
```

Below, the controller that processes the answers to test questions determines whether or not to accept an answer based on whether the user submitted the answer within the allotted time:

```
// ...
// in the answer processing controller
$testSpace = new Zend_Session_Namespace('testSpace');
if ($testSpace->accept_answer === true) {
    // within time
}
else {
    // not within time
}
// ...
```

Preventing Multiple Instances per Namespace

Although session locking provides a good degree of protection against unintended use of namespaced session data, `Zend_Session_Namespace` also features the ability to prevent the creation of multiple instances corresponding to a single namespace.

To enable this behavior, pass `true` to the second constructor argument when creating the last allowed instance of `Zend_Session_Namespace`. Any subsequent attempt to instantiate the same namespace would result in a thrown exception.

Example 41.10. Limiting Session Namespace Access to a Single Instance

```
// create an instance of a namespace
$authSpaceAccessor1 = new Zend_Session_Namespace('Zend_Auth');

// create another instance of the same namespace, but disallow any
// new instances
$authSpaceAccessor2 = new Zend_Session_Namespace('Zend_Auth', true);

// making a reference is still possible
$authSpaceAccessor3 = $authSpaceAccessor2;

$authSpaceAccessor1->foo = 'bar';

assert($authSpaceAccessor2->foo, 'bar');

try {
    $aNamespaceObject = new Zend_Session_Namespace('Zend_Auth');
} catch (Zend_Session_Exception $e) {
    echo 'Cannot instantiate this namespace since ' .
         '$authSpaceAccessor2 was created\n';
}
```

The second parameter in the constructor above tells `Zend_Session_Namespace` that any future instances with the "`Zend_Auth`" namespace are not allowed. Attempting to create such an instance causes an exception to be thrown by the constructor. The developer therefore becomes responsible for storing a reference to an instance object (`$authSpaceAccessor1`, `$authSpaceAccessor2`, or `$authSpaceAccessor3` in the example above) somewhere, if access to the session namespace is needed at a later time during the same request. For example, a developer may store the reference in a static variable, add the reference to a registry [http://www.martinfowler.com/eaaCatalog/registry.html] (see Chapter 36, *Zend_Registry*), or otherwise make it available to other methods that may need access to the session namespace.

Working with Arrays

Due to the implementation history of PHP magic methods, modifying an array inside a namespace may not work under PHP versions before 5.2.1. If you will only be working with PHP 5.2.1 or later, then you may skip to the next section.

Example 41.11. Modifying Array Data with a Session Namespace

The following illustrates how the problem may be reproduced:

```
$sessionNamespace = new Zend_Session_Namespace();
$sessionNamespace->array = array();

// may not work as expected before PHP 5.2.1
$sessionNamespace->array['testKey'] = 1;
echo $sessionNamespace->array['testKey'];
```

Example 41.12. Building Arrays Prior to Session Storage

If possible, avoid the problem altogether by storing arrays into a session namespace only after all desired array values have been set.

```
$sessionNamespace = new Zend_Session_Namespace('Foo');
$sessionNamespace->array = array('a', 'b', 'c');
```

If you are using an affected version of PHP and need to modify the array after assigning it to a session namespace key, you may use either or both of the following workarounds.

Example 41.13. Workaround: Reassign a Modified Array

In the code that follows, a copy of the stored array is created, modified, and reassigned to the location from which the copy was created, overwriting the original array.

```
$sessionNamespace = new Zend_Session_Namespace();

// assign the initial array
$sessionNamespace->array = array('tree' => 'apple');

// make a copy of the array
$tmp = $sessionNamespace->array;

// modfiy the array copy
$tmp['fruit'] = 'peach';

// assign a copy of the array back to the session namespace
$sessionNamespace->array = $tmp;

echo $sessionNamespace->array['fruit']; // prints "peach"
```

Example 41.14. Workaround: store array containing reference

Alternatively, store an array containing a reference to the desired array, and then access it indirectly.

```
$myNamespace = new Zend_Session_Namespace('myNamespace');
$a = array(1, 2, 3);
$myNamespace->someArray = array( &$a );
$a['foo'] = 'bar';
echo $myNamespace->someArray['foo']; // prints "bar"
```

Using Sessions with Objects

If you plan to persist objects in the PHP session, know that they will be serialized [http://www.php.net/manual/en/language.oop.serialization.php] for storage. Thus, any object persisted with the PHP session must be unserialized upon retrieval from storage. The implication is that the developer must ensure that the classes for the persisted objects must have been defined before the object is unserialized from session storage. If an unserialized object's class is not defined, then it becomes an instance of `stdClass`.

Using Sessions with Unit Tests

The Zend Framework relies on PHPUnit to facilitate testing of itself. Many developers extend the existing suite of unit tests to cover the code in their applications. The exception "**Zend_Session is currently marked as read-only**" is thrown while performing unit tests, if any write-related methods are used after ending the session. However, unit tests using Zend_Session require extra attention, because closing (`Zend_Session::writeClose()`), or destroying a session (`Zend_Session::destroy()`) prevents any further setting or unsetting of keys in any instance of `Zend_Session_Namespace`. This behavior is a direct result of the underlying ext/session mechanism and PHP's `session_destroy()` and `session_write_close()`, which have no "undo" mechanism to facilitate setup/teardown with unit tests.

To work around this, see the unit test `testSetExpirationSeconds()` in `SessionTest.php` and `SessionTestHelper.php`, both located in `tests/Zend/Session`, which make use of PHP's `exec()` to launch a separate process. The new process more accurately simulates a second, successive request from a browser. The separate process begins with a "clean" session, just like any PHP script execution for a web request. Also, any changes to `$_SESSION` made in the calling process become available to the child process, provided the parent closed the session before using `exec()`.

Example 41.15. PHPUnit Testing Code Dependent on Zend_Session

```
// testing setExpirationSeconds()
$script = 'SessionTestHelper.php';
$s = new Zend_Session_Namespace('space');
$s->a = 'apple';
$s->o = 'orange';
$s->setExpirationSeconds(5);

Zend_Session::regenerateId();
$id = Zend_Session::getId();
session_write_close(); // release session so process below can use it
sleep(4); // not long enough for things to expire
exec($script . "expireAll $id expireAll", $result);
$result = $this->sortResult($result);
$expect = ';a === apple;o === orange;p === pear';
$this->assertTrue($result === $expect,
    "iteration over default Zend_Session namespace failed; " .
    "expecting result === '$expect', but got '$result'");

sleep(2); // long enough for things to expire (total of 6 seconds
          // waiting, but expires in 5)
exec($script . "expireAll $id expireAll", $result);
$result = array_pop($result);
$this->assertTrue($result === '',
    "iteration over default Zend_Session namespace failed; " .
    "expecting result === '', but got '$result')");
session_start(); // resume artificially suspended session

// We could split this into a separate test, but actually, if anything
// leftover from above contaminates the tests below, that is also a
// bug that we want to know about.
$s = new Zend_Session_Namespace('expireGuava');
$s->setExpirationSeconds(5, 'g'); // now try to expire only 1 of the
                                  // keys in the namespace
$s->g = 'guava';
$s->p = 'peach';
$s->p = 'plum';

session_write_close(); // release session so process below can use it
sleep(6); // not long enough for things to expire
exec($script . "expireAll $id expireGuava", $result);
$result = $this->sortResult($result);
session_start(); // resume artificially suspended session
$this->assertTrue($result === ';p === plum',
    "iteration over named Zend_Session namespace failed (result=$result)");
```

Global Session Management

The default behavior of sessions can be modified using the static methods of Zend_Session. All management and manipulation of global session management occurs using Zend_Session, including configuration of the usual options provided by ext/session [http://www.php.net/session#session.configuration], using `Zend_Session::setOptions()`. For example, failure to insure the use of a safe `save_path` or a unique cookie name by ext/session using `Zend_Session::setOptions()` may result in security issues.

Configuration Options

When the first session namespace is requested, Zend_Session will automatically start the PHP session, unless already started with `Zend_Session::start()`. The underlying PHP session will use defaults from Zend_Session, unless modified first by `Zend_Session::setOptions()`.

To set a session configuration option, include the basename (the part of the name after "`session.`") as a key of an array passed to `Zend_Session::setOptions()`. The corresponding value in the array is used to set the session option value. If no options are set by the developer, Zend_Session will utilize recommended default options first, then the default php.ini settings. Community feedback about best practices for these options should be sent to fw-auth@lists.zend.com [mailto:fw-auth@lists.zend.com].

Example 41.16. Using Zend_Config to Configure Zend_Session

To configure this component using `Zend_Config_Ini`, first add the configuration options to the INI file:

```
; Accept defaults for production
[production]
; bug_compat_42
; bug_compat_warn
; cache_expire
; cache_limiter
; cookie_domain
; cookie_lifetime
; cookie_path
; cookie_secure
; entropy_file
; entropy_length
; gc_divisor
; gc_maxlifetime
; gc_probability
; hash_bits_per_character
; hash_function
; name should be unique for each PHP application sharing the same
; domain name
name = UNIQUE_NAME
; referer_check
; save_handler
; save_path
; serialize_handler
; use_cookies
; use_only_cookies
; use_trans_sid

; remember_me_seconds = <integer seconds>
; strict = on|off


; Development inherits configuration from production, but overrides
; several values
[development : production]
; Don't forget to create this directory and make it rwx (readable and
; modifiable) by PHP.
save_path = /home/myaccount/zend_sessions/myapp
use_only_cookies = on
; When persisting session id cookies, request a TTL of 10 days
remember_me_seconds = 864000
```

Next, load the configuration file and pass its array representation to `Zend_Session::setOptions()`:

```
$config = new Zend_Config_Ini('myapp.ini', 'development');
```

```
Zend_Session::setOptions($config->toArray());
```

Most options shown above need no explanation beyond that found in the standard PHP documentation, but those of particular interest are noted below.

- boolean `strict` - disables automatic starting of `Zend_Session` when using `new Zend_Session_Namespace()`.
- integer `remember_me_seconds` - how long should session id cookie persist, after user agent has ended (e.g., browser application terminated).
- string `save_path` - The correct value is system dependent, and should be provided by the developer using an **absolute path** to a directory readable and writable by the PHP process. If a writable path is not supplied, then `Zend_Session` will throw an exception when started (i.e., when `start()` is called).

Security Risk

If the path is readable by other applications, then session hijacking might be possible. If the path is writable by other applications, then session poisoning [http://en.wikipedia.org/wiki/Session_poisoning] might be possible. If this path is shared with other users or other PHP applications, various security issues might occur, including theft of session content, hijacking of sessions, and collision of garbage collection (e.g., another user's application might cause PHP to delete your application's session files).

For example, an attacker can visit the victim's website to obtain a session cookie. Then, he edits the cookie path to his own domain on the same server, before visiting his own website to execute `var_dump($_SESSION)`. Armed with detailed knowledge of the victim's use of data in their sessions, the attacker can then modify the session state (poisoning the session), alter the cookie path back to the victim's website, and then make requests from the victim's website using the poisoned session. Even if two applications on the same server do not have read/write access to the other application's `save_path`, if the `save_path` is guessable, and the attacker has control over one of these two websites, the attacker could alter their website's `save_path` to use the other's save_path, and thus accomplish session poisoning, under some common configurations of PHP. Thus, the value for `save_path` should not be made public knowledge and should be altered to a secure location unique to each application.

- string `name` - The correct value is system dependent and should be provided by the developer using a value **unique** to the application.

Security Risk

If the `php.ini` setting for `session.name` is the same (e.g., the default "PHPSESSID"), and there are two or more PHP applications accessible through the same domain name then they will share the same session data for visitors to both websites. Additionally, possible corruption of session data may result.

- boolean `use_only_cookies` - In order to avoid introducing additional security risks, do not alter the default value of this option.

Security Risk

If this setting is not enabled, an attacker can easily fix victim's session ids, using links on the attacker's website, such as `http://www.example.com/index.php?PHPSESSID=fixed_session_id`. The fixation works, if the victim does not already have a session id cookie for example.com. Once a victim is using a known session id, the attacker can then attempt to hijack the session by pretending to be the victim, and emulating the victim's user agent.

Error: Headers Already Sent

If you see the error message, "Cannot modify header information - headers already sent", or, "You must call ... before any output has been sent to the browser; output started in ...", then carefully examine the immediate cause (function or method) associated with the message. Any actions that require sending HTTP headers, such as sending a cookie, must be done before sending normal output (unbuffered output), except when using PHP's output buffering.

- Using output buffering [http://php.net/outcontrol] often is sufficient to prevent this issue, and may help improve performance. For example, in `php.ini`, "`output_buffering = 65535`" enables output buffering with a 64K buffer. Even though output buffering might be a good tactic on production servers to increase performance, relying only on buffering to resolve the "headers already sent" problem is not sufficient. The application must not exceed the buffer size, or the problem will occur whenever the output sent (prior to the HTTP headers) exceeds the buffer size.
- Alternatively, try rearranging the application logic so that actions manipulating headers are performed prior to sending any output whatsoever.
- If a Zend_Session method is involved in causing the error message, examine the method carefully, and make sure its use really is needed in the application. For example, the default usage of `destroy()` also sends an HTTP header to expire the client-side session cookie. If this is not needed, then use `destroy(false)`, since the instructions to set cookies are sent with HTTP headers.
- Alternatively, try rearranging the application logic so that all actions manipulating headers are performed prior to sending any output whatsoever.
- Remove any closing "`?>`" tags, if they occur at the end of a PHP source file. They are not needed, and newlines and other nearly invisible whitespace following the closing tag can trigger output to the client.

Session Identifiers

Introduction: Best practice in relation to using sessions with ZF calls for using a browser cookie (i.e. a normal cookie stored in your web browser), instead of embedding a unique session identifier in URLs as a means to track individual users. By default this component uses only cookies to maintain session identifiers. The cookie's value is the unique identifier of your browser's session. PHP's ext/session uses this identifier to maintain a unique one-to-one relationship between website visitors, and persistent session data storage unique to each visitor. Zend_Session* wraps this storage mechanism (`$_SESSION`) with an object-oriented interface. Unfortunately, if an attacker gains access to the value of the cookie (the session id), an attacker might be able to hijack a visitor's session. This problem is not unique to PHP, or the Zend Framework. The `regenerateId()` method allows an application to change the session id (stored in the visitor's cookie) to a new, random, unpredictable value. Note: Although not the same, to make this section easier to read, we use the terms "user agent" and "web browser" interchangeably.

Why?: If an attacker obtains a valid session identifier, an attacker might be able to impersonate a valid user (the victim), and then obtain access to confidential information or otherwise manipulate the victim's data managed by your application. Changing session ids helps protect against session hijacking. If the session id is changed, and an attacker does not know the new value, the attacker can not use the new session id in their attempts to hijack the visitor's session. Even if an attacker gains access to an old session id, `regenerateId()` also moves the session data from the old session id "handle" to the new one, so no data remains accessible via the old session id.

When to use regenerateId(): Adding `Zend_Session::regenerateId()` to your Zend Framework bootstrap yields one of the safest and most secure ways to regenerate session id's in user agent cookies. If there is no conditional logic to determine when to regenerate the session id, then there are no flaws in that logic. Although regenerating on every request prevents several possible avenues of attack, not everyone wants the associated small performance and bandwidth cost. Thus, applications commonly try to dynamically determine situations of greater risk, and only regenerate the session ids in those situations. Whenever a website visitor's session's privileges are "escalated" (e.g. a visitor re-authenticates their identity before editing their personal "profile"), or whenever a security "sensitive" session parameter change occurs, consider using `regenerateId()` to create a new session id. If you call the `rememberMe()` function, then don't use `regenerateId()`, since the former calls the latter. If a user has successfully logged into your website, use `rememberMe()` instead of `regenerateId()`.

Session Hijacking and Fixation

Avoiding cross-site script (XSS) vulnerabilities [http://en.wikipedia.org/wiki/Cross_site_scripting] helps preventing session hijacking. According to Secunia's [http://secunia.com/] statistics XSS problems occur frequently, regardless of the languages used to create web applications. Rather than expecting to never have a XSS problem with an application, plan for it by following best practices to help minimize damage, if it occurs. With XSS, an attacker does not need direct access to a victim's network traffic. If the victim already has a session cookie, Javascript XSS might allow an attacker to read the cookie and steal the session. For victims with no session cookies, using XSS to inject Javascript, an attacker could create a session id cookie on the victim's browser with a known value, then set an identical cookie on the attacker's system, in order to hijack the victim's session. If the victim visited an attacker's website, then the attacker can also emulate most other identifiable characteristics of the victim's user agent. If your website has an XSS vulnerability, the attacker might be able to insert an AJAX Javascript that secretly "visits" the attacker's website, so that the attacker knows the victim's browser characteristics and becomes aware of a compromised session at the victim website. However, the attacker can not arbitrarily alter the server-side state of PHP sessions, provided the developer has correctly set the value for the `save_path` option.

By itself, calling `Zend_Session::regenerateId()` when the user's session is first used, does not prevent session fixation attacks, unless you can distinguish between a session originated by an attacker emulating the victim. At first, this might sound contradictory to the previous statement above, until we consider an attacker who first initiates a real session on your website. The session is "first used" by the attacker, who then knows the result of the initialization (`regenerateId()`). The attacker then uses the new session id in combination with an XSS vulnerability, or injects the session id via a link on the attacker's website (works if `use_only_cookies = off`).

If you can distinguish between an attacker and victim using the same session id, then session hijacking can be dealt with directly. However, such distinctions usually involve some form of usability tradeoffs, because the methods of distinction are often imprecise. For example, if a request is received from an IP in a different country than the IP of the request when the session was created, then the new request probably belongs to an attacker. Under the following conditions, there might not be any way for a website application to distinguish between a victim and an attacker:

- attacker first initiates a session on your website to obtain a valid session id

- attacker uses XSS vulnerability on your website to create a cookie on the victim's browser with the same, valid session id (i.e. session fixation)
- both the victim and attacker originate from the same proxy farm (e.g. both are behind the same firewall at a large company, like AOL)

The sample code below makes it much harder for an attacker to know the current victim's session id, unless the attacker has already performed the first two steps above.

Example 41.17. Session Fixation

```
$defaultNamespace = new Zend_Session_Namespace();

if (!isset($defaultNamespace->initialized)) {
    Zend_Session::regenerateId();
    $defaultNamespace->initialized = true;
}
```

rememberMe(integer $seconds)

Ordinarily, sessions end when the user agent terminates, such as when an end user exits a web browser program. However, your application may provide the ability to extend user sessions beyond the lifetime of the client program through the use of persistent cookies. Use `Zend_Session::rememberMe()` before a session is started to control the length of time before a persisted session cookie expires. If you do not specify a number of seconds, then the session cookie lifetime defaults to `remember_me_seconds`, which may be set using `Zend_Session::setOptions()`. To help thwart session fixation/hijacking, use this function when a user successfully authenticates with your application (e.g., from a "login" form).

forgetMe()

This function complements `rememberMe()` by writing a session cookie that has a lifetime ending when the user agent terminates.

sessionExists()

Use this method to determine if a session already exists for the current user agent/request. It may be used before starting a session, and independently of all other `Zend_Session` and `Zend_Session_Namespace` methods.

destroy(bool $remove_cookie = true, bool $readonly = true)

`Zend_Session::destroy()` destroys all of the persistent data associated with the current session. However, no variables in PHP are affected, so your namespaced sessions (instances of `Zend_Session_Namespace`) remain readable. To complete a "logout", set the optional parameter to `true` (the default) to also delete the user agent's session id cookie. The optional `$readonly` parameter removes the ability to create new `Zend_Session_Namespace` instances and for `Zend_Session` methods to write to the session data store.

If you see the error message, "Cannot modify header information - headers already sent", then either avoid using `true` as the value for the first argument (requesting removal of the session cookie), or see the section called “Error: Headers Already Sent”. Thus, `Zend_Session::destroy(true)` must either be called before PHP has sent HTTP headers, or output buffering must be enabled. Also, the total output sent must not exceed the set buffer size, in order to prevent triggering sending the output before the call to `destroy()`.

Throws

By default, `$readonly` is enabled and further actions involving writing to the session data store will throw an exception.

stop()

This method does absolutely nothing more than toggle a flag in Zend_Session to prevent further writing to the session data store. We are specifically requesting feedback on this feature. Potential uses/abuses might include temporarily disabling the use of `Zend_Session_Namespace` instances or `Zend_Session` methods to write to the session data store, while execution is transfered to view- related code. Attempts to perform actions involving writes via these instances or methods will throw an exception.

writeClose($readonly = true)

Shutdown the session, close writing and detach `$_SESSION` from the back-end storage mechanism. This will complete the internal data transformation on this request. The optional `$readonly` boolean parameter can remove write access by throwing an exception upon any attempt to write to the session via `Zend_Session` or `Zend_Session_Namespace`.

Throws

By default, `$readonly` is enabled and further actions involving writing to the session data store will throw an exception. However, some legacy application might expect `$_SESSION` to remain writable after ending the session via `session_write_close()`. Although not considered "best practice", the `$readonly` option is available for those who need it.

expireSessionCookie()

This method sends an expired session id cookie, causing the client to delete the session cookie. Sometimes this technique is used to perform a client-side logout.

setSaveHandler(Zend_Session_SaveHandler_Interface $interface)

Most developers will find the default save handler sufficient. This method provides an object-oriented wrapper for `session_set_save_handler()` [http://php.net/session_set_save_handler].

namespaceIsset($namespace)

Use this method to determine if a session namespace exists, or if a particular index exists in a particular namespace.

Throws

An exception will be thrown if `Zend_Session` is not marked as readable (e.g., before `Zend_Session` has been started).

namespaceUnset($namespace)

Use `Zend_Session::namespaceUnset($namespace)` to efficiently remove an entire namespace and its contents. As with all arrays in PHP, if a variable containing an array is unset, and the array contains other objects, those objects will remain available, if they were also stored by reference in other array/objects that remain accessible via other variables. So `namespaceUnset()` does not perform a "deep" unsetting/deleting of the contents of the entries in the namespace. For a more detailed explanation, please see References Explained [http://php.net/references] in the PHP manual.

Throws

An exception will be thrown if the namespace is not writable (e.g., after `destroy()`).

namespaceGet($namespace)

DEPRECATED: Use `getIterator()` in `Zend_Session_Namespace`. This method returns an array of the contents of `$namespace`. If you have logical reasons to keep this method publicly accessible, please provide feedback to the fw-auth@lists.zend.com [mailto:fw-auth@lists.zend.com] mail list. Actually, all participation on any relevant topic is welcome :)

Throws

An exception will be thrown if `Zend_Session` is not marked as readable (e.g., before `Zend_Session` has been started).

getIterator()

Use `getIterator()` to obtain an array containing the names of all namespaces.

Throws

An exception will be thrown if `Zend_Session` is not marked as readable (e.g., before `Zend_Session` has been started).

Zend_Session_SaveHandler_DbTable

The basic setup for Zend_Session_SaveHandler_DbTable must at least have four columns, denoted in the config array/Zend_Config object: primary, which is the primary key and defaults to just the session id which by default is a string of length 32; modified, which is the unix timestamp of the last modified date; lifetime, which is the lifetime of the session (modified + lifetime > time()); and data, which is the serialized data stored in the session

Example 41.18. Basic Setup

```
CREATE TABLE `session` (
  `id` char(32),
  `modified` int,
  `lifetime` int,
  `data` text,
  PRIMARY KEY (`id`)
);
```

```
//get your database connection ready
$db = Zend_Db::factory('Pdo_Mysql', array(
    'host'        =>'example.com',
    'username'    => 'dbuser',
    'password'    => '******',
    'dbname'    => 'dbname'
));

//you can either set the Zend_Db_Table default adapter
//or you can pass the db connection straight to the save handler $config
Zend_Db_Table_Abstract::setDefaultAdapter($db);
$config = array(
    'name'           => 'session',
    'primary'        => 'id',
    'modifiedColumn' => 'modified',
    'dataColumn'     => 'data',
    'lifetimeColumn' => 'lifetime'
);

//create your Zend_Session_SaveHandler_DbTable and
//set the save handler for Zend_Session
Zend_Session::setSaveHandler(new Zend_Session_SaveHandler_DbTable($config));

//start your session!
Zend_Session::start();

//now you can use Zend_Session like any other time
```

You can also use Multiple Columns in your primary key for Zend_Session_SaveHandler_DbTable.

Example 41.19. Using a Multi-Column Primary Key

```
CREATE TABLE `session` (
    `session_id` char(32) NOT NULL,
    `save_path` varchar(32) NOT NULL,
    `name` varchar(32) NOT NULL DEFAULT '',
    `modified` int,
    `lifetime` int,
    `session_data` text,
    PRIMARY KEY (`Session_ID`, `save_path`, `name`)
);
```

```
//setup your DB connection like before
//NOTE: this config is also passed to Zend_Db_Table so anything specific
//to the table can be put in the config as well
$config = array(
    'name'              => 'session', //table name as per Zend_Db_Table
    'primary'           => array(
        'session_id',   //the sessionID given by php
        'save_path',    //session.save_path
        'name',         //session name
    ),
    'primaryAssignment' => array( //you must tell the save handler which columns y
                                  //are using as the primary key. ORDER IS IMPORTA
        'sessionId',          //first column of the primary key is of the sessionI
        'sessionSavePath',    //second column of the primary key is the save path
        'sessionName',        //third column of the primary key is the session nam
    ),
    'modifiedColumn'    => 'modified',     //time the session should expire
    'dataColumn'        => 'session_data', //serialized data
    'lifetimeColumn'    => 'lifetime',     //end of life for a specific record
);

//Tell Zend_Session to use your Save Handler
Zend_Session::setSaveHandler(new Zend_Session_SaveHandler_DbTable($config));

//start your session
Zend_Session::start();

//use Zend_Session as normal
```

Chapter 42. Zend_Soap

Zend_Soap_Server

`Zend_Soap_Server` class is intended to simplify Web Services server part development for PHP programmers.

It may be used in WSDL or non-WSDL mode, and using classes or functions to define Web Service API.

When Zend_Soap_Server component works in the WSDL mode, it uses already prepared WSDL document to define server object behavior and transport layer options.

WSDL document may be auto-generated with functionality provided by Zend_Soap_AutoDiscovery component or should be constructed manually using `Zend_Soap_Wsdl` class or any other XML generating tool.

If the non-WSDL mode is used, then all protocol options have to be set using options mechanism.

`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:

1. `$wsdl`, which is an URI of a WSDL file[1].
2. `$options` - options to create SOAP server object[2].

   The following options are recognized in the WSDL mode:

   - 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or SOAP_1_2).
   - 'actor' - the actor URI for the server.
   - 'classmap' ('classMap') which can be used to map some WSDL types to PHP classes.

     The option must be an array with WSDL types as keys and names of PHP classes as values.
   - 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
   - 'wsdl' which is equivalent to `setWsdl($wsdlValue)` call.

`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).

[1] May be set later using `setWsdl($wsdl)` method.
[2] Options may be set later using `setOptions($options)` method.

The second constructor parameter (`$options`) is an array with options to create SOAP server object[3].

The following options are recognized in the non-WSDL mode:

- 'soap_version' ('soapVersion') - soap version to use (SOAP_1_1 or SOAP_1_2).
- 'actor' - the actor URI for the server.
- 'classmap' ('classMap') which can be used to map some WSDL types to PHP classes.

  The option must be an array with WSDL types as keys and names of PHP classes as values.
- 'encoding' - internal character encoding (UTF-8 is always used as an external encoding).
- 'uri' (required) - URI namespace for SOAP server.

Methods to define Web Service API.

There are two ways to define Web Service API when your want to give access to your PHP code through SOAP.

The first one is to attach some class to the `Zend_Soap_Server` object which has to completely describe Web Service API:

```
...
class MyClass {
    /**
     * This method takes ...
     *
     * @param integer $inputParam
     * @return string
     */
    public function method1($inputParam) {
        ...
    }

    /**
     * This method takes ...
     *
     * @param integer $inputParam1
     * @param string  $inputParam2
     * @return float
     */
    public function method2($inputParam1, $inputParam2) {
        ...
    }

    ...
}
...
$server = new Zend_Soap_Server(null, $options);
$server->setClass('MyClass');
```

[3] Options may be set later using `setOptions($options)` method.

```
...
$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:

```
...
/**
 * This function ...
 *
 * @param integer $inputParam
 * @return string
 */
function function1($inputParam) {
    ...
}

/**
 * This function ...
 *
 * @param integer $inputParam1
 * @param string  $inputParam2
 * @return float
 */
function function2($inputParam1, $inputParam2) {
    ...
}
...
$server = new Zend_Soap_Server(null, $options);
$server->addFunction('function1');
$server->addFunction('function2');
...
$server->handle();
```

Request and response objects handling.

Advanced

This section describes advanced request/response processing options and may be skipped.

Zend_Soap_Server component performs request/response processing automatically, but allows to catch it and do some pre- and post-processing.

Request processing.

`Zend_Soap_Server::handle()` method takes request from the standard input stream ('php://input'). It may be overridden either by supplying optional parameter to the `handle()` method or by setting request using `setRequest()` method:

```
...
$server = new Zend_Soap_Server(...);
...
// Set request using optional $request parameter
$server->handle($request);
...
// Set request using setRequest() method
$server->setRequest();
$server->handle();
```

Request object may be represented using any of the following:

- DOMDocument (casted to XML)
- DOMNode (owner document is grabbed and casted to XML)
- SimpleXMLElement (casted to XML)
- stdClass (__toString() is called and verified to be valid XML)
- string (verified to be valid XML)

Last processed request may be retrieved using `getLastRequest()` method as an XML string:

```
...
$server = new Zend_Soap_Server(...);
...
$server->handle();
$request = $server->getLastRequest();
```

Response pre-processing.

`Zend_Soap_Server::handle()` method automatically emits generated response to the output stream. It may be blocked using `setReturnResponse()` with `true` or `false` as a parameter[4]. Generated response is returned by `handle()` method in this case.

```
...
$server = new Zend_Soap_Server(...);
...
// Get a response as a return value of handle() method instead of emitting it to t
```

[4] Current state of the Return Response flag may be requested with `setReturnResponse()` method.

```
$server->setReturnResponse(true);
...
$response = $server->handle();
...
```

Last response may be also retrieved by `getLastResponse()` method for some post-processing:

```
...
$server = new Zend_Soap_Server(...);
...
$server->handle();
$response = $server->getLastResyponse();
...
```

Zend_Soap_Client

The `Zend_Soap_Client` class simplifies SOAP client development for PHP programmers.

It may be used in WSDL or non-WSDL mode.

Under the WSDL mode, the Zend_Soap_Client component uses a WSDL document to define transport layer options.

The WSDL description is usually provided by the web service the client will access. If the WSDL description is not made avaiable, you may want to use Zend_Soap_Client in non-WSDL mode. Under this mode, all SOAP protocol options have to be set explicitely 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:

```
// Accept response compression
$client = new SoapClient("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT));
...

// Compress requests using gzip with compression level 5
$client = new SoapClient("some.wsdl",
  array('compression' => SOAP_COMPRESSION_ACCEPT | SOAP_COMPRESSION_GZIP | 5));
...

// Compress requests using deflate compression
$client = new SoapClient("some.wsdl",
  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:

```
//****************************************************************
//                Server code
//****************************************************************
// class MyClass {
//     /**
//      * This method takes ...
//      *
//      * @param integer $inputParam
//      * @return string
//      */
//     public function method1($inputParam) {
//         ...
//     }
//
//     /**
//      * This method takes ...
//      *
//      * @param integer $inputParam1
//      * @param string  $inputParam2
//      * @return float
//      */
//     public function method2($inputParam1, $inputParam2) {
//         ...
//     }
//
//     ...
// }
// ...
// $server = new Zend_Soap_Server(null, $options);
// $server->setClass('MyClass');
// ...
// $server->handle();
//
//****************************************************************
//                End of server code
//****************************************************************

$client = new SoapClient("MyService.wsdl");
...

// $result1 is a string
$result1 = $client->method1(10);
...

// $result2 is a float
$result2 = $client->method2(22, 'some string');
```

WSDL Accessor

Note

`Zend_Soap_Wsdl` class is used by Zend_Soap_Server component internally to operate with WSDL documents. Nevertheless, you could also use functionality provided by this class for your own needs.

If you don't plan to do this, you can skip this documentation section.

`Zend_Soap_Wsdl` constructor.

`Zend_Soap_Wsdl` constructor takes three parameters:

1. `$name` - name of the Web Service being described.
2. `$uri` - URI where the WSDL will be available (could also be a reference to the file in the filesystem.)
3. `$extractComplexTypes` - optional flag used to identify that complex types (objects) should be extracted.

`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 message name.

`$parts` parameter is an array of message parts which describe SOAP call parameters. It's an associative array: 'part name' (SOAP call parameter name) => 'part type'.

Type mapping management is performed using `addTypes()`, `addTypes()` and `addComplexType()` methods (see below).

Note

Messages parts can use either 'element' or 'type' attribute for typing (see http://www.w3.org/TR/wsdl#_messages).

'element' attribute must refer to a corresponding element of data type definition. 'type' attribute refers to a corresponding complexType entry.

All standard XSD types have both 'element' and 'complexType' definitions (see http://schemas.xmlsoap.org/soap/encoding/).

All non-standard types, which may be added using `Zend_Soap_Wsdl::addComplexType()` method, are described using 'complexType' node of '/definitions/types/schema/' section of WSDL document.

So `addMessage()` method always uses 'type' attribute to describe types.

`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 `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 st 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 Wed Service name, port name, binding, and location.

WSDL 1.1 allows to have several port types (sets of operations) per service. This ability is not used by Zend_Soap_Server implementation and not supported by `Zend_Soap_Wsdl` class.

Zend_Soap_Server implementation uses:

- `$name . 'Service'` as a Web Service name,
- `$name . 'Port'` as a port type name,
- `'tns:' . $name . 'Binding'` [5] as binding name,
- script URI[6] as a service URI for Web Service definition using classes.

[5] `'tns:' namespace` is defined as script URI (`'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']`).
[6] `'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']`

where `$name` is a class name for the Web Service definition mode using class and script name for the Web Service definition mode using set of functions.

See http://www.w3.org/TR/wsdl#_services for the details.

Type mapping.

Zend_Soap 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 <-> `tns:$className` [7].
- 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:

```
...
$wsdl = new Zend_Soap_Wsdl('My_Web_Service', $myWebServiceUri);

...
$soapIntType = $wsdl->getType('int');

...
class MyClass {
    ...
}
...
$soapMyClassType = $wsdl->getType('MyClass');
```

[7] If `Zend_Soap_Wsdl` object is created with `$extractComplexTypes` parameter turned off, then classes are translated to `xsd:anyType`. Otherwise, `tns:$className` is used and type is described in details in <types> WSDL section.

Retrieving 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 type if `$extractComplexTypes` parameter of the constructor is set to `true` (otherwise classes are mapped to 'xsd:anyType' SOAP type).

`addComplexType()` method creates '/definitions/types/xsd:schema/xsd:complexType' element for each described complex type with name of the specified PHP class.

All class public properties are described as corresponding elements of 'xsd:all' node attached to 'xsd:complexType'.

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 finilised 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.

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:

1. SOAP server PHP application <---> SOAP client PHP application

2. SOAP server non-PHP application <---> SOAP client PHP application

3. 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 [http://www.w3.org/TR/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.

Zend_Soap_AutoDiscovery class also supports datatypes mapping from PHP to XSD types [http://www.w3.org/TR/xmlschema-2/].

Here is an example of common usage of the autodiscovery functionality:

```
class My_SoapServer_Class {
...
}

$autodiscover = new Zend_Soap_AutoDiscover();
$autodiscover->setClass('My_SoapServer_Class');
$autodiscover->handle();
```

Class autodiscovering.

If class is used to provide SOAP server functionality, then the same class should be provided to `Zend_Soap_AutoDiscovery` for WSDL generation:

```
$autodiscover = new Zend_Soap_AutoDiscover();
$autodiscover->setClass('My_SoapServer_Class');
$autodiscover->handle();
```

The following rules are used while WSDL generation:

- Generated WSDL describes an RPC style Web Service.
- Class name is used as a name of the Web Service being described.

- `'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']` is used as an URI where the WSDL is available.

  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 [http://www.w3.org/TR/wsdl#_porttypes].

  `$className . 'Port'` is used as Port Type name.

- Each class method is registered as a corresponding port operation.

- Each method prototype genrerates corresponding Request/Response messages.

  Method may have several prototypes if some method parameters are optional.

Important!

WSDL autodiscovery 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:

```
$autodiscover = new Zend_Soap_AutoDiscover();
$autodiscover->addFunction('function1');
$autodiscover->addFunction('function2');
$autodiscover->addFunction('function3');
...
$autodiscover->handle();
```

The following rules are used while WSDL generation:

- Generated WSDL describes an RPC style Web Service.

- Current script name is used as a name of the Web Service being described.

- `'http://' .$_SERVER['HTTP_HOST'] . $_SERVER['SCRIPT_NAME']` is used as an URI where the WSDL is available.

  It's also used as a target namespace for all service related names (including described complex types).

- Functions are joined into one Port Type [http://www.w3.org/TR/wsdl#_porttypes].

  `$functionName . 'Port'` is used as Port Type name.

- Each function is registered as a corresponding port operation.

- Each function prototype genrerates corresponding Request/Response messages.

  Function may have several prototypes if some method parameters are optional.

Important!

WSDL autodiscovery 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.

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 <-> `tns:$className` [8].
- 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.

[8] If `Zend_Soap_AutoDiscover` object is created with `$extractComplexTypes` parameter turned off, then classes are translated to `xsd:anyType`.

Otherwise, `tns:$className` is used and type is described in details in <types> WSDL section.

Chapter 43. Zend_Test

Introduction

Zend_Test 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

Zend_Test_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.

Example 43.1. Application Login TestCase example

The following is a simple test case for a `UserController` to verify several things:

- The login form should be displayed to non-authenticated users.
- When a user logs in, they should be redirected to their profile page, and that profile page should show relevant information.

This particular example assumes a few things. First, we're moving most of our bootstrapping to a plugin. This simplifies setup of the test case as it allows us to specify our environment succinctly, and also allows us to bootstrap the application in a single line. Also, our particular example is assuming that autoloading is setup so we do not need to worry about requiring the appropriate classes (such as the correct controller, plugin, etc).

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function setUp()
    {
        $this->bootstrap = array($this, 'appBootstrap');
        parent::setUp();
    }

    public function appBootstrap()
    {
        $this->frontController->registerPlugin(new Bugapp_Plugin_Initialize('devel
    }

    public function testCallWithoutActionShouldPullFromIndexAction()
    {
        $this->dispatch('/user');
        $this->assertController('user');
        $this->assertAction('index');
    }

    public function testIndexActionShouldContainLoginForm()
    {
        $this->dispatch('/user');
        $this->assertAction('index');
        $this->assertQueryCount('form#loginForm', 1);
    }

    public function testValidLoginShouldGoToProfilePage()
    {
        $this->request->setMethod('POST')
              ->setPost(array(
                  'username' => 'foobar',
                  'password' => 'foobar'
              ));
        $this->dispatch('/user/login');
        $this->assertRedirectTo('/user/view');

        $this->resetResponse();
        $this->request->setMethod('GET')
             ->setPost(array());
        $this->dispatch('/user/view');
        $this->assertRoute('default');
        $this->assertModule('default');
        $this->assertController('user');
        $this->assertAction('view');
        $this->assertNotRedirect();
        $this->assertQuery('dl');
        $this->assertQueryContentContains('h2', 'User: foobar');
    }
}
```

This example could be written somewhat simpler -- not all the assertions shown are necessary, and are provided for illustration purposes only. Hopefully, it shows how simple it can be to test your applications.

Bootstrapping your TestCase

As noted in the Login example, all MVC test cases should extend `Zend_Test_PHPUnit_ControllerTestCase`. 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 bootstrap it. There are several ways to do this, all of which hinge on the public `$bootstrap` property.

First, you can set this property to point to a file. If you do this, the file should *not* dispatch the front controller, but merely setup the front controller and any application specific needs.

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public $bootstrap = '/path/to/bootstrap/file.php'

    // ...
}
```

Second, you can provide a PHP callback to execute in order to bootstrap your application. This method is seen in the Login example. If the callback is a function or static method, this could be set at the class level:

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public $bootstrap = array('App', 'bootstrap');

    // ...
}
```

In cases where an object instance is necessary, we recommend performing this in your `setUp()` method:

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function setUp()
    {
        // Use the 'start' method of a Bootstrap object instance:
        $bootstrap = new Bootstrap('test');
        $this->bootstrap = array($bootstrap, 'start');
        parent::setUp();
    }
}
```

Note the call to `parent::setUp()`; this is necessary, as the `setUp()` method of `Zend_Test_PHPUnit_Controller_TestCase` will perform the remainder of the bootstrapping process (which includes calling the callback).

During normal operation, the `setUp()` method will bootstrap the application. This process first will include cleaning up the environment to a clean request state, resetting any plugins and helpers, resetting the front controller instance, and creating new request and response objects. Once this is done, it will then either `include` the file specified in `$bootstrap`, or call the callback specified.

Bootstrapping should be as close as possible to how the application will be bootstrapped. However, there are several caveats:

- Do not provide alternate implementations of the Request and Response objects; they will not be used. `Zend_Test_PHPUnit_Controller_TestCase` uses custom request and response objects, `Zend_Controller_Request_HttpTestCase` and `Zend_Controller_Response_HttpTestCase`, respectively. These objects provide methods for setting up the request environment in targetted ways, and pulling response artifacts in specific ways.
- Do not expect to test server specifics. In other words, the tests are not a guarantee that the code will run on a specific server configuration, but merely that the application should run as expected should the router be able to route the given request. To this end, do not set server-specific headers in the request object.

Once the application is bootstrapped, you can then start creating your tests.

Testing your Controllers and MVC Applications

Once you have your bootstrap 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:

```
class IndexControllerTest extends Zend_Test_PHPUnit_Controller_TestCase
{
    // ...

    public function testHomePage()
    {
        $this->dispatch('/');
        // ...
    }
}
```

There will be times, however, that you need to provide extra information -- GET and POST variables, COOKIE information, etc. You can populate the request with that information:

```
class FooControllerTest extends Zend_Test_PHPUnit_Controller_TestCase
{
    // ...

    public function testBarActionShouldReceiveAllParameters()
    {
        // Set GET variables:
        $this->request->setQuery(array(
            'foo' => 'bar',
            'bar' => 'baz',
        ));

        // Set POST variables:
        $this->request->setPost(array(
            'baz'  => 'bat',
            'lame' => 'bogus',
        ));

        // Set a cookie value:
        $this->request->setCookie('user', 'matthew');
        // or many:
        $this->request->setCookies(array(
            'timestamp' => time(),
            'host'      => 'foobar',
        ));

        // Set headers, even:
        $this->request->setHeader('X-Requested-With', 'XmlHttpRequest');

        // Set the request method:
        $this->request->setMethod('POST');

        // Dispatch:
        $this->dispatch('/foo/bar');

        // ...
    }
}
```

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_ControllerTestCase` provides a number of assertions to make testing your MVC apps and controllers simpler.

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, $message = '')`: assert that one or more DOM elements matching the given CSS selector are present. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryContentContains($path, $match, $message = '')`: assert that one or more DOM elements matching the given CSS selector are present, and that at least one contains the content provided in `$match`. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryContentRegex($path, $pattern, $message = '')`: assert that one or more DOM elements matching the given CSS selector are present, and that at least one matches the regular expression provided in `$pattern`. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryCount($path, $count, $message = '')`: assert that there are exactly `$count` DOM elements matching the given CSS selector present. If a `$message` is present, it will be prepended to any failed assertion message.
- `assertQueryCountMin($path, $count, $message = '')`: assert that there are at least `$count` DOM elements matching the given CSS selector present. If a `$message` is present, it will be prepended to any failed assertion message. *Note:* specifying a value of 1 for `$count` is the same as simply using `assertQuery()`.
- `assertQueryCountMax($path, $count, $message = '')`: assert that there are no more than `$count` DOM elements matching the given CSS selector present. If a `$message` is present, it will be prepended to any failed assertion message. *Note:* specifying a value of 1 for `$count` is the same as simply using `assertQuery()`.

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:

- `assertXpath($path, $message = '')`
- `assertNotXpath($path, $message = '')`
- `assertXpathContentContains($path, $match, $message = '')`
- `assertNotXpathContentContains($path, $match, $message = '')`
- `assertXpathContentRegex($path, $pattern, $message = '')`
- `assertNotXpathContentRegex($path, $pattern, $message = '')`
- `assertXpathCount($path, $count, $message = '')`
- `assertNotXpathCount($path, $count, $message = '')`
- `assertXpathCountMin($path, $count, $message = '')`
- `assertNotXpathCountMax($path, $count, $message = '')`

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($message = '')`: assert simply that a redirect has occurred.
- `assertNotRedirect($message = '')`: assert that no redirect has occurred.
- `assertRedirectTo($url, $message = '')`: assert that a redirect has occurred, and that the value of the Location header is the `$url` provided.
- `assertNotRedirectTo($url, $message = '')`: assert that a redirect has either NOT occurred, or that the value of the Location header is NOT the `$url` provided.
- `assertRedirectRegex($pattern, $message = '')`: assert that a redirect has occurred, and that the value of the Location header matches the regular expression provided by `$pattern`.
- `assertNotRedirectRegex($pattern, $message = '')`: assert that a redirect has either NOT occurred, or that the value of the Location header does NOT match the regular expression provided by `$pattern`.

Response Header Assertions

In addition to checking for redirect headers, you will often need to check for specific HTTP response codes and headers -- for instance, to determine whether an action results in a 404 or 500 response, or to ensure that JSON responses contain the appropriate Content-Type header. The following assertions are available.

- `assertResponseCode($code, $message = '')`: assert that the response resulted in the given HTTP response code.

- `assertHeader($header, $message = '')`: assert that the response contains the given header.
- `assertHeaderContains($header, $match, $message = '')`: assert that the response contains the given header and that its content contains the given string.
- `assertHeaderRegex($header, $pattern, $message = '')`: 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.

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:

- `assertModule($module, $message = '')`: Assert that the given module was used in the last dispatched action.
- `assertController($controller, $message = '')`: Assert that the given controller was selected in the last dispatched action.
- `assertAction($action, $message = '')`: Assert that the given action was last dispatched.
- `assertRoute($route, $message = '')`: Assert that the given named route was matched by the router.

Each also has a 'Not' variant for negative assertions.

Examples

Knowing how to setup your testing infrastructure and how to make assertions is only half the battle; now it's time to start looking at some actual testing scenarios to see how you can leverage them.

Example 43.2. Testing a UserController

Let's consider a standard task for a website: authenticating and registering users. In our example, we'll define a UserController for handling this, and have the following requirements:

- If a user is not authenticated, they will always be redirected to the login page of the controller, regardless of the action specified.
- The login form page will show both the login form and the registration form.
- Providing invalid credentials should result in returning to the login form.
- Valid credentials should result in redirecting to the user profile page.
- The profile page should be customized to contain the user's username.
- Authenticated users who visit the login page should be redirected to their profile page.
- On logout, a user should be redirected to the login page.
- With invalid data, registration should fail.

We could, and should define further tests, but these will do for now.

For our application, we will define a plugin, 'Initialize', that runs at `routeStartup()`. This allows us to encapsulate our bootstrap in an OOP interface, which also provides an easy way to provide a callback. Let's look at the basics of this class first:

```
class Bugapp_Plugin_Initialize extends Zend_Controller_Plugin_Abstract
{
    /**
     * @var Zend_Config
     */
    protected static $_config;

    /**
     * @var string Current environment
     */
    protected $_env;

    /**
     * @var Zend_Controller_Front
     */
    protected $_front;

    /**
     * @var string Path to application root
     */
    protected $_root;

    /**
     * Constructor
     *
     * Initialize environment, root path, and configuration.
     *
     * @param  string $env
     * @param  string|null $root
     * @return void
     */
    public function __construct($env, $root = null)
    {
        $this->_setEnv($env);
        if (null === $root) {
            $root = realpath(dirname(__FILE__) . '/../../../');
        }
        $this->_root = $root;

        $this->initPhpConfig();

        $this->_front = Zend_Controller_Front::getInstance();
    }

    /**
     * Route startup
     *
     * @return void
     */
    public function routeStartup(Zend_Controller_Request_Abstract $request)
    {
        $this->initDb();
        $this->initHelpers();
        $this->initView();
        $this->initPlugins();
        $this->initRoutes();
        $this->initControllers();
    }

    // definition of methods would follow...
}
```

This allows us to create a bootstrap callback like the following:

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    public function appBootstrap()
    {
        $controller = $this->getFrontController();
        $controller->registerPlugin(new Bugapp_Plugin_Initialize('development'));
    }

    public function setUp()
    {
        $this->bootstrap = array($this, 'appBootstrap');
        parent::setUp();
    }

    // ...
}
```

Once we have that in place, we can write our tests. However, what about those tests that require a user is logged in? The easy solution is to use our application logic to do so... and fudge a little by using the `resetResponse()` method, which will allow us to dispatch another request.

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    // ...

    public function loginUser($user, $password)
    {
        $request = $this->getRequest();
        $request->setMethod('POST')
                ->setPost(array(
                    'username' => $user,
                    'password' => $password,
                ));
        $this->dispatch('/user/login');
        $this->assertRedirectTo('/user/view');
        $this->resetResponse();
        $request->setPost(array());
    }

    // ...
}
```

Now let's write tests:

```
class UserControllerTest extends Zend_Test_PHPUnit_ControllerTestCase
{
    // ...

    public function testCallWithoutActionShouldPullFromIndexAction()
    {
        $this->dispatch('/user');
        $this->assertController('user');
        $this->assertAction('index');
    }

    public function testLoginFormShouldContainLoginAndRegistrationForms()
    {
        $this->dispatch('/user');
        $this->assertQueryCount('form', 2);
    }

    public function testInvalidCredentialsShouldResultInRedisplayOfLoginForm()
    {
        $request = $this->getRequest();
        $request->setMethod('POST')
                ->setPost(array(
                    'username' => 'bogus',
                    'password' => 'reallyReallyBogus',
                ));
        $this->dispatch('/user/login');
        $this->assertNotRedirect();
        $this->assertQuery('form');
    }

    public function testValidLoginShouldRedirectToProfilePage()
    {
        $this->loginUser('foobar', 'foobar');
    }

    public function testAuthenticatedUserShouldHaveCustomizedProfilePage()
    {
        $this->loginUser('foobar', 'foobar');
        $this->request->setMethod('GET');
        $this->dispatch('/user/view');
        $this->assertNotRedirect();
        $this->assertQueryContentContains('h2', 'foobar');
    }

    public function testAuthenticatedUsersShouldBeRedirectedToProfilePageWhenVisit
    {
        $this->loginUser('foobar', 'foobar');
        $this->request->setMethod('GET');
        $this->dispatch('/user');
        $this->assertRedirectTo('/user/view');
    }

    public function testUserShouldRedirectToLoginPageOnLogout()
    {
        $this->loginUser('foobar', 'foobar');
        $this->request->setMethod('GET');
        $this->dispatch('/user/logout');
        $this->assertRedirectTo('/user');
    }

    public function testRegistrationShouldFailWithInvalidData()
    {
        $data = array(
            'username' => 'This will not work',
            'email'    => 'this is an invalid email',
            'password' => 'Th1s!s!nv@l1d',
            'passwordVerification' => 'wrong!',
        );
        $request = $this->getRequest();
        $request->setMethod('POST')
                ->setPost($data);
        $this->dispatch('/user/register');
        $this->assertNotRedirect();
        $this->assertQuery('form .errors');
    }
}
```

Notice that these are terse, and, for the most part, don't look for actual content. Instead, they look for artifacts within the response -- response codes and headers, and DOM nodes. This allows you to verify that the structure is as expected -- preventing your tests from choking every time new content is added to the site.

Also notice that we use the structure of the document in our tests. For instance, in the final test, we look for a form that has a node with the class of "errors"; this allows us to test merely for the presence of form validation errors, and not worry about what specific errors might have been thrown.

This application *may* utilize a database. If so, you will probably need some scaffolding to ensure that the database is in a pristine, testable configuration at the beginning of each test. PHPUnit already provides functionality for doing so; read about it in the PHPUnit documentation [http://www.phpunit.de/pocket_guide/3.3/en/database.html]. We recommend using a separate database for testing versus production, and in particular recommend using either a SQLite file or in-memory database, as both options perform very well, do not require a separate server, and can utilize most SQL syntax.

Chapter 44. Zend_Text

Zend_Text_Figlet

Zend_Text_Figlet is a component which enables developers to create a so called FIGlet text. A FIGlet text is a string, which is represented as ASCII art. FIGlets use a special font format, called FLT (FigLet Font). By default, one standard font is shipped with Zend_Text_Figlet, but you can download additional fonts at http://www.figlet.org [http://www.figlet.org/fontdb.cgi].

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.

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, 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::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::DIRECTION_LEFT_TO_RIGHT or Zend_Text_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::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 [http://www.jave.de/figlet/figfont.txt]. 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.

Example 44.1. Using Zend_Text_Figlet

This example illustrates the basic use of `Zend_Text_Figlet` to create a simple FIGlet text:

```
$figlet = new Zend_Text_Figlet();
echo $figlet->render('Zend');
```

Assuming you are using a monospace font, this would look as follows:

```
 ______    ______    _  __   ______
|__  //   |  ___|| | \| || |  __ \\
  / //    | ||__    |  ' || | |  \ ||
 / //__   | ||___   | .  || | |__/ ||
/_____||  |_____||  |_|\_|| |_____//
`-----`'  `-----`   `-` -`'  -----`
```

Chapter 45. Zend_TimeSync

Introduction

`Zend_TimeSync` is able to receive internet or network time from a timeserver using the **NTP** or **SNTP** protocol. With `Zend_TimeSync` the Zend Framework is able to act indepentendly from the timesettings of the server where it is running.

To be independent from the actual time of the server, `Zend_TimeSync` does internally work just with the difference of the real time which is send through NTP or SNTP and the internal servers time.

Background

`Zend_TimeSync` is not able to change the server's time, but it will return a Zend_Date instance from which the difference to the servers time can be worked with.

Why `Zend_TimeSync` ?

So why would someone use `Zend_TimeSync` ?

Normally every server within a multiserver farm will have a service running which syncronises the own time with a timeserver. So within a standard environment it should not be necessary to use `Zend_TimeSync`. But it can become handy if there is no service available and if you don't have the right to install such a service.

Here are some example usecases, where `Zend_TimeSync` is perfect suited for:

- **Server without timeservice**

  If your application is running on a server and this server does not have any timeservice running it can be good to implement `Zend_TimeSync` within the own application.

- **Seperated database server**

  If your database is running on a seperated server and the other server is not connected with **NTP** or **SNTP** to the application server you would expect problems with data stored into the database where timestamps are used.

- **Multiple servers**

  If your application is running on more than one server and the timebase of this servers are not coupled together you can expect problems within your application when part of the application are coming from one server and others from other servers.

- **Batch processing**

  If your want to include or work with a timeservice within a batch file or within a command line application.

In all this cases `Zend_TimeSync` is a perfect solution and can be used if you are not able to run any service on your server.

What is NTP ?

The `Network Time Protocol` (**NTP**) is a protocol for synchronizing the clocks of computer systems over packet-switched, variable-latency data networks. NTP uses UDP port 123 as it's transport layer. See this wikipedia article [http://en.wikipedia.org/wiki/Network_Time_Protocol] for details about this protocol.

What is SNTP?

The `Simple Network Time Protocol` (**SNTP**) is a protocol for syncronising with clocks of computer systems over packet-switched, variable-latency data networks. SNTP uses UDP port 37 as it's transport layer. It is nearly related to the `NTP` Protocol but simpler.

Problematic usage

Be warned that when you are using `Zend_TimeSync` you will have to think about some details related to the structure of timesync and the web itself. How problems can be avoided and best practice will be described here. Read carefully before using `Zend_TimeSync`.

Decide which server to use

You have to select the timeserver which you want to use very carefully. This has several reasons which are described here:

- Distance

  The distance from the server where your application is running to the timeserver you are requesting. If your server is in europe it would make no sense to use a timeserver in tahiti. Select always a server which is not far away. This reduces the time for the request and reduced network load.

- Speed

  How long it takes to receive the request is also relevant. Try some servers to get the best result. If you are requesting a server which is never accessible you will always have a unnecessary delay.

- Splitting

  Do not use always the same server. All timeservers will lock request from servers which are flooding the server. If your application makes excessive use of timeservers you should not use a single timeserver but one of the pools described later.

So where can you find a timeserver ? Generally you can use any timeserver you know. This can be a timeserver within you LAN or any public timeserver you know. If you decide to use a public timeserver you should use a server pool. Serverpools are public addresses where you will get a random timeserver from the pool if you request the time. This way you will not have to split your requests. There are public serverpools available for different regions so you will not have any of the problems mentioned above.

Take a look at pool.ntp.org [http://www.pool.ntp.org] to get your nearest serverpool. So if your server is located within germany for example you can connect to `0.europe.pool.ntp.org` and so on.

Working with Zend_TimeSync

`Zend_TimeSync` can return the actual time from any given **NTP** or **SNTP** timeserver. It can automatically handle multiple servers and provides a simple interface.

Note

In all examples within this chapter we are using one of the available public generic timeservers. In our case **0.europe.pool.ntp.org**. For your environment it is recommended that you are using a public generic timeserver which is nearly to the location of your server. See http://www.pool.ntp.org for details.

Generic timeserver request

Requesting the time from a timeserver is quite simple. All you have to give is the timeserver from which you want to have the time.

```
$server = new Zend_TimeSync('0.pool.ntp.org');

print $server->getDate()->getIso();
```

So what is happening in the background of `Zend_TimeSync`? First the syntax of the given server is checked. So in our example '`0.pool.ntp.org`' is checked and recognised as possible correct adress for a timeserver. Then when calling `getDate()` the actual set timeserver is requested and it will return it's own time. `Zend_TimeSync` then calculates the difference to the actual time of the server running the script and returns a `Zend_Date` object with the actual, corrected time.

For details about `Zend_Date` and it's methods you can refer to Zend_Date.

Multiple timeservers

Not all timeservers are always available and will return their time. Servers will have a time where they can not be reached, for example when having a maintenance. In such cases, when the time can not be requested from the timeserver, you would get an exception.

As simple solution `Zend_TimeSync` can handle multiple timeservers and supports a automatic fallback machanism. There are two supported ways. You can either give an array of timeservers when creating the instance. Or you can add additionally timeservers afterwards with the `addServer()` method.

```
$server = new Zend_TimeSync(array('0.pool.ntp.org',
                                  '1.pool.ntp.org',
                                  '2.pool.ntp.org'));
$server->addServer('3.pool.ntp.org');

print $server->getDate()->getIso();
```

There is no limitation in the ammount of timeservers you can add. When a timeserver can not be reached `Zend_TimeSync` will fallback and try to connect to the next given timeserver.

When you give more than one timeserver, which should be your default behaviour, you should name your servers. You can either name your servers with the array key, but also with the second parameter at initiation or addition of an other timeserver.

```
$server = new Zend_TimeSync(array('generic'  => '0.pool.ntp.org',
                                  'fallback' => '1.pool.ntp.org',
                                  'reserve'  => '2.pool.ntp.org'));
$server->addServer('3.pool.ntp.org', 'additional');

print $server->getDate()->getIso();
```

Naming the timeservers gives you the ability to request a specific timeserver as we will see later in this chapter.

Protocols of timeservers

There are different types of timeservers. The most public timeservers are using **NTP** as protocol. But there are different other protocols available.

You can set the proper protocol within the address of the timeserver. Actual there are two protocols which are supported by `Zend_TimeSync`. The default protocol is **NTP**. If you are only using NTP you can ommit the protocol within the address as show in the previous examples.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org',
                                  'reserve'  => 'ntp:\\2.pool.ntp.org'));
$server->addServer('sntp:\\internal.myserver.com', 'additional');

print $server->getDate()->getIso();
```

`Zend_TimeSync` is able to handle mixed timeservers. So you are not restricted to only one protocol, but you can add any server independently from it's protocol.

Using ports for timeservers

As every protocol within the world wide web, the **NTP** and **SNTP** protocols are using standard ports. NTP uses port **123** and SNTP uses **37**.

But sometimes the used port differ from the standard one. You can define the port which has to be used for each server within the address. Just add the number of the port behind the address. If no port is defined, then `Zend_TimeSync` will use the standard port.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org:200',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));
```

```
$server->addServer('sntp:\\internal.myserver.com:399', 'additional');

print $server->getDate()->getIso();
```

Options for timeservers

Actually there is only one option within `Zend_TimeSync` which will be used internally. But you can set any self defined option you are in need for and request it.

The option **timeout** defines the number of seconds after which a connection is detected as broken when there was no response. The default value is **1**, which means that `Zend_TimeSync` will fallback to the next timeserver is the actual requested timeserver does not respond in one second.

With the `setOptions()` method, you can set any option. It accepts an array where the key is the option to set and the value is the value of that option. Any previous set option will be overwritten by the new value. If you want to know which options are set, use the `getOptions()` method. It accepts either a key which returns the given option if set or, if no key is set, it will return all set options.

```
Zend_TimeSync::setOptions(array('timeout' => 3, 'myoption' => 'timesync'));
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));
$server->addServer('sntp:\\internal.myserver.com', 'additional');

print $server->getDate()->getIso();
print_r(Zend_TimeSync::getOptions();
print "Timeout = " . Zend_TimeSync::getOptions('timeout');
```

As you can see the options for `Zend_TimeSync` are static, which means that each instance of `Zend_TimeSync` will act with the same options.

Using different timeservers

The default behaviour for requesting a time is to request it from the first given server. But sometimes it is useful to set a different timeserver from which to request the time. This can be done with the `setServer()` method. To define the used timeserver just set the alias as parameter within the method. And to get the actual used timeserver just call the `getServer()` method. It accepts an alias as parameter which defined the timeserver to be returned. If no parameter is given, the current timeserver will be returned.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));
$server->addServer('sntp:\\internal.myserver.com', 'additional');

$actual = $server->getServer();
$server = $server->setServer('additional');
```

Informations from timeservers

Timeservers offer not only the time itself but also additionally informations. You can get these informations with the `getInfo()` method.

```
$server = new Zend_TimeSync(array('generic'  => 'ntp:\\0.pool.ntp.org',
                                  'fallback' => 'ntp:\\1.pool.ntp.org'));

print_r ($server->getInfo());
```

The returned informations differ with the used protocols and they can also differ with the used servers.

Taking care of exceptions

Exceptions are collected for all timeserver and will be returned as array. So you are able to iterate through all throwed exceptions like shown in the following example:

```
$serverlist = array(
        // invalid servers
        'invalid_a'  => 'ntp://a.foo.bar.org',
        'invalid_b'  => 'sntp://b.foo.bar.org',
);

$server = new Zend_TimeSync($serverlist);

try {
    $result = $server->getDate();
    echo $result->getIso();
} catch (Zend_TimeSync_Exception $e) {

    $exceptions = $e->get();

    foreach ($exceptions as $key => $myException) {
        echo $myException->getMessage();
        echo '<br />';
    }
}
```

Chapter 46. Zend_Translate

Introduction

Zend_Translate is the Zend Framework's solution for multilingual applications.

In multilingual applications, the content must be translated into several languages and display content depending on the user's language. PHP offers already several ways to handle such problems, however the PHP solution has some problems:

- **Inconsistent API:** There is no single API for the different source formats. The usage of gettext for example is very complicated.
- **PHP supports only gettext and native array:** PHP itself offers only support for array or gettext. All other source formats have to be coded manually, because there is no native support.
- **No detection of the default language:** The default language of the user cannot be detected without deeper knowledge of the backgrounds for the different web browsers.
- **Gettext is not thread-safe:** PHP's gettext library is not thread safe, and it should not be used in a multithreaded environment. This is due to problems with gettext itself, not PHP, but it is an existing problem.

Zend_Translate does not have the above problems. This is why we recommend using Zend_Translate instead of PHP's native functions. The benefits of Zend_Translate are:

- **Supports multiple source formats:** Zend_Translate supports several source formats, including those supported by PHP, and other formats including TMX and CSV files.
- **Thread-safe gettext:** The gettext reader of Zend_Translate is thread-safe. There are no problems using it in multi-threaded environments.
- **Easy and generic API:** The API of Zend_Translate is very simple and requires only a handful of functions. So it's easy to learn and easy to maintain. All source formats are handled the same way, so if the format of your source files change from Gettext to TMX, you only need to change one line of code to specify the storage adapter.
- **Detection of the user's standard language:** The preferred language of the user accessing the site can be detected and used by Zend_Translate.
- **Automatic source detection:** Zend_Translate is capable of detecting and integrating multiple source files and additionally detect the locale to be used depending on directory or filenames.

Starting multi-lingual

So let's get started with multi-lingual business. What we want to do is translate our string output so the view produces the translated output. Otherwise we would have to write one view for each language, and no one would like to do this. Generally, multi-lingual sites are very simple in their design. There are only four steps you would have to do:

1. Decide which adapter you want to use;
2. Create your view and integrate Zend_Translate in your code;

3. Create the source file from your code;

4. Translate your source file to the desired language.

The following sections guide you through all four steps. Read through the next few pages to create your own multi-lingual web application.

Adapters for Zend_Translate

Zend_Translate can handle different adapters for translation. Each adapter has its own advantages and disadvantages. Below is a comprehensive list of all supported adapters for translation source files.

Table 46.1. Adapters for Zend_Translate

Adapter	Description	Usage
Array	Use php arrays	Small pages; simplest usage; only for programmers
Csv	Use comma seperated (*.csv/*.txt) files	Simple text file format; fast; possible problems with unicode characters
Gettext	Use binary gettext (*.mo) files	GNU standard for linux; thread-safe; needs tools for translation
Ini	Use simple ini (*.ini) files	Simple text file format; fast; possible problems with unicode characters
Tbx	Use termbase exchange (*.tbx/*.xml) files	Industry standard for inter application terminology strings; XML format
Tmx	Use tmx (*.tmx/*.xml) files	Industry standard for inter application translation; XML format; human readable
Qt	Use qt linguist (*.ts) files	Cross platform application framework; XML format; human readable
Xliff	Use xliff (*.xliff/*.xml) files	A simpler format as TMX but related to it; XML format; human readable
XmlTm	Use xmltm (*.xml) files	Industry standard for XML document translation memory; XML format; human readable
Others	*.sql	Different other adapters may be implemented in the future

How to decide which translation adapter to use

You should decide which Adapter you want to use for Zend_Translate. Frequently, external criteria such as a project requirement or a customer requirement determines this for you, but if you are in the position to do this yourself, the following hints may simplify your decision.

Note

When deciding your adapter you should also be aware of the used encoding. Even if the Zend Framework declares UTF-8 as default encoding you will sometimes be in the need of other encoding. `Zend_Translate` will not change any encoding which is defined within the source file which means that if your Gettext source is build upon ISO-8859-1 it will also return strings in this encoding without converting them. There is only one restriction:

When you use a xml based source format like TMX or XLIFF you must define the encoding within the xml files header because xml files without defined encoding will be treated as UTF-8 by any xml parser by default. You should also be aware that actually the encoding of xml files is limited to the encodings supported by PHP which are UTF-8, ISO-8859-1 and US-ASCII.

Zend_Translate_Adapter_Array

The Array Adapter is the Adapter which is simplest to use for programmers. But when you have numerous translation strings or many languages you should think about another Adapter. For example, if you have 5000 translation strings, the Array Adapter is possibly not the best choice for you.

You should only use this Adapter for small sites with a handful of languages, and if you or your programmer team creates the translations yourselves.

Zend_Translate_Adapter_Csv

The Csv Adapter is the Adapter which is simplest to use for customers. CSV files are readable by standard text editors, but text editors often do not support utf8 character sets.

You should only use this Adapter if your customer wants to do translations himself.

Note

Beware that the Csv Adapter has problems when your Csv files are encoded differently than the locale setting of your environment. This is due to a Bug of PHP itself which will not be fixed before PHP 6.0 (http://bugs.php.net/bug.php?id=38471). So you should be aware that the Csv Adapter due to PHP restrictions is not locale aware.

Zend_Translate_Adapter_Gettext

The Gettext Adapter is the Adapter which is used most frequently. Gettext is a translation source format which was introduced by GNU, and is now used worldwide. It is not human readable, but there are several freeware tools (for instance, POEdit [http://sourceforge.net/projects/poedit/]), which are very helpful. The Zend_Translate Gettext Adapter is not implemented using PHP's gettext extension. You can use the Gettext Adapter even if you do not have the PHP gettext extension installed. Also the Adapter is thread-safe and the PHP gettext extension is currently not thread-safe.

Most people will use this adapter. With the available tools, professional translation is very simple. But gettext data are is stored in a machine-readable format, which is not readable without tools.

Zend_Translate_Adapter_Ini

The Ini Adapter is a very simple Adapter which can even be used directly by customers. INI files are readable by standard text editors, but text editors often do not support utf8 character sets.

You should only use this Adapter when your customer wants to do translations himself. Do not use this adapter as generic translation source.

Zend_Translate_Adapter_Tbx

The Tbx Adapter is an Adapter which will be used by customers which already use the TBX format for their internal translation system. Tbx is no standard translation format but more a collection of already translated and pre translated source strings. When you use this adapter you have to be sure that all your needed source string are translated. TBX is a XML file based format and a completly new format. XML files are human-readable, but the parsing is not as fast as with gettext files.

This adapter is perfect for companies when pre translated source files already exist. The files are human readable and system-independent.

Zend_Translate_Adapter_Tmx

The Tmx Adapter is the Adapter which will be used by most customers which have multiple systems which use the same translation source, or when the translation source must be system-independent. TMX is a XML file based format, which is announced to be the next industry standard. XML files are human-readable, but the parsing is not as fast as with gettext files.

Most medium to large companies use this adapter. The files are human readable and system-independent.

Zend_Translate_Adapter_Qt

The Qt Adapter is for all customers which have TS files as their translation source which are made by Qt-Linguist. QT is a XML file based format. XML files are human-readable, but the parsing is not as fast as with gettext files.

Several big players have build software upon the QT framework. The files are human readable and system-independent.

Zend_Translate_Adapter_Xliff

The Xliff Adapter is the Adapter which will be used by most customers which want to have XML files but do not have tools for TMX. XLIFF is a XML file based format, which is related to TMX but simpler as it does not support all possibilities of it. XML files are human-readable, but the parsing is not as fast as with gettext files.

Most medium companies use this adapter. The files are human readable and system-independent.

Zend_Translate_Adapter_XmlTm

The XmlTm Adapter is the Adapter which will be used by customers which do their layout themself. XmlTm is a format which allows the complete html source to be included in the translation source, so the translation is coupled with the layout. XLIFF is a XML file based format, which is related to XLIFF but its not as simple to read.

This adapter sould only be used when source files already exist. The files are human readable and system-independent.

Integrate self written Adapters

Zend_Translate allows you to integrate and use self written Adapter classes. They can be used like the standard Adapter classes which are already included within Zend_Translate.

Any adapter class you want to use with Zend_Translate must be a subclass of Zend_Translate_Adapter. Zend_Translate_Adapter is an abstract class which already defines all what is needed for translation. What has to be done by you, is the definition of the reader for translation datas.

The usage of the prefix "Zend" should be limited to the Zend_Framework. If you extend Zend_Translate with your own adapter, you should name it like "Company_Translate_Adapter_MyFormat". The following code shows an example of how a self written adapter class could be implemented:

```
try {
    $translate = new Zend_Translate('Company_Translate_Adapter_MyFormat',
                                   '/path/to/translate.xx',
                                   'en',
                                   array('myoption' => 'myvalue'));
} catch (Exception $e) {
    // File not found, no adapter class...
    // General failure
}
```

Speedup all Adapters

`Zend_Translate` allows you use internally `Zend_Cache` to fasten the loading of translation sources. This comes very handy if you use many translation sources or extensive source formats like XML based files.

To use caching you will just have to give a cache object to the `Zend_Translate::setCache()` method. It takes a instance of `Zend_Cache` as only parameter. Also if you use any adapter direct you can use the `setCache()` method. For convenience there is the static method `Zend_Translate::getCache()`.

```
$cache = Zend_Cache::factory('Page',
                             'File',
                             $frontendOptions,
                             $backendOptions);
Zend_Translate::setCache($cache);
$translate = new Zend_Translate('gettext',
                                '/path/to/translate.mo',
                                'en');
```

Note

You must set the cache **before** you use or initiate any adapter or instance of `Zend_Translate`. Otherwise your translation source will not be cached until you add a new source with the `addTranslation()` method.

Using Translation Adapters

The next step is to use the adapter within your code.

Example 46.1. Example of single-language PHP code

```
print "Example\n";
print "=======\n";
print "Here is line one\n";
print "Today is the " . date("d.m.Y") . "\n";
print "\n";
print "Fix language here is line two\n";
```

The example above shows some output with no support for translation. You probably write your code in your native language. Generally you need to translate not only the output, but also error messages and log messages.

The next step is to include Zend Translate in your existing code. Of course it is much easier if you are writing your code using Zend_Translate instead of changing your code afterwards.

Example 46.2. Example of multi-lingual PHP code

```
$translate = new Zend_Translate('gettext', '/my/path/source-de.mo', 'de');
$translate->addTranslation('//my/path/fr-source.mo', 'fr');

print $translate->_("Example")."\n";
print "=======\n";
print $translate->_("Here is line one")."\n";
printf($translate->_("Today is the %1\$s") . "\n", date("d.m.Y"));
print "\n";

$translate->setLocale('fr');
print $translate->_("Fix language here is line two") . "\n";
```

Now let's get a deeper look into what has been done and how to integrate Zend_Translate into your code.

Create a new Translation object and define the base adapter:

```
$translate = new Zend_Translate('gettext', '/my/path/source-de.mo', 'de');
```

In this example we decided the **Gettext Adapter**. We place our file **source-de.mo** into the directory **/my/path**. The gettext file will have German translation included. And we also added another language source for French.

The next step is to wrap all strings which are to be translated. The simplest approach is to have only simple strings or sentences like this:

```
print $translate->_("Example")."\n";
print "=======\n";
print $translate->_("Here is line one")."\n";
```

Some strings do not needed to be translated. The seperating line is always a seperating line, even in other languages.

Having data values integrated into a translation string is also supported through the use of embedded parameters.

```
printf($translate->_("Today is the %1\$s") . "\n", date("d.m.Y"));
```

Instead of `print()`, use the `printf()` function and replace all parameters with `%1\$s` parts. The first is `%1\$s`, the second `%2\$s`, and so on. This way a translation can be done without knowing the exact value. In our example, the date is always the actual day, but the string can be translated without the knowledge of the actual day.

Each string is identified in the translation storage by a message id. You can use message id's instead of strings in your code, like this:

```
print $translate->_(1)."\n";
print "=======\n";
print $translate->_(2)."\n";
```

But doing this has several disadvantages:

You can not see what your code should output just by viewing your code.

Also you will get problems if some strings are not translated. You always must imagine how translation works. First Zend_Translate looks if the set language has a translation for the given message id or string. If no translation string has been found it refers to the next lower language as defined within Zend_Locale. So "**de_AT**" becomes "**de**" only. If there is no translation found for "**de**" either, then the original message is returned. This way you always have an output, in case the message translation does not exist in your message storage. Zend_Translate never throws an error or exception when translating strings.

Translation Source Structures

Your next step is to create the translation sources for the several languages to which you translate. Every adapter is created its own way as described here. But there are some general features that are relevant for all adapters.

You should know where to store your translation source files. With Zend_Translate you are not bound to any restriction. The following structures are preferable:

- Single structured source

```
/application
/languages
```

```
  lang.en
  lang.de
/library
```

Positive: All source files for every languages can be found in one directory. No splitting of related files.

- Language structured source

```
/application
/languages
  /en
    lang.en
    other.en
  /de
    lang.de
    other.de
/library
```

Positive: Every language is based in one directory. Easy translation as only one directory has to be translated by a language team. Also the usage of multiple files is transparent.

- Application structured source

```
/application
  /languages
    lang.en
    lang.de
    other.en
    other.de
```

Positive: All source files for every languages can be found in one directory. No splitting of related files.

Negative: Having multiple files for the same language is problematic.

- Gettext structured source

```
/languages
  /de
    /LC_MESSAGES
      lang.mo
      other.mo
  /en
    /LC_MESSAGES
      lang.mo
      other.mo
```

Positive: Old gettext sources can be used without changing structure.

Negative: Having sub-sub directories may be confusing for people who have not used gettext before.

- File structured source

  ```
  /application
    /models
      mymodel.php
      mymodel.de
      mymodel.en
    /views
    /controllers
      mycontroller.de
  /document_root
    /images
    /styles
    .htaccess
    index.php
    index.de
  /library
    /Zend
  ```

  Positive: Every file is related to its own translation source.

  Negative: Multiple small translation source files make it harder to translate. Also every file has to be added as translation source.

Single structured and language structured source files are most usable for Zend_Translate.

So now, that we know which structure we want to have, we should create our translation source files.

Creating array source files

Array source files are just arrays. But you have to define them manually because there is no tool for this. But because they are so simple, it's the fastest way to look up messages if your code works as expected. It's generally the best adapter to get started with translation business.

```
$english = array('message1' => 'message1',
                 'message2' => 'message2',
                 'message3' => 'message3');
$german = array('message1' => 'Nachricht1',
                'message2' => 'Nachricht2',
                'message3' => 'Nachricht3');

$translate = new Zend_Translate('array', $english, 'en');
$translate->addTranslation($deutsch, 'de');
```

Since Release 1.5 it is also supported to have arrays included within a external file. You just have to give the filename and `Zend_Translate` will automatically include it and look for the array. See the following example for details:

```
// myarray.php
return array(
    'message1' => 'Nachricht1',
    'message2' => 'Nachricht2',
    'message3' => 'Nachricht3');

// controller
$translate = new Zend_Translate('array', 'path/to/myarray.php', 'de');
```

Note

Files which do not return an array will fail to be included. Also any output within this file will be ignored and suppressed.

Creating Gettext Source Files

Gettext source files are created by GNU's gettext library. There are several free tools available that can parse your code files and create the needed gettext source files. These files have the ending ***.mo** and they are binary files. One freeware tool for creating the files is poEdit [http://sourceforge.net/projects/poedit/]. This tool also supports you for the translation process itself.

```
// We expect that we have created the mo files and translated them
$translate = new Zend_Translate('gettext', 'path/to/english.mo', 'en');
$translate->addTranslation('path/to/german.mo', 'de');
```

As you can see the adapters are used exactly the same way, with only just one small difference. Change 'array' to 'gettext'. All other usages are exactly the same as with all other adapters. With the gettext adapter you no longer have to be aware of gettext's standard directory structure, bindtextdomain and textdomain. Just give the path and filename to the adapter.

Note

You should always use UTF-8 as source encoding. Otherwise you will have problems if you are using two different source encodings. For example, if one of your source files is encoded with ISO-8815-11 and another file is encoded with CP815. You can set only one encoding for your source file, so one of your languages probably will not display correctly.

UTF-8 is a portable format which supports all languages. If you use UTF-8 encoding for all languages, you eliminate the problem of incompatible encodings.

Many gettext editors add adapter informations as empty translation string. This is the reason why empty string are not translated when using the gettext adapter. Instead they are erased from the translation table and provided by the `getAdapterInfo()` method. It will return the adapter informations for all added gettext files as array where the filename is used as key.

```
// How to get the adapter informations
$translate = new Zend_Translate('gettext', 'path/to/english.mo', 'en');
```

```
print_r $translate->getAdapterInfo();
```

Creating TMX Source Files

TMX source files are a new industry standard. They have the advantage of being XML files and so they are readable by every editor and of course they are human-readable. You can either create TMX files manually with a text editor, or you can use a tool. But most tools currently available for developing TMX source files are not freeware.

Example 46.3. Example TMX file

```
<?xml version="1.0" ?>
<!DOCTYPE tmx SYSTEM "tmx14.dtd">
<tmx version="1.4">
 <header creationtoolversion="1.0.0" datatype="winres" segtype="sentence"
         adminlang="en-us" srclang="de-at" o-tmf="abc"
         creationtool="XYZTool" >
 </header>
 <body>
  <tu tuid='message1'>
   <tuv xml:lang="de"><seg>Nachricht1</seg></tuv>
   <tuv xml:lang="en"><seg>message1</seg></tuv>
  </tu>
  <tu tuid='message2'>
   <tuv xml:lang="en"><seg>message2</seg></tuv>
   <tuv xml:lang="de"><seg>Nachricht2</seg></tuv>
  </tu>
```

```
$translate = new Zend_Translate('tmx', 'path/to/mytranslation.tmx', 'en');
// TMX can have several languages within one TMX file.
```

TMX files can have several languages within the same file. All other included languages are added automatically, so you do not have to call `addLanguage()`.

If you want to have only specified languages from the source translated you can set the option `defined_language` to `true`. With this option you can add the wished languages explicit with `addLanguage()`. The default value for this option is to add all languages.

Creating CSV Source Files

CSV source files are small and human readable. If your customers want to translate their own, you will probably use the CSV adapter.

Example 46.4. Example CSV file

```
#Example csv file
message1;Nachricht1
message2;Nachricht2
```

```
$translate = new Zend_Translate('csv', 'path/to/mytranslation.csv', 'de');
$translate->addTranslation('path/to/other.csv', 'fr');
```

There are three different options for the CSV adapter. You can set `'delimiter'`, `'limit'` and `'enclosure'`.

The default delimiter for CSV string is the `';'` sign. But it has not to be that sign. With the option '`delimiter`' you can decide to use another delimiter sign.

The default limit for a line within a CSV file is '`0`'. This means that the end of a CSV line is searched automatically. If you set the '`limit`' option to any value, then the CSV file will be read faster, but any line exceeding this limit will be truncated.

The default enclosure to use for CSV files is '`"`'. You can set a different one with the option '`enclosure`'.

Example 46.5. Example CSV file two

```
# Example csv file
# original 'message,1'
"message,1",Nachricht1
# translation 'Nachricht,2'
message2,"Nachricht,2"
# original 'message3,'
"message3,",Nachricht3
```

```
$translate = new Zend_Translate('csv',
                                'path/to/mytranslation.csv',
                                'de',
                                array('delimiter' => ','));
$translate->addTranslation('path/to/other.csv', 'fr');
```

Creating INI Source Files

INI source files are human readable but normally not very small as they also include other data beside translations. If you have data which shall be editable by your customers you could also use the INI adapter for this purpose.

Example 46.6. Example INI file

```
[Test]
;TestPage Comment
Message_1="Nachricht 1 (de)"
Message_2="Nachricht 2 (de)"
Message_3="Nachricht :3 (de)"
```

```
$translate = new Zend_Translate('ini', 'path/to/mytranslation.ini', 'de');
$translate->addTranslation('path/to/other.ini', 'it');
```

INI files have several restrictions. If a value in the ini file contains any non-alphanumeric characters it needs to be enclosed in double-quotes ("). There are also reserved words which must not be used as keys for ini files. These include: null, yes, no, true, and false. Values null, no and false results in "", yes and true results in "1". Characters {}|&~![()" must not be used anywhere in the key and have a special meaning in the value. Do not use them as you will have unexpected behaviour.

Options for adapters

Options can be used with all adapters. Of course the options are different for all adapters. You can set options when you create the adapter. Actually there is one option which is available to all adapters. '`clear`' decides if translation data should be added to existing one or not. Standard behaviour is to add new translation data to existing one. But the translation data is only cleared for the selected language. So all other languages will not be touched.

You can set options temporary when using `addTranslation($data, $locale, array $options = array())`. as third and optional parameter. And you can use the `setOptions()` function to set the options fix.

Example 46.7. Using translation options

```
// define ':' as separator for the translation source files
$options = array('separator' => ':');
$translate = new Zend_Translate('csv',
                                'path/to/mytranslation.csv',
                                'de',
                                $options);

...

// clear the defined language and use new translation data
$options = array('clear' => true);
$translate->addTranslation('path/to/new.csv', 'fr', $options);
```

Here you can find all available options for the different adapters with a description of their usage:

Table 46.2. Options for Translation Adapters

Adapter	Option	Standard value	Description
all	clear	**false**	If set to true, the already read translations will be cleared. This can be used instead of creating a new instance when reading new translation data
all	ignore	**.**	All directories and files beginning with this prefix will be ignored when searching for files. This value defaults to **'.'** which leads to the behavior that all hidden files will be ignored. Setting this value to 'tmp' would mean that directories and files like 'tmpImages' and 'tmpFiles' would be ignored and also all subsequent directories
all	scan	**null**	If set to null, no scanning of the directory structure will be done. If set to Zend_Translate::LOCALE_DIRECTORY the locale will be detected within the directory. It set to Zend_Translate::LOCALE_FILENAME the locale will be detected within the filename. See the section called “Automatic source detection” for details
Csv	delimiter	**;**	Defines which sign is used as delimiter for seperating source and translation
Csv	length	**0**	Defines the maximum length of a csv line. When set to 0 it will be detected automatically
Csv	enclosure	**"**	Defines the enclosure character to be used. Defaults to a doublequote

When you want to have self defined options, you are also able to use them within all adapters. The `setOptions()` method can be used to define your option. `setOptions()` needs an array with the options you want to set. If an given option exists it will be signed over. You can define as much options as needed as they will not be checked by the adapter. Just get sure that you do not sign over any existing option which is used by an adapter.

To return the set option you can use the `getOptions()` method. When `getOptions()` is called without an parameter it will return all set options. When the optional parameter is given you will only get the particular option returned.

Handling languages

When working with different languages there are a few methods which will be useful.

The `getLocale()` method can be used to get the actual set language. It can eigther hold an instance of `Zend_Locale` or the identifier of a locale.

The `setLocale()` method sets a new standard language for translation. This prevents the need of setting the optional language parameter more than once to the `translate()` method. If the given language does not exist, or no translation data is available for the language, `setLocale()` tries to downgrade to the language without the region if any was given. A language of `en_US` would be downgraded to `en`. When also the downgraded language can not be found an exception will be thrown.

The `isAvailable()` method checks if a given language is already available. It returns `true` if data for the given language exist.

And finally the `getList()` method can be used to get all actual set languages for an adapter returned as array.

Example 46.8. Handling languages with adapters

```
...
// returns the actual set language
$actual = $translate->getLocale();

...
// you can use the optional parameter while translating
echo $translate->_("my_text", "fr");
// or set a new standard language
$translate->setLocale("fr");
echo $translate->_("my_text");
// refer to the base language... fr_CH will be downgraded to fr and be
// used
$translate->setLocale("fr_CH");
echo $translate->_("my_text");

...
// check if this language exist
if ($translate->isAvailable("fr")) {
    // language exists
}
```

Automatically handling of languages

Note that as long as you only add new translation sources with the `addTranslation()` method `Zend_Translate` will automatically set the best fitting language for your environment. So normally you will not need to call `setLocale()`.

The algorithmus will search for the best fitting locale depending on the users browser and your environment. See the following example for details:

Example 46.9. How automatically language detection works

```
// Let's expect the browser returns this language settings
HTTP_ACCEPT_LANGUAGE = "de_AT=1;fr=1;en_US=0.8";

// Example 1:
$translate = new Zend_Translate("gettext", "\my_it.mo", "it_IT");
$translate->addTranslation("\my_es.mo","es_UG");
// no fitting language found, return the messageid

// Example 2:
$translate = new Zend_Translate("gettext", "\my_en.mo", "en_US");
$translate->addTranslation("\my_it.mo","it_IT");
// best found fitting language is "en_US"

// Example 3:
$translate = new Zend_Translate("gettext", "\my_it.mo", "it_IT");
$translate->addTranslation("\my_de.mo","de");
// best found fitting language is "de" because "de_AT" will be
// degraded to "de"

// Example 4:
$translate = new Zend_Translate("gettext", "\my_it.mo", "it_IT");
$translate->addTranslation("\my_ru.mo","ru");
$translate->setLocale("it_IT");
$translate->addTranslation("\my_de.mo","de");
// returns "it_IT" as translation source
```

After setting a language manually with the `setLocale()` method the automatically detection will be switched off and overridden.

If you want to use the automatic again, you can set the language **auto** with `setLocale()` which will reactivate the automatically detection for `Zend_Translate`.

Since Zend Framework 1.6 `Zend_Translate` recognises also an application wide locale. You can simply set a `Zend_Locale` instance to the registry like shown below. With this notation you can forget about setting the locale manually with each instance when you want to use the same locale multiple times.

```
// in your bootstrap file
$locale = new Zend_Locale('de_AT');
Zend_Registry::set('Zend_Locale', $locale);

// somewhere in your application
$translate = new Zend_Translate("gettext", "\my_de.mo");
$translate->getLocale();
```

Automatic source detection

Zend_Translate can detect translation sources automatically. So you don't have to declare each source file manually. You can let Zend_Translate do this job and scan the complete directory structure for source files.

Note

Automatic source detection is available since Zend Framework version 1.5 .

The usage is quite the same as initiating a single translation source with one difference. You must give a directory which has to be scanned instead a file.

Example 46.10. Scanning a directory structure for sources

```
// expect we have the following structure
//  /language
//  /language/login/login.tmx
//  /language/logout/logout.tmx
//  /language/error/loginerror.tmx
//  /language/error/logouterror.tmx

$translate = new Zend_Translate('tmx', '/language');
```

So Zend_Translate does not only search the given directory, but also all subdirectories for translation source files. This makes the usage quite simple. But Zend_Translate will ignore all files which are not sources or which produce failures while reading the translation data. So you have to make sure that all of your translation sources are correct and readable because you will not get any failure if a file is bogus or can not be read.

Note

Depending on how deep your directory structure is and how much files are within this structure it can take a long time for Zend_Translate to complete.

In our example we have used the TMX format which includes the language to be used within the source. But many of the other source formats are not able to include the language within the file. Even this sources can be used with automatic scanning if you do some pre-requisits as described below:

Language through naming directories

One way to include automatic language detection is to name the directories related to the language which is used for the sources within this directory. This is the easiest way and is used for example within standard gettext implementations.

Zend_Translate needs the 'scan' option to know that it should search the names of all directories for languages. See the following example for details:

Example 46.11. Directory scanning for languages

```
// expect we have the following structure
//  /language
//  /language/de/login/login.mo
//  /language/de/error/loginerror.mo
//  /language/en/login/login.mo
//  /language/en/error/loginerror.mo

$translate = new Zend_Translate('gettext',
                                '/language',
                                null,
                                array('scan' =>
                                      Zend_Translate::LOCALE_DIRECTORY));
```

Note

This works only for adapters which do not include the language within the source file. Using this option for example with TMX will be ignored. Also language definitions within the filename will be ignored when using this option.

Note

You should be aware if you have several subdirectories under the same structure. Expect we have a structure like `/language/module/de/en/file.mo`. The path contains in this case multiple strings which would be detected as locale. It could be eigther `de` or `en`. As the behaviour is, in this case, not declared it is recommended that you use file detection in such situations.

Language through filenames

Another way to detect the langage automatically is to use special filenames. You can either name the complete file or parts of a file with the used language. To use this way of detection you will have to set the 'scan' option at initiation. There are several ways of naming the sourcefiles which are described below:

Example 46.12. Filename scanning for languages

```
// expect we have the following structure
//  /language
//  /language/login/login_en.mo
//  /language/login/login_de.mo
//  /language/error/loginerror_en.mo
//  /language/error/loginerror_de.mo

$translate = new Zend_Translate('gettext',
                                '/language',
                                null,
                                array('scan' =>
                                      Zend_Translate::LOCALE_FILENAME));
```

Complete Filename

Having the whole file named after the language is the simplest way but only usable if you have only one file per directory.

```
/languages
  en.mo
  de.mo
  es.mo
```

Extension of the file

Another very simple way if to use the extension of the file for the language detection. But this may be confusing because you will no longer know which file extension the file originally was.

```
/languages
  view.en
  view.de
  view.es
```

Filename tokens

Zend_Translate is also captable of detecting the language if it is included within the filename. But if you use this way you will have to seperate the language with a token. There are three supported tokens which can be used: A point '.', a underline '_', or a hyphen '-'.

```
/languages
  view_en.mo  -> detects english
  view_de.mo  -> detects german
  view_it.mo  -> detects italian
```

The first found token which can be detected as locale will be used. See the following example for details.

```
/languages
  view_en_de.mo  -> detects english
  view_en_es.mo  -> detects english and overwrites the first file
  view_it_it.mo  -> detects italian
```

All three tokens are used to detect the locale. The first one is the point '.', the second is the underline '_' and the third the hyphen '-'. If you have several tokens within the filename the first found depending on the order of the tokens will be used. See the following example for details.

```
/languages
  view_en-it.mo  -> detects english because '_' will be used before '-'
  view-en_it.mo  -> detects italian because '_' will be used before '-'
  view_en.it.mo  -> detects italian because '.' will be used before '_'
```

Checking for translations

Normally text will be translated without any computations. But sometimes it is necessary to know if a text is translated or not within the source. Therefor the `isTranslated()` method can be used.

`isTranslated($messageId, $original = false, $locale = null)` takes as first parameter the text from which you want to know if it can be translated. And as optional third parameter the locale for which you want to know the translation. The optional second parameter declares if translation is fixed to the declared language or a lower set of translations can be used. If you have a text which can be translated by 'en' but not for 'en_US' you will normally get the translation returned, but by setting `$original` to true, the `isTranslated()` method will return false in such cases.

Example 46.13. Checking if a text is translatable

```
$english = array('message1' => 'Nachricht 1',
                 'message2' => 'Nachricht 2',
                 'message3' => 'Nachricht 3');
$translate = new Zend_Translate('array', $english, 'de_AT');

if ($translate->isTranslated('message1')) {
    print "'message1' can be translated";
}
if (!($translate->isTranslated('message1', true, 'de'))) {
    print "'message1' can not be translated in 'de' as it's only " .
    "available in 'de_AT'";
}
if ($translate->isTranslated('message1', false, 'de')) {
    print "'message1' can be translated in 'de_AT' falls back to 'de'";
}
```

Access to the source data

Of course sometimes it is useful to have access to the translation source data. Therefor two functions exist.

The `getMessageIds($locale = null)` method returns all known message ids as array.

And the `getMessages($locale = null)` method returns the complete translation source as array. The message id is used as key and the translation data as value.

Both methods accept an optional parameter `$locale` which, when set, returns the translation data for the specified language. If this parameter is not given, the actual set language will be used. Keep in mind that normally all translations should be available in all languages. Which means that in a normal situation you will not have to set this parameter.

Additionally the `getMessages()` method is able to return the complete translation dictionary with the pseudo-locale 'all'. This will return all available translation data for each added locale.

Note

Attention: The returned array can be **very big**, depending on the count of added locales and the amount of translation data.

Example 46.14. Handling languages with adapters

```
...
// returns all known message ids
$messageids = $translate->getMessageIds();
print_r($messageids);

...
// or just for the specified language
$messageids = $translate->getMessageIds('en_US');
print_r($messageids);

...
// returns all the complete translation data
$source = $translate->getMessages();
print_r($source);
```

Chapter 47. Zend_Uri

Zend_Uri

Overview

`Zend_Uri` is a component that aids in manipulating and validating Uniform Resource Identifiers [http://www.w3.org/Addressing/] (URIs). `Zend_Uri` exists primarily to service other components such as `Zend_Http_Client` 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` class provides a factory that returns a subclass of itself which specializes in each scheme. The subclass will be named `Zend_Uri_<scheme>`, where `<scheme>` is the scheme lowercased with the first letter capitalized. An exception to this rule is HTTPS, which is also handled by `Zend_Uri_Http`.

Creating a New URI

`Zend_Uri` will build a new URI from scratch if only a scheme is passed to `Zend_Uri::factory()`.

Example 47.1. Creating a New URI with `Zend_Uri::factory()`

```
// To create a new URI from scratch, pass only the scheme.
$uri = Zend_Uri::factory('http');

// $uri instanceof Zend_Uri_Http
```

To create a new URI from scratch, pass only the scheme to `Zend_Uri::factory()`[1]. If an unsupported scheme is passed, a `Zend_Uri_Exception` will be thrown.

If the scheme or URI passed is supported, `Zend_Uri::factory()` will return a subclass of itself that specializes in the scheme to be created.

Manipulating an Existing URI

To manipulate an existing URI, pass the entire URI to `Zend_Uri::factory()`.

Example 47.2. Manipulating an Existing URI with `Zend_Uri::factory()`

```
// To manipulate an existing URI, pass it in.
$uri = Zend_Uri::factory('http://www.zend.com');

// $uri instanceof Zend_Uri_Http
```

[1]At the time of writing, Zend_Uri only supports the HTTP and HTTPS schemes.

The URI will be parsed and validated. If it is found to be invalid, a `Zend_Uri_Exception` will be thrown immediately. Otherwise, `Zend_Uri::factory()` will return a subclass of itself that specializes in the scheme to be manipulated.

URI Validation

The `Zend_Uri::check()` function can be used if only validation of an existing URI is needed.

Example 47.3. URI Validation with `Zend_Uri::check()`

```
// Validate whether a given URI is well formed
$valid = Zend_Uri::check('http://uri.in.question');

// $valid is TRUE for a valid URI, or FALSE otherwise.
```

`Zend_Uri::check()` returns a boolean, which is more convenient than using `Zend_Uri::factory()` and catching the exception.

Common Instance Methods

Every instance of a `Zend_Uri` subclass (e.g. `Zend_Uri_Http`) has 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://www.zend.com` is `http`.

Example 47.4. Getting the Scheme from a `Zend_Uri_*` Object

```
$uri = Zend_Uri::factory('http://www.zend.com');

$scheme = $uri->getScheme();  // "http"
```

The `getScheme()` instance method returns only the scheme part of the URI object.

Getting the Entire URI

Example 47.5. Getting the Entire URI from a `Zend_Uri_*` Object

```
$uri = Zend_Uri::factory('http://www.zend.com');

echo $uri->getUri();  // "http://www.zend.com"
```

The `getUri()` method returns the string representation of the entire URI.

Validating the URI

`Zend_Uri::factory()` will always validate any URI passed to it and will not instantiate a new `Zend_Uri` subclass if the given URI is found to be invalid. However, after the `Zend_Uri` subclass is instantiated for a new URI or a valid existing one, it is possible that the URI can then later become invalid after it is manipulated.

Example 47.6. Validating a `Zend_Uri_*` Object

```
$uri = Zend_Uri::factory('http://www.zend.com');

$isValid = $uri->valid();  // TRUE
```

The `valid()` instance method provides a means to check that the URI object is still valid.

Chapter 48. Zend_Validate

Introduction

The Zend_Validate 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 usernames meet 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_Validate_Interface`, which defines two methods, `isValid()` and `getMessages()`. The `isValid()` method performs validation upon the provided value, returning `true` if and only if the value passes against the validation criteria.

If `isValid()` returns `false`, the `getMessages()` returns an array of messages explaining the reason(s) for validation failure. The array keys are short strings that identify the reasons for validation failure, and the array values are the corresponding human-readable string messages. The keys and values are class-dependent; each validation class defines its own set of validation failure messages and the unique keys that identify them. Each class also has a `const` definition that matches each identifier for a validation failure cause.

Note

The `getMessages()` methods return validation failure information only for the most recent `isValid()` call. Each call to `isValid()` clears any messages and errors caused by a previous `isValid()` call, because it's likely that each call to `isValid()` is made for a different input value.

The following example illustrates validation of an e-mail address:

```
$validator = new Zend_Validate_EmailAddress();

if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $messageId => $message) {
        echo "Validation failure '$messageId': $message\n";
    }
}
```

Customizing messages

Validate 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_Validate_LessThan`. The `getMessageVariables()` method returns an array of variable tokens supported by the validator.

The second optional argument is a string that identifies the validation failure message template to be set, which is useful when a validation class defines more than one cause for failure. If you omit the second argument, `setMessage()` assumes the message you specify should be used for the first message template declared in the validation class. Many validation classes only have one error message template defined, so there is no need to specify which message template you are changing.

```
$validator = new Zend_Validate_StringLength(8);

$validator->setMessage(
    'The string \'%value%\' is too short; it must be at least %min% ' .
    'characters',
    Zend_Validate_StringLength::TOO_SHORT);

if (!$validator->isValid('word')) {
    $messages = $validator->getMessages();
    echo current($messages);

    // "The string 'word' is too short; it must be at least 8 characters"
}
```

You can set multiple messages using the `setMessages()` method. Its argument is an array containing key/message pairs.

```
$validator = new Zend_Validate_StringLength(8, 12);

$validator->setMessages( array(
    Zend_Validate_StringLength::TOO_SHORT =>
        'The string \'%value%\' is too short',
    Zend_Validate_StringLength::TOO_LONG  =>
        'The string \'%value%\' is too long'
));
```

If your application requires even greater flexibility with which it reports validation failures, you can access properties by the same name as the message tokens supported by a given validation class. The `value` property is always available in a validator; it is the value you specified as the argument of `isValid()`. Other properties may be supported on a case-by-case basis in each validation class.

```
$validator = new Zend_Validate_StringLength(8, 12);

if (!validator->isValid('word')) {
    echo 'Word failed: '
        . $validator->value
        . '; its length is not between '
        . $validator->min
        . ' and '
        . $validator->max
        . "\n";
}
```

Using the static `is()` method

If it's inconvenient to load a given validation class and create an instance of the validator, you can use the static method `Zend_Validate::is()` as an alternative invocation style. The first argument of this method is a data input value, that you would pass to the `isValid()` method. The second argument is a string, which corresponds to the basename of the validation class, relative to the `Zend_Validate` namespace. The `is()` method automatically loads the class, creates an instance, and applies the `isValid()` method to the data input.

```
if (Zend_Validate::is($email, 'EmailAddress')) {
    // Yes, email appears to be valid
}
```

You can also pass an array of constructor arguments, if they are needed for the validator.

```
if (Zend_Validate::is($value, 'Between', array(1, 12))) {
    // Yes, $value is between 1 and 12
}
```

The `is()` method returns a boolean value, the same as the `isValid()` method. When using the static `is()` method, validation failure messages are not available.

The static usage can be convenient for invoking a validator ad hoc, but if you have the need to run a validator for multiple inputs, it's more efficient to use the non-static usage, creating an instance of the validator object and calling its `isValid()` method.

Also, the `Zend_Filter_Input` class allows you to instantiate and run multiple filter and validator classes on demand to process sets of input data. See the section called “Zend_Filter_Input”.

Standard Validation Classes

The Zend Framework comes with a standard set of validation classes, which are ready for you to use.

Alnum

Returns `true` if and only if `$value` contains only alphabetic and digit characters. This validator includes an option to also consider white space characters as valid.

Alpha

Returns `true` if and only if `$value` contains only alphabetic characters. This validator includes an option to also consider white space characters as valid.

Barcode

This validator is instantiated with a barcode type against which you wish to validate a barcode value. It currently supports "`UPC-A`" (Universal Product Code) and "`EAN-13`" (European Article Number) barcode types, and the `isValid()` method returns true if and only if the input successfully validates against the barcode validation algorithm. You should remove all characters other than the digits zero through nine (0-9) from the input value before passing it on to the validator.

Between

Returns `true` if and only if `$value` is between the minimum and maximum boundary values. The comparison is inclusive by default (`$value` may equal a boundary value), though this may be overridden in order to do a strict comparison, where `$value` must be strictly greater than the minimum and strictly less than the maximum.

Ccnum

Returns `true` if and only if `$value` follows the Luhn algorithm (mod-10 checksum) for credit card numbers.

Date

Returns `true` if `$value` is a valid date of the format `YYYY-MM-DD`. If the optional `locale` option is set then the date will be validated according to the set locale. And if the optional `format` option is set this format is used for the validation. For details about the optional parameters see Zend_Date::isDate().

Digits

Returns `true` if and only if `$value` only contains digit characters.

EmailAddress

`Zend_Validate_EmailAddress` allows you to validate an email address. The validator first splits the email address on local-part @ hostname and attempts to match these against known specifications for email addresses and hostnames.

Basic usage

A basic example of usage is below:

```
$validator = new Zend_Validate_EmailAddress();
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This will match the email address `$email` and on failure populate `$validator->getMessages()` with useful error messages.

Complex local parts

`Zend_Validate_EmailAddress` will match any valid email address according to RFC2822. For example, valid emails include `bob@domain.com`, `bob+jones@domain.us`, `"bob@jones"@domain.com` and `"bob jones"@domain.com`

Some obsolete email formats will not currently validate (e.g. carriage returns or a "\" character in an email address).

Validating different types of hostnames

The hostname part of an email address is validated against `Zend_Validate_Hostname`. By default only DNS hostnames of the form `domain.com` are accepted, though if you wish you can accept IP addresses and Local hostnames too.

To do this you need to instantiate `Zend_Validate_EmailAddress` passing a parameter to indicate the type of hostnames you want to accept. More details are included in `Zend_Validate_Hostname`, though an example of how to accept both DNS and Local hostnames appears below:

```
$validator = new Zend_Validate_EmailAddress(Zend_Validate_Hostname::ALLOW_DNS | Ze
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

Checking if the hostname actually accepts email

Just because an email address is in the correct format, it doesn't necessarily mean that email address actually exists. To help solve this problem, you can use MX validation to check whether an MX (email) entry exists in the DNS record for the email's hostname. This tells you that the hostname accepts email, but doesn't tell you the exact email address itself is valid.

MX checking is not enabled by default and at this time is only supported by UNIX platforms. To enable MX checking you can pass a second parameter to the `Zend_Validate_EmailAddress` constructor.

```
$validator = new Zend_Validate_EmailAddress(Zend_Validate_Hostname::ALLOW_DNS, tru
```

Alternatively you can either pass `true` or `false` to `$validator->setValidateMx()` to enable or disable MX validation.

By enabling this setting network functions will be used to check for the presence of an MX record on the hostname of the email address you wish to validate. Please be aware this will likely slow your script down.

Validating International Domains Names

`Zend_Validate_EmailAddress` will also match international characters that exist in some domains. This is known as International Domain Name (IDN) support. This is enabled by default, though you can disable this by changing the setting via the internal `Zend_Validate_Hostname` object that exists within `Zend_Validate_EmailAddress`.

```
$validator->hostnameValidator->setValidateIdn(false);
```

More information on the usage of `setValidateIdn()` appears in the `Zend_Validate_Hostname` documentation.

Please note 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. This is enabled by default, though you can disable this by changing the setting via the internal `Zend_Validate_Hostname` object that exists within `Zend_Validate_EmailAddress`.

```
$validator->hostnameValidator->setValidateTld(false);
```

More information on the usage of `setValidateTld()` appears in the `Zend_Validate_Hostname` documentation.

Please note TLDs are only validated if you allow DNS hostnames to be validated.

Float

Returns `true` if and only if `$value` is a floating-point value.

GreaterThan

Returns `true` if and only if `$value` is greater than the minimum boundary.

Hex

Returns `true` if and only if `$value` contains only hexadecimal digit characters.

Hostname

Zend_Validate_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.

Basic usage

A basic example of usage is below:

```
$validator = new Zend_Validate_Hostname();
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This will match the hostname `$hostname` and on failure populate `$validator->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_Validate_Hostname when you instantiate it. The paramter should be an integer which determines what types of hostnames are allowed. You are encouraged to use the Zend_Validate_Hostname constants to do this.

The Zend_Validate_Hostname constants are: `ALLOW_DNS` to allow only DNS hostnames, `ALLOW_IP` to allow IP addresses, `ALLOW_LOCAL` to allow local network names, and `ALLOW_ALL` to allow all three types. To just check for IP addresses you can use the example below:

```
$validator = new Zend_Validate_Hostname(Zend_Validate_Hostname::ALLOW_IP);
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

As well as using `ALLOW_ALL` to accept all hostnames types you can combine these types to allow for combinations. For example, to accept DNS and Local hostnames instantiate your Zend_Validate_Hostname object as so:

```
$validator = new Zend_Validate_Hostname(Zend_Validate_Hostname::ALLOW_DNS |
                                        Zend_Validate_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_Validate_Hostname via extended characters that are used in the validation process.

At present the list of supported ccTLDs include:

- at (Austria)
- ch (Switzerland)
- li (Liechtenstein)
- de (Germany)
- fi (Finland)
- hu (Hungary)
- no (Norway)
- se (Sweden)

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 by either passing a parameter to the Zend_Validate_Hostname constructor or via the `$validator->setValidateIdn()` method.

You can disable IDN validation by passing a second parameter to the Zend_Validate_Hostname constructor in the following way.

```
$validator =
    new Zend_Validate_Hostname(Zend_Validate_Hostname::ALLOW_DNS, false);
```

Alternatively you can either pass TRUE or FALSE to `$validator->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/Validate/Hostname specifying the additional characters a normal hostname validation is performed.

Please note 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_Validate_Hostname constructor. In the example below we are supporting IDN validation via the second parameter.

```
$validator =
    new Zend_Validate_Hostname(Zend_Validate_Hostname::ALLOW_DNS,
                               true,
                               false);
```

Alternatively you can either pass TRUE or FALSE to `$validator->setValidateTld()` to enable or disable TLD validation.

Please note TLDs are only validated if you allow DNS hostnames to be validated.

InArray

Returns `true` if and only if a "needle" `$value` is contained in a "haystack" array. If the strict option is `true`, then the type of `$value` is also checked.

Int

Returns `true` if and only if `$value` is a valid integer.

Ip

Returns `true` if and only if `$value` is a valid IP address.

LessThan

Returns `true` if and only if `$value` is less than the maximum boundary.

NotEmpty

Returns `true` if and only if `$value` is not an empty value.

Regex

Returns `true` if and only if `$value` matches against a regular expression pattern.

StringLength

Returns `true` if and only if the string length of `$value` is at least a minimum and no greater than a maximum (when the max option is not `null`). Since version 1.5.0, the `setMin()` method throws an exception if the minimum length is set to a value greater than the set maximum length, and the `setMax()` method throws an exception if the maximum length is set to a value less than than the set minimum length. Since version 1.0.2, this class supports UTF-8 and other character encodings, based on the current value of `iconv.internal_encoding` [http://www.php.net/manual/en/ref.iconv.php#iconv.configuration].

Validator Chains

Often multiple validations should be applied to some value in a particular order. The following code demonstrates a way to solve the example from the introduction, where a username must be between 6 and 12 alphanumeric characters:

```
// Create a validator chain and add validators to it
$validatorChain = new Zend_Validate();
$validatorChain->addValidator(new Zend_Validate_StringLength(6, 12))
               ->addValidator(new Zend_Validate_Alnum());

// Validate the username
if ($validatorChain->isValid($username)) {
    // username passed validation
} else {
    // username failed validation; print reasons
    foreach ($validatorChain->getMessages() as $message) {
        echo "$message\n";
    }
}
```

Validators are run in the order they were added to `Zend_Validate`. 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_Validate` supports such use cases with the second parameter to the `addValidator()` method. By setting `$breakChainOnFailure` to `true`, the added validator will break the chain execution upon failure, which avoids running any other validations that are determined to be unnecessary or inappropriate for the situation. If the above example were written as follows, then the alphanumeric validation would not occur if the string length validation fails:

```
$validatorChain->addValidator(new Zend_Validate_StringLength(6, 12), true)
        ->addValidator(new Zend_Validate_Alnum());
```

Any object that implements `Zend_Validate_Interface` may be used in a validator chain.

Writing Validators

Zend_Validate 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_Validate_Interface` defines three methods, `isValid()`, `getMessages()`, and `getErrors()`, that may be implemented by user classes in order to create custom validation objects. An object that implements `Zend_Validate_Interface` interface may be added to a validator chain with `Zend_Validate::addValidator()`. Such objects may also be used with `Zend_Filter_Input`.

As you may already have inferred from the above description of `Zend_Validate_Interface`, 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_Validate_Abstract`. To include this functionality when creating a validation class, simply extend `Zend_Validate_Abstract`. 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.

Example 48.1. Creating a Simple Validation Class

The following example demonstrates how a very simple custom validator might be written. In this case the validation rules are simply that the input value must be a floating point value.

```
class MyValid_Float extends Zend_Validate_Abstract
{
    const FLOAT = 'float';

    protected $_messageTemplates = array(
        self::FLOAT => "'%value%' is not a floating point value"
    );

    public function isValid($value)
    {
        $this->_setValue($value);

        if (!is_float($value)) {
            $this->_error();
            return false;
        }

        return true;
    }
}
```

The class defines a template for its single validation failure message, which includes the built-in magic parameter, `%value%`. The call to `_setValue()` prepares the object to insert the tested value into the failure message automatically, should the value fail validation. The call to `_error()` tracks a reason for validation failure. Since this class only defines one failure message, it is not necessary to provide `_error()` with the name of the failure message template.

Example 48.2. Writing a Validation Class having Dependent Conditions

The following example demonstrates a more complex set of validation rules, where it is required that the input value be numeric and within the range of minimum and maximum boundary values. An input value would fail validation for exactly one of the following reasons:

- The input value is not numeric.
- The input value is less than the minimum allowed value.
- The input value is more than the maximum allowed value.

These validation failure reasons are then translated to definitions in the class:

```
class MyValid_NumericBetween extends Zend_Validate_Abstract
{
    const MSG_NUMERIC = 'msgNumeric';
    const MSG_MINIMUM = 'msgMinimum';
    const MSG_MAXIMUM = 'msgMaximum';

    public $minimum = 0;
    public $maximum = 100;

    protected $_messageVariables = array(
        'min' => 'minimum',
        'max' => 'maximum'
    );

    protected $_messageTemplates = array(
        self::MSG_NUMERIC => "'%value%' is not numeric",
        self::MSG_MINIMUM => "'%value%' must be at least '%min%'",
        self::MSG_MAXIMUM => "'%value%' must be no more than '%max%'"
    );

    public function isValid($value)
    {
        $this->_setValue($value);

        if (!is_numeric($value)) {
            $this->_error(self::MSG_NUMERIC);
            return false;
        }

        if ($value < $this->minimum) {
            $this->_error(self::MSG_MINIMUM);
            return false;
        }

        if ($value > $this->maximum) {
            $this->_error(self::MSG_MAXIMUM);
            return false;
        }
        return true;
    }
}
```

The public properties `$minimum` and `$maximum` have been established to provide the minimum and maximum boundaries, respectively, for a value to successfully validate. The class also defines two message variables that correspond to the public properties and allow `min` and `max` to be used in message templates as magic parameters, just as with `value`.

Note that if any one of the validation checks in `isValid()` fails, an appropriate failure message is prepared, and the method immediately returns `false`. These validation rules are therefore sequentially dependent. That is, if one test should fail, there is no need to test any subsequent validation rules. This need not be the case, however. The following example illustrates how to write a class having independent validation rules, where the validation object may return multiple reasons why a particular validation attempt failed.

Example 48.3. Validation with Independent Conditions, Multiple Reasons for Failure

Consider writing a validation class for password strength enforcement - when a user is required to choose a password that meets certain criteria for helping secure user accounts. Let us assume that the password security criteria enforce that the password:

- is at least 8 characters in length,
- contains at least one uppercase letter,
- contains at least one lowercase letter,
- and contains at least one digit character.

The following class implements these validation criteria:

```
class MyValid_PasswordStrength extends Zend_Validate_Abstract
{
    const LENGTH = 'length';
    const UPPER  = 'upper';
    const LOWER  = 'lower';
    const DIGIT  = 'digit';

    protected $_messageTemplates = array(
        self::LENGTH => "'%value%' must be at least 8 characters in length",
        self::UPPER  => "'%value%' must contain at least one uppercase letter",
        self::LOWER  => "'%value%' must contain at least one lowercase letter",
        self::DIGIT  => "'%value%' must contain at least one digit character"
    );

    public function isValid($value)
    {
        $this->_setValue($value);

        $isValid = true;

        if (strlen($value) < 8) {
            $this->_error(self::LENGTH);
            $isValid = false;
        }

        if (!preg_match('/[A-Z]/', $value)) {
            $this->_error(self::UPPER);
            $isValid = false;
        }

        if (!preg_match('/[a-z]/', $value)) {
            $this->_error(self::LOWER);
            $isValid = false;
        }

        if (!preg_match('/\d/', $value)) {
            $this->_error(self::DIGIT);
            $isValid = false;
        }

        return $isValid;
    }
}
```

Note that the four criteria tests in `isValid()` do not immediately return `false`. This allows the validation class to provide **all** of the reasons that the input password failed to meet the validation requirements. If, for example, a user were to input the string "`#$%`" as a password, `isValid()` would cause all four validation failure messages to be returned by a subsequent call to `getMessages()`.

Chapter 49. Zend_Version

Reading the Zend Framework Version

The class constant `Zend_Version::VERSION` contains a string that identifies the current version number of Zend Framework. For example, "0.9.0beta".

The static method `Zend_Version::compareVersion($version)` is based on the PHP function `version_compare()` [http://php.net/version_compare]. The method returns -1 if the specified `$version` is older than the Zend Framework version, 0 if they are the same, and +1 if the specified `$version` is newer than the Zend Framework version.

Example 49.1. Example of compareVersion() method

```
// returns -1, 0 or 1
$cmp = Zend_Version::compareVersion('1.0.0');
```

Chapter 50. Zend_View

Introduction

Zend_View is a class for working with the "view" portion of the model-view-controller pattern. That is, it exists to help keep the view script separate from the model and controller scripts. It provides a system of helpers, output filters, and variable escaping.

Zend_View 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.

Essentially, using Zend_View happens in two major steps: 1. Your controller script creates an instance of Zend_View and assigns variables to that instance. 2. The controller tells the Zend_View to render a particular view, thereby handing control over the view script, which generates the view output.

Controller Script

As a simple example, let us say your controller has a list of book data that it wants to have rendered by a view. The controller script might look something like this:

```
// use a model to get the data for book authors and titles.
$data = array(
    array(
        'author' => 'Hernando de Soto',
        'title' => 'The Mystery of Capitalism'
    ),
    array(
        'author' => 'Henry Hazlitt',
        'title' => 'Economics in One Lesson'
    ),
    array(
        'author' => 'Milton Friedman',
        'title' => 'Free to Choose'
    )
);

// now assign the book data to a Zend_View instance
Zend_Loader::loadClass('Zend_View');
$view = new Zend_View();
$view->books = $data;

// and render a view script called "booklist.php"
echo $view->render('booklist.php');
```

View Script

Now we need the associated view script, "booklist.php". This is a PHP script like any other, with one exception: it executes inside the scope of the Zend_View instance, which means that references to $this point to the Zend_View instance properties and methods. (Variables assigned to the instance by the controller are public properties of the Zend_View instance.) Thus, a very basic view script could look like this:

```
 if ($this->books): ?>

    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escape($val['author']) ?></td>
            <td><?php echo $this->escape($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>

    </table>

<?php else: ?>

    <p>There are no books to display.</p>

<?php endif;?>
```

Note how we use the "escape()" method to apply output escaping to variables.

Options

`Zend_View` has several options that may be set to configure the behaviour of your view scripts.

- `basePath`: indicate a base path from which to set the script, helper, and filter path. It assumes a directory structure of:

  ```
  base/path/
      helpers/
      filters/
      scripts/
  ```

  This may be set via `setBasePath()`, `addBasePath()`, or the `basePath` option to the constructor.

- `encoding`: indicate the character encoding to use with `htmlentities()`, `htmlspecialchars()`, and other operations. Defaults to ISO-8859-1 (latin1). May be set via `setEncoding()` or the `encoding` option to the constructor.

- `escape`: indicate a callback to be used by `escape()`. May be set via `setEscape()` or the `escape` option to the constructor.
- `filter`: indicate a filter to use after rendering a view script. May be set via `setFilter()`, `addFilter()`, or the `filter` option to the constructor.
- `strictVars:` force `Zend_View` to emit notices and warnings when uninitialized view variables are accessed. This may be set by calling `strictVars(true)` or passing the `strictVars` option to the constructor.

Short Tags with View Scripts

In our examples and documentation, we make use of PHP short tags: `<?` and `<?=`. In addition, we typically use the alternate syntax for control structures [http://us.php.net/manual/en/control-structures.alternative-syntax.php]. These are convenient shorthands to use when writing view scripts, as they make the constructs more terse, and keep statements on single lines.

That said, many developers prefer to use full tags for purposes of validation or portability. For instance, `short_open_tag` is disabled in the php.ini.recommended file, and if you template XML in view scripts, short open tags will cause the templates to fail validation.

Additionally, if you use short tags when the setting is off, then the view scripts will either cause errors or simply echo code to the user.

For this latter case, where you wish to use short tags but they are disabled, you have two options:

- Turn on short tags in your `.htaccess` file:

  ```
  php_value "short_open_tag" "on"
  ```

  This will only be possible if you are allowed to create and utilize `.htaccess` files. This directive can also be added to your `httpd.conf` file.

- Enable an optional stream wrapper to convert short tags to long tags on the fly:

  ```
  $view->setUseStreamWrapper(true);
  ```

  This registers `Zend_View_Stream` as a stream wrapper for view scripts, and will ensure that your code continues to work as if short tags were enabled.

View Stream Wrapper Degrades Performance

Usage of the stream wrapper *will* degrade performance of your application, though actual benchmarks are unavailable to quantify the amount of degradation. We recommend that you either enable short tags, convert your scripts to use full tags, or have a good partial and/or full page content caching strategy in place.

Utility Accessors

Typically, you'll only ever need to call on `assign()`, `render()`, or one of the methods for setting/adding filter, helper, and script paths. However, if you wish to extend `Zend_View` yourself, or need access to some of its internals, a number of accessors exist:

- `getVars()` will return all assigned variables.
- `clearVars()` will clear all assigned variables; useful when you wish to re-use a view object, but want to control what variables are available.
- `getScriptPath($script)` will retrieve the resolved path to a given view script.
- `getScriptPaths()` will retrieve all registered script paths.
- `getHelperPath($helper)` will retrieve the resolved path to the named helper class.
- `getHelperPaths()` will retrieve all registered helper paths.
- `getFilterPath($filter)` will retrieve the resolved path to the named filter class.
- `getFilterPaths()` will retrieve all registered filter paths.

Controller Scripts

The controller is where you instantiate and configure Zend_View. You then assign variables to the view, and tell the view to render output using a particular script.

Assigning Variables

Your controller script should assign necessary variables to the view before it hands over control to the view script. Normally, you can do assignments one at a time by assigning to property names of the view instance:

```
$view = new Zend_View();
$view->a = "Hay";
$view->b = "Bee";
$view->c = "Sea";
```

However, this can be tedious when you have already collected the values to be assigned into an array or object.

The assign() method lets you assign from an array or object "in bulk." The following examples have the same effect as the above one-by-one property assignments.

```
$view = new Zend_View();

// assign an array of key-value pairs, where the
// key is the variable name, and the value is
// the assigned value.
```

```
$array = array(
    'a' => "Hay",
    'b' => "Bee",
    'c' => "Sea",
);
$view->assign($array);

// do the same with an object's public properties;
// note how we cast it to an array when assigning.
$obj = new StdClass;
$obj->a = "Hay";
$obj->b = "Bee";
$obj->c = "Sea";
$view->assign((array) $obj);
```

Alternatively, you can use the assign method to assign one-by-one by passing a string variable name, and then the variable value.

```
$view = new Zend_View();
$view->assign('a', "Hay");
$view->assign('b', "Bee");
$view->assign('c', "Sea");
```

Rendering a View Script

Once you have assigned all needed variables, the controller should tell Zend_View to render a particular view script. Do so by calling the render() method. Note that the method will return the rendered view, not print it, so you need to print or echo it yourself at the appropriate time.

```
$view = new Zend_View();
$view->a = "Hay";
$view->b = "Bee";
$view->c = "Sea";
echo $view->render('someView.php');
```

View Script Paths

By default, Zend_View expects your view scripts to be relative to your calling script. For example, if your controller script is at "/path/to/app/controllers" and it calls $view->render('someView.php'), Zend_View will look for "/path/to/app/controllers/someView.php".

Obviously, your view scripts are probably located elsewhere. To tell Zend_View where it should look for view scripts, use the setScriptPath() method.

```
$view = new Zend_View();
```

```
$view->setScriptPath('/path/to/app/views');
```

Now when you call $view->render('someView.php'), it will look for "/path/to/app/views/someView.php".

In fact, you can "stack" paths using the addScriptPath() method. As you add paths to the stack, Zend_View will look at the most-recently-added path for the requested view script. This allows you override default views with custom views so that you may create custom "themes" or "skins" for some views, while leaving others alone.

```
$view = new Zend_View();
$view->addScriptPath('/path/to/app/views');
$view->addScriptPath('/path/to/custom/');

// now when you call $view->render('booklist.php'), Zend_View will
// look first for "/path/to/custom/booklist.php", then for
// "/path/to/app/views/booklist.php", and finally in the current
// directory for "booklist.php".
```

View Scripts

Once your controller has assigned variables and called render(), Zend_View then includes the requested view script and executes it "inside" the scope of the Zend_View instance. Therefore, in your view scripts, references to $this actually point to the Zend_View instance itself.

Variables assigned to the view from the controller are referred to as instance properties. For example, if the controller were to assign a variable 'something', you would refer to it as $this->something in the view script. (This allows you to keep track of which values were assigned to the script, and which are internal to the script itself.)

By way of reminder, here is the example view script from the Zend_View introduction.

```
<?php if ($this->books): ?>

    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escape($val['author']) ?></td>
            <td><?php echo $this->escape($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>

    </table>
```

```
<?php else: ?>

    <p>There are no books to display.</p>

<?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.

Zend_View comes with a method called escape() that does such escaping for you.

```
// bad view-script practice:
echo $this->variable;

// good view-script practice:
echo $this->escape($this->variable);
```

By default, the escape() method uses the PHP htmlspecialchars() function for escaping. However, depending on your environment, you may wish for escaping to occur in a different way. Use the setEscape() method at the controller level to tell Zend_View what escaping callback to use.

```
// create a Zend_View instance
$view = new Zend_View();

// tell it to use htmlentities as the escaping callback
$view->setEscape('htmlentities');

// or tell it to use a static class method as the callback
$view->setEscape(array('SomeClass', 'methodName'));

// or even an instance method
$obj = new SomeClass();
$view->setEscape(array($obj, 'methodName'));

// and then render your view
echo $view->render(...);
```

The callback function or method should take the value to be escaped as its first parameter, and all other parameters should be optional.

Using Alternate Template Systems

Although PHP is itself a powerful template system, many developers feel it is too powerful or complex for their template designers and will want to use an alternate template engine. Zend_View provides two mechanisms for doing so, the first through view scripts, the second by implementing Zend_View_Interface.

Template Systems Using View Scripts

A view script may be used to instantiate and manipulate a separate template object, such as a PHPLIB-style template. The view script for that kind of activity might look something like this:

```
include_once 'template.inc';
$tpl = new Template();

if ($this->books) {
    $tpl->setFile(array(
        "booklist" => "booklist.tpl",
        "eachbook" => "eachbook.tpl",
    ));

    foreach ($this->books as $key => $val) {
        $tpl->set_var('author', $this->escape($val['author']);
        $tpl->set_var('title', $this->escape($val['title']);
        $tpl->parse("books", "eachbook", true);
    }

    $tpl->pparse("output", "booklist");
} else {
    $tpl->setFile("nobooks", "nobooks.tpl")
    $tpl->pparse("output", "nobooks");
}
```

These would be the related template files:

```
<!-- booklist.tpl -->
<table>
    <tr>
        <th>Author</th>
        <th>Title</th>
    </tr>
    {books}
</table>

<!-- eachbook.tpl -->
    <tr>
        <td>{author}</td>
        <td>{title}</td>
    </tr>
```

```
<!-- nobooks.tpl -->
<p>There are no books to display.</p>
```

Template Systems Using Zend_View_Interface

Some may find it easier to simply provide a Zend_View-compatible template engine. `Zend_View_Interface` defines the minimum interface needed for compatability:

```
/**
 * Return the actual template engine object
 */
public function getEngine();

/**
 * Set the path to view scripts/templates
 */
public function setScriptPath($path);

/**
 * Set a base path to all view resources
 */
public function setBasePath($path, $prefix = 'Zend_View');

/**
 * Add an additional base path to view resources
 */
public function addBasePath($path, $prefix = 'Zend_View');

/**
 * Retrieve the current script paths
 */
public function getScriptPaths();

/**
 * Overloading methods for assigning template variables as object
 * properties
 */
public function __set($key, $value);
public function __get($key);
public function __isset($key);
public function __unset($key);

/**
 * Manual assignment of template variables, or ability to assign
 * multiple variables en masse.
 */
public function assign($spec, $value = null);

/**
 * Unset all assigned template variables
 */
```

```
public function clearVars();

/**
 * Render the template named $name
 */
public function render($name);
```

Using this interface, it becomes relatively easy to wrap a third-party template engine as a Zend_View-compatible class. As an example, the following is one potential wrapper for Smarty:

```
class Zend_View_Smarty implements Zend_View_Interface
{
    /**
     * Smarty object
     * @var Smarty
     */
    protected $_smarty;

    /**
     * Constructor
     *
     * @param string $tmplPath
     * @param array $extraParams
     * @return void
     */
    public function __construct($tmplPath = null, $extraParams = array())
    {
        $this->_smarty = new Smarty;

        if (null !== $tmplPath) {
            $this->setScriptPath($tmplPath);
        }

        foreach ($extraParams as $key => $value) {
            $this->_smarty->$key = $value;
        }
    }

    /**
     * Return the template engine object
     *
     * @return Smarty
     */
    public function getEngine()
    {
        return $this->_smarty;
    }

    /**
     * Set the path to the templates
     *
     * @param string $path The directory to set as the path.
     * @return void
     */
    public function setScriptPath($path)
    {
        if (is_readable($path)) {
            $this->_smarty->template_dir = $path;
            return;
        }

        throw new Exception('Invalid path provided');
    }

    /**
     * Retrieve the current template directory
     *
     * @return string
     */
    public function getScriptPaths()
    {
        return array($this->_smarty->template_dir);
    }

    /**
     * Alias for setScriptPath
     *
     * @param string $path
     * @param string $prefix Unused
     * @return void
     */
    public function setBasePath($path, $prefix = 'Zend_View')
    {
        return $this->setScriptPath($path);
    }

    /**
     * Alias for setScriptPath
     *
     * @param string $path
     * @param string $prefix Unused
     * @return void
     */
    public function addBasePath($path, $prefix = 'Zend_View')
    {
        return $this->setScriptPath($path);
    }

    /**
     * Assign a variable to the template
     *
     * @param string $key The variable name.
     * @param mixed $val The variable value.
     * @return void
     */
    public function __set($key, $val)
    {
        $this->_smarty->assign($key, $val);
    }

    /**
     * Retrieve an assigned variable
     *
     * @param string $key The variable name.
     * @return mixed The variable value.
     */
    public function __get($key)
    {
        return $this->_smarty->get_template_vars($key);
    }

    /**
     * Allows testing with empty() and isset() to work
     *
     * @param string $key
     * @return boolean
     */
    public function __isset($key)
    {
        return (null !== $this->_smarty->get_template_vars($key));
    }

    /**
     * Allows unset() on object properties to work
     *
     * @param string $key
     * @return void
     */
    public function __unset($key)
    {
        $this->_smarty->clear_assign($key);
    }

    /**
     * Assign variables to the template
     *
     * Allows setting a specific key to the specified value, OR passing
     * an array of key => value pairs to set en masse.
     *
     * @see __set()
     * @param string|array $spec The assignment strategy to use (key or
     * array of key => value pairs)
     * @param mixed $value (Optional) If assigning a named variable,
     * use this as the value.
     * @return void
     */
    public function assign($spec, $value = null)
    {
        if (is_array($spec)) {
            $this->_smarty->assign($spec);
            return;
        }

        $this->_smarty->assign($spec, $value);
    }

    /**
     * Clear all assigned variables
     *
     * Clears all variables assigned to Zend_View either via
     * {@link assign()} or property overloading
     * ({@link __get()}/{@link __set()}).
     *
     * @return void
     */
    public function clearVars()
    {
        $this->_smarty->clear_all_assign();
    }

    /**
     * Processes a template and returns the output.
     *
     * @param string $name The template to process.
     * @return string The output.
     */
    public function render($name)
    {
        return $this->_smarty->fetch($name);
    }
}
```

In this example, you would instantiate the `Zend_View_Smarty` class instead of `Zend_View`, and then use it in roughly the same fashion as `Zend_View`:

```
$view = new Zend_View_Smarty();
$view->setScriptPath('/path/to/templates');
$view->book = 'Zend PHP 5 Certification Study Guide';
$view->author = 'Davey Shafik and Ben Ramsey'
$rendered = $view->render('bookinfo.tpl');
```

View Helpers

In your view scripts, often it is necessary to perform certain complex functions over and over: e.g., formatting a date, generating form elements, or displaying action links. You can use helper classes to perform these behaviors for you.

A helper is simply a class. Let's say we want a helper named 'fooBar'. By default, the class is prefixed with `'Zend_View_Helper_'` (you can specify a custom prefix when setting a helper path), and the last segment of the class name is the helper name; this segment should be TitleCapped; the full class name is then: `Zend_View_Helper_FooBar`. This class should contain at the minimum a single method, named after the helper, and camelCased: `fooBar()`.

Watch the Case

Helper names are always camelCased, i.e., they never begin with an uppercase character. The class name itself is MixedCased, but the method that is actually executed is camelCased.

Default Helper Path

The default helper path always points to the Zend Framework view helpers, i.e., 'Zend/View/Helper/'. Even if you call `setHelperPath()` to overwrite the existing paths, this path will be set to ensure the default helpers work.

To use a helper in your view script, call it using `$this->helperName()`. Behind the scenes, `Zend_View` will load the `Zend_View_Helper_HelperName` class, create an object instance of it, and call its `helperName()` method. The object instance is persistent within the `Zend_View` instance, and is reused for all future calls to `$this->helperName()`.

Initial Helpers

`Zend_View` comes with an initial set of helper classes, most of which relate to form element generation and perform the appropriate output escaping automatically. In addition, there are helpers for creating route-based URLs and HTML lists, as well as declaring variables. The currently shipped helpers include:

- `declareVars()`: Primarily for use when using `strictVars()`, this helper can be used to declare template variables that may or may not already be set in the view object, as well as to set default values. Arrays passed as arguments to the method will be used to set default values; otherwise, if the variable does not exist, it is set to an empty string.
- `fieldset($name, $content, $attribs)`: Creates an XHTML fieldset. If `$attribs` contains a 'legend' key, that value will be used for the fieldset legend. The fieldset will surround the `$content` as provided to the helper.
- `form($name, $attribs, $content)`: Generates an XHTML form. All `$attribs` are escaped and rendered as XHTML attributes of the form tag. If `$content` is present and not a boolean false, then that content is rendered within the start and close form tags; if `$content` is a boolean false (the default), only the opening form tag is generated.
- `formButton($name, $value, $attribs)`: Creates an <button /> element.
- `formCheckbox($name, $value, $attribs, $options)`: Creates an <input type="checkbox" /> element.

  By default, when no $value is provided and no $options are present, '0' is assumed to be the unchecked value, and '1' the checked value. If a $value is passed, but no $options are present, the checked value is assumed to be the value passed.

  $options should be an array. If the array is indexed, the first value is the checked value, and the second the unchecked value; all other values are ignored. You may also pass an associative array with the keys 'checked' and 'unChecked'.

If $options has been passed, if $value matches the checked value, then the element will be marked as checked. You may also mark the element as checked or unchecked by passing a boolean value for the attribute 'checked'.

The above is probably best summed up with some examples:

```
// '1' and '0' as checked/unchecked options; not checked
echo $this->formCheckbox('foo');

// '1' and '0' as checked/unchecked options; checked
echo $this->formCheckbox('foo', null, array('checked' => true));

// 'bar' and '0' as checked/unchecked options; not checked
echo $this->formCheckbox('foo', 'bar');

// 'bar' and '0' as checked/unchecked options; checked
echo $this->formCheckbox('foo', 'bar', array('checked' => true));

// 'bar' and 'baz' as checked/unchecked options; unchecked
echo $this->formCheckbox('foo', null, null, array('bar', 'baz');

// 'bar' and 'baz' as checked/unchecked options; unchecked
echo $this->formCheckbox('foo', null, null, array(
    'checked' => 'bar',
    'unChecked' => 'baz'
));

// 'bar' and 'baz' as checked/unchecked options; checked
echo $this->formCheckbox('foo', 'bar', null, array('bar', 'baz');
echo $this->formCheckbox('foo',
                         null,
                         array('checked' => true),
                         array('bar', 'baz');

// 'bar' and 'baz' as checked/unchecked options; unchecked
echo $this->formCheckbox('foo', 'baz', null, array('bar', 'baz');
echo $this->formCheckbox('foo',
                         null,
                         array('checked' => false),
                         array('bar', 'baz');
```

In all cases, the markup prepends a hidden element with the unchecked value; this way, if the value is unchecked, you will still get a valid value returned to your form.

- `formErrors($errors, $options)`: Generates an XHTML unordered list to show errors. `$errors` should be a string or an array of strings; `$options` should be any attributes you want placed in the opening list tag.

  You can specify alternate opening, closing, and separator content when rendering the errors by calling several methods on the helper:

  - `setElementStart($string)`; default is '<ul class="errors"%s"><li>', where %s is replaced with the attributes as specified in `$options`.
  - `setElementSeparator($string)`; default is '</li><li>'.
  - `setElementEnd($string)`; default is '</li></ul>'.
- `formFile($name, $value, $attribs):` Creates an <input type="file" /> element.
- `formHidden($name, $value, $attribs):` Creates an <input type="hidden" /> element.
- `formLabel($name, $value, $attribs):` Creates a <label> element, setting the `for` attribute to `$name`, and the actual label text to `$value`. If `disable` is passed in `attribs`, nothing will be returned.
- `formMultiCheckbox($name, $value, $attribs, $options, $listsep):` Creates a list of checkboxes. `$options` should be an associative array, and may be arbitrarily deep. `$value` may be a single value or an array of selected values that match the keys in the `$options` array. `$listsep` is an HTML break ("<br />") by default. By default, this element is treated as an array; all checkboxes share the same name, and are submitted as an array.
- `formPassword($name, $value, $attribs):` Creates an <input type="password" /> element.
- `formRadio($name, $value, $attribs, $options):` Creates a series of <input type="radio" /> elements, one for each of the $options elements. In the $options array, the element key is the radio value, and the element value is the radio label. The $value radio will be preselected for you.
- `formReset($name, $value, $attribs):` Creates an <input type="reset" /> element.
- `formSelect($name, $value, $attribs, $options):` Creates a <select>...</select> block, with one <option>one for each of the $options elements. In the $options array, the element key is the option value, and the element value is the option label. The $value option(s) will be preselected for you.
- `formSubmit($name, $value, $attribs):` Creates an <input type="submit" /> element.
- `formText($name, $value, $attribs):` Creates an <input type="text" /> element.
- `formTextarea($name, $value, $attribs):` Creates a <textarea>...</textarea> block.
- `url($urlOptions, $name, $reset):` Creates a URL string based on a named route. `$urlOptions` should be an associative array of key/value pairs used by the particular route.
- `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.

Using these in your view scripts is very easy, here is an example. Note that you all you need to do is call them; they will load and instantiate themselves as they are needed.

```
// inside your view script, $this refers to the Zend_View instance.
//
// say that you have already assigned a series of select options under
// the name $countries as array('us' => 'United States', 'il' =>
// 'Israel', 'de' => 'Germany').
?>
<form action="action.php" method="post">
    <p><label>Your Email:
<?php echo $this->formText('email', 'you@example.com', array('size' => 32)) ?>
    </label></p>
    <p><label>Your Country:
<?php echo $this->formSelect('country', 'us', null, $this->countries) ?>
    </label></p>
    <p><label>Would you like to opt in?
<?php echo $this->formCheckbox('opt_in', 'yes', null, array('yes', 'no')) ?>
    </label></p>
</form>
```

The resulting output from the view script will look something like this:

```
<form action="action.php" method="post">
    <p><label>Your Email:
        <input type="text" name="email" value="you@example.com" size="32" />
    </label></p>
    <p><label>Your Country:
        <select name="country">
            <option value="us" selected="selected">United States</option>
            <option value="il">Israel</option>
            <option value="de">Germany</option>
        </select>
    </label></p>
    <p><label>Would you like to opt in?
        <input type="hidden" name="opt_in" value="no" />
        <input type="checkbox" name="opt_in" value="yes" checked="checked" />
    </label></p>
</form>
```

Action View Helper

The `Action` view helper enables view scripts to dispatch a given controller action; the result of the response object following the dispatch is then returned. These can be used when a particular action could generate re-usable content or "widget-ized" content.

Actions that result in a `_forward()` or redirect are considered invalid, and will return an empty string.

The API for the `Action` view helper follows that of most MVC components that invoke controller actions: `action($action, $controller, $module = null, array $params = array())`. `$action` and `$controller` are required; if no module is specified, the default module is assumed.

Example 50.1. Basic Usage of Action View Helper

As an example, you may have a `CommentController` with a `listAction()` method you wish to invoke in order to pull a list of comments for the current request:

```
<div id="sidebar right">
    <div class="item">
        <?= $this->action('list', 'comment', null, array('count' => 10)); ?>
    </div>
</div>
```

Partial Helper

The `Partial` view helper is used to render a specified template within its own variable scope. The primary use is for reusable template fragments with which you do not need to worry about variable name clashes. Additionally, they allow you to specify partial view scripts from specific modules.

A sibling to the `Partial`, the `PartialLoop` view helper allows you to pass iterable data, and render a partial for each item.

Example 50.2. Basic Usage of Partials

Basic usage of partials is to render a template fragment in its own view scope. Consider the following partial script:

```
<?php // partial.phtml ?>
<ul>
    <li>From: <?= $this->escape($this->from) ?></li>
    <li>Subject: <?= $this->escape($this->subject) ?></li>
</ul>
```

You would then call it from your view script using the following:

```
<?= $this->partial('partial.phtml', array(
    'from' => 'Team Framework',
    'subject' => 'view partials')); ?>
```

Which would then render:

```
<ul>
    <li>From: Team Framework</li>
    <li>Subject: view partials</li>
</ul>
```

What is a model?

A model used with the `Partial` view helper can be one of the following:

- *Array*. If an array is passed, it should be associative, as its key/value pairs are assigned to the view with keys as view variables.
- *Object implementing toArray() method*. If an object is passed an has a `toArray()` method, the results of `toArray()` will be assigned to the view object as view variables.
- *Standard object*. Any other object will assign the results of `object_get_vars()` (essentially all public properties of the object) to the view object.

If your model is an object, you may want to have it passed *as an object* to the partial script, instead of serializing it to an array of variables. You can do this by setting the 'objectKey' property of the appropriate helper:

```
// Tell partial to pass objects as 'model' variable
$view->partial()->setObjectKey('model');
```

```
// Tell partial to pass objects from partialLoop as 'model' variable
// in final partial view script:
$view->partialLoop()->setObjectKey('model');
```

This technique is particularly useful when passing `Zend_Db_Table_Rowset`s to `partial-Loop()`, 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).

Example 50.3. 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:

```
<? // partialLoop.phtml ?>
    <dt><?= $this->key ?></dt>
    <dd><?= $this->value ?></dd>
```

And the following "model":

```
$model = array(
    array('key' => 'Mammal', 'value' => 'Camel'),
    array('key' => 'Bird', 'value' => 'Penguin'),
    array('key' => 'Reptile', 'value' => 'Asp'),
    array('key' => 'Fish', 'value' => 'Flounder'),
);
```

In your view script, you could then invoke the `PartialLoop` helper:

```
<dl>
<?= $this->partialLoop('partialLoop.phtml', $model) ?>
</dl>
```

```
<dl>
    <dt>Mammal</dt>
    <dd>Camel</dd>

    <dt>Bird</dt>
    <dd>Penguin</dd>

    <dt>Reptile</dt>
    <dd>Asp</dd>

    <dt>Fish</dt>
    <dd>Flounder</dd>
</dl>
```

Example 50.4. Rendering Partials in Other Modules

Sometime a partial will exist in a different module. If you know the name of the module, you can pass it as the second argument to either `partial()` or `partialLoop()`, moving the `$model` argument to third position.

For instance, if there's a pager partial you wish to use that's in the 'list' module, you could grab it as follows:

```
<?= $this->partial('pager.phtml', 'list', $pagerData) ?>
```

In this way, you can re-use partials created specifically for other modules. That said, it's likely a better practice to put re-usable partials in shared view script paths.

Placeholder Helper

The `Placeholder` view helper is used to persist content between view scripts and view instances. It also offers some useful features such as aggregating content, capturing view script content for later use, and adding pre- and post-text to content (and custom separators for aggregated content).

Example 50.5. Basic Usage of Placeholders

Basic usage of placeholders is to persist view data. Each invocation of the `Placeholder` helper expects a placeholder name; the helper then returns a placeholder container object that you can either manipulate or simply echo out.

```
<?php $this->placeholder('foo')->set("Some text for later") ?>

<?php
    echo $this->placeholder('foo');
    // outputs "Some text for later"
?>
```

Example 50.6. Using Placeholders to Aggregate Content

Aggregating content via placeholders can be useful at times as well. For instance, your view script may have a variable array from which you wish to retrieve messages to display later; a later view script can then determine how those will be rendered.

The `Placeholder` view helper uses containers that extend `ArrayObject`, providing a rich featureset for manipulating arrays. In addition, it offers a variety of methods for formatting the content stored in the container:

- `setPrefix($prefix)` sets text with which to prefix the content. Use `getPrefix()` at any time to determine what the current setting is.
- `setPostfix($prefix)` sets text with which to append the content. Use `getPostfix()` at any time to determine what the current setting is.
- `setSeparator($prefix)` sets text with which to separate aggregated content. Use `getSeparator()` at any time to determine what the current setting is.
- `setIndent($prefix)` can be used to set an indentation value for content. If an integer is passed, that number of spaces will be used; if a string is passed, the string will be used. Use `getIndent()` at any time to determine what the current setting is.

```
<!-- first view script -->
<?php $this->placeholder('foo')->exchangeArray($this->data) ?>
```

```
<!-- later view script -->
<?php
$this->placeholder('foo')->setPrefix("<ul>\n    <li>")
                         ->setSeparator("</li><li>\n")
                         ->setIndent(4)
                         ->setPostfix("</li></ul>\n");
?>

<?php
    echo $this->placeholder('foo');
    // outputs as unordered list with pretty indentation
?>
```

Because the `Placeholder` container objects extend `ArrayObject`, you can also assign content to a specific key in the container easily, instead of simply pushing it into the container. Keys may be accessed either as object properties or as array keys.

```
<?php $this->placeholder('foo')->bar = $this->data ?>
<?php echo $this->placeholder('foo')->bar ?>

<?php
$foo = $this->placeholder('foo');
```

```
echo $foo['bar'];
?>
```

Example 50.7. Using Placeholders to Capture Content

Occasionally you may have content for a placeholder in a view script that is easiest to template; the `Placeholder` view helper allows you to capture arbitrary content for later rendering using the following API.

- `captureStart($type, $key)` begins capturing content.

  `$type` should be one of the `Placeholder` constants `APPEND` or `SET`. If `APPEND`, captured content is appended to the list of current content in the placeholder; if `SET`, captured content is used as the sole value of the placeholder (potentially replacing any previous content). By default, `$type` is `APPEND`.

  `$key` can be used to specify a specific key in the placeholder container to which you want content captured.

  `captureStart()` locks capturing until `captureEnd()` is called; you cannot nest capturing with the same placholder container. Doing so will raise an exception.

- `captureEnd()` stops capturing content, and places it in the container object according to how `captureStart()` was called.

```
<!-- Default capture: append -->
<?php $this->placeholder('foo')->captureStart();
foreach ($this->data as $datum): ?>
<div class="foo">
    <h2><?= $datum->title ?></h2>
    <p><?= $datum->content ?></p>
</div>
<?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>

<?php echo $this->placeholder('foo') ?>
```

```
<!-- Capture to key -->
<?php $this->placeholder('foo')->captureStart('SET', 'data');
foreach ($this->data as $datum): ?>
<div class="foo">
    <h2><?= $datum->title ?></h2>
    <p><?= $datum->content ?></p>
</div>
 <?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>

<?php echo $this->placeholder('foo')->data ?>
```

Concrete Placeholder Implementations

Zend Framework ships with a number of "concrete" placeholder implementations. These are for commonly used placeholders: doctype, page title, and various <head> elements. In all cases, calling the placeholder with no arguments returns the element itself.

Documentation for each element is covered separately, as linked below:

- Doctype
- HeadLink
- HeadMeta
- HeadScript
- HeadStyle
- HeadTitle
- InlineScript

Doctype Helper

Valid HTML and XHTML documents should include a `DOCTYPE` declaration. Besides being difficult to remember, these can also affect how certain elements in your document should be rendered (for instance, CDATA escaping in `<script>` and `<style>` elements.

The `Doctype` helper allows you to specify one of the following types:

- `XHTML11`
- `XHTML1_STRICT`
- `XHTML1_TRANSITIONAL`
- `XHTML1_FRAMESET`
- `XHTML_BASIC1`
- `HTML4_STRICT`
- `HTML4_LOOSE`
- `HTML4_FRAMESET`

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.

Example 50.8. Doctype Helper Basic Usage

You may specify the doctype at any time. However, helpers that depend on the doctype for their output will recognize it only after you have set it, so the easyest approach is to specify it in your bootstrap:

```
$doctypeHelper = new Zend_View_Helper_Doctype();
$doctypeHelper->doctype('XHTML1_STRICT');
```

And then print it out on top of your layout script:

```
<?php echo $this->doctype() ?>
```

Example 50.9. Retrieving the Doctype

If you need to know the doctype, you can do so by calling `getDoctype()` on the object, which is returned by invoking the helper.

```
$doctype = $view->doctype()->getDoctype();
```

Typically, you'll simply want to know if the doctype is XHTML or not; for this, the `isXhtml()` method will suffice:

```
if ($view->doctype()->isXhtml()) {
    // do something differently
}
```

HeadLink Helper

The HTML `<link>` element is increasingly used for linking a variety of resources for your site: stylesheets, feeds, favicons, trackbacks, and more. The `HeadLink` helper provides a simple interface for creating and aggregating these elements for later retrieval and output in your layout script.

The `HeadLink` helper has special methods for adding stylesheet links to its stack:

- `appendStylesheet($href, $media, $conditionalStylesheet)`
- `offsetSetStylesheet($index, $href, $media, $conditionalStylesheet)`
- `prependStylesheet($href, $media, $conditionalStylesheet)`
- `setStylesheet($href, $media, $conditionalStylesheet)`

The `$media` value defaults to 'screen', but may be any valid media value. `$conditionalStylesheet` is a boolean, and will be used at rendering time to determine if special comments should be included to prevent loading of the stylesheet on certain platforms.

Additionally, the `HeadLink` helper has special methods for adding 'alternate' links to its stack:

- `appendAlternate($href, $type, $title)`
- `offsetSetAlternate($index, $href, $type, $title)`
- `prependAlternate($href, $type, $title)`
- `setAlternate($href, $type, $title)`

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.

Example 50.10. HeadLink Helper Basic Usage

You may specify a `headLink` at any time. Typically, you will specify global links in your layout script, and application specific links in your application view scripts. In your layout script, in the <head> section, you will then echo the helper to output it.

```
<?php // setting links in a view script:
$this->headLink()->appendStylesheet('/styles/basic.css')
                 ->headLink(array('rel' => 'favicon',
                                  'href' => '/img/favicon.ico'),
                                  'PREPEND')
                 ->prependStylesheet('/styles/moz.css', 'screen', true);
?>
<?php // rendering the links: ?>
<?= $this->headLink() ?>
```

HeadMeta Helper

The HTML `<meta>` element is used to provide meta information about your HTML document -- typically keywords, document character set, caching pragmas, etc. Meta tags may be either of the 'http-equiv' or 'name' types, must contain a 'content' attribute, and can also have either of the 'lang' or 'scheme' modifier attributes.

The `HeadMeta` helper supports the following methods for setting and adding meta tags:

- `appendName($keyValue, $content, $conditionalName)`
- `offsetSetName($index, $keyValue, $content, $conditionalName)`
- `prependName($keyValue, $content, $conditionalName)`
- `setName($keyValue, $content, $modifiers)`

- `appendHttpEquiv($keyValue, $content, $conditionalHttpEquiv)`
- `offsetSetHttpEquiv($index, $keyValue, $content, $conditionalHttpEquiv)`
- `prependHttpEquiv($keyValue, $content, $conditionalHttpEquiv)`
- `setHttpEquiv($keyValue, $content, $modifiers)`

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 `$keyType`, which should be either 'name' or 'http-equiv'. `$placement` can be either '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.

Example 50.11. HeadMeta Helper Basic Usage

You may specify a new meta tag at any time. Typically, you will specify client-side caching rules or SEO keywords.

For instance, if you wish to specify SEO keywords, you'd be creating a meta name tag with the name 'keywords' and the content the keywords you wish to associate with your page:

```
// setting meta keywords
$this->headMeta()->appendName('keywords', 'framework php productivity');
```

If you wishedto set some client-side caching rules, you'd set http-equiv tags with the rules you wish to enforce:

```
// disabling client-side cache
$this->headMeta()->appendHttpEquiv('expires',
                                   'Wed, 26 Feb 1997 08:21:57 GMT')
                 ->appendHttpEquiv('pragma', 'no-cache')
                 ->appendHttpEquiv('Cache-Control', 'no-cache');
```

Another popular use for meta tags is setting the content type, character set, and language:

```
// setting content type and character set
$this->headMeta()->appendHttpEquiv('Content-Type',
                                   'text/html; charset=UTF-8')
                 ->appendHttpEquiv('Content-Language', 'en-US');
```

As a final example, an easy way to display a transitional message before a redirect is using a "meta refresh":

```
// setting a meta refresh for 3 seconds to a new url:
$this->headMeta()->appendHttpEquiv('Refresh',
                                   '3;URL=http://www.some.org/some.html');
```

When you're ready to place your meta tags in the layout, simply echo the helper:

```
<?= $this->headMeta() ?>
```

HeadScript Helper

The HTML `<script>` element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `HeadScript` helper allows you to manage both.

The `HeadScript` helper supports the following methods for setting and adding scripts:

- `appendFile($src, $type = 'text/javascript', $attrs = array())`
- `offsetSetFile($index, $src, $type = 'text/javascript', $attrs = array())`
- `prependFile($src, $type = 'text/javascript', $attrs = array())`
- `setFile($src, $type = 'text/javascript', $attrs = array())`
- `appendScript($script, $type = 'text/javascript', $attrs = array())`
- `offsetSetScript($index, $script, $type = 'text/javascript', $attrs = array())`
- `prependScript($script, $type = 'text/javascript', $attrs = array())`
- `setScript($script, $type = 'text/javascript', $attrs = array())`

In the case of the `*File()` methods, `$src` is the remote location of the script to load; this is usually in the form of a URL or a path. For the `*Script()` methods, `$script` is the client-side scripting directives you wish to use in the element.

`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, $placement = 'APPEND')`. The `$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'.

`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.

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.

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 [http://www.dojotoolkit.org/], utilize custom attributes in order to modify behavior. To allow such attributes, you can enable them via the `setAllowArbitraryAttributes()` method:

```
$this->headScript()->setAllowArbitraryAttributes(true);
```

Example 50.12. HeadScript Helper Basic Usage

You may specify a new script tag at any time. As noted above, these may be links to outside resource files or scripts themselves.

```
// adding scripts
$this->headScript()->appendFile('/js/prototype.js')
                   ->appendScript($onloadScript);
```

Order is often important with client-side scripting; you may need to ensure that libraries are loaded in a specific order due to dependencies each have; use the various append, prepend, and offsetSet directives to aid in this task:

```
// Putting scripts in order

// place at a particular offset to ensure loaded last
$this->headScript()->offsetSetScript(100, '/js/myfuncs.js');

// use scriptaculous effects (append uses next index, 101)
$this->headScript()->appendScript('/js/scriptaculous.js');

// but always have base prototype script load first:
$this->headScript()->prependScript('/js/prototype.js');
```

When you're finally ready to output all scripts in your layout script, simply echo the helper:

```
<?= $this->headScript() ?>
```

Example 50.13. Capturing Scripts Using the HeadScript Helper

Sometimes you need to generate client-side scripts programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the script and sprinkling in PHP tags. `HeadScript` lets you do just that, capturing it to the stack:

```
<?php $this->headScript()->captureStart() ?>
var action = '<?= $this->baseUrl ?>';
$('foo_form').action = action;
<?php $this->headScript()->captureEnd() ?>
```

The following assumptions are made:

- The script will be appended to the stack. If you wish for it to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- The script MIME type is assumed to be 'text/javascript'; if you wish to specify a different type, you will need to pass it as the second argument to `captureStart()`.
- If you wish to specify any additional attributes for the `<script>` tag, pass them in an array as the third argument to `captureStart()`.

HeadStyle Helper

The HTML `<style>` element is used to include CSS stylesheets inline in the HTML `<head>` element.

Use HeadLink to link CSS files

HeadLink should be used to create `<link>` elements for including external stylesheets. `HeadScript` 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 permissable.

`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($content$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.

Example 50.14. HeadStyle Helper Basic Usage

You may specify a new style tag at any time:

```
// adding styles
$this->headStyle()->appendStyle($styles);
```

Order is very important with CSS; you may need to ensure that declarations are loaded in a specific order due to the order of the cascade; use the various append, prepend, and offsetSet directives to aid in this task:

```
// Putting styles in order

// place at a particular offset:
$this->headStyle()->offsetSetStyle(100, $customStyles);

// place at end:
$this->headStyle()->appendStyle($finalStyles);

// place at beginning
$this->headStyle()->prependStyle($firstStyles);
```

When you're finally ready to output all style declarations in your layout script, simply echo the helper:

```
<?= $this->headStyle() ?>
```

Example 50.15. Capturing Style Declarations Using the HeadStyle Helper

Sometimes you need to generate CSS style declarations programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the styles and sprinkling in PHP tags. `HeadStyle` lets you do just that, capturing it to the stack:

```
<?php $this->headStyle()->captureStart() ?>
body {
    background-color: <?= $this->bgColor ?>;
}
<?php $this->headStyle()->captureEnd() ?>
```

The following assumptions are made:

- The style declarations will be appended to the stack. If you wish for them to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- If you wish to specify any additional attributes for the `<style>` tag, pass them in an array as the second argument to `captureStart()`.

HeadTitle Helper

The HTML `<title>` element is used to provide a title for an HTML document. The `HeadTitle` helper allows you to programmatically create and store the title for later retrieval and output.

The `HeadTitle` helper is a concrete implementation of the Placeholder helper. It overrides the `toString()` method to enforce generating a `<title>` element, and adds a `headTitle()` method for quick and easy setting and aggregation of title elements. The signature for that method is `headTitle($title, $setType = 'APPEND')`; by default, the value is appended to the stack (aggregating title segments), but you may also specify either 'PREPEND' (place at top of stack) or 'SET' (overwrite stack).

Example 50.16. HeadTitle Helper Basic Usage

You may specify a title tag at any time. A typical usage would have you setting title segments for each level of depth in your application: site, controller, action, and potentially resource.

```
 // setting the controller and action name as title segments:
$request = Zend_Controller_Front::getInstance()->getRequest();
$this->headTitle($request->getActionName())
     ->headTitle($request->getControllerName());

// setting the site in the title; possibly in the layout script:
$this->headTitle('Zend Framework');

// setting a separator string for segments:
$this->headTitle()->setSeparator(' / ');
```

When you're finally ready to render the title in your layout script, simply echo the helper:

```
<!-- renders <action> / <controller> / Zend Framework -->
<?= $this->headTitle() ?>
```

HTML Object Helpers

The HTML `<object>` element is used for embedding media like Flash or QuickTime in web pages. The object view helpers take care of embedding media with minimum effort.

There are four initial Object helpers:

- `formFlash` Generates markup for embedding Flash files.
- `formObject` Generates markup for embedding a custom Object.
- `formPage` Generates markup for embedding other (X)HTML pages.
- `formQuicktime` 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.

Example 50.17. Flash helper

Embedding Flash in your page using the helper is pretty straight-forward. The only required argument is the resource URI.

```
<?php echo $this->htmlFlash('/path/to/flash.swf'); ?>
```

This outputs the following HTML:

```
<object data="/path/to/flash.swf"
        type="application/x-shockwave-flash"
        classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.
</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.

Example 50.18. 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. The `classid` and `codebase` are examples of such attributes. The fourth argument also only takes a key-value array and uses them to create `<param>` elements. You will see an example of this shortly. Lastly, there is the option of providing additional content to the object. Now for an example which utilizes all arguments.

```
echo $this->htmlObject(
    '/path/to/file.ext',
    'mime/type',
    array(
        'attr1' => 'aval1',
        'attr2' => 'aval2'
    ),
    array(
        'param1' => 'pval1',
        'param2' => 'pval2'
    ),
    'some content'
);

/*
This would output:

<object data="/path/to/file.ext" type="mime/type"
    attr1="aval1" attr2="aval2">
    <param name="param1" value="pval1" />
    <param name="param2" value="pval2" />
    some content
</object>
*/
```

InlineScript Helper

The HTML `<script>` element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `InlineScript` helper allows you to manage both. It is derived from HeadScript, and any method of that helper is available; however, use the `inlineScript()` method in place of `headScript()`.

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.

JSON Helper

When creating views that return JSON, it's important to also set the appropriate response header. The JSON view helper does exactly that. In addition, by default, it disables layouts (if currently enabled), as layouts generally aren't used with JSON responses.

The JSON helper sets the following header:

```
Content-Type: application/json
```

Most AJAX libraries look for this header when parsing responses to determine how to handle the content.

Usage of the JSON helper is very straightforward:

```
<?= $this->json($this->data) ?>
```

Translate Helper

Often web sites are available in several languages. To translate the content of a site you should simply use Zend Translate and to integrate `Zend Translate` within your view you should use the `Translate` View Helper.

In all following examples we are using the simple Array Translation Adapter. Of course you can also use any instance of `Zend_Translate` and also any subclasses of `Zend_Translate_Adapter`. There are several ways to initiate the `Translate` View Helper:

- Registered, through a previously registered instance in `Zend_Registry`
- Afterwards, through the fluent interface
- Directly, through initiating the class

A registered instance of `Zend_Translate` is the preferred usage for this helper. You can also select the locale to be used simply before you add the adapter to the registry.

Note

We are speaking of locales instead of languages because a language also may contain a region. For example English is spoken in different dialects. There may be a translation for British and one for American English. Therefore, we say "locale" instead of "language."

Example 50.19. Registered instance

To use a registered instance just create an instance of `Zend_Translate` or `Zend_Translate_Adapter` and register it within `Zend_Registry` using `Zend_Translate` as its key.

```
// our example adapter
$adapter = new Zend_Translate('array', array('simple' => 'einfach'), 'de');
Zend_Registry::set('Zend_Translate', $adapter);

// within your view
echo $this->translate('simple');
// this returns 'einfach'
```

If you are more familiar with the fluent interface, then you can also create an instace within your view and initiate the helper afterwards.

Example 50.20. Within the view

To use the fluent interface, create an instance of `Zend_Translate` or `Zend_Translate_Adapter`, call the helper without a parameter, and call the `setTranslator()` method.

```
// within your view
$adapter = new Zend_Translate('array', array('simple' => 'einfach'), 'de');
$this->translate()->setTranslator($adapter)->translate('simple');
// this returns 'einfach'
```

If you are using the helper without `Zend_View` then you can also use it directly.

Example 50.21. Direct usage

```
// our example adapter
$adapter = new Zend_Translate('array', array('simple' => 'einfach'), 'de');

// initiate the adapter
$translate = new Zend_View_Helper_Translate($adapter);
print $translate->translate('simple'); // this returns 'einfach'
```

You would use this way if you are not working with `Zend_View` and need to create translated output.

As already seen, the `translate()` method is used to return the translation. Just call it with the needed messageid of your translation adapter. But it can also replace parameters within the translation string. Therefore, it accepts variable parameters in two ways: either as a list of parameters, or as an array of parameters. As examples:

Example 50.22. Single parameter

To use a single parameter just add it to the method.

```
// within your view
$date = "Monday";
$this->translate("Today is %1\$s", $date);
// could return 'Heute ist Monday'
```

Note

Keep in mind that if you are using parameters which are also text, you may also need to translate these parameters.

Example 50.23. List of parameters

Or use a list of parameters and add it to the method.

```
// within your view
$date = "Monday";
$month = "April";
$time = "11:20:55";
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s",
                 $date,
                 $month,
                 $time);
// Could return 'Heute ist Monday in April. Aktuelle Zeit: 11:20:55'
```

Example 50.24. Array of parameters

Or use an array of parameters and add it to the method.

```
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date);
// Could return 'Heute ist Monday in April. Aktuelle Zeit: 11:20:55'
```

Sometimes it is necessary to change the locale of the translation. This can be done either dynamically per translation or statically for all following translations. And you can use it with both a parameter list and an array of parameters. In both cases the locale must be given as the last single parameter.

Example 50.25. Change locale dynamically

```
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date, 'it');
```

This example returns the Italian translation for the messageid. But it will only be used once. The next translation will use the locale from the adapter. Normally you will set the desired locale within the translation adapter before you add it to the registry. But you can also set the locale from within the helper:

Example 50.26. Change locale statically

```
// within your view
$date = array("Monday", "April", "11:20:55");
$this->translate()->setLocale('it');
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date);
```

The above example sets `'it'` as the new default locale which will be used for all further translations.

Of course there is also a `getLocale()` method to get the currently set locale.

Example 50.27. Get the currently set locale

```
// within your view
$date = array("Monday", "April", "11:20:55");

// returns 'de' as set default locale from our above examples
$this->translate()->getLocale();

$this->translate()->setLocale('it');
$this->translate("Today is %1\$s in %2\$s. Actual time: %3\$s", $date);

// returns 'it' as new set default locale
$this->translate()->getLocale();
```

Helper Paths

As with view scripts, your controller can specify a stack of paths for `Zend_View` to search for helper classes. By default, `Zend_View` looks in "Zend/View/Helper/*" for helper classes. You can tell `Zend_View` to look in other locations using the `setHelperPath()` and `addHelperPath()` methods. Additionally, you can indicate a class prefix to use for helpers in the path provided, to allow namespacing your helper classes. By default, if no class prefix is provided, 'Zend_View_Helper_' is assumed.

```
$view = new Zend_View();
```

```
// Set path to /path/to/more/helpers, with prefix 'My_View_Helper'
$view->setHelperPath('/path/to/more/helpers', 'My_View_Helper');
```

In fact, you can "stack" paths using the `addHelperPath()` method. As you add paths to the stack, `Zend_View` will look at the most-recently-added path for the requested helper class. This allows you to add to (or even override) the initial distribution of helpers with your own custom helpers.

```
$view = new Zend_View();
// Add /path/to/some/helpers with class prefix 'My_View_Helper'
$view->addHelperPath('/path/to/some/helpers', 'My_View_Helper');
// Add /other/path/to/helpers with class prefix 'Your_View_Helper'
$view->addHelperPath('/other/path/to/helpers', 'Your_View_Helper');

// now when you call $this->helperName(), Zend_View will look first for
// "/path/to/some/helpers/HelperName" using class name
// "Your_View_Helper_HelperName", then for
// "/other/path/to/helpers/HelperName.php" using class name
// "My_View_Helper_HelperName", and finally for
// "Zend/View/Helper/HelperName.php" using class name
// "Zend_View_Helper_HelperName".
```

Writing Custom Helpers

Writing custom helpers is easy; just follow these rules:

- While not strictly necessary, we recommend either implementing `Zend_View_Helper_Interface` or extending `Zend_View_Helper_Abstract` when creating your helpers. Introduced in 1.6.0, these simply define a `setView()` method; however, in upcoming releases, we plan to implement a strategy pattern that will simplify much of the naming schema detailed below. Building off these now will help you future-proof your code.
- The class name must, at the very minimum, end with the helper name itself, using MixedCaps. E.g., if you were writing a helper called "specialPurpose", the class name would minimally need to be "SpecialPurpose". You may, and should, give the class name a prefix, and it is recommended that you use 'View_Helper' as part of that prefix: "My_View_Helper_SpecialPurpose". (You will need to pass in the prefix, with or without the trailing underscore, to `addHelperPath()` or `setHelperPath()`).
- The class must have a public method that matches the helper name; this is the method that will be called when your template calls "$this->specialPurpose()". In our "specialPurpose" helper example, the required method declaration would be "public function specialPurpose()".
- In general, the class should not echo or print or otherwise generate output. Instead, it should return values to be printed or echoed. The returned values should be escaped appropriately.
- The class must be in a file named after the helper class. Again using our "specialPurpose" helper example, the file has to be named "SpecialPurpose.php".

Place the helper class file somewhere in your helper path stack, and `Zend_View` will automatically load, instantiate, persist, and execute it for you.

Here is an example of our `SpecialPurpose` helper code:

```
class My_View_Helper_SpecialPurpose extends Zend_View_Helper_Abstract
{
    protected $_count = 0;
    public function specialPurpose()
    {
        $this->_count++;
        $output = "I have seen 'The Jerk' {$this->_count} time(s).";
        return htmlspecialchars($output);
    }
}
```

Then in 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 `Zend_View` instance.

```
// remember, in a view script, $this refers to the Zend_View instance.
echo $this->specialPurpose();
echo $this->specialPurpose();
echo $this->specialPurpose();
```

The output would look something like this:

```
I have seen 'The Jerk' 1 time(s).
I have seen 'The Jerk' 2 time(s).
I have seen 'The Jerk' 3 time(s).
```

Sometimes you will need access to the calling `Zend_View` object -- for instance, if you need to use the registered encoding, or want to render another view script as part of your helper. To get access to the view object, your helper class should have a `setView($view)` method, like the following:

```
class My_View_Helper_ScriptPath
{
    public $view;

    public function setView(Zend_View_Interface $view)
    {
        $this->view = $view;
    }

    public function scriptPath($script)
    {
        return $this->view->getScriptPath($script);
    }
}
```

If your helper class has a `setView()` method, it will be called when the helper class is first instantiated, and passed the current view object. It is up to you to persist the object in your class, as well as determine how it should be accessed.

If you are extending `Zend_View_Helper_Abstract`, you do not need to define this method, as it is defined for you.

Zend_View_Abstract

`Zend_View_Abstract` is the base class on which `Zend_View` is built; `Zend_View` itself simply extends it and declares a concrete implementation of the `_run()` method (which is invoked by `render()`).

Many developers find that they want to extend `Zend_View_Abstract` to add custom functionality, and inevitably run into issues with its design, which includes a number of private members. This document aims to explain the decision behind the design.

`Zend_View` is something of an anti-templating engine in that it uses PHP natively for its templating. As a result, all of PHP is available, and view scripts inherit the scope of their calling object.

It is this latter point that is salient to the design decisions. Internally, `Zend_View::_run()` does the following:

```
protected function _run()
{
    include func_get_arg(0);
}
```

As such, the view scripts have access to the current object (`$this`), *and any methods or members of that object*. Since many operations depend on members with limited visibility, this poses a problem: the view scripts could potentially make calls to such methods or modify critical properties directly. Imagine a script overwriting `$_path` or `$_file` inadvertently -- any further calls to `render()` or view helpers would break!

Fortunately, PHP 5 has an answer to this with its visibility declarations: private members are not accessible by objects extending a given class. This led to the current design: since `Zend_View` *extends* `Zend_View_Abstract`, view scripts are thus limited to only protected or public methods and members of `Zend_View_Abstract` -- effectively limiting the actions it can perform, and allowing us to secure critical areas from abuse by view scripts.

Chapter 51. Zend_Wildfire

Zend_Wildfire

`Zend_Wildfire` is a component that facilitates communication between PHP code and Wildfire [http://www.wildfirehq.org/] client components.

The purpose of the Wildfire Project is to develop standardized communication channels between a large variety of components and a dynamic and scriptable plugin architecture. At this time the primary focus is to provide a system to allow server-side PHP code to inject logging messages into the Firebug [http://www.getfirebug.com/] Console [http://getfirebug.com/logging.html].

For the purpose of logging to Firebug the `Zend_Log_Writer_Firebug` component is provided and a communication protocol has been developed that uses HTTP request and response headers to send data between the server and client components. It is great for logging intelligence data, generated during script execution, to the browser without interfering with the page content. Debugging AJAX requests that require clean JSON and XML responses is possible with this approach.

There is also a `Zend_Db_Profiler_Firebug` component to log database profiling information to Firebug.

Chapter 52. Zend_XmlRpc

Introduction

From its home page [http://www.xmlrpc.com/], 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."

The Zend Framework provides support for both consuming remote XML-RPC services and building new XML-RPC servers.

Zend_XmlRpc_Client

Introduction

The 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.

Example 52.1. XML-RPC Method Call

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

echo $client->call('test.sayHello');

// hello
```

The XML-RPC value returned from the remote method call will be automatically unmarshaled and cast to the equivalent PHP native type. In the example above, a PHP `string` is returned and is immediately ready to be used.

The first parameter of the `call()` method receives the name of the remote method to call. If the remote method requires any parameters, these can be sent by supplying a second, optional parameter to `call()` with an `array` of values to pass to the remote method:

Example 52.2. XML-RPC Method Call with Parameters

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$arg1 = 1.1;
$arg2 = 'foo';

$result = $client->call('test.sayHello', array($arg1, $arg2));

// $result is a native PHP type
```

If the remote method doesn't require parameters, this optional parameter may either be left out or an empty `array()` passed to it. The array of parameters for the remote method can contain native PHP types, `Zend_XmlRpc_Value` objects, or a mix of each.

The `call()` method will automatically convert the XML-RPC response and return its equivalent PHP native type. A `Zend_XmlRpc_Response` object for the return value will also be available by calling the `getLastResponse()` method after the call.

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 52.1. PHP and XML-RPC Type Conversions

PHP Native Type	XML-RPC Type
integer	int
double	double
boolean	boolean
string	string
array	array
associative array	struct
object	array

What type do empty arrays get cast to?

Passing an empty array to an XML-RPC method is problematic, as it could represent either an array or a struct. `Zend_XmlRpc_Client` detects such conditions and makes a request to the server's `system.methodSignature` method to determine the appropriate XML-RPC type to cast to.

However, this in itself can lead to issues. First off, servers that do not support `system.methodSignature` will log failed requests, and `Zend_XmlRpc_Client` will resort to casting the value to an XML-RPC array type. Additionally, this means that any call with array arguments will result in an additional call to the remote server.

To disable the lookup entirely, you can call the `setSkipSystemLookup()` method prior to making your XML-RPC call:

```
$client->setSkipSystemLookup(true);
$result = $client->call('foo.bar', array(array()));
```

`Zend_XmlRpc_Value` Objects as Parameters

Parameters may also be created as `Zend_XmlRpc_Value` instances to specify an exact XML-RPC type. The primary reasons for doing this are:

- When you want to make sure the correct parameter type is passed to the procedure (i.e. the procedure requires an integer and you may get it from a database as a string)
- When the procedure requires `base64` or `dateTime.iso8601` type (which doesn't exists as a PHP native type)
- When auto-conversion may fail (i.e. you want to pass an empty XML-RPC struct as a parameter. Empty structs are represented as empty arrays in PHP but, if you give an empty array as a parameter it will be auto-converted to an XML-RPC array since it's not an associative array)

There are two ways to create a `Zend_XmlRpc_Value` object: instantiate one of the `Zend_XmlRpc_Value` subclasses directly, or use the static factory method `Zend_XmlRpc_Value::getXmlRpcValue()`.

Table 52.2. `Zend_XmlRpc_Value` Objects for XML-RPC Types

XML-RPC Type	`Zend_XmlRpc_Value` Constant	`Zend_XmlRpc_Value` Object
int	`Zend_XmlRpc_Value::XMLRPC_TYPE_INTEGER`	`Zend_XmlRpc_Value_Integer`
double	`Zend_XmlRpc_Value::XMLRPC_TYPE_DOUBLE`	`Zend_XmlRpc_Value_Double`
boolean	`Zend_XmlRpc_Value::XMLRPC_TYPE_BOOLEAN`	`Zend_XmlRpc_Value_Boolean`
string	`Zend_XmlRpc_Value::XMLRPC_TYPE_STRING`	`Zend_XmlRpc_Value_String`
base64	`Zend_XmlRpc_Value::XMLRPC_TYPE_BASE64`	`Zend_XmlRpc_Value_Base64`
dateTime.iso8601	`Zend_XmlRpc_Value::XMLRPC_TYPE_DATETIME`	`Zend_XmlRpc_Value_DateTime`
array	`Zend_XmlRpc_Value::XMLRPC_TYPE_ARRAY`	`Zend_XmlRpc_Value_Array`
struct	`Zend_XmlRpc_Value::XMLRPC_TYPE_STRUCT`	`Zend_XmlRpc_Value_Struct`

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.

Example 52.3. Proxy the Default Namespace

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$server = $client->getProxy();           // Proxy the default namespace

$hello = $server->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:

Example 52.4. Proxy Any Namespace

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$test  = $client->getProxy('test');     // Proxy the "test" namespace

$hello = $test->sayHello(1, 2);         // test.Hello(1,2) returns "hello"
```

If the remote server supports nested namespaces of any depth, these can also be used through the server proxy. For example, if the server in the example above had a method `test.foo.bar()`, it could be called as `$test->foo->bar()`.

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_HttpException` will be thrown.

Example 52.5. Handling HTTP Errors

```
$client = new Zend_XmlRpc_Client('http://foo/404');

try {

    $client->call('bar', array($arg1, $arg2));

} catch (Zend_XmlRpc_Client_HttpException $e) {

    // $e->getCode() returns 404
    // $e->getMessage() returns "Not Found"

}
```

Regardless of how the XML-RPC client is used, the `Zend_XmlRpc_Client_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_FaultException` being thrown. The code and message of the exception will map directly to their respective values in the original XML-RPC fault response.

Example 52.6. Handling XML-RPC Faults

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

try {

    $client->call('badMethod');

} catch (Zend_XmlRpc_Client_FaultException $e) {

    // $e->getCode() returns 1
    // $e->getMessage() returns "Unknown method"

}
```

When the `call()` method is used to make the request, the `Zend_XmlRpc_Client_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_XmlRpcClient`. It can then be used to perform introspection operations 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:

Example 52.7. Processing Request to Response

```
$client = new Zend_XmlRpc_Client('http://framework.zend.com/xmlrpc');

$request = new Zend_XmlRpc_Request();
$request->setMethod('test.sayHello');
$request->setParams(array('foo', 'bar'));

$client->doRequest($request);

// $server->getLastRequest() returns instanceof Zend_XmlRpc_Request
// $server->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.

Zend_XmlRpc_Server

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/spec]. Additionally, it implements the system.multicall() method, allowing boxcarring of requests.

Basic Usage

An example of the most basic use case:

```
$server = new Zend_XmlRpc_Server();
$server->setClass('My_Service_Class');
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_Fault`object. These objects both have `__toString()` methods that create valid XML-RPC XML responses, allowing them to be directly echoed.

Conventions

Zend_XmlRpc_Server allows the developer to attach functions and class method calls as dispatchable XML-RPC methods. Via Zend_Server_Reflection, it does introspection on all attached methods, using the function and method docblocks to determine the method help text and method signatures.

XML-RPC types do not necessarily map one-to-one to PHP types. However, the code will do its best to guess the appropriate type based on the values listed in @param and @return lines. Some XML-RPC types have no immediate PHP equivalent, however, and should be hinted using the XML-RPC type in the phpdoc. These include:

- dateTime.iso8601, a string formatted as YYYYMMDDTHH:mm:ss
- base64, base64 encoded data
- struct, any associative array

An example of how to hint follows:

```
/**
* This is a sample function
*
* @param base64 $val1 Base64-encoded data
* @param dateTime.iso8601 $val2 An ISO date
* @param struct $val3 An associative array
* @return struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

PhpDocumentor does no validation of the types specified for params or return values, so this will have no impact on your API documentation. Providing the hinting is necessary, however, when the server is validating the parameters provided to the method call.

It is perfectly valid to specify multiple types for both params and return values; the XML-RPC specification even suggests that system.methodSignature should return an array of all possible method signatures (i.e., all possible combinations of param and return values). You may do so just as you normally would with PhpDocumentor, using the '|' operator:

```
/**
* This is a sample function
*
* @param string|base64 $val1 String or base64-encoded data
* @param string|dateTime.iso8601 $val2 String or an ISO date
* @param array|struct $val3 Normal indexed array or an associative array
* @return boolean|struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

One note, however: allowing multiple signatures can lead to confusion for developers using the services; generally speaking, an XML-RPC 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:

```
// All public methods in My_Service_Class will be accessible as
// myservice.METHODNAME
$server->setClass('My_Service_Class', 'myservice');

// Function 'somefunc' will be accessible as funcs.somefunc
$server->addFunction('somefunc', 'funcs');
```

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:

```
function __autoload($class)
{
    Zend_Loader::loadClass($class);
}
```

```
$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
$server = new Zend_XmlRpc_Server();

if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {
    require_once 'My/Services/Glue.php';
    require_once 'My/Services/Paste.php';
    require_once 'My/Services/Tape.php';

    $server->setClass('My_Services_Glue', 'glue');   // glue. namespace
    $server->setClass('My_Services_Paste', 'paste'); // paste. namespace
    $server->setClass('My_Services_Tape', 'tape');   // tape. namespace

    Zend_XmlRpc_Server_Cache::save($cacheFile, $server);
}

echo $server->handle();
```

The above example attempts to retrieve a server definition from xmlrpc.cache in the same directory as the script. If unsuccessful, it loads the service classes it needs, attaches them to the server instance, and then attempts to create a new cache file with the server definition.

Usage Examples

Below are several usage examples, showing the full spectrum of options available to developers. Usage examples will each build on the previous example provided.

Basic Usage

The example below attaches a function as a dispatchable XML-RPC method and handles incoming calls.

```
/**
 * Return the MD5 sum of a value
 *
 * @param string $value Value to md5sum
 * @return string MD5 sum of value
 */
function md5Value($value)
{
    return md5($value);
}

$server = new Zend_XmlRpc_Server();
$server->addFunction('md5Value');
echo $server->handle();
```

Attaching a class

The example below illustrates attaching a class' public methods as dispatchable XML-RPC methods.

```
require_once 'Services/Comb.php';

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb');
echo $server->handle();
```

Attaching several classes using namespaces

The example below illustrates attaching several classes, each with their own namespace.

```
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*
echo $server->handle();
```

Specifying exceptions to use as valid fault responses

The example below allows any Services_Exception-derived class to report its code and message in the fault response.

```
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*
echo $server->handle();
```

Utilizing a custom request object

The example below instantiates a custom request object and passes it to the server to handle.

```
require_once 'Services/Request.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*

// Create a request object
$request = new Services_Request();

echo $server->handle($request);
```

Utilizing a custom response object

The example below illustrates specifying a custom response class for the returned response.

```
require_once 'Services/Request.php';
require_once 'Services/Response.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();
$server->setClass('Services_Comb', 'comb');   // methods called as comb.*
$server->setClass('Services_Brush', 'brush'); // methods called as brush.*
$server->setClass('Services_Pick', 'pick');   // methods called as pick.*

// Create a request object
$request = new Services_Request();

// Utilize a custom response
$server->setResponseClass('Services_Response');

echo $server->handle($request);
```

Cache server definitions between requests

The example below illustrates caching server definitions between requests.

```
// Specify a cache file
$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';

// Allow Services_Exceptions to report as fault responses
Zend_XmlRpc_Server_Fault::attachFaultException('Services_Exception');

$server = new Zend_XmlRpc_Server();

// Attempt to retrieve server definition from cache
if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {
    $server->setClass('Services_Comb', 'comb');   // methods called as comb.*
    $server->setClass('Services_Brush', 'brush'); // methods called as brush.*
    $server->setClass('Services_Pick', 'pick');   // methods called as pick.*

    // Save cache
    Zend_XmlRpc_Server_Cache::save($cacheFile, $server));
}

// Create a request object
$request = new Services_Request();

// Utilize a custom response
$server->setResponseClass('Services_Response');

echo $server->handle($request);
```

Appendix A. Zend Framework Requirements

Zend Framework requires a PHP 5 interpreter with a web server configured to handle PHP scripts correctly. Some features require additional extensions or web server features; in most cases the framework can be used without them, although performance may suffer or ancillary features may not be fully functional. An example of such a dependency is mod_rewrite in an Apache environment, which can be used to implement "pretty URL's" like "http://www.example.com/user/edit". If mod_rewrite is not enabled, ZF can be configured to support URL's such as "http://www.example.com?controller=user&action=edit". Pretty URL's may be used to shorten URL's for textual representation or search engine optimization (SEO), but they do not directly affect the functionality of the application.

PHP Version

Zend recommends PHP 5.2.3 or higher for critical security and performance enhancements, although Zend Framework requires only PHP 5.1.4 or later.

Zend Framework has an extensive collection of unit tests, which you can run using PHPUnit 3.0 or later.

PHP Extensions

You will find a table listing all extensions typically found in PHP and how they are used in Zend Framework below. You should verify that the extensions on which ZF components you'll be using in your application are available in your PHP environments. Many applications will not require every extension listed below.

A dependency of type "hard" indicates that the components or classes cannot function properly if the respective extension is not available, while a dependency of type "soft" indicates that the component may use the extension if it is available but will function properly if it is not. Many components will automatically use certain extensions if they are available to optimize performance but will execute code with similar functionality in the component itself if the extensions are unavailable.

Table A.1. PHP Extensions Used in Zend Framework by Component

Extension	Dependency Type	Used by Zend Framework Components
`apc` [http://www.php.net/manual/en/ref.apc.php]	Hard	`Zend_Cache_Back` [http://framework.zend.com/manual/en/zend.cach
`bcmath` [http://www.php.net/manual/en/ref.bc.php]	Soft	`Zend_Locale` [http://framework.zend.com/ma
`bitset` [http://pecl.php.net/package/Bitset]	Soft	`Zend_Search_` [http://framework.zend.com/manual/en/zend.sear
`bz2` [http://www.php.net/manual/en/ref.bzip2.php]	---	---
`calendar` [http://www.php.net/manual/en/ref.calendar.php]	---	---
`com_dotnet` [http://www.php.net/manual/en/ref.com.php]	---	---
`ctype` [http://www.php.net/manual/en/ref.ctype.php]	Hard	`Zend_Auth_Adapt` [http://framework.zend.com/manual/en/zend.auth
		`Zend_Gdata` [http://framework.zend.com/manu
		`Zend_Http_C` [http://framework.zend.com/manual/en/zend.http
		`Zend_Pdf` [http://framework.zend.com/manual/
		`Zend_Rest_C` [http://framework.zend.com/manual/en/zend.rest
		`Zend_Rest_S` [http://framework.zend.com/manual/en/zend.rest
		`Zend_Search_` [http://framework.zend.com/manual/en/zend.sear
		`Zend_Uri` [http://framework.zend.com/manual/
		`Zend_Vali` [http://framework.zend.com/manual/en/zend.vali
`curl` [http://www.php.net/manual/en/ref.curl.php]	Hard	`Zend_Http_Client_Ad` [http://framework.zend.com/manual/en/zend.http
`date` [http://www.php.net/manual/en/ref.datetime.php]	---	---
`dba` [http://www.php.net/manual/en/ref.dba.php]	---	---
`dbase` [http://www.php.net/manual/en/ref.dbase.php]	---	---

Extension	Dependency Type	Used by Zend Framework Components
dom [http://www.php.net/manual/en/ref.dom.php]	Hard	Zend_Feed [http://framework.zend.com/manua
		Zend_Gdata [http://framework.zend.com/manu
		Zend_Log_Format [http://framework.zend.com/manual/en/zend.log.
		Zend_Rest_S [http://framework.zend.com/manual/en/zend.rest
		Zend_Search_ [http://framework.zend.com/manual/en/zend.sear
		Zend_Service_ [http://framework.zend.com/manual/en/zend.serv
		Zend_Service_De [http://framework.zend.com/manual/en/zend.serv
		Zend_Service_ [http://framework.zend.com/manual/en/zend.serv
		Zend_Service [http://framework.zend.com/manual/en/zend.serv
		Zend_Service [http://framework.zend.com/manual/en/zend.serv
		Zend_XmlRpc [http://framework.zend.com/ma
exif [http://www.php.net/manual/en/ref.exif.php]	---	---
fbsql [http://www.php.net/manual/en/ref.fbsql.php]	---	---
fdf [http://www.php.net/manual/en/ref.fdf.php]	---	---
filter [http://www.php.net/manual/en/ref.filter.php]	---	---
ftp [http://www.php.net/manual/en/ref.ftp.php]	---	---
gd [http://www.php.net/manual/en/ref.image.php]	Hard	Zend_Pdf [http://framework.zend.com/manual/
gettext [http://www.php.net/manual/en/ref.gettext.php]	---	---
gmp [http://www.php.net/manual/en/ref.gmp.php]	---	---
hash [http://www.php.net/manual/en/ref.hash.php]	Hard	Zend_Auth_Adapt [http://framework.zend.com/manual/en/zend.auth
ibm_db2 [http://www.php.net/manual/en/ref.ibm-db2.php]	Hard	Zend_Db_Adapt [http://framework.zend.com/manual/en/zend.db.h

Extension	**Dependency Type**	**Used by Zend Framework Components**
`iconv` [http://www.php.net/manual/en/ref.iconv.php]	Hard	`Zend_Curr` [http://framework.zend.com/manual/en/zend.curr
		`Zend_Locale_` [http://framework.zend.com/manual/en/zend.loca
		`Zend_Mime` [http://framework.zend.com/manua
		`Zend_Pdf` [http://framework.zend.com/manual/
		`Zend_Search_` [http://framework.zend.com/manual/en/zend.sear
		`Zend_Service_Audio` [http://framework.zend.com/manual/en/zend.serv
		`Zend_Service_` [http://framework.zend.com/manual/en/zend.serv
		`Zend_XmlRpc_` [http://framework.zend.com/manual/en/zend.xml
`imap` [http://www.php.net/manual/en/ref.imap.php]	---	---
`informix` [http://www.php.net/manual/en/ref.ifx.php]	---	---
`interbase` [http://www.php.net/manual/en/ref.ibase.php]	Hard	Zend_Db_Adapter_Firebird
`json` [http://www.php.net/manual/en/ref.json.php]	Soft	`Zend_Json` [http://framework.zend.com/manua
`ldap` [http://www.php.net/manual/en/ref.ldap.php]	---	---
`libxml` [http://www.php.net/manual/en/ref.libxml.php]	Hard	`DOM` [http://www.php.net/manual/en/ref.dom.php
		`SimpleXML` [http://www.php.net/manual/en/ref
		`XSLT` [http://www.php.net/manual/en/ref.xslt.ph
`mbstring` [http://www.php.net/manual/en/ref.mbstring.php]	Hard	`Zend_Feed` [http://framework.zend.com/manua
`mcrypt` [http://www.php.net/manual/en/ref.mcrypt.php]	---	---
`memcache` [http://www.php.net/manual/en/ref.memcache.php]	Hard	`Zend_Cache_Backend_` [http://framework.zend.com/manual/en/zend.cach
`mhash` [http://www.php.net/manual/en/ref.mhash.php]	---	---
`mime_magic` [http://www.php.net/manual/en/ref.mime-magic.php]	Hard	`Zend_Http_C` [http://framework.zend.com/manual/en/zend.http
`ming` [http://www.php.net/manual/en/ref.ming.php]	---	---
`msql` [http://www.php.net/manual/en/ref.msql.php]	---	---
`mssql` [http://www.php.net/manual/en/ref.mssql.php]	---	---
`mysql` [http://www.php.net/manual/en/ref.mysql.php]	---	---
`mysqli` [http://www.php.net/manual/en/ref.mysqli.php]	Hard	`Zend_Db_Adapter` [http://framework.zend.com/manual/en/zend.db.h
`ncurses` [http://www.php.net/manual/en/ref.ncurses.php]	---	---
`oci8` [http://www.php.net/manual/en/ref.oci8.php]	Hard	`Zend_Db_Adapter` [http://framework.zend.com/manual/en/zend.db.h
`odbc` [http://www.php.net/manual/en/ref.uodbc.php]	---	---

Extension	Dependency Type	Used by Zend Framework Components
`openssl` [http://www.php.net/manual/en/ref.openssl.php]	---	---
`pcntl` [http://www.php.net/manual/en/ref.pcntl.php]	---	---
`pcre` [http://www.php.net/manual/en/ref.pcre.php]	Hard	Virtually all components
`pdo` [http://www.php.net/manual/en/ref.pdo.php]	Hard	All PDO database adapters
`pdo_dblib` [http://www.php.net/manual/en/ref.pdo-dblib.php]	---	---
`pdo_firebird` [http://www.php.net/manual/en/ref.pdo-firebird.php]	---	---
`pdo_mssql`	Hard	`Zend_Db_Adapter_P` [http://framework.zend.com/manual/en/zend.db.h
`pdo_mysql` [http://www.php.net/manual/en/ref.pdo-mysql.php]	Hard	`Zend_Db_Adapter_P` [http://framework.zend.com/manual/en/zend.db.h
`pdo_oci` [http://www.php.net/manual/en/ref.pdo-oci.php]	Hard	`Zend_Db_Adapter_` [http://framework.zend.com/manual/en/zend.db.h
`pdo_pgsql` [http://www.php.net/manual/en/ref.pdo-pgsql.php]	Hard	`Zend_Db_Adapter_P` [http://framework.zend.com/manual/en/zend.db.h
`pdo_sqlite` [http://www.php.net/manual/en/ref.pdo-sqlite.php]	Hard	`Zend_Db_Adapter_P` [http://framework.zend.com/manual/en/zend.db.h
`pgsql` [http://www.php.net/manual/en/ref.pgsql.php]	---	---
`posix` [http://www.php.net/manual/en/ref.posix.php]	Soft	`Zend_Mail` [http://framework.zend.com/manua
`pspell` [http://www.php.net/manual/en/ref.pspell.php]	---	---
`readline` [http://www.php.net/manual/en/ref.readline.php]	---	---
`recode` [http://www.php.net/manual/en/ref.recode.php]	---	---
`Reflection` [http://www.php.net/manual/en/language.oop5.reflection.php]	Hard	`Zend_Contr` [http://framework.zend.com/manual/en/zend.con
		`Zend_Filter` [http://framework.zend.com/ma
		`Zend_Filter_` [http://framework.zend.com/manual/en/zend.filte
		`Zend_Json` [http://framework.zend.com/manua
		`Zend_Log` [http://framework.zend.com/manual/
		`Zend_Rest_S` [http://framework.zend.com/manual/en/zend.rest
		`Zend_Server_Ref` [http://framework.zend.com/manual/en/zend.serv
		`Zend_Vali` [http://framework.zend.com/manual/en/zend.vali
		`Zend_View` [http://framework.zend.com/manua
		`Zend_XmlRpc_` [http://framework.zend.com/manual/en/zend.xml

Extension	Dependency Type	Used by Zend Framework Components
`session` [http://www.php.net/manual/en/ref.session.php]	Hard	`Zend_Controller_Action_Hel` [http://framework.zend.com/manual/en/zend.con
		`Zend_Session` [http://framework.zend.com/ma
`shmop` [http://www.php.net/manual/en/ref.shmop.php]	---	
`SimpleXML` [http://www.php.net/manual/en/ref.simplexml.php]	Hard	`Zend_Confi` [http://framework.zend.com/manual/en/zend.con
		`Zend_Feed` [http://framework.zend.com/manua
		`Zend_Rest_C` [http://framework.zend.com/manual/en/zend.rest
		`Zend_Service_Audio` [http://framework.zend.com/manual/en/zend.serv
		`Zend_XmlRpc` [http://framework.zend.com/ma
`soap` [http://www.php.net/manual/en/ref.soap.php]	Hard	`Zend_Service_St` [http://framework.zend.com/manual/en/zend.serv
`sockets` [http://www.php.net/manual/en/ref.sockets.php]	---	---
`SPL` [http://www.php.net/manual/en/ref.spl.php]	Hard	Virtually all components
`SQLite` [http://www.php.net/manual/en/ref.sqlite.php]	Hard	`Zend_Cache_Backe` [http://framework.zend.com/manual/en/zend.cac
`standard`	Hard	Virtually all components
`sybase` [http://www.php.net/manual/en/ref.sybase.php]	---	---
sysvmsg	---	---
sysvsem	---	--
sysvshm	---	---
`tidy` [http://www.php.net/manual/en/ref.tidy.php]	---	---
`tokenizer` [http://www.php.net/manual/en/ref.tokenizer.php]	---	---
`wddx` [http://www.php.net/manual/en/ref.wddx.php]	---	---
`xml` [http://www.php.net/manual/en/ref.xml.php]	Hard	`Zend_Translate_Ad` [http://framework.zend.com/manual/en/zend.tran
		`Zend_Translate_Ad` [http://framework.zend.com/manual/en/zend.tran
		`Zend_Translate_Adap` [http://framework.zend.com/manual/en/zend.tran
`XMLReader` [http://www.php.net/manual/en/ref.xmlreader.php]	---	---
`xmlrpc` [http://www.php.net/manual/en/ref.xmlrpc.php]	---	---
`XMLWriter` [http://www.php.net/manual/en/ref.xmlwriter.php]	---	---
`xsl` [http://www.php.net/manual/en/ref.xsl.php]	---	---
`zip` [http://www.php.net/manual/en/ref.zip.php]	---	---

Extension	Dependency Type	Used by Zend Framework Components
`zlib` [http://www.php.net/manual/en/ref.zlib.php]	Hard	`Zend_Pdf` [http://framework.zend.com/manual/
		`Memcache` [http://www.php.net/manual/en/ref.m

Zend Framework Components

Below is a table that lists all available Zend Framework Components and which PHP extension they need. This can help guide you to know which extensions are required for your application. Not all extensions used by Zend Framework are required for every application.

A dependency of type "hard" indicates that the components or classes cannot function properly if the respective extension is not available, while a dependency of type "soft" indicates that the component may use the extension if it is available but will function properly if it is not. Many components will automatically use certain extensions if they are available to optimize performance but will execute code with similar functionality in the component itself if the extensions are unavailable.

Table A.2. Zend Framework Components and the PHP Extensions they use

Zend Framework Components	Dependency Type	Subclass
All Components	Hard	---
`Zend_Acl` [http://framework.zend.com/manual/en/zend.acl.html]	---	---
`Zend_Auth` [http://framework.zend.com/manual/en/zend.auth.html]	Hard	Z e n d _ A u t h _ A [http://framework.zend.com/manual
`Zend_Cache` [http://framework.zend.com/manual/en/zend.cache.html]	Hard	Z e n d _ C a c h e _ [http://framework.zend.com/manual
		Z e n d _ C a c h e _ B a c [http://framework.zend.com/manual
		Z e n d _ C a c h e _ B [http://framework.zend.com/manual
		Z e n d _ C a c h e _ [http://framework.zend.com/manual
`Zend_Config` [http://framework.zend.com/manual/en/zend.config.html]	Hard	Z e n d _ C o r [http://framework.zend.com/manual
`Zend_Console_Getopt` [http://framework.zend.com/manual/en/zend.console.getopt.html]	---	---
`Zend_Controller` [http://framework.zend.com/manual/en/zend.controller.html]	Hard	---
		Zend_Controller_Acti [http://framework.zend.com/manual
`Zend_Currency` [http://framework.zend.com/manual/en/zend.currency.html]	Hard	---
`Zend_Date` [http://framework.zend.com/manual/en/zend.date.html]	---	---

Zend Framework Components	**Dependency Type**	**Subclass**
`Zend_Db` [http://framework.zend.com/manual/en/zend.db.html]	Hard	All PDO Adapters
		`Zend_Db_A` [http://framework.zend.com/manual
		`Zend_Db_Ada` [http://framework.zend.com/manual
		`Zend_Db_Ada` [http://framework.zend.com/manual
		`Zend_Db_Adap` [http://framework.zend.com/manual
		`Zend_Db_Adap` [http://framework.zend.com/manual
		`Zend_Db_Ada` [http://framework.zend.com/manual
		`Zend_Db_Adap` [http://framework.zend.com/manual
		`Zend_Db_Adapt` [http://framework.zend.com/manual
`Zend_Debug` [http://framework.zend.com/manual/en/zend.debug.html]	---	---
`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]	---	---
`Zend_Feed` [http://framework.zend.com/manual/en/zend.feed.html]	Hard	---
`Zend_Filter` [http://framework.zend.com/manual/en/zend.filter.html]	Hard	---
`Zend_Form` [http://framework.zend.com/manual/en/zend.form.html]	---	---
`Zend_Gdata` [http://framework.zend.com/manual/en/zend.gdata.html]	Hard	`Zend_Gdata_App` [http://framewor
		---
`Zend_Http` [http://framework.zend.com/manual/en/zend.http.html]	Hard	`Zend_Http_Clie` [http://framework.zend.com/manual
		`Zend_Htt` [http://framework.zend.com/manual
`Zend_InfoCard` [http://framework.zend.com/manual/en/zend.infocard.html]	---	---

Zend Framework Components	**Dependency Type**	**Subclass**
`Zend_Json` **[http://framework.zend.com/manual/en/zend.json.html]**	Soft	---
	Hard	---
`Zend_Layout` **[http://framework.zend.com/manual/en/zend.layout.html]**	---	---
`Zend_Ldap` **[http://framework.zend.com/manual/en/zend.ldap.html]**	---	---
`Zend_Loader` **[http://framework.zend.com/manual/en/zend.loader.html]**	---	---
`Zend_Locale` **[http://framework.zend.com/manual/en/zend.locale.html]**	Soft	`Zend_Loc` [http://framework.zend.com/manual
	Hard	`Zend_Loca` [http://framework.zend.com/manual
`Zend_Log` **[http://framework.zend.com/manual/en/zend.log.html]**	Hard	`Zend_Log_Fo` [http://framework.zend.com/manual
		---
`Zend_Mail` **[http://framework.zend.com/manual/en/zend.mail.html]**	Soft	---
`Zend_Measure` **[http://framework.zend.com/manual/en/zend.measure.html]**	---	---
`Zend_Memory` **[http://framework.zend.com/manual/en/zend.memory.html]**	---	---
`Zend_Mime` **[http://framework.zend.com/manual/en/zend.mime.html]**	Hard	`Zend_Mim` [http://framework.zend.com/manual
`Zend_OpenId` **[http://framework.zend.com/manual/en/zend.openid.html]**	---	---
`Zend_Pdf` **[http://framework.zend.com/manual/en/zend.pdf.html]**	Hard	---
`Zend_Registry` **[http://framework.zend.com/manual/en/zend.registry.html]**	---	---
`Zend_Request` **[http://framework.zend.com/manual/en/zend.request.html]**	---	---
`Zend_Rest` **[http://framework.zend.com/manual/en/zend.rest.html]**	Hard	`Zend_Res` [http://framework.zend.com/manual
		`Zend_Res` [http://framework.zend.com/manual

Zend Framework Components	Dependency Type	Subclass
Zend_Search_Lucene [http://framework.zend.com/manual/en/zend.search.lucene.html]	Soft	---
	Hard	
Zend_Server_Reflection [http://framework.zend.com/manual/en/zend.server.reflection.html]	Hard	---
Zend_Service_Akismet [http://framework.zend.com/manual/en/zend.service.akismet.html]	---	---
Zend_Service_Amazon [http://framework.zend.com/manual/en/zend.service.amazon.html]	Hard	---
Zend_Service_Audioscrobbler [http://framework.zend.com/manual/en/zend.service.audioscrobbler.html]	Hard	---
Zend_Service_Delicious [http://framework.zend.com/manual/en/zend.service.delicious.html]	Hard	---
Zend_Service_Flickr [http://framework.zend.com/manual/en/zend.service.flickr.html]	Hard	---
Zend_Service_Nirvanix [http://framework.zend.com/manual/en/zend.service.nirvanix.html]	---	---
Zend_Service_Simpy [http://framework.zend.com/manual/en/zend.service.simpy.html]	Hard	---
Zend_Service_SlideShare [http://framework.zend.com/manual/en/zend.service.slideshare.html]	---	---
Zend_Service_StrikeIron [http://framework.zend.com/manual/en/zend.service.strikeiron.html]	Hard	---
Zend_Service_Technorati [http://framework.zend.com/manual/en/zend.service.technorati.html]	---	---
Zend_Service_Yahoo [http://framework.zend.com/manual/en/zend.service.yahoo.html]	Hard	---
Zend_Session [http://framework.zend.com/manual/en/zend.session.html]	Hard	---
Zend_TimeSync [http://framework.zend.com/manual/en/zend.timesync.html]	---	---

Zend Framework Components	Dependency Type	Subclass
Zend_Translate [http://framework.zend.com/manual/en/zend.translate.html]	Hard	Zend_Transla [http://framework.zend.com/manual
		Zend_Translat [http://framework.zend.com/manual
		Zend_Translate [http://framework.zend.com/manual
Zend_Uri [http://framework.zend.com/manual/en/zend.uri.html]	Hard	---
Zend_Validate [http://framework.zend.com/manual/en/zend.validate.html]	Hard	---
Zend_Version [http://framework.zend.com/manual/en/zend.version.html]	---	---
Zend_Validate [http://framework.zend.com/manual/en/zend.validate.html]	Hard	---
Zend_XmlRpc [http://framework.zend.com/manual/en/zend.xmlrpc.html]	Hard	---
		Zend_XmlR [http://framework.zend.com/manual
		Zend_XmlR [http://framework.zend.com/manual

Zend Framework Dependencies

Below you can find a table listing Zend Framework Components and their dependencies to other Zend Framework Components. This can help you if you need to have only single components instead of the complete Zend Framework.

A dependency of type "hard" indicates that the components or classes cannot function properly if the respective dependent component is not available, while a dependency of type "soft" indicates that the component may need the dependent component in special situations or with special adapters.

Note

Even if it's possible to seperate single components for usage from the complete Zend Framework you should keep in mind that this can lead to problems when files are missed or components are used dynamically.

Table A.3. Zend Framework Components and their dependency to other Zend Framework Components

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
`Zend_Acl` [http://framework.zend.com/manual/en/zend.acl.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
`Zend_Auth` [http://framework.zend.com/manual/en/zend.auth.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
	Soft	`Zend_Db` [http://framework.zend.com/manual/en/zend.db.html]
		`Zend_InfoCard` [http://framework.zend.com/manual/en/zend.infocard.html]
		`Zend_Ldap` [http://framework.zend.com/manual/en/zend.ldap.html
		`Zend_OpenId` [http://framework.zend.com/manual/en/zend.openid.html]
		`Zend_Session` [http://framework.zend.com/manual/en/zend.session.html]
`Zend_Cache` [http://framework.zend.com/manual/en/zend.cache.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Loader` [http://framework.zend.com/manual/en/zend.loader.html]
`Zend_Config` [http://framework.zend.com/manual/en/zend.config.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
`Zend_Console_Getopt` [http://framework.zend.com/manual/en/zend.console.getopt.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Json` [http://framework.zend.com/manual/en/zend.json.html
`Zend_Controller` [http://framework.zend.com/manual/en/zend.controller.html]	Hard	`Zend_Config` [http://framework.zend.com/manual/en/zend.config.html]
		`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Filter` [http://framework.zend.com/manual/en/zend.filter.htm
		`Zend_Json` [http://framework.zend.com/manual/en/zend.json.html
		`Zend_Layout` [http://framework.zend.com/manual/en/zend.layout.html]
		`Zend_Loader` [http://framework.zend.com/manual/en/zend.loader.html]
		`Zend_Registry` [http://framework.zend.com/manual/en/zend.registry.html]
		`Zend_Session` [http://framework.zend.com/manual/en/zend.session.html]
		`Zend_Uri` [http://framework.zend.com/manual/en/zend.uri.html]
		`Zend_View` [http://framework.zend.com/manual/en/zend.view.html
`Zend_Currency` [http://framework.zend.com/manual/en/zend.currency.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Locale` [http://framework.zend.com/manual/en/zend.locale.htm

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
Zend_Date [http://framework.zend.com/manual/en/zend.date.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Locale [http://framework.zend.com/manual/en/zend.locale.htm
Zend_Db [http://framework.zend.com/manual/en/zend.db.html]	Hard	Zend_Config [http://framework.zend.com/manual/en/zend.config.html]
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Registry [http://framework.zend.com/manual/en/zend.registry.html]
Zend_Debug [http://framework.zend.com/manual/en/zend.debug.html]	---	---
Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]	---	---
Zend_Feed [http://framework.zend.com/manual/en/zend.feed.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Uri [http://framework.zend.com/manual/en/zend.uri.html]
Zend_Filter [http://framework.zend.com/manual/en/zend.filter.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Locale [http://framework.zend.com/manual/en/zend.locale.htm
		Zend_Validate [http://framework.zend.com/manual/en/zend.validate.html]
Zend_Form [http://framework.zend.com/manual/en/zend.form.html]	Hard	Zend_Controller [http://framework.zend.com/manual/en/zend.controller.html]
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Filter [http://framework.zend.com/manual/en/zend.filter.htm
		Zend_Json [http://framework.zend.com/manual/en/zend.json.html
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Registry [http://framework.zend.com/manual/en/zend.registry.html]
		Zend_Session [http://framework.zend.com/manual/en/zend.session.html]
		Zend_Validate [http://framework.zend.com/manual/en/zend.validate.html]

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
Zend_Gdata [http://framework.zend.com/manual/en/zend.gdata.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Mime [http://framework.zend.com/manual/en/zend.mime.htm
		Zend_Version [http://framework.zend.com/manual/en/zend.version.html]
Zend_Http [http://framework.zend.com/manual/en/zend.http.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Uri [http://framework.zend.com/manual/en/zend.uri.html]
Zend_InfoCard [http://framework.zend.com/manual/en/zend.infocard.html]	Hard	Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
Zend_Json [http://framework.zend.com/manual/en/zend.json.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Layout [http://framework.zend.com/manual/en/zend.layout.html]	Hard	Zend_Controller [http://framework.zend.com/manual/en/zend.controller.html]
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Filter [http://framework.zend.com/manual/en/zend.filter.htm
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_View [http://framework.zend.com/manual/en/zend.view.htm
Zend_Ldap [http://framework.zend.com/manual/en/zend.ldap.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Locale [http://framework.zend.com/manual/en/zend.locale.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Log [http://framework.zend.com/manual/en/zend.log.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Mail [http://framework.zend.com/manual/en/zend.mail.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Mime [http://framework.zend.com/manual/en/zend.mime.htm
		Zend_Validate [http://framework.zend.com/manual/en/zend.validate.html]

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
Zend_Measure [http://framework.zend.com/manual/en/zend.measure.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Locale [http://framework.zend.com/manual/en/zend.locale.htm
Zend_Memory [http://framework.zend.com/manual/en/zend.memory.html]	Hard	Zend_Cache [http://framework.zend.com/manual/en/zend.cache.htm
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Mime [http://framework.zend.com/manual/en/zend.mime.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_OpenId [http://framework.zend.com/manual/en/zend.openid.html]	Hard	Zend_Controller [http://framework.zend.com/manual/en/zend.controller.html]
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html
		Zend_Session [http://framework.zend.com/manual/en/zend.session.html]
Zend_Pdf [http://framework.zend.com/manual/en/zend.pdf.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Log [http://framework.zend.com/manual/en/zend.log.html]
		Zend_Memory [http://framework.zend.com/manual/en/zend.memory.html]
Zend_Registry [http://framework.zend.com/manual/en/zend.registry.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
Zend_Request [http://framework.zend.com/manual/en/zend.request.html]	---	---
Zend_Rest [http://framework.zend.com/manual/en/zend.rest.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Server [http://framework.zend.com/manual/en/zend.server.htm
		Zend_Service [http://framework.zend.com/manual/en/zend.service.html]
		Zend_Uri [http://framework.zend.com/manual/en/zend.uri.html]
Zend_Search_Lucene [http://framework.zend.com/manual/en/zend.search.lucene.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Server_Reflection [http://framework.zend.com/manual/en/zend.server.reflection.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
Zend_Service_Akismet [http://framework.zend.com/manual/en/zend.service.akismet.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html
		Zend_Uri [http://framework.zend.com/manual/en/zend.uri.html]
		Zend_Version [http://framework.zend.com/manual/en/zend.version.html]

Zend Framework Component	Dependency Type	Dependent Zend Framework Component
Zend_Service_Amazon [http://framework.zend.com/manual/en/zend.service.amazon.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
		Zend_Rest [http://framework.zend.com/manual/en/zend.rest.html]
Zend_Service_Audioscrob-bler [http://framework.zend.com/manual/en/zend.service.audioscrobbler.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
Zend_Service_Delicious [http://framework.zend.com/manual/en/zend.service.delicious.html]	Hard	Zend_Date [http://framework.zend.com/manual/en/zend.date.html
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
		Zend_Json [http://framework.zend.com/manual/en/zend.json.html
		Zend_Rest [http://framework.zend.com/manual/en/zend.rest.html]
Zend_Service_Flickr [http://framework.zend.com/manual/en/zend.service.flickr.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
		Zend_Rest [http://framework.zend.com/manual/en/zend.rest.html]
		Zend_Validate [http://framework.zend.com/manual/en/zend.validate.html]
Zend_Service_Nirvanix [http://framework.zend.com/manual/en/zend.service.nirvanix.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
Zend_Service_Simpy [http://framework.zend.com/manual/en/zend.service.simpy.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
		Zend_Rest [http://framework.zend.com/manual/en/zend.rest.html]
Zend_Service_SlideShare [http://framework.zend.com/manual/en/zend.service.slideshare.html]	Hard	Zend_Cache [http://framework.zend.com/manual/en/zend.cache.htm
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
Zend_Service_StrikeIron [http://framework.zend.com/manual/en/zend.service.strikeiron.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Http [http://framework.zend.com/manual/en/zend.http.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]

Zend Framework Component	**Dependency Type**	**Dependent Zend Framework Component**
`Zend_Service_Technorati` [http://framework.zend.com/manual/en/zend.service.technorati.html]	Hard	`Zend_Date` [http://framework.zend.com/manual/en/zend.date.html
		`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Http` [http://framework.zend.com/manual/en/zend.http.html]
		`Zend_Locale` [http://framework.zend.com/manual/en/zend.locale.htm
		`Zend_Rest` [http://framework.zend.com/manual/en/zend.rest.html]
		`Zend_Uri` [http://framework.zend.com/manual/en/zend.uri.html]
`Zend_Service_Yahoo` [http://framework.zend.com/manual/en/zend.service.yahoo.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Http` [http://framework.zend.com/manual/en/zend.http.html]
		`Zend_Rest` [http://framework.zend.com/manual/en/zend.rest.html]
		`Zend_Validate` [http://framework.zend.com/manual/en/zend.validate.html]
`Zend_Session` [http://framework.zend.com/manual/en/zend.session.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Loader` [http://framework.zend.com/manual/en/zend.loader.html]
`Zend_TimeSync` [http://framework.zend.com/manual/en/zend.timesync.html]	Hard	`Zend_Date` [http://framework.zend.com/manual/en/zend.date.html
		`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Loader` [http://framework.zend.com/manual/en/zend.loader.html]
`Zend_Translate` [http://framework.zend.com/manual/en/zend.translate.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Loader` [http://framework.zend.com/manual/en/zend.loader.html]
		`Zend_Locale` [http://framework.zend.com/manual/en/zend.locale.htm
`Zend_Uri` [http://framework.zend.com/manual/en/zend.uri.html]	Hard	`Zend_Exception` [http://framework.zend.com/manual/en/zend.exception.html]
		`Zend_Loader` [http://framework.zend.com/manual/en/zend.loader.html]
		`Zend_Validate` [http://framework.zend.com/manual/en/zend.validate.html]

Zend Framework Component	**Dependency Type**	**Dependent Zend Framework Component**
Zend_Validate [http://framework.zend.com/manual/en/zend.validate.html]	Soft	Zend_Date [http://framework.zend.com/manual/en/zend.date.html
		Zend_Filter [http://framework.zend.com/manual/en/zend.filter.htm
		Zend_Locale [http://framework.zend.com/manual/en/zend.locale.htm
		Zend_Registry [http://framework.zend.com/manual/en/zend.registry.html]
	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
Zend_Version [http://framework.zend.com/manual/en/zend.version.html]	---	---
Zend_View [http://framework.zend.com/manual/en/zend.view.html]	Hard	Zend_Controller [http://framework.zend.com/manual/en/zend.controller.html]
		Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Json [http://framework.zend.com/manual/en/zend.json.html
		Zend_Layout [http://framework.zend.com/manual/en/zend.layout.html]
		Zend_Loader [http://framework.zend.com/manual/en/zend.loader.html]
		Zend_Locale [http://framework.zend.com/manual/en/zend.locale.htm
		Zend_Registry [http://framework.zend.com/manual/en/zend.registry.html]
Zend_XmlRpc [http://framework.zend.com/manual/en/zend.xmlrpc.html]	Hard	Zend_Exception [http://framework.zend.com/manual/en/zend.exception.html]
		Zend_Registry [http://framework.zend.com/manual/en/zend.http.html]
		Zend_Server [http://framework.zend.com/manual/en/zend.server.htm

Appendix B. Zend Framework Coding Standard for PHP

Overview

Scope

This document provides guidelines for code formatting and documentation to individuals and teams contributing to Zend Framework. Many developers using Zend Framework have also found these coding standards useful because their code's style remains consistent with all Zend Framework code. It is also worth noting that it requires significant effort to fully specify coding standards. Note: Sometimes developers consider the establishment of a standard more important than what that standard actually suggests at the most detailed level of design. The guidelines in the Zend Framework coding standards capture practices that have worked well on the ZF project. You may modify these standards or use them as is in accordance with the terms of our license [http://framework.zend.com/license]

Topics covered in the ZF coding standards include:

- PHP File Formatting
- Naming Conventions
- Coding Style
- Inline Documentation

Goals

Coding standards are important in any development project, but they are particularly important when many developers are working on the same project. Coding standards help ensure that the code is high quality, has fewer bugs, and can be easily maintained.

PHP File Formatting

General

For files that contain only PHP code, the closing tag ("?>") is never permitted. It is not required by PHP, and omitting it prevents the accidental injection of trailing whitespace into the response.

IMPORTANT: Inclusion of arbitrary binary data as permitted by `__HALT_COMPILER()` is prohibited from PHP files in the Zend Framework project or files derived from them. Use of this feature is only permitted for some installation scripts.

Indentation

Indentation should consist of 4 spaces. Tabs are not allowed.

Maximum Line Length

The target line length is 80 characters. That is to say, ZF developers should strive keep each line of their code under 80 characters where possible and practical. However, longer lines are acceptable in some circumstances. The maximum length of any line of PHP code is 120 characters.

Line Termination

Line termination follows the Unix text file convention. Lines must end with a single linefeed (LF) character. Linefeed characters are represented as ordinal 10, or hexadecimal 0x0A.

Note: Do not use carriage returns (CR) as is the convention in Apple OS's (0x0D) or the carriage return/linefeed combination (CRLF) as is standard for the Windows OS (0x0D, 0x0A).

Naming Conventions

Classes

Zend Framework standardizes on a class naming convention whereby the names of the classes directly map to the directories in which they are stored. The root level directory of the ZF standard library is the "Zend/" directory, whereas the root level directory of the ZF extras library is the "ZendX/" directory. All Zend Framework classes are stored hierarchially under these root directories..

Class names may only contain alphanumeric characters. Numbers are permitted in class names but are discouraged in most cases. Underscores are only permitted in place of the path separator; the filename "Zend/Db/Table.php" must map to the class name "Zend_Db_Table".

If a class name is comprised of more than one word, the first letter of each new word must be capitalized. Successive capitalized letters are not allowed, e.g. a class "Zend_PDF" is not allowed while "Zend_Pdf" is acceptable.

These conventions define a pseudo-namespace mechanism for Zend Framework. Zend Framework will adopt the PHP namespace feature when it becomes available and is feasible for our developers to use in their applications.

See the class names in the standard and extras libraries for examples of this classname convention. *IMPORTANT:* Code that must be deployed alongside ZF libraries but is not part of the standard or extras libraries (e.g. application code or libraries that are not distributed by Zend) must never start with "Zend_" or "ZendX_".

Filenames

For all other files, only alphanumeric characters, underscores, and the dash character ("-") are permitted. Spaces are strictly prohibited.

Any file that contains PHP code should end with the extension ".php", with the notable exception of view scripts. The following examples show acceptable filenames for Zend Framework classes.:

```
Zend/Db.php

Zend/Controller/Front.php
```

```
Zend/View/Helper/FormRadio.php
```

File names must map to class names as described above.

Functions and Methods

Function names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in function names but are discouraged in most cases.

Function names must always start with a lowercase letter. When a function name consists of more than one word, the first letter of each new word must be capitalized. This is commonly called "camelCase" formatting.

Verbosity is generally encouraged. Function names should be as verbose as is practical to fully describe their purpose and behavior.

These are examples of acceptable names for functions:

```
filterInput()

getElementById()

widgetFactory()
```

For object-oriented programming, accessors for instance or static variables should always be prefixed with "get" or "set". In implementing design patterns, such as the singleton or factory patterns, the name of the method should contain the pattern name where practical to more thoroughly describe behavior.

For methods on objects that are declared with the "private" or "protected" modified, the first character of the variable name must be an underscore. This is the only acceptable application of an underscore in a method name. Methods declared "public" should never contain an underscore.

Functions in the global scope (a.k.a "floating functions") are permitted but discouraged in most cases. Consider wrapping these functions in a static class.

Variables

Variable names may only contain alphanumeric characters. Underscores are not permitted. Numbers are permitted in variable names but are discouraged in most cases.

For instance variables that are declared with the "private" or "protected" modifier, the first character of the variable name must be a single underscore. This is the only acceptable application of an underscore in a variable name. Member variables declared "public" should never start with an underscore.

As with function names (see section 3.3) variable names must always start with a lowercase letter and follow the "camelCaps" capitalization convention.

Verbosity is generally encouraged. Variables should always be as verbose as practical to describe the data that the developer intends to store in them. Terse variable names such as "$i" and "$n" are discouraged for all but the smallest loop contexts. If a loop contains more than 20 lines of code, the index variables should have more descriptive names.

Constants

Constants may contain both alphanumeric characters and underscores. Numbers are permitted in constant names.

All letters used in a constant name must be capitalized.

Words in constant names must be separated by underscore characters. For example, `EMBED_SUPPRESS_EMBED_EXCEPTION` is permitted but `EMBED_SUPPRESSEMBEDEXCEPTION` is not.

Constants must be defined as class members with the "const" modifier. Defining constants in the global scope with the "define" function is permitted but strongly discouraged.

Coding Style

PHP Code Demarcation

PHP code must always be delimited by the full-form, standard PHP tags:

```
<?php

?>
```

Short tags are never allowed. For files containing only PHP code, the closing tag must always be omitted (See the section called “General”).

Strings

String Literals

When a string is literal (contains no variable substitutions), the apostrophe or "single quote" should always be used to demarcate the string:

```
$a = 'Example String';
```

String Literals Containing Apostrophes

When a literal string itself contains apostrophes, it is permitted to demarcate the string with quotation marks or "double quotes". This is especially useful for SQL statements:

```
$sql = "SELECT `id`, `name` from `people` WHERE `name`='Fred' OR `name`='Susan'";
```

This syntax is preferred over escaping apostrophes as it is much easier to read.

Variable Substitution

Variable substitution is permitted using either of these forms:

```
$greeting = "Hello $name, welcome back!";

$greeting = "Hello {$name}, welcome back!";
```

For consistency, this form is not permitted:

```
$greeting = "Hello ${name}, welcome back!";
```

String Concatenation

Strings must be concatenated using the "." operator. A space must always be added before and after the "." operator to improve readability:

```
$company = 'Zend' . ' ' . 'Technologies';
```

When concatenating strings with the "." operator, it is encouraged to break the statement into multiple lines to improve readability. In these cases, each successive line should be padded with whitespace such that the "."; operator is aligned under the "=" operator:

```
$sql = "SELECT `id`, `name` FROM `people` "
     . "WHERE `name` = 'Susan' "
     . "ORDER BY `name` ASC ";
```

Arrays

Numerically Indexed Arrays

Negative numbers are not permitted as indices.

An indexed array may start with any non-negative number, however all base indices besides 0 are discouraged.

When declaring indexed arrays with the `array` function, a trailing space must be added after each comma delimiter to improve readability:

```
$sampleArray = array(1, 2, 3, 'Zend', 'Studio');
```

It is permitted to declare multiline indexed arrays using the "array" construct. In this case, each successive line must be padded with spaces such that beginning of each line is aligned:

```
$sampleArray = array(1, 2, 3, 'Zend', 'Studio',
                     $a, $b, $c,
                     56.44, $d, 500);
```

Associative Arrays

When declaring associative arrays with the `array` construct, breaking the statement into multiple lines is encouraged. In this case, each successive line must be padded with whitespace such that both the keys and the values are aligned:

```
$sampleArray = array('firstKey'  => 'firstValue',
                     'secondKey' => 'secondValue');
```

Classes

Class Declaration

Classes must be named according to Zend Framework's naming conventions.

The brace should always be written on the line underneath the class name (the "one true brace" form).

Every class must have a documentation block that conforms to the PHPDocumentor standard.

All code in a class must be indented with four spaces.

Only one class is permitted in each PHP file.

Placing additional code in class files is permitted but discouraged. In such files, two blank lines must separate the class from any additional PHP code in the class file.

The following is an example of an acceptable class declaration:

```
/**
 * Documentation Block Here
 */
class SampleClass
{
    // all contents of class
    // must be indented four spaces
}
```

Class Member Variables

Member variables must be named according to Zend Framework's variable naming conventions.

Any variables declared in a class must be listed at the top of the class, above the declaration of any methods.

The `var` constuct is not permitted. Member variables always declare their visibility by using one of the `private`, `protected`, or `public` modifiers. Giving access to member variables directly by declaring them as public is permitted but discouraged in favor of accessor methods (set/get).

Functions and Methods

Function and Method Declaration

Functions must be named according to the Zend Framework function naming conventions.

Methods inside classes must always declare their visibility by using one of the `private`, `protected`, or `public` modifiers.

As with classes, the brace should always be written on the line underneath the function name (the "one true brace" form). Space between the function name and the opening parenthesis for the arguments is not permitted.

Functions in the global scope are strongly discouraged.

The following is an example of an acceptable function declaration in a class:

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar()
    {
        // all contents of function
        // must be indented four spaces
    }
}
```

NOTE: Pass-by-reference is the only parameter passing mechanism permitted in a method declaration.

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * Documentation Block Here
     */
    public function bar(&$baz)
    {}
}
```

Call-time pass-by-reference is strictly prohibited.

The return value must not be enclosed in parentheses. This can hinder readability, in addtional to breaking code if a method is later changed to return by reference.

```
/**
 * Documentation Block Here
 */
class Foo
{
    /**
     * WRONG
     */
    public function bar()
    {
        return($this->bar);
    }

    /**
     * RIGHT
     */
    public function bar()
    {
        return $this->bar;
    }
}
```

Function and Method Usage

Function arguments should be separated by a single trailing space after the comma delimiter. The following is an example of an acceptable invocation of a function that takes three arguments:

```
threeArguments(1, 2, 3);
```

Call-time pass-by-reference is strictly prohibited. See the function declarations section for the proper way to pass function arguments by-reference.

In passing arrays as arguments to a function, the function call may include the "array" hint and may be split into multiple lines to improve readability. In such cases, the normal guidelines for writing arrays still apply:

```
threeArguments(array(1, 2, 3), 2, 3);

threeArguments(array(1, 2, 3, 'Zend', 'Studio',
                     $a, $b, $c,
                     56.44, $d, 500), 2, 3);
```

Control Statements

If/Else/Elseif

Control statements based on the `if` and `elseif` constructs must have a single space before the opening parenthesis of the conditional and a single space after the closing parenthesis.

Within the conditional statements between the parentheses, operators must be separated by spaces for readability. Inner parentheses are encouraged to improve logical grouping for larger conditional expressions.

The opening brace is written on the same line as the conditional statement. The closing brace is always written on its own line. Any content within the braces must be indented using four spaces.

```
if ($a != 2) {
    $a = 2;
}
```

For "if" statements that include "elseif" or "else", the formatting conventions are similar to the "if" construct. The following examples demonstrate proper formatting for "if" statements with "else" and/or "elseif" constructs:

```
if ($a != 2) {
    $a = 2;
} else {
   $a = 7;
}

if ($a != 2) {
    $a = 2;
} elseif ($a == 3) {
   $a = 4;
} else {
   $a = 7;
}
```

PHP allows statements to be written without braces in some circumstances. This coding standard makes no differentiation- all "if", "elseif" or "else" statements must use braces.

Use of the "elseif" construct is permitted but strongly discouraged in favor of the "else if" combination.

Switch

Control statements written with the "switch" statement must have a single space before the opening parenthesis of the conditional statement and after the closing parenthesis.

All content within the "switch" statement must be indented using four spaces. Content under each "case" statement must be indented using an additional four spaces.

```
switch ($numPeople) {
    case 1:
        break;

    case 2:
        break;

    default:
        break;
}
```

The construct `default` should never be omitted from a `switch` statement.

NOTE: It is sometimes useful to write a `case` statement which falls through to the next case by not including a `break` or `return` within that case. To distinguish these cases from bugs, any `case` statement where `break` or `return` are omitted should contain a comment indicating that the break was intentionally omitted.

Inline Documentation

Documentation Format

All documentation blocks ("docblocks") must be compatible with the phpDocumentor format. Describing the phpDocumentor format is beyond the scope of this document. For more information, visit: http://phpdoc.org/

All class files must contain a "file-level" docblock at the top of each file and a "class-level" docblock immediately above each class. Examples of such docblocks can be found below.

Files

Every file that contains PHP code must have a docblock at the top of the file that contains these phpDocumentor tags at a minimum:

```
/**
 * Short description for file
 *
 * Long description for file (if any)...
 *
 * LICENSE: Some license information
 *
 * @copyright  2008 Zend Technologies
 * @license    http://framework.zend.com/license   BSD License
 * @version    $Id:$
 * @link       http://framework.zend.com/package/PackageName
 * @since      File available since Release 1.5.0
*/
```

Classes

Every class must have a docblock that contains these phpDocumentor tags at a minimum:

```
/**
 * Short description for class
 *
 * Long description for class (if any)...
 *
 * @copyright  2008 Zend Technologies
 * @license    http://framework.zend.com/license   BSD License
 * @version    Release: @package_version@
 * @link       http://framework.zend.com/package/PackageName
 * @since      Class available since Release 1.5.0
 * @deprecated Class deprecated in Release 2.0.0
 */
```

Functions

Every function, including object methods, must have a docblock that contains at a minimum:

- A description of the function
- All of the arguments
- All of the possible return values

It is not necessary to use the "@access" tag because the access level is already known from the "public", "private", or "protected" modifier used to declare the function.

If a function/method may throw an exception, use @throws for all known exception classes:

```
@throws exceptionclass [description]
```

Appendix C. Copyright Information

The following copyrights are applicable to portions of Zend Framework.

Index