Zend Framework 2 Documentation

Release

Zend Technologies Ltd.

Aug 01, 2017

Contents

CHAPTER 1

Overview

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

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

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

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

We use PHPUnit to test our code and Travis CI as a continuous integration service.

Some of the features offered by Zend Framework: - Robust, high performance M-V-C implementation. - Database abstraction that is simple to use. - Forms component that implements valid HTML form rendering, validation, and filtering in an easy-to-use, *OOP* interface. - User authentication and authorization, such as `Zend\Authentication` and `Zend\Permissions\Acl`. - Comprehensive documentation

While they can be used separately, Zend Framework components in the standard library form a powerful and extensible web application framework when combined. Zend Framework offers a robust, high performance *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\Authentication` and `Zend\Permissions\Acl`, provide user authentication and authorization against all common credential stores.

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

The principal sponsor of the project 'Zend Framework' is Zend Technologies, but many companies have contributed components or significant features to the framework. Companies such as Google, Microsoft, and StrikeIron have partnered with Zend to provide interfaces to web services and other technologies 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 Zend Framework community. Community members, including contributors, make themselves available on mailing lists, IRC channels, and other forums. Whatever question you have about Zend Framework, the community is always available to address it.

CHAPTER 2

Installation

See the requirements appendix for a detailed list of requirements for Zend Framework.

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

  The *URL* for Zend Framework's *Git* repository is: https://github.com/zendframework/zf2

Once you have a copy of Zend Framework available, your application needs to be able to access the framework classes found in the library folder. Though there are several ways to achieve this, your *PHP* include_path needs to contain the path to Zend Framework's library.

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

CHAPTER 3

Getting Started with Zend Framework 2

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

Some assumptions

This tutorial assumes that you are running PHP 5.3.10 with the Apache web server and MySQL, accessible via the PDO extension. Your Apache installation must have the mod_rewrite extension installed and congured.

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

```
AllowOverride None
```

to

```
AllowOverride  All
```

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

The tutorial application

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

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

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

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

CHAPTER 4

Getting started: A skeleton application

In order to build our application, we will start with the ZendSkeletonApplication available on github. Go to https://github.com/zendframework/ZendSkeletonApplication and click the "Zip" button. This will download a le with a name like `zendframework-ZendSkeletonApplication-zfrelease-2.0.0beta5-2-gc2c7315.zip` or similar.

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

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

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

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

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

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

Generating autoload files
```

Note: If you see this message:

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

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

```
php composer.phar install
```

run instead:

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

We can now move on to the virtual host.

Virtual host

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

Setting up the virtual host is usually done within `httpd.conf` or `extra/httpd-vhosts.conf`. (If you are using `httpd-vhosts.conf`, ensure that this le is included by your main `httpd.conf` le.)

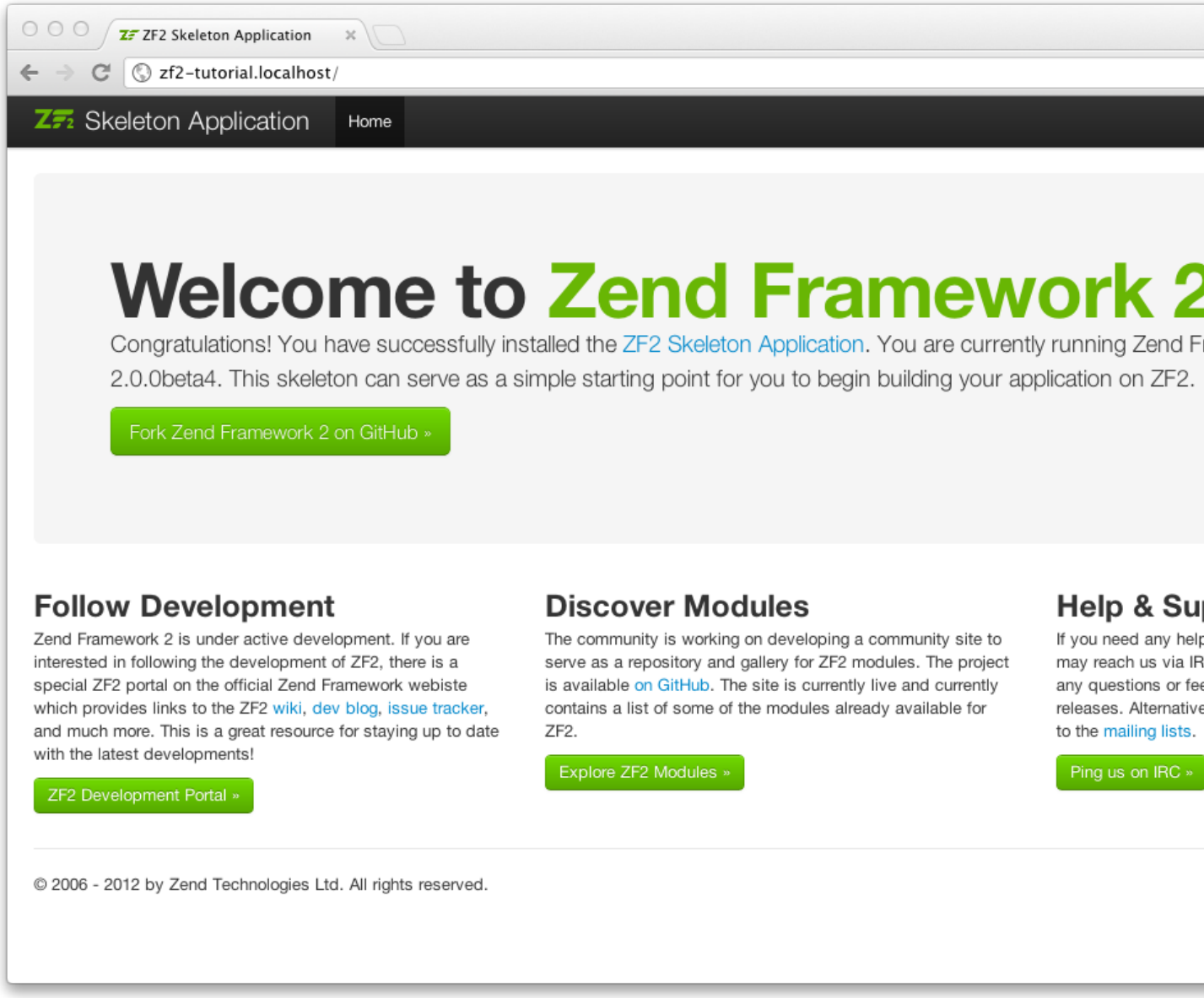
Ensure that `NameVirtualHost` is dened and set to "*:80" or similar, and then dene a virtual host along these lines:

```
<VirtualHost \*:80>
    ServerName zf2-tutorial.localhost
    DocumentRoot /path/to/zf-2tutorial/public
    SetEnv APPLICATION_ENV "development"
    <Directory /path/to/zf2-tutorial/public>
        DirectoryIndex index.php
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

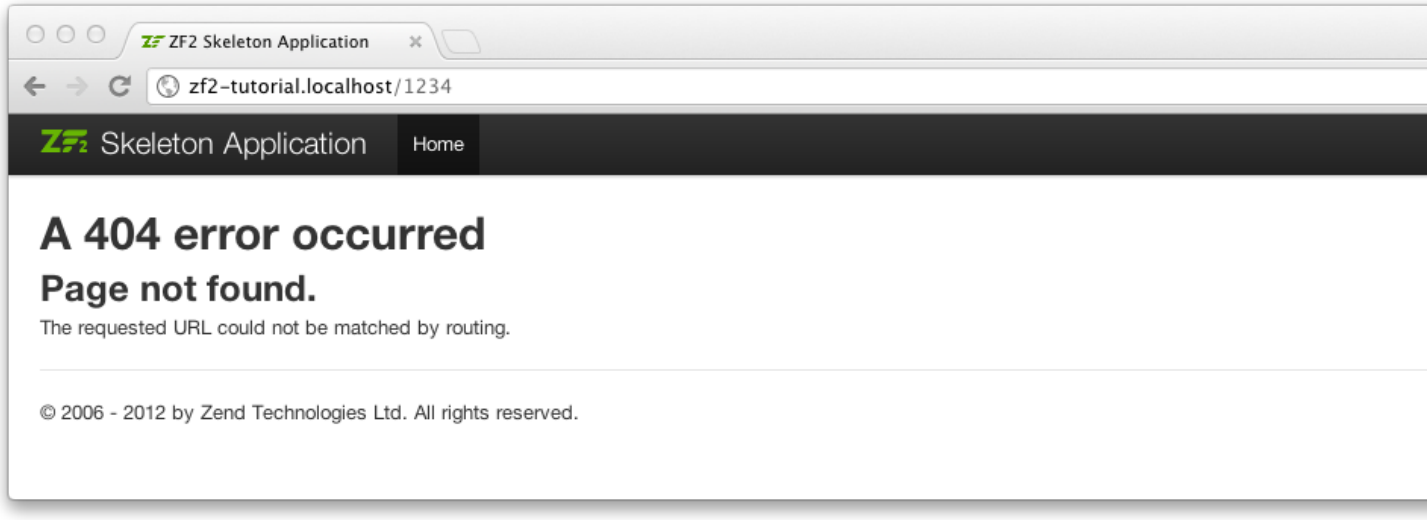
Make sure that you update your `/etc/hosts` or `c:\windows\system32\drivers\etc\hosts` le so that `zf2-tutorial.localhost` is mapped to `127.0.0.1`. The website can then be accessed using http://zf2-tutorial.localhost.

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

If you've done it right, you should see something like this:



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



If you see a standard Apache 404 error, then you need to x `.htaccess` usage before continuing.

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

CHAPTER 5

Modules

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

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

Let's start with the directories required.

Setting up the Album module

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

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

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

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

Create `Module.php` in the `Album` module:

```
// module/Album/Module.php
namespace Album;

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

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

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

Autoloading les

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

As we are in development, we don't need to load les via the classmap, so we provide an empty array for the classmap autoloader. Create `autoload_classmap.php` with these contents:

```
<?php
// module/Album/autoload_classmap.php:
return array();
```

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

Note: Note that as we are using Composer, as an alternative, you could not implement `getAutoloaderConfig()` and instead add `"Application": "module/Application/src"` to the `psr-0` key in `composer.json`. If you go this way, then you need to run `php composer.phar update` to update the composer autoloading les.

Configuration

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

Create the following conguration le for the `Album` module:

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

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

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

Informing the application about our new module

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

(Changes required are highlighted using comments.)

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

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

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

CHAPTER 6

Routing and controllers

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

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

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

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

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

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

```
// module/Album/cong/module.cong.php:
return array(
    'controllers' => array(
        'invokables' => array(
            'Album\Controller\Album' => 'Album\Controller\AlbumController',
        ),
```

```
    ),

    // The following section is new and should be added to your file
    'router' => array(
        'routes' => array(
            'album' => array(
                'type'    => 'segment',
                'options' => array(
                    'route'    => '/album[/:action][/:id]',
                    'constraints' => array(
                        'action' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'id'     => '[0-9]+',
                    ),
                    'defaults' => array(
                        'controller' => 'Album\Controller\Album',
                        'action'     => 'index',
                    ),
                ),
            ),
        ),
    ),

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

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

This route allows us to have the following URLs:

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

Create the controller

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

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

Let's go ahead and create our controller class:

```
// module/Album/src/Album/Controller/AlbumController.php:
namespace Album\Controller;

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

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

    public function addAction()
    {
    }

    public function editAction()
    {
    }

    public function deleteAction()
    {
    }
}
```

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

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

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

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

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

Initialise the view scripts

To integrate the view into our application all we need to do is create some view script files. These les will be executed by the `DefaultViewStrategy` and will be passed any variables or view models that are returned from the con-

troller action method. These view scripts are stored in our module's views directory within a directory named after the controller. Create these four empty files now:

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

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

CHAPTER 7

Database and models

The database

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

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

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

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

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

The model files

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

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

Let's start with our `Album` entity class within the `Model` directory:

```
// module/Album/src/Album/Model/Album.php:
namespace Album\Model;

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

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

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

Next, we extend `Zend\Db\TableGateway\AbstractTableGateway` and create our own `AlbumTable` class in the module's `Model` directory like this:

```
// module/Album/src/Album/Model/AlbumTable.php:
namespace Album\Model;

use Zend\Db\Adapter\Adapter;
use Zend\Db\ResultSet\ResultSet;
use Zend\Db\TableGateway\AbstractTableGateway;

class AlbumTable extends AbstractTableGateway
{
    protected $table ='album';

    public function __construct(Adapter $adapter)
    {
        $this->adapter = $adapter;
        $this->resultSetPrototype = new ResultSet();
        $this->resultSetPrototype->setArrayObjectPrototype(new Album());
        $this->initialize();
    }
```

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

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

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

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

There's a lot going on here. Firstly, we set the protected property `$table` to the name of the database table, 'album' in this case. We then write a constructor that takes a database adapter as its only parameter and assigns it to the adapter property of our class. We then need to tell the table gateway's result set that whenever it creates a new row object, it should use an `Album` object to do so. The `TableGateway` classes use the prototype pattern for creation of result sets and entities. This means that instead of instantiating when required, the system clones a previously instantiated object. See PHP Constructor Best Practices and the Prototype Pattern for more details.

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

Using ServiceManager to configure the database credentials and inject into the controller

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

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

```
// module/Album/Module.php:
namespace Album;

// Add this import statement:
use Album\Model\AlbumTable;

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

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

This method returns an array of `factories` that are all merged together by the `ModuleManager` before passing to the `ServiceManager`. We also need to configure the `ServiceManager` so that it knows how to get a `Zend\Db\Adapter\Adapter`. This is done using a factory called `Zend\Db\Adapter\AdapterServiceFactory` which we can configure within the merged config system. Zend Framework 2's `ModuleManager` merges all the configuration from each module's `module.config.php` file and then merges in the files in `config/autoload` (`*.global.php` and then `*.local.php` files). We'll add our database configuration information to `global.php` which you should commit to your version control system.You can use `local.php` (outside of the VCS) to store the credentials for your database if you want to.

```
// config/autoload/global.php:
return array(
    'db' => array(
        'driver'         => 'Pdo',
        'dsn'            => 'mysql:dbname=zf2tutorial;host=localhost',
        'driver_options' => array(
            PDO::MYSQL_ATTR_INIT_COMMAND => 'SET NAMES \'UTF8\''
        ),
    ),
```

```
    'service_manager' => array(
        'factories' => array(
            'Zend\Db\Adapter\Adapter'
                    => 'Zend\Db\Adapter\AdapterServiceFactory',
        ),
    ),
);
```

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

```
// config.autoload/local.php:
return array(
    'db' => array(
        'username' => 'YOUR USERNAME HERE',
        'password' => 'YOUR PASSWORD HERE',
    ),
);
```

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

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

You should also add:

```
protected $albumTable;
```

to the top of the class.

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

Listing albums

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

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

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

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

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

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

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

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

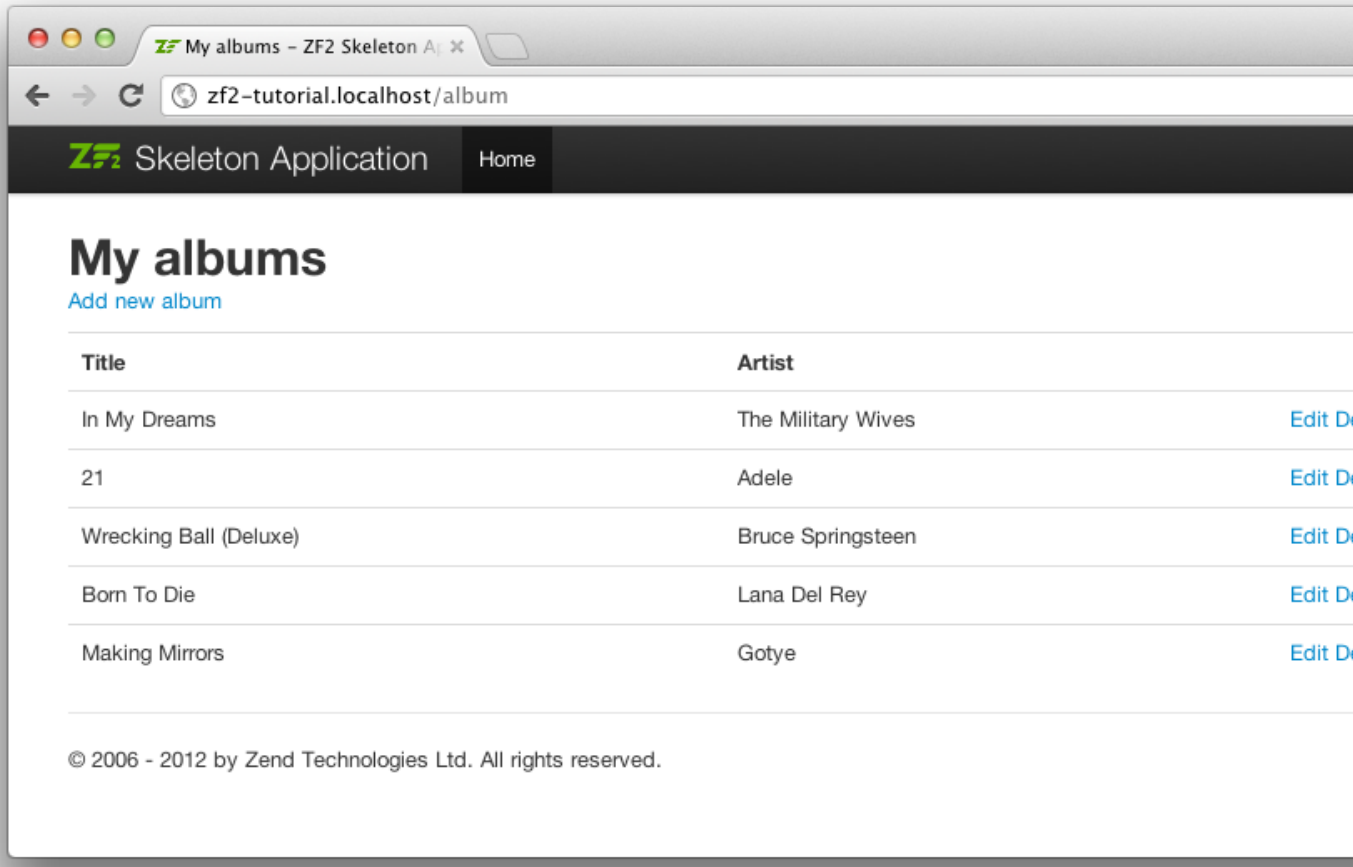
The `url()` view helper is provided by Zend Framework 2 and is used to create the links we need. The first parameter to `url()` is the route name we wish to use for construction of the URL, and the the second parameter is an array of all the variables to fit into the placeholders to use. In this case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

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

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

Note: We always use the `escapeHtml()` view helper to help protect ourselves from XSS vulnerabilities.

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

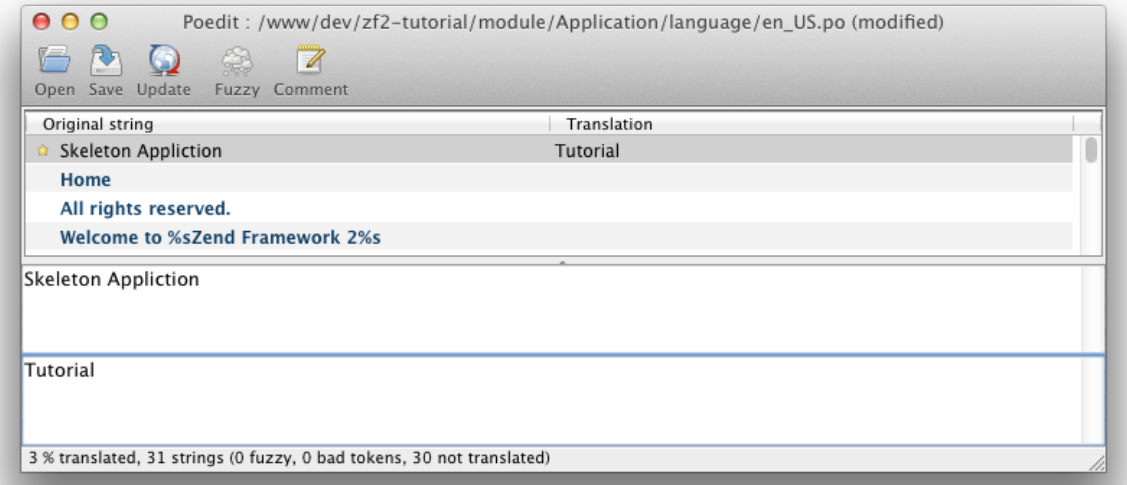

My albums - ZF2 Skeleton
zf2-tutorial.localhost/album
Skeleton Application
Home
My albums
Add new album
Title
Artist
In My Dreams
The Military Wives
21
Adele
Wrecking Ball (Deluxe)
Bruce Springsteen
Born To Die
Lana Del Rey
Making Mirrors
Gotye
© 2006 - 2012 by Zend Technologies Ltd. All rights reserved.

CHAPTER 8

Styling and Translations

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

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

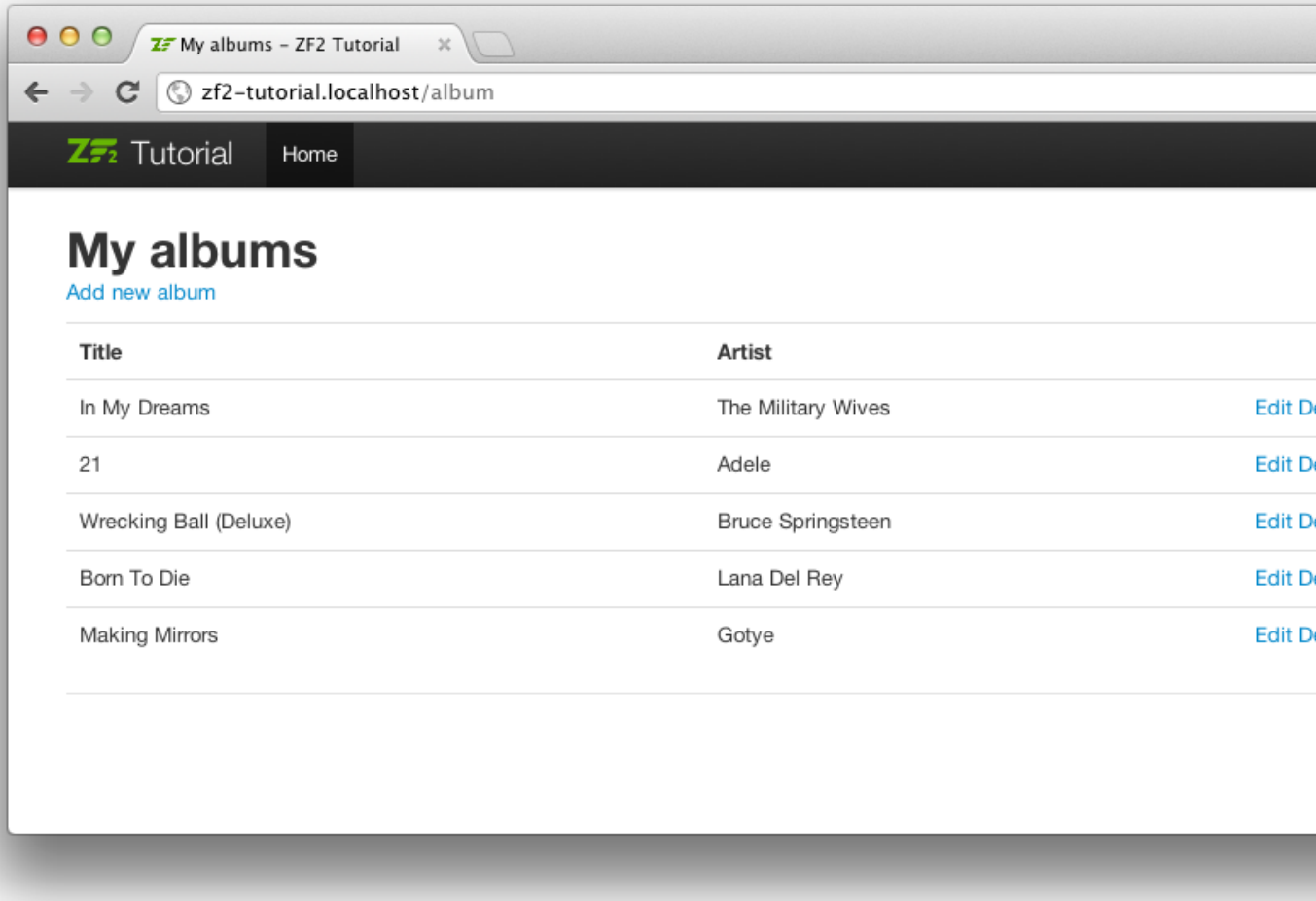


Press Save in the toolbar and poedit will create an `en_US.mo` file for us.

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

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

The page now looks ever so slightly better now!



CHAPTER 9

Forms and actions

Adding new albums

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

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

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

Create this file file now:

```
// module/Album/src/Album/Form/AlbumForm.php:
namespace Album\Form;

use Zend\Form\Form;

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

```
            'attributes' => array(
                'type'  => 'text',
            ),
            'options' => array(
                'label' => 'Artist',
            ),
        ));
        $this->add(array(
            'name' => 'title',
            'attributes' => array(
                'type'  => 'text',
            ),
            'options' => array(
                'label' => 'Title',
            ),
        ));
        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type'  => 'submit',
                'value' => 'Go',
                'id' => 'submitbutton',
            ),
        ));
    }
}
```

Within the constructor of `AlbumForm`, we set the name when we call the parent's constructor and then set the method and then create four form elements for the id, artist, title, and submit button. For each item we set various attributes and options, including the label to be displayed.

We also need to set up validation for this form. In Zend Framework 2 is this done using an input filter which can either be standalone or within any class that implements `InputFilterAwareInterface`, such as a model entity. We are going to add the input filter to our `Album` entity:

```
// module/Album/src/Album/Model/Album.php:
namespace Album\Model;

use Zend\InputFilter\Factory as InputFactory;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\InputFilterAwareInterface;
use Zend\InputFilter\InputFilterInterface;

class Album implements InputFilterAwareInterface
{
    public $id;
    public $artist;
    public $title;
    protected $inputFilter;

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

    public function setInputFilter(InputFilterInterface $inputFilter)
```

```
    {
        throw new \Exception("Not used");
    }

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

            $inputFilter->add($factory->createInput(array(
                'name'     => 'id',
                'required' => true,
                'filters'  => array(
                    array('name' => 'Int'),
                ),
            )));

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

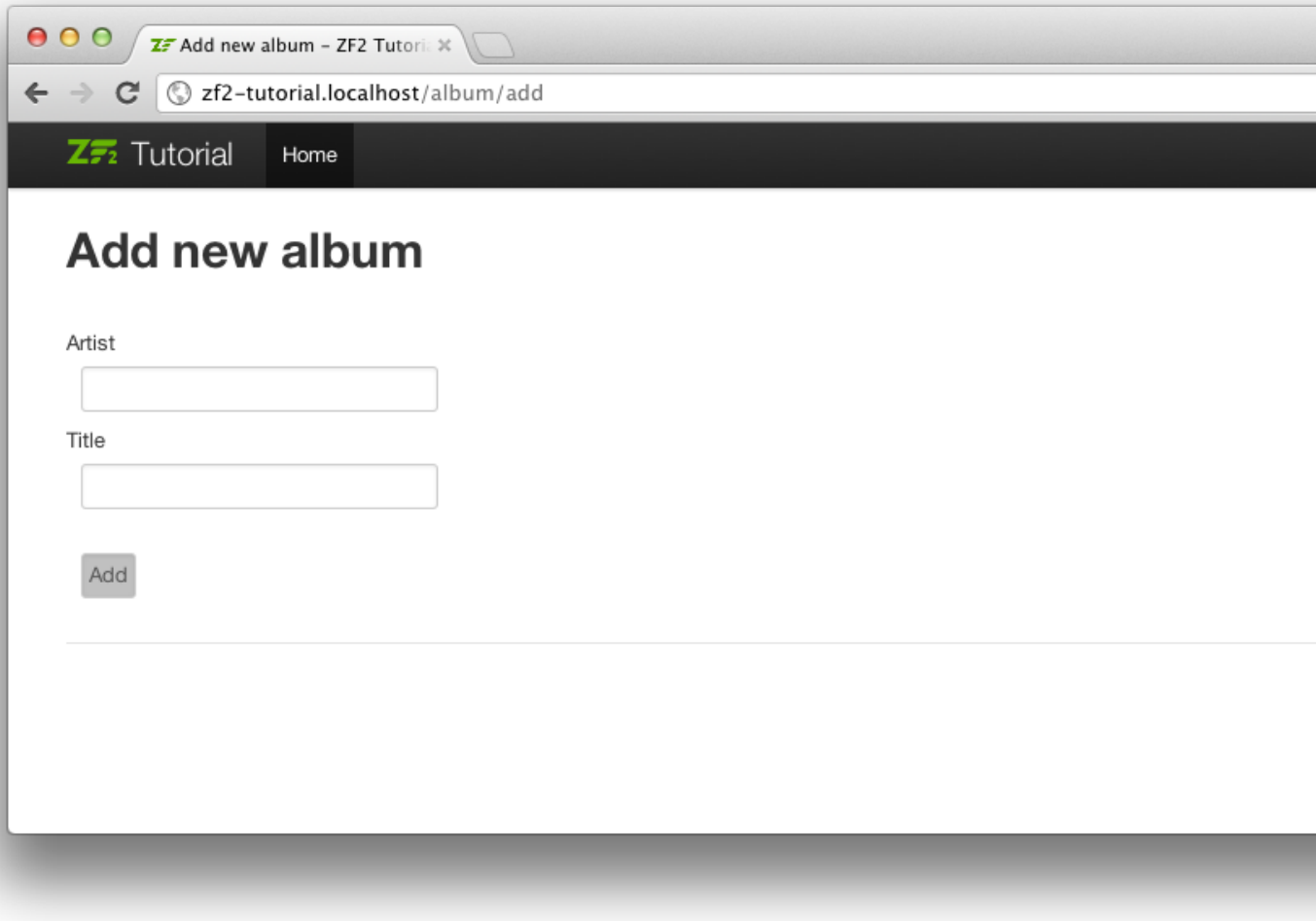
```
return array('form' => $form);
```

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

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

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

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

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

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



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

Editing an album

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

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

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

```
        $album = $this->getAlbumTable()->getAlbum($id);

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

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

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

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

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

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

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

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

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

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

 # When displaying the form, the initial values for each element are extracted from the model.

 # After successful validation in isValid(), the data from the form is put back into the model.

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

```
// module/Album/src/Album/Model/Album.php:
// ...
```

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

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

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

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

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

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

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

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

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

You should now be able to edit albums.

Deleting an album

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

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

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

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

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

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

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

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

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

The view script is a simple form:

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

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

<p>Are you sure that you want to delete
    '<?php echo $this->escapeHtml($album->title); ?>' by
    '<?php echo $this->escapeHtml($album->artist); ?>'?
</p>
<?php
$url = $this->url('album', array(
    'action' => 'delete',
    'id'     => $this->id,
));
?>
<form action="<?php echo $url; ?>" method="post">
<div>
```

```
    <input type="hidden" name="id" value="<?php echo (int) $album->id; ?>" />
    <input type="submit" name="del" value="Yes" />
    <input type="submit" name="del" value="No" />
</div>
</form>
```

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

Ensuring that the home page displays the list of albums

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

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

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

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

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

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

CHAPTER 10

Conclusion

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

CHAPTER 11

Learning Dependency Injection

Very brief introduction to Di.

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

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

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

Very brief introduction to Di Container.

```
TBD.
```

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

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

```
namespace My {

    class A
    {
        /* Some useful functionality */
    }

```

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

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

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

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

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

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

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

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

```
namespace My {

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

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

    class C
    {
        protected $b = null;
```

```
        public function __construct(B $b)
        {
            $this->b = $b;
        }
    }

}
```

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

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

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

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

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

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

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

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

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

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

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

Simplest Usage Case Without Type-hints

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

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

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

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

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

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

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

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

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

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

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

// and finally, create C
```

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

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

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

Simplest usage case with Compiled Definition

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

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

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

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

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

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

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

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

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

```php
if (!file_exists(__DIR__ . '/di-definition.php') && $isProduction) {
    $compiler = new Zend\Di\Definition\Compiler();
    $compiler->addCodeScannerDirectory(
        new Zend\Code\Scanner\ScannerDirectory('path/to/library/My/')
    );
    $definition = $compiler->compile();
    file_put_contents(
        __DIR__ . '/di-definition.php',
        '<?php return ' . var_export($definition->toArray(), true) . ';'
    );
} else {
    $definition = new Zend\Di\Definition\ArrayDefinition(
        include __DIR__ . '/di-definition.php'
    );
}

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

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

Creating a precompiled definition for others to use

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

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

// Now, create a Definition class for this information
$codeGenerator = new Zend\CodeGenerator\Php\PhpFile();
$codeGenerator->setClass(($class = new Zend\CodeGenerator\Php\PhpClass()));
$class->setNamespaceName('My');
$class->setName('DiDefinition');
$class->setExtendedClass('\Zend\Di\Definition\ArrayDefinition');
$class->setMethod(array(
    'name' => '__construct',
    'body' => 'parent::__construct(' . var_export($definition->toArray(), true) . ');'
));
file_put_contents(__DIR__ . '/My/DiDefinition.php', $codeGenerator->generate());
```

Using Multiple Definitions From Multiple Sources

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

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

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

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

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

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

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

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

Generating Service Locators

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

things up and remove those lookups?

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

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

As an example, consider the following:

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

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

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

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

```
<?php

namespace Application;

use Zend\Di\ServiceLocator;

class Context extends ServiceLocator
{

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

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

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

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

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

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

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

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

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

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

```
$container = new Application\Context;

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

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

CHAPTER 12

Introduction

The `Zend\Authentication` component provides an *API* for authentication and includes concrete authentication adapters for common use case scenarios.

`Zend\Authentication` is concerned only with **authentication** and not with **authorization**. Authentication is loosely defined as determining whether an entity actually is what it purports to be (i.e., identification), based on some set of credentials. Authorization, the process of deciding whether to allow an entity access to, or to perform operations upon, other entities is outside the scope of `Zend\Authentication`. For more information about authorization and access control with Zend Framework, please see the *Zend\Permissions\Acl* component.

Note: There is no `Zend\Authentication\Authentication` class, instead the class `Zend\Authentication\AuthenticationService` is provided. This class uses underlying authentication adapters and persistent storage backends.

Adapters

`Zend\Authentication` adapters are used to authenticate against a particular type of authentication service, such as *LDAP*, *RDBMS*, or file-based storage. Different adapters are likely to have vastly different options and behaviors, but some basic things are common among authentication adapters. For example, accepting authentication credentials (including a purported identity), performing queries against the authentication service, and returning results are common to `Zend\Authentication` adapters.

Each `Zend\Authentication` adapter class implements `Zend\Authentication\Adapter\AdapterInterface`. This interface defines one method, `authenticate()`, that an adapter class must implement for performing an authentication query. Each adapter class must be prepared prior to calling `authenticate()`. Such adapter preparation includes setting up credentials (e.g., username and password) and defining values for adapter-specific configuration options, such as database connection settings for a database table adapter.

The following is an example authentication adapter that requires a username and password to be set for authentication. Other details, such as how the authentication service is queried, have been omitted for brevity:

```php
use Zend\Authentication\Adapter\AdapterInterface;

class My\Auth\Adapter implements AdapterInterface
{
    /**
     * Sets username and password for authentication
     *
     * @return void
     */
    public function __construct($username, $password)
    {
        // ...
    }

    /**
     * Performs an authentication attempt
     *
     * @return \Zend\Authentication\Result
     * @throws \Zend\Authentication\Adapter\Exception\ExceptionInterface
     *               If authentication cannot be performed
     */
    public function authenticate()
    {
        // ...
    }
}
```

As indicated in its docblock, `authenticate()` must return an instance of `Zend\Authentication\Result` (or of a class derived from `Zend\Authentication\Result`). If for some reason performing an authentication query is impossible, `authenticate()` should throw an exception that derives from `Zend\Authentication\Adapter\Exception\ExceptionInterface`.

Results

`Zend\Authentication` adapters return an instance of `Zend\Authentication\Result` with `authenticate()` in order to represent the results of an authentication attempt. Adapters populate the `Zend\Authentication\Result` object upon construction, so that the following four methods provide a basic set of user-facing operations that are common to the results of `Zend\Authentication` adapters:

- `isValid()` - returns `TRUE` if and only if the result represents a successful authentication attempt
- `getCode()` - returns a `Zend\Authentication\Result` constant identifier for determining the type of authentication failure or whether success has occurred. This may be used in situations where the developer wishes to distinguish among several authentication result types. This allows developers to maintain detailed authentication result statistics, for example. Another use of this feature is to provide specific, customized messages to users for usability reasons, though developers are encouraged to consider the risks of providing such detailed reasons to users, instead of a general authentication failure message. For more information, see the notes below.
- `getIdentity()` - returns the identity of the authentication attempt
- `getMessages()` - returns an array of messages regarding a failed authentication attempt

A developer may wish to branch based on the type of authentication result in order to perform more specific operations. Some operations developers might find useful are locking accounts after too many unsuccessful password attempts, flagging an IP address after too many nonexistent identities are attempted, and providing specific, customized authentication result messages to the user. The following result codes are available:

```
use Zend\Authentication\Result;

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

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

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

switch ($result->getCode()) {

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

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

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

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

Identity Persistence

Authenticating a request that includes authentication credentials is useful per se, but it is also important to support maintaining the authenticated identity without having to present the authentication credentials with each request.

HTTP is a stateless protocol, however, and techniques such as cookies and sessions have been developed in order to facilitate maintaining state across multiple requests in server-side web applications.

Default Persistence in the PHP Session

By default, `Zend\Authentication` provides persistent storage of the identity from a successful authentication attempt using the *PHP* session. Upon a successful authentication attempt, `Zend\Authentication\AuthenticationService::authenticate()` stores the identity from the authentication result into persistent storage. Unless specified otherwise, `Zend\Authentication\AuthenticationService` uses a storage class named `Zend\Authentication\Storage\Session`, which, in turn, uses Zend\Session. A custom class may instead be used by providing an object that implements `Zend\Authentication\Storage\StorageInterface` to `Zend\Authentication\AuthenticationService::setStorage()`.

Note: If automatic persistent storage of the identity is not appropriate for a particular use case, then developers

may forego using the Zend\Authentication\AuthenticationService class altogether, instead using an adapter class directly.

Modifying the Session Namespace

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

```
use Zend\Authentication\AuthenticationService;
use Zend\Authentication\Storage\Session as SessionStorage;

$auth = new AuthenticationService();

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

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

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

Implementing Customized Storage

Sometimes developers may need to use a different identity storage mechanism than that provided by Zend\Authentication\Storage\Session. For such cases developers may simply implement Zend\Authentication\Storage\StorageInterface and supply an instance of the class to Zend\Authentication\AuthenticationService::setStorage().

Using a Custom Storage Class

In order to use an identity persistence storage class other than Zend\Authentication\Storage\Session, a developer implements Zend\Authentication\Storage\StorageInterface:

```
use Zend\Authentication\Storage\StorageInterface;

class My\Storage implements StorageInterface
{
    /**
     * Returns true if and only if storage is empty
     *
     * @throws \Zend\Authentication\Exception\ExceptionInterface
     *               If it is impossible to
     *               determine whether storage is empty
     * @return boolean
     */
```

```
13      public function isEmpty()
14      {
15          /**
16           * @todo implementation
17           */
18      }
19
20      /**
21       * Returns the contents of storage
22       *
23       * Behavior is undefined when storage is empty.
24       *
25       * @throws \Zend\Authentication\Exception\ExceptionInterface
26       *               If reading contents from storage is impossible
27       * @return mixed
28       */
29
30      public function read()
31      {
32          /**
33           * @todo implementation
34           */
35      }
36
37      /**
38       * Writes $contents to storage
39       *
40       * @param  mixed $contents
41       * @throws \Zend\Authentication\Exception\ExceptionInterface
42       *               If writing $contents to storage is impossible
43       * @return void
44       */
45
46      public function write($contents)
47      {
48          /**
49           * @todo implementation
50           */
51      }
52
53      /**
54       * Clears contents from storage
55       *
56       * @throws \Zend\Authentication\Exception\ExceptionInterface
57       *               If clearing contents from storage is impossible
58       * @return void
59       */
60
61      public function clear()
62      {
63          /**
64           * @todo implementation
65           */
66      }
67  }
```

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

```php
use Zend\Authentication\AuthenticationService;

// Instruct AuthenticationService to use the custom storage class
$auth = new AuthenticationService();

$auth->setStorage(new My\Storage());

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

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

Usage

There are two provided ways to use `Zend\Authentication` adapters:

. indirectly, through `Zend\Authentication\AuthenticationService::authenticate()`

. directly, through the adapter's `authenticate()` method

The following example illustrates how to use a `Zend\Authentication` adapter indirectly, through the use of the `Zend\Authentication\AuthenticationService` class:

```php
use Zend\Authentication\AuthenticationService;

// instantiate the authentication service
$auth = new AuthenticationService();

// Set up the authentication adapter
$authAdapter = new My\Auth\Adapter($username, $password);

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

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

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

```php
use Zend\Authentication\AuthenticationService;

$auth = new AuthenticationService();

```

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

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

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

```php
$auth->clearIdentity();
```

When the automatic use of persistent storage is inappropriate for a particular use case, a developer may simply bypass the use of the `Zend\Authentication\AuthenticationService` class, using an adapter class directly. Direct use of an adapter class involves configuring and preparing an adapter object and then calling its `authenticate()` method. Adapter-specific details are discussed in the documentation for each adapter. The following example directly utilizes `My\Auth\Adapter`:

```php
// Set up the authentication adapter
$authAdapter = new My\Auth\Adapter($username, $password);

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

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

CHAPTER 13

Database Table Authentication

Introduction

`Zend\Authentication\Adapter\DbTable` provides the ability to authenticate against credentials stored in a database table. Because `Zend\Authentication\Adapter\DbTable` requires an instance of `Zend\Db\Adapter\Adapter` to be passed to its constructor, each instance is bound to a particular database connection. Other configuration options may be set through the constructor and through instance methods, one for each option.

The available configuration options include:

- **tableName**: This is the name of the database table that contains the authentication credentials, and against which the database authentication query is performed.
- **identityColumn**: This is the name of the database table column used to represent the identity. The identity column must contain unique values, such as a username or e-mail address.
- **credentialColumn**: This is the name of the database table column used to represent the credential. Under a simple identity and password authentication scheme, the credential value corresponds to the password. See also the `credentialTreatment` option.
- **credentialTreatment**: In many cases, passwords and other sensitive data are encrypted, hashed, encoded, obscured, salted or otherwise treated through some function or algorithm. By specifying a parameterized treatment string with this method, such as '`MD5(?)`' or '`PASSWORD(?)`', a developer may apply such arbitrary *SQL* upon input credential data. Since these functions are specific to the underlying *RDBMS*, check the database manual for the availability of such functions for your database system.

Basic Usage

As explained in the introduction, the `Zend\Authentication\Adapter\DbTable` constructor requires an instance of `Zend\Db\Adapter\Adapter` that serves as the database connection to which the authentication adapter instance is bound. First, the database connection should be created.

The following code creates an adapter for an in-memory database, creates a simple table schema, and inserts a row against which we can perform an authentication query later. This example requires the *PDO* SQLite extension to be available:

```
use Zend\Db\Adapter\Adapter as DbAdapter;

// Create a SQLite database connection
$dbAdapter = new DbAdapter(array(
                'driver' => 'Pdo_Sqlite',
                'database' => 'path/to/sqlite.db'
            ));

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

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

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

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

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

```
use Zend\Authentication\Adapter\DbTable as AuthAdapter;

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

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

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

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

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

```
    ->setCredential('my_password')
;

// Perform the authentication query, saving the result
```

In addition to the availability of the `getIdentity()` method upon the authentication result object, `Zend\Authentication\Adapter\DbTable` also supports retrieving the table row upon authentication success:

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

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

/* Output:
my_username

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

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

Advanced Usage: Persisting a DbTable Result Object

By default, `Zend\Authentication\Adapter\DbTable` returns the identity supplied back to the auth object upon successful authentication. Another use case scenario, where developers want to store to the persistent storage mechanism of `Zend\Authentication` an identity object containing other useful information, is solved by using the `getResultRowObject()` method to return a **stdClass** object. The following code snippet illustrates its use:

```
// authenticate with Zend\Authentication\Adapter\DbTable
$result = $this->_auth->authenticate($adapter);

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

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

    /* ... */
```

```
21
22 } else {
23
24     /* ... */
25
26 }
```

Advanced Usage By Example

While the primary purpose of the `Zend\Authentication` component (and consequently `Zend\Authentication\Adapter\DbTable`) is primarily **authentication** and not **authorization**, there are a few instances and problems that toe the line between which domain they fit within. Depending on how you've decided to explain your problem, it sometimes makes sense to solve what could look like an authorization problem within the authentication adapter.

With that disclaimer out of the way, `Zend\Authentication\Adapter\DbTable` has some built in mechanisms that can be leveraged for additional checks at authentication time to solve some common user problems.

```
1  use Zend\Authentication\Adapter\DbTable as AuthAdapter;
2
3  // The status field value of an account is not equal to "compromised"
4  $adapter = new AuthAdapter($db,
5                             'users',
6                             'username',
7                             'password',
8                             'MD5(?) AND status != "compromised"'
9                             );
10
11 // The active field value of an account is equal to "TRUE"
12 $adapter = new AuthAdapter($db,
13                             'users',
14                             'username',
15                             'password',
16                             'MD5(?) AND active = "TRUE"'
17                             );
```

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

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

```
1 $sqlAlter = "ALTER TABLE [users] "
2           . "ADD COLUMN [password_salt] "
3           . "AFTER [password]";
```

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

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

And now let's build the adapter:

```
$adapter = new AuthAdapter($db,
                           'users',
                           'username',
                           'password',
                           "MD5(CONCAT('staticSalt', ?, password_salt))"
                          );
```

Note: You can improve security even more by using a static salt value hard coded into your application. In the case that your database is compromised (e. g. by an *SQL* injection attack) but your web server is intact your data is still unusable for the attacker.

Another alternative is to use the `getDbSelect()` method of the `Zend\Authentication\Adapter\DbTable` after the adapter has been constructed. This method will return the `Zend\Db\Sql\Select` object instance it will use to complete the `authenticate()` routine. It is important to note that this method will always return the same object regardless if `authenticate()` has been called or not. This object **will not** have any of the identity or credential information in it as those values are placed into the select object at `authenticate()` time.

An example of a situation where one might want to use the `getDbSelect()` method would check the status of a user, in other words to see if that user's account is enabled.

```
// Continuing with the example from above
$adapter = new AuthAdapter($db,
                           'users',
                           'username',
                           'password',
                           'MD5(?)'
                           );

// get select object (by reference)
$select = $adapter->getDbSelect();
$select->where('active = "TRUE"');

// authenticate, this ensures that users.active = TRUE
$adapter->authenticate();
```

CHAPTER 14

Digest Authentication

Introduction

Digest authentication is a method of *HTTP* authentication that improves upon Basic authentication by providing a way to authenticate without having to transmit the password in clear text across the network.

This adapter allows authentication against text files containing lines having the basic elements of Digest authentication:

- username, such as "**joe.user**"
- realm, such as "**Administrative Area**"
- *MD5* hash of the username, realm, and password, separated by colons

The above elements are separated by colons, as in the following example (in which the password is "**somePassword**"):

```
someUser:Some Realm:fde17b91c3a510ecbaf7dbd37f59d4f8
```

Specifics

The digest authentication adapter, `Zend\Authentication\Adapter\Digest`, requires several input parameters:

- filename - Filename against which authentication queries are performed
- realm - Digest authentication realm
- username - Digest authentication user
- password - Password for the user of the realm

These parameters must be set prior to calling `authenticate()`.

Identity

The digest authentication adapter returns a `Zend\Authentication\Result` object, which has been populated with the identity as an array having keys of **realm** and **username**. The respective array values associated with these keys correspond to the values set before `authenticate()` is called.

```
use Zend\Authentication\Adapter\Digest as AuthAdapter;

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

$result = $adapter->authenticate();

$identity = $result->getIdentity();

print_r($identity);

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

CHAPTER 15

HTTP Authentication Adapter

Introduction

`Zend\Authentication\Adapter\Http` provides a mostly-compliant implementation of RFC-2617, Basic and Digest *HTTP* Authentication. Digest authentication is a method of *HTTP* authentication that improves upon Basic authentication by providing a way to authenticate without having to transmit the password in clear text across the network.

Major Features:

- Supports both Basic and Digest authentication.
- Issues challenges in all supported schemes, so client can respond with any scheme it supports.
- Supports proxy authentication.
- Includes support for authenticating against text files and provides an interface for authenticating against other sources, such as databases.

There are a few notable features of *RFC-2617* that are not implemented yet:

- Nonce tracking, which would allow for "stale" support, and increased replay attack protection.
- Authentication with integrity checking, or "auth-int".
- Authentication-Info *HTTP* header.

Design Overview

This adapter consists of two sub-components, the *HTTP* authentication class itself, and the so-called "Resolvers." The *HTTP* authentication class encapsulates the logic for carrying out both Basic and Digest authentication. It uses a Resolver to look up a client's identity in some data store (text file by default), and retrieve the credentials from the data store. The "resolved" credentials are then compared to the values submitted by the client to determine whether authentication is successful.

Configuration Options

The `Zend\Authentication\Adapter\Http` class requires a configuration array passed to its constructor. There are several configuration options available, and some are required:

Table 15.1: Configuration Options

Option Name	Required	Description
accept_schemes	Yes	Determines which authentication schemes the adapter will accept from the client. Must be a space=separated list containing 'basic' and/or 'digest'.
realm	Yes	Sets the authentication realm; usernames should be unique within a given realm.
digest_domains	Yes, when accept_schemes contains digest	Space-separated list of URIs for which the same authentication information is valid. The URIs need not all point to the same server.
nonce_timeout	Yes, when accept_schemes contains digest	Sets the number of seconds for which the nonce is valid. See notes below.
use_opaque	No	Specifies whether to send the opaque value in the header. True by default.
algorithm	No	Specified the algorithm. Defaults to MD5, the only supported option (for now).
proxy_auth	No	Disabled by default. Enable to perform Proxy authentication, instead of normal origin server authentication.

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

Resolvers

The resolver's job is to take a username and realm, and return some kind of credential value. Basic authentication expects to receive the Base64 encoded version of the user's password. Digest authentication expects to receive a hash of the user's username, the realm, and their password (each separated by colons). Currently, the only supported hash algorithm is *MD5*.

`Zend\Authentication\Adapter\Http` relies on objects implementing `Zend\Authentication\Adapter\Http\ResolverInterface`. A text file resolver class is included with this adapter, but any other kind of resolver can be created simply by implementing the resolver interface.

File Resolver

The file resolver is a very simple class. It has a single property specifying a filename, which can also be passed to the constructor. Its `resolve()` method walks through the text file, searching for a line with a matching username and realm. The text file format similar to Apache htpasswd files:

```
1 <username>:<realm>:<credentials>\n
```

Each line consists of three fields - username, realm, and credentials - each separated by a colon. The credentials field is opaque to the file resolver; it simply returns that value as-is to the caller. Therefore, this same file format serves both Basic and Digest authentication. In Basic authentication, the credentials field should be written in clear text. In Digest authentication, it should be the *MD5* hash described above.

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

```
use Zend\Authentication\Adapter\Http\FileResolver;
$path     = 'files/passwd.txt';
$resolver = new FileResolver($path);
```

or

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

If the given path is empty or not readable, an exception is thrown.

Basic Usage

First, set up an array with the required configuration values:

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

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

Next, create the `Zend\Authentication\Adapter\Http` object:

```
$adapter = new Zend\Authentication\Adapter\Http($config);
```

Since we're supporting both Basic and Digest authentication, we need two different resolver objects. Note that this could just as easily be two different classes:

```
use Zend\Authentication\Adapter\Http\FileResolver;

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

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

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

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

```php
assert($request instanceof Zend\Http\Request);
assert($response instanceof Zend\Http\Response);

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

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

CHAPTER 16

LDAP Authentication

Introduction

`Zend\Authentication\Adapter\Ldap` supports web application authentication with *LDAP* services. Its features include username and domain name canonicalization, multi-domain authentication, and failover capabilities. It has been tested to work with Microsoft Active Directory and OpenLDAP, but it should also work with other *LDAP* service providers.

This documentation includes a guide on using `Zend\Authentication\Adapter\Ldap`, an exploration of its *API*, an outline of the various available options, diagnostic information for troubleshooting authentication problems, and example options for both Active Directory and OpenLDAP servers.

Usage

To incorporate `Zend\Authentication\Adapter\Ldap` authentication into your application quickly, even if you're not using `Zend\Mvc`, the meat of your code should look something like the following:

```
1  use Zend\Authentication\AuthenticationService;
2  use Zend\Authentication\Adapter\Ldap as AuthAdapter;
3  use Zend\Config\Reader\Ini as ConfigReader;
4  use Zend\Log\Logger;
5  use Zend\Log\Writer\Stream as LogWriter;
6  use Zend\Log\Filter\Priority as LogFilter;
7
8  $username = $this->_request->getParam('username');
9  $password = $this->_request->getParam('password');
10
11
12 $auth = new AuthenticationService();
13
14 $config = new ConfigReader('./ldap-config.ini','production');
15
```

```php
$log_path = $config->ldap->log_path;
$options = $config->ldap->toArray();
unset($options['log_path']);

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

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

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

    $logger = new Logger;
    $writer = new LogWriter($log_path);

    $logger->addWriter($writer);

    $filter = new LogFilter(Logger::DEBUG);
    $logger->addFilter($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", Logger::DEBUG);
        }
    }
}
```

Of course, the logging code is optional, but it is highly recommended that you use a logger. `Zend\Authentication\Adapter\Ldap` will record just about every bit of information anyone could want in `$messages` (more below), which is a nice feature in itself for something that has a history of being notoriously difficult to debug.

The `Zend\Config\Reader\Ini` code is used above to load the adapter options. It is also optional. A regular array would work equally well. The following is an example `ldap-config.ini` file that has options for two separate servers. With multiple sets of server options the adapter will try each, in order, until the credentials are successfully authenticated. The names of the servers (e.g., 'server1' and 'server2') are largely arbitrary. For details regarding the options array, see the **Server Options** section below. Note that `Zend\Config\Reader\Ini` requires that any values with "equals" characters (**=**) will need to be quoted (like the DNs shown below).

```
[production]

ldap.log_path = /tmp/ldap.log

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

; Typical options for Active Directory
ldap.server2.host = dc1.w.net
ldap.server2.useStartTls = true
```

```
ldap.server2.accountDomainName = w.net
ldap.server2.accountDomainNameShort = W
ldap.server2.accountCanonicalForm = 3
ldap.server2.baseDn = "CN=Users,DC=w,DC=net"
```

The above configuration will instruct `Zend\Authentication\Adapter\Ldap` to attempt to authenticate users with the OpenLDAP server `s0.foo.net` first. If the authentication fails for any reason, the AD server `dc1.w.net` will be tried.

With servers in different domains, this configuration illustrates multi-domain authentication. You can also have multiple servers in the same domain to provide redundancy.

Note that in this case, even though OpenLDAP has no need for the short NetBIOS style domain name used by Windows, we provide it here for name canonicalization purposes (described in the **Username Canonicalization** section below).

The API

The `Zend\Authentication\Adapter\Ldap` constructor accepts three parameters.

The `$options` parameter is required and must be an array containing one or more sets of options. Note that it is **an array of arrays** of *Zend\Ldap\Ldap* options. Even if you will be using only one *LDAP* server, the options must still be within another array.

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

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

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

)
```

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

Note: What is a Distinguished Name?

A DN or "distinguished name" is a string that represents the path to an object within the *LDAP* directory. Each comma-separated component is an attribute and value representing a node. The components are evaluated in reverse. For example, the user account **CN=Bob Carter,CN=Users,DC=w,DC=net** is located directly within the **CN=Users,DC=w,DC=net container**. This structure is best explored with an *LDAP* browser like the *ADSI* Edit *MMC* snap-in for Active Directory or phpLDAPadmin.

The names of servers (e.g. 'server1' and 'server2' shown above) are largely arbitrary, but for the sake of using `Zend\Config\Reader\Ini`, the identifiers should be present (as opposed to being numeric indexes) and should not contain any special characters used by the associated file formats (e.g. the '**.**'*INI* property separator, '**&**' for *XML* entity references, etc).

With multiple sets of server options, the adapter can authenticate users in multiple domains and provide failover so that if one server is not available, another will be queried.

Note: The Gory Details: What Happens in the Authenticate Method?

When the `authenticate()` method is called, the adapter iterates over each set of server options, sets them on the internal `Zend\Ldap\Ldap` instance, and calls the `Zend\Ldap\Ldap::bind()` method with the username and password being authenticated. The `Zend\Ldap\Ldap` class checks to see if the username is qualified with a domain (e.g., has a domain component like `alice@foo.net` or `FOO\alice`). If a domain is present, but does not match either of the server's domain names (`foo.net` or *FOO*), a special exception is thrown and caught by `Zend\Authentication\Adapter\Ldap` that causes that server to be ignored and the next set of server options is selected. If a domain **does** match, or if the user did not supply a qualified username, `Zend\Ldap\Ldap` proceeds to try to bind with the supplied credentials. if the bind is not successful, `Zend\Ldap\Ldap` throws a `Zend\Ldap\Exception\LdapException` which is caught by `Zend\Authentication\Adapter\Ldap` and the next set of server options is tried. If the bind is successful, the iteration stops, and the adapter's `authenticate()` method returns a successful result. If all server options have been tried without success, the authentication fails, and `authenticate()` returns a failure result with error messages from the last iteration.

The username and password parameters of the `Zend\Authentication\Adapter\Ldap` constructor represent the credentials being authenticated (i.e., the credentials supplied by the user through your *HTML* login form). Alternatively, they may also be set with the `setUsername()` and `setPassword()` methods.

Server Options

Each set of server options **in the context of ZendAuthenticationAdapterLdap** consists of the following options, which are passed, largely unmodified, to `Zend\Ldap\Ldap::setOptions()`:

Table 16.1: Server Options

Name	Description
host	The hostname of LDAP server that these options represent. This option is required.
port	The port on which the LDAP server is listening. If useSsl is TRUE, the default port value is 636. If useSsl is FALSE, the default port value is 389.
useStartTls	Whether or not the LDAP client should use TLS (aka SSLv2) encrypted transport. A value of TRUE is strongly favored in production environments to prevent passwords from be transmitted in clear text. The default value is FALSE, as servers frequently require that a certificate be installed separately after installation. The useSsl and useStartTls options are mutually exclusive. The useStartTls option should be favored over useSsl but not all servers support this newer mechanism.
useSsl	Whether or not the LDAP client should use SSL encrypted transport. The useSsl and useStartTls options are mutually exclusive, but useStartTls should be favored if the server and LDAP client library support it. This value also changes the default port value (see port description above).
username	The DN of the account used to perform account DN lookups. LDAP servers that require the username to be in DN form when performing the "bind" require this option. Meaning, if bindRequiresDn is TRUE, this option is required. This account does not need to be a privileged account; an account with read-only access to objects under the baseDn is all that is necessary (and preferred based on the Principle of Least Privilege).
password	The password of the account used to perform account DN lookups. If this option is not supplied, the LDAP client will attempt an "anonymous bind" when performing account DN lookups.
bindRequiresDn	Some LDAP servers require that the username used to bind be in DN form like CN=Alice Baker,OU=Sales,DC=foo,DC=net (basically all servers except AD). If this option is TRUE, this instructs Zend\Ldap\Ldap to automatically retrieve the DN corresponding to the username being authenticated, if it is not already in DN form, and then re-bind with the proper DN. The default value is FALSE. Currently only Microsoft Active Directory Server (ADS) is known not to require usernames to be in DN form when binding, and therefore this option may be FALSE with AD (and it should be, as retrieving the DN requires an extra round trip to the server). Otherwise, this option must be set to TRUE (e.g. for OpenLDAP). This option also controls the default 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\Authentication\Result::getIdentity() (and Zend\Authentication\AuthenticationService::getIdentity(), if Zend\Authentication\AuthenticationService was used) will always be FOO\alice, regardless of what form Alice supplied, whether it be alice, alice@foo.net, FOO\alice, FoO\aLicE, foo.net\alice, etc. See the Account Name Canonicalization section in the Zend\Ldap\Ldap documentation for details. Note that when using multiple sets of server options it is recommended, but not required, that the same accountCanonicalForm be used with all server options so that the resulting usernames are always canonicalized to the same form (e.g., if you canonicalize to EXAMPLE\username with an AD server but to username@example.com with an OpenLDAP server, that may be awkward for the application's high-level logic).
accountDomainName	The FQDN domain name for which the target LDAP server is an authority (e.g., example.com). This option is used to canonicalize names so that the username supplied by the user can be converted as necessary for binding. It is also used to determine if the server is an authority for the supplied username (e.g., if accountDomainName is foo.net and the user supplies bob@bar.net, the server will not be queried, and a failure will result). This option is not required, but if it is not supplied, usernames in principal name form (e.g., alice@foo.net) are not supported. It is strongly recommended that you supply this option, as there are many

Note: If you enable **useStartTls = TRUE** or **useSsl = TRUE** you may find that the *LDAP* client generates an error claiming that it cannot validate the server's certificate. Assuming the *PHP LDAP* extension is ultimately linked to the OpenLDAP client libraries, to resolve this issue you can set "`TLS_REQCERT never`" in the OpenLDAP client `ldap.conf` (and restart the web server) to indicate to the OpenLDAP client library that you trust the server. Alternatively, if you are concerned that the server could be spoofed, you can export the *LDAP* server's root certificate and put it on the web server so that the OpenLDAP client can validate the server's identity.

Collecting Debugging Messages

`Zend\Authentication\Adapter\Ldap` collects debugging information within its `authenticate()` method. This information is stored in the `Zend\Authentication\Result` object as messages. The array returned by `Zend\Authentication\Result::getMessages()` is described as follows

Table 16.2: Debugging Messages

Messages Array Index	Description
Index 0	A generic, user=friendly message that is suitable for displaying to users (e.g., "Invalid credentials"). If the authentication is successful, this string is empty.
Index 1	A more detailed error message that is not suitable to be displayed to users but should be logged for the benefit of server operators. If the authentication is successful, this string is empty.
Indexes 2 and higher	All log messages in order starting at index 2.

In practice, index 0 should be displayed to the user (e.g., using the FlashMessenger helper), index 1 should be logged and, if debugging information is being collected, indexes 2 and higher could be logged as well (although the final message always includes the string from index 1).

Common Options for Specific Servers

Options for Active Directory

For *ADS*, the following options are noteworthy:

Table 16.3: Options for Active Directory

Name	Additional Notes
host	As with all servers, this option is required.
useStartTls	For the sake of security, this should be TRUE if the server has the necessary certificate installed.
useSsl	Possibly used as an alternative to useStartTls (see above).
baseDn	As with all servers, this option is required. By default AD places all user accounts under the Users container (e.g., CN=Users,DC=foo,DC=net), but the default is not common in larger organizations. Ask your AD administrator what the best DN for accounts for your application would be.
accountCanonicalForm	You almost certainly want this to be 3 for backslash style names (e.g., FOO\alice), which are most familiar to Windows users. You should not use the unqualified form 2 (e.g., alice), as this may grant access to your application to users with the same username in other trusted domains (e.g., BAR\alice and FOO\alice will be treated as the same user). (See also note below.)
accountDomainName	This is required with AD unless accountCanonicalForm 2 is used, which, again, is discouraged.
accountDomainNameShort	The NetBIOS name of the domain that users are in and for which the AD server is an authority. This is required if the backslash style accountCanonicalForm is used.

Note: Technically there should be no danger of accidental cross-domain authentication with the current `Zend\Authentication\Adapter\Ldap` implementation, since server domains are explicitly checked, but this may not be true of a future implementation that discovers the domain at runtime, or if an alternative adapter is used (e.g., Kerberos). In general, account name ambiguity is known to be the source of security issues, so always try to use qualified account names.

Options for OpenLDAP

For OpenLDAP or a generic *LDAP* server using a typical posixAccount style schema, the following options are noteworthy:

Table 16.4: Options for OpenLDAP

Name	Additional Notes
host	As with all servers, this option is required.
useStartTls	For the sake of security, this should be TRUE if the server has the necessary certificate installed.
useSsl	Possibly used as an alternative to useStartTls (see above).
username	Required and must be a DN, as OpenLDAP requires that usernames be in DN form when performing a bind. Try to use an unprivileged account.
password	The password corresponding to the username above, but this may be omitted if the LDAP server permits an anonymous binding to query user accounts.
bindRequiresDn	Required and must be TRUE, as OpenLDAP requires that usernames be in DN form when performing a bind.
baseDn	As with all servers, this option is required and indicates the DN under which all accounts being authenticated are located.
accountCanonicalForm	Optional, but the default value is 4 (principal style names like alice@foo.net), which may not be ideal if your users are used to backslash style names (e.g., FOO\alice). For backslash style names use value 3.
accountDomainName	Required unless you're using accountCanonicalForm 2, which is not recommended.
accountDomainNameShort	If AD is not also being used, this value is not required. Otherwise, if accountCanonicalForm 3 is used, this option is required and should be a short name that corresponds adequately to the accountDomainName (e.g., if your accountDomainName is foo.net, a good accountDomainNameShort value might be FOO).

CHAPTER 17

Introduction

`Zend\Barcode\Barcode` provides a generic way to generate barcodes. The `Zend\Barcode` component is divided into two subcomponents: barcode objects and renderers. Objects allow you to create barcodes independently of the renderer. Renderer allow you to draw barcodes based on the support required.

CHAPTER 18

Barcode creation using Zend\Barcode\Barcode class

Using Zend\Barcode\Barcode::factory

`Zend_Barcode` uses a factory method to create an instance of a renderer that extends `Zend\Barcode\Renderer\AbstractRenderer`. The factory method accepts five arguments.

. The name of the barcode format (e.g., "code39") or a Traversable object (required)

. The name of the renderer (e.g., "image") (required)

. Options to pass to the barcode object (an array or a Traversable object) (optional)

. Options to pass to the renderer object (an array or a Traversable object) (optional)

. Boolean to indicate whether or not to automatically render errors. If an exception occurs, the provided barcode object will be replaced with an Error representation (optional default `TRUE`)

Getting a Renderer with Zend\Barcode\Barcode::factory()

`Zend\Barcode\Barcode::factory()` instantiates barcode objects and renderers and ties them together. In this first example, we will use the **Code39** barcode type together with the **Image** renderer.

```
use Zend\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();
$renderer = Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

Using Zend\Barcode\Barcode::factory() with Zend\Config\Config objects

You may pass a `Zend\Config\Config` object to the factory in order to create the necessary objects. The following example is functionally equivalent to the previous.

```
use Zend\Config;
use Zend\Barcode;

// Using only one Zend\Config\Config object
$config = new Config(array(
    'barcode'        => 'code39',
    'barcodeParams'  => array('text' => 'ZEND-FRAMEWORK'),
    'renderer'       => 'image',
    'rendererParams' => array('imageType' => 'gif'),
));

$renderer = Barcode::factory($config);
```

Drawing a barcode

When you **draw** the barcode, you retrieve the resource in which the barcode is drawn. To draw a barcode, you can call the `draw()` of the renderer, or simply use the proxy method provided by `Zend\Barcode\Barcode`.

Drawing a barcode with the renderer object

```
use Zend\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
$imageResource = Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
)->draw();
```

Drawing a barcode with Zend\Barcode\Barcode::draw()

```
use Zend\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
$imageResource = Barcode::draw(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

Renderering a barcode

When you render a barcode, you draw the barcode, you send the headers and you send the resource (e.g. to a browser). To render a barcode, you can call the `render()` method of the renderer or simply use the proxy method provided by `Zend\Barcode\Barcode`.

Renderering a barcode with the renderer object

```
use Zend\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
// send the headers and the image
Barcode::factory(
    'code39', 'image', $barcodeOptions, $rendererOptions
)->render();
```

This will generate this barcode:

Renderering a barcode with Zend\Barcode\Barcode::render()

```
use Zend\Barcode;

// Only the text to draw is required
$barcodeOptions = array('text' => 'ZEND-FRAMEWORK');

// No required options
$rendererOptions = array();

// Draw the barcode in a new image,
// send the headers and the image
Barcode::render(
    'code39', 'image', $barcodeOptions, $rendererOptions
);
```

This will generate the same barcode as the previous example.

CHAPTER 19

Zend\Barcode\Barcode Objects

Barcode objects allow you to generate barcodes independently of the rendering support. After generation, you can retrieve the barcode as an array of drawing instructions that you can provide to a renderer.

Objects have a large number of options. Most of them are common to all objects. These options can be set in three ways:

- As an array or a Traversable object) object passed to the constructor.
- As an array passed to the `setOptions()` method.
- Via individual setters for each configuration type.

Different ways to parameterize a barcode object

```
use Zend\Barcode;

$options = array('text' => 'ZEND-FRAMEWORK', 'barHeight' => 40);

// Case 1: constructor
$barcode = new Object\Code39($options);

// Case 2: setOptions()
$barcode = new Object\Code39();
$barcode->setOptions($options);

// Case 3: individual setters
$barcode = new Object\Code39();
$barcode->setText('ZEND-FRAMEWORK')
        ->setBarHeight(40);
```

Common Options

In the following list, the values have no units; we will use the term "unit." For example, the default value of the "thin bar" is "1 unit". The real units depend on the rendering support (see *the renderers documentation* for more information). Setters are each named by uppercasing the initial letter of the option and prefixing the name with "set" (e.g. "barHeight" becomes "setBarHeight"). All options have a corresponding getter prefixed with "get" (e.g. "getBarHeight"). Available options are:

Table 19.1: Common Options

Option	Data Type	Default Value	Description
barcode-Namespace	String	Zend\Barcode\Object	Namespace of the barcode; for example, if you need to extend the embedding objects
barHeight	Integer	50	Height of the bars
barThick-Width	Integer	3	Width of the thick bar
barThin-Width	Integer	1	Width of the thin bar
factor	Integer	1	Factor by which to multiply bar widths and font sizes (barHeight, barThinWidth, barThickWidth and fontSize)
foreColor	Integer	0x000000 (black)	Color of the bar and the text. Could be provided as an integer or as a HTML value (e.g. "#333333")
background-Color	Integer or String	0xFFFFFF (white)	Color of the background. Could be provided as an integer or as a HTML value (e.g. "#333333")
orientation	Float	0	Orientation of the barcode
font	String or Integer	NULL	Font path to a TTF font or a number between 1 and 5 if using image generation with GD (internal fonts)
fontSize	Float	10	Size of the font (not applicable with numeric fonts)
withBorder	Boolean	FALSE	Draw a border around the barcode and the quiet zones
withQuiet-Zones	Boolean	TRUE	Leave a quiet zone before and after the barcode
drawText	Boolean	TRUE	Set if the text is displayed below the barcode
stretchText	Boolean	FALSE	Specify if the text is stretched all along the barcode
withCheck-sum	Boolean	FALSE	Indicate whether or not the checksum is automatically added to the barcode
withCheck-sumInText	Boolean	FALSE	Indicate whether or not the checksum is displayed in the textual representation
text	String	NULL	The text to represent as a barcode

Particular case of static setBarcodeFont()

You can set a commont font for all your objects by using the static method `Zend\Barcode\Barcode::setBarcodeFont()`. This value can be always be overridden for individual objects by using the `setFont()` method.

```
use Zend\Barcode;

// In your bootstrap:
Barcode::setBarcodeFont('my_font.ttf');

// Later in your code:
```

```
Barcode::render(
    'code39',
    'pdf',
    array('text' => 'ZEND-FRAMEWORK')
); // will use 'my_font.ttf'

// or:
Barcode::render(
    'code39',
    'image',
    array(
        'text' => 'ZEND-FRAMEWORK',
        'font' => 3
    )
); // will use the 3rd GD internal font
```

Common Additional Getters

Table 19.2: Common Getters

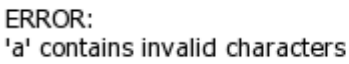
Getter	Data Type	Description
getType()	String	Return the name of the barcode class without the namespace (e.g. Zend\Barcode\Object\Code39 returns simply "code39")
getRawText()	String	Return the original text provided to the object
getTextToDisplay()	String	Return the text to display, including, if activated, the checksum value
getQuietZone()	Integer	Return the size of the space needed before and after the barcode without any drawing
getInstructions()	Array	Return drawing instructions as an array.
getHeight($recalculate = false)	Integer	Return the height of the barcode calculated after possible rotation
getWidth($recalculate = false)	Integer	Return the width of the barcode calculated after possible rotation
getOffsetTop($recalculate = false)	Integer	Return the position of the top of the barcode calculated after possible rotation
getOffsetLeft($recalculate = false)	Integer	Return the position of the left of the barcode calculated after possible rotation

CHAPTER 20

Description of shipped barcodes

You will find below detailed information about all barcode types shipped by default with Zend Framework.

Zend\Barcode\Object\Error



This barcode is a special case. It is internally used to automatically render an exception caught by the `Zend\Barcode` component.

Zend\Barcode\Object\Code128

- **Name:** Code 128
- **Allowed characters:** the complete ASCII-character set
- **Checksum:** optional (modulo 103)
- **Length:** variable

There are no particular options for this barcode.

Zend\Barcode\Object\Codabar

- **Name:** Codabar (or Code 2 of 7)
- **Allowed characters:** '0123456789-$:/.+' with 'ABCD' as start and stop characters
- **Checksum:** none
- **Length:** variable

There are no particular options for this barcode.

Zend\Barcode\Object\Code25

- **Name:** Code 25 (or Code 2 of 5 or Code 25 Industrial)
- **Allowed characters:** '0123456789'
- **Checksum:** optional (modulo 10)
- **Length:** variable

There are no particular options for this barcode.

Zend\Barcode\Object\Code25interleaved

This barcode extends `Zend\Barcode\Object\Code25` (Code 2 of 5), and has the same particulars and options, and adds the following:

- **Name:** Code 2 of 5 Interleaved
- **Allowed characters:** '0123456789'
- **Checksum:** optional (modulo 10)
- **Length:** variable (always even number of characters)

Available options include:

Table 20.1: Zend\Barcode\Object\Code25interleaved Options

Option	Data Type	Default Value	Description
withBearerBars	Boolean	FALSE	Draw a thick bar at the top and the bottom of the barcode.

Note: If the number of characters is not even, `Zend\Barcode\Object\Code25interleaved` will automatically prepend the missing zero to the barcode text.

Zend\Barcode\Object\Ean2

This barcode extends `Zend\Barcode\Object\Ean5` (*EAN* 5), and has the same particulars and options, and adds the following:

- **Name:** *EAN*-2
- **Allowed characters:** '0123456789'
- **Checksum:** only use internally but not displayed
- **Length:** 2 characters

There are no particular options for this barcode.

Note: If the number of characters is lower than 2, `Zend\Barcode\Object\Ean2` will automatically prepend the missing zero to the barcode text.

Zend\Barcode\Object\Ean5

This barcode extends `Zend\Barcode\Object\Ean13` (*EAN* 13), and has the same particulars and options, and adds the following:

- **Name:** *EAN*-5
- **Allowed characters:** '0123456789'
- **Checksum:** only use internally but not displayed
- **Length:** 5 characters

There are no particular options for this barcode.

Note: If the number of characters is lower than 5, `Zend\Barcode\Object\Ean5` will automatically prepend the missing zero to the barcode text.

Zend\Barcode\Object\Ean8

This barcode extends `Zend\Barcode\Object\Ean13` (*EAN* 13), and has the same particulars and options, and adds the following:

- **Name:** *EAN*-8
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 8 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 8, `Zend\Barcode\Object\Ean8` will automatically prepend the missing zero to the barcode text.

Zend\Barcode\Object\Ean13

- **Name:** *EAN*-13
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 13 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 13, `Zend\Barcode\Object\Ean13` will automatically prepend the missing zero to the barcode text.

The option `withQuietZones` has no effect with this barcode.

Zend\Barcode\Object\Code39

- **Name:** Code 39
- **Allowed characters:** '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ -.$/+%'
- **Checksum:** optional (modulo 43)
- **Length:** variable

Note: `Zend\Barcode\Object\Code39` will automatically add the start and stop characters ('*') for you.

There are no particular options for this barcode.

Zend\Barcode\Object\Identcode

This barcode extends `Zend\Barcode\Object\Code25interleaved` (Code 2 of 5 Interleaved), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** Identcode (Deutsche Post Identcode)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10 different from Code25)
- **Length:** 12 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 12, `Zend\Barcode\Object\Identcode` will automatically prepend missing zeros to the barcode text.

Zend\Barcode\Object\Itf14

This barcode extends `Zend\Barcode\Object\Code25interleaved` (Code 2 of 5 Interleaved), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** *ITF*-14

- **Allowed characters:**'0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 14 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 14, `Zend\Barcode\Object\Itf14` will automatically prepend missing zeros to the barcode text.

Zend\Barcode\Object\Leitcode

This barcode extends `Zend\Barcode\Object\Identcode` (Deutsche Post Identcode), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** Leitcode (Deutsche Post Leitcode)
- **Allowed characters:**'0123456789'
- **Checksum:** mandatory (modulo 10 different from Code25)
- **Length:** 14 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 14, `Zend\Barcode\Object\Leitcode` will automatically prepend missing zeros to the barcode text.

Zend\Barcode\Object\Planet

- **Name:** Planet (PostaL Alpha Numeric Encoding Technique)
- **Allowed characters:**'0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 12 or 14 characters (including checksum)

There are no particular options for this barcode.

Zend\Barcode\Object\Postnet

- **Name:** Postnet (POSTal Numeric Encoding Technique)
- **Allowed characters:**'0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 6, 7, 10 or 12 characters (including checksum)

There are no particular options for this barcode.

Zend\Barcode\Object\Royalmail

- **Name:** Royal Mail or *RM4SCC* (Royal Mail 4-State Customer Code)
- **Allowed characters:**'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ'
- **Checksum:** mandatory
- **Length:** variable

There are no particular options for this barcode.

Zend\Barcode\Object\Upca

This barcode extends `Zend\Barcode\Object\Ean13` (*EAN*-13), and inherits some of its capabilities; it also has a few particulars of its own.

- **Name:** *UPC*-A (Universal Product Code)
- **Allowed characters:**'0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 12 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 12, `Zend\Barcode\Object\Upca` will automatically prepend missing zeros to the barcode text.

The option `withQuietZones` has no effect with this barcode.

Zend\Barcode\Object\Upce

This barcode extends `Zend\Barcode\Object\Upca` (*UPC*-A), and inherits some of its capabilities; it also has a few particulars of its own. The first character of the text to encode is the system (0 or 1).

- **Name:** *UPC*-E (Universal Product Code)
- **Allowed characters:** '0123456789'
- **Checksum:** mandatory (modulo 10)
- **Length:** 8 characters (including checksum)

There are no particular options for this barcode.

Note: If the number of characters is lower than 8, `Zend\Barcode\Object\Upce` will automatically prepend missing zeros to the barcode text.

Note: If the first character of the text to encode is not 0 or 1, `Zend\Barcode\Object\Upce` will automatically replace it by 0.

The option `withQuietZones` has no effect with this barcode.

CHAPTER 21

Zend\Barcode Renderers

Renderers have some common options. These options can be set in three ways:

- As an array or a Traversable object passed to the constructor.
- As an array passed to the `setOptions()` method.
- As discrete values passed to individual setters.

Different ways to parameterize a renderer object

```
use Zend\Barcode;

$options = array('topOffset' => 10);

// Case 1
$renderer = new Renderer\Pdf($options);

// Case 2
$renderer = new Renderer\Pdf();
$renderer->setOptions($options);

// Case 3
$renderer = new Renderer\Pdf();
$renderer->setTopOffset(10);
```

Common Options

In the following list, the values have no unit; we will use the term “unit.” For example, the default value of the “thin bar” is “1 unit.” The real units depend on the rendering support. The individual setters are obtained by uppercasing the initial letter of the option and prefixing the name with “set” (e.g. “barHeight” => “setBarHeight”). All options have a correspondant getter prefixed with “get” (e.g. “getBarHeight”). Available options are:

Table 21.1: Common Options

Option	Data Type	Default Value	Description
rendererNamespace	String	Zend\Barcode\Renderer	Namespace of the renderer; for example, if you need to extend the renderers
horizontalPosition	String	"left"	Can be "left", "center" or "right". Can be useful with PDF or if the setWidth() method is used with an image renderer.
verticalPosition	String	"top"	Can be "top", "middle" or "bottom". Can be useful with PDF or if the setHeight() method is used with an image renderer.
leftOffset	Integer	0	Top position of the barcode inside the renderer. If used, this value will override the "horizontalPosition" option.
topOffset	Integer	0	Top position of the barcode inside the renderer. If used, this value will override the "verticalPosition" option.
automaticRenderError	Boolean	FALSE	Whether or not to automatically render errors. If an exception occurs, the provided barcode object will be replaced with an Error representation. Note that some errors (or exceptions) can not be rendered.
moduleSize	Float	1	Size of a rendering module in the support.
barcode	Zend\Barcode\Object	NULL	The barcode object to render.

An additional getter exists: `getType()`. It returns the name of the renderer class without the namespace (e.g. `Zend\Barcode\Renderer\Image` returns "image").

Zend\Barcode\Renderer\Image

The Image renderer will draw the instruction list of the barcode object in an image resource. The component requires the GD extension. The default width of a module is 1 pixel.

Available options are:

Table 21.2: Zend\Barcode\Renderer\Image Options

Option	Data Type	Default Value	Description
height	Integer	0	Allow you to specify the height of the result image. If "0", the height will be calculated by the barcode object.
width	Integer	0	Allow you to specify the width of the result image. If "0", the width will be calculated by the barcode object.
imageType	String	"png"	Specify the image format. Can be "png", "jpeg", "jpg" or "gif".

Zend\Barcode\Renderer\Pdf

The *PDF* renderer will draw the instruction list of the barcode object in a *PDF* document. The default width of a module is 0.5 point.

There are no particular options for this renderer.

CHAPTER 22

Zend\Cache\Storage\Adapter

Overview

Storage adapters are wrappers for real storage resources such as memory and the filesystem, using the well known adapter pattern.

They comes with tons of methods to read, write and modify stored items and to get information about stored items and the storage.

All adapters implements the interface `Zend\Cache\Storage\StorageInterface` and most extend `Zend\Cache\Storage\Adapter\AbstractAdapter`, which comes with basic logic.

Configuration is handled by either `Zend\Cache\Storage\Adapter\AdapterOptions`, or an adapter-specific options class if it exists. You may pass the options instance to the class at instantiation or via the `setOptions()` method, or alternately pass an associative array of options in either place (internally, these are then passed to an options class instance). Alternately, you can pass either the options instance or associative array to the `Zend\Cache\StorageFactory::factory` method.

Note: Many methods throw exceptions

Because many caching methods can throw exceptions, you need to catch them manually or you can use the plug-in `Zend\Cache\Storage\Plugin\ExceptionHandler` to automatically catch them and redirect exceptions into a log file using the option "exception_callback".

Quick Start

Caching adapters can either be created from the provided `Zend\Cache\StorageFactory` factory, or by simply instantiating one of the `Zend\Cache\Storage\Adapter\*`classes.

To make life easier, the `Zend\Cache\StorageFactory` comes with a `factory` method to create an adapter and create/add all requested plugins at once.

```php
use Zend\Cache\StorageFactory;

// Via factory:
$cache = StorageFactory::factory(array(
    'adapter' => 'apc',
    'plugins' => array(
        'exception_handler' => array('throw_exceptions' => false),
    ),
));

// Alternately:
$cache  = StorageFactory::adapterFactory('apc');
$plugin = StorageFactory::pluginFactory('exception_handler', array(
    'throw_exceptions' => false,
));
$cache->addPlugin($plugin);

// Or manually:
$cache  = new Zend\Cache\Storage\Adapter\Apc();
$plugin = new Zend\Cache\Storage\Plugin\ExceptionHandler(array(
    'throw_exceptions' => false,
));
$cache->addPlugin($plugin);
```

Basic configuration Options

key_pattern Pattern against which to validate cache keys.

- `setKeyPattern(null|string $pattern)` Implements a fluent interface.
- `getKeyPattern()` Returns string

namespace The "namespace" in which cache items will live.

- `setNamespace(string $namespace)` Implements a fluent interface.
- `getNamespace()` Returns string

readable Enable/Disable reading data from cache.

- `setReadable(boolean $flag)` Implements a fluent interface.
- `getReadable()` Returns boolean

ttl Set time to live.

- `setTtl(int|float $ttl)` Implements a fluent interface.
- `getTtl()` Returns float

writable Enable/Disable writing data to cache.

- `setWritable(boolean $flag)` Implements a fluent interface.
- `getWritable()` Returns boolean

Available Methods defined by `Zend\Cache\Storage\StorageInterface`

setOptions `setOptions(array|Traversable|Zend\Cache\Storage\Adapter\AdapterOptions $options)`

Set options.

Implements a fluent interface.

getOptions `getOptions()`

Get options

Returns Zend\Cache\Storage\Adapter\AdapterOptions

getItem `getItem(string $key, boolean & $success = null, mixed & $casToken = null)`

Load an item with the given $key, set parameter $success to TRUE and set parameter $casToken.

If item can't load this method returns NULL and set parameter $success to FALSE.

getItems `getItems(array $keys)`

Load all items given by $keys.

Returns an array of key-value pairs of available items.

hasItem `hasItem(string $key)`

Test if an item exists.

Returns boolean

hasItems `hasItems(array $keys)`

Test multiple items.

Returns array

getMetadata `getMetadata(string $key)`

Get metadata of an item.

Returns array|boolean

getMetadatas `getMetadatas(array $keys)`

Get multiple metadata

Returns array

setItem `setItem(string $key, mixed $value)`

Store an item.

Returns boolean

setItems `setItems(array $keyValuePairs)`

Store multiple items.

Returns boolean

addItem `addItem(string $key, mixed $value)`

Add an item.

Returns boolean

addItems `addItems(array $keyValuePairs)`

Add multiple items.

Returns boolean

replaceItem `replaceItem(string $key, mixed $value)`

Replace an item.

Returns boolean

replaceItems `replaceItems(array $keyValuePairs)`

Replace multiple items.

Returns boolean

checkAndSetItem `checkAndSetItem(mixed $token, string $key, mixed $value)`

Set item only if token matches

It uses the token from received from `getItem()` to check if the item has changed before overwriting it.

Returns boolean

touchItem `touchItem(string $key)`

Reset lifetime of an item

Returns boolean

touchItems `touchItems(array $keys)`

Reset lifetime of multiple items.

Returns boolean

removeItem `removeItem(string $key)`

Remove an item.

Returns boolean

removeItems `removeItems(array $keys)`

Remove multiple items.

Returns boolean

incrementItem `incrementItem(string $key, int $value)`

Increment an item.

Returns int|boolean

incrementItems `incrementItems(array $keyValuePairs)`

Increment multiple items.

Returns boolean

decrementItem `decrementItem(string $key, int $value)`

Decrement an item.

Returns int|boolean

decrementItems `decrementItems(array $keyValuePairs)`

Decrement multiple items.

Returns boolean

getCapabilities `getCapabilities()`

Capabilities of this storage

Returns Zend\Cache\Storage\Capabilities

Available Methods defined by `Zend\Cache\Storage\AvailableSpaceCapable`

getAvailableSpace `getAvailableSpace()`

Get available space in bytes

Returns int|float

Available Methods defined by `Zend\Cache\Storage\TotalSpaceCapableInte`

getTotalSpace `getTotalSpace()`

Get total space in bytes

Returns int|float

Available Methods defined by `Zend\Cache\Storage\ClearByNamespaceInter`

clearByNamespace `clearByNamespace(string $namespace)`

Remove items of given namespace

Returns boolean

Available Methods defined by `Zend\Cache\Storage\ClearByPrefixInterfac`

clearByPrefix `clearByPrefix(string $prefix)`

Remove items matching given prefix

Returns boolean

Available Methods defined by `Zend\Cache\Storage\ClearExpiredInterface`

clearExpired `clearExpired()`

Remove expired items

Returns boolean

Available Methods defined by `Zend\Cache\Storage\FlushableInterface`

flush `flush()`

Flush the whole storage

Returns boolean

Available Methods defined by `Zend\Cache\Storage\IterableInterface` (extends `IteratorAggregate`)

getIterator `getIterator()`

Get an Iterator

Returns `Zend\Cache\Storage\IteratorInterface`

Available Methods defined by `Zend\Cache\Storage\OptimizableInterface`

optimize `optimize()`

Optimize the storage

Returns boolean

Available Methods defined by `Zend\Cache\Storage\TaggableInterface`

setTags `setTags(string $key, string[] $tags)`

Set tags to an item by given key. An empty array will remove all tags.

Returns boolean

getTags `getTags(string $key)`

Get tags of an item by given key

Returns string[]|FALSE

clearByTags `clearByTags(string[] $tags, boolean $disjunction = false)`

Remove items matching given tags.

If $disjunction only one of the given tags must match else all given tags must match.

Returns boolean

TODO: Examples

CHAPTER 23

Zend\Cache\Storage\Capabilities

Overview

Storage capabilities describes how a storage adapter works and which features it supports.

To get capabilities of a storage adapter, you can use the method `getCapabilities()` of the storage adapter but only the storage adapter and its plugins have permissions to change them.

Because capabilities are mutable, for example, by changing some options, you can subscribe to the "change" event to get notifications; see the examples for details.

If you are writing your own plugin or adapter, you can also change capabilities because you have access to the marker object and can create your own marker to instantiate a new object of `Zend\Cache\Storage\Capabilities`.

Available Methods

__construct `__construct(stdClass $marker, array $capabilities = array ( ), null|Zend\Cache\Storage\Capabilities $baseCapabilities) __construct(Zend\Cache\Storage\StorageInterface $storage, stdClass $marker, array $capabilities = array(), Capabilities $baseCapabilities = null)`

Constructor

getSupportedDatatypes `getSupportedDatatypes()`

Get supported datatypes.

Returns array.

setSupportedDatatypes `setSupportedDatatypes(stdClass $marker, array $datatypes)`

Set supported datatypes.

Implements a fluent interface.

getSupportedMetadata `getSupportedMetadata()`

Get supported metadata.

Returns array.

setSupportedMetadata `setSupportedMetadata(stdClass $marker, string $metadata)`

Set supported metadata

Implements a fluent interface.

getMinTtl `getMinTtl()`

Get minimum supported time-to-live

Returns int (0 means items never expire)

setMinTtl `setMinTtl(stdClass $marker, int $minTtl)`

Set minimum supported time-to-live

Implements a fluent interface.

getMaxTtl `getMaxTtl()`

Get maximum supported time-to-live

Returns int

setMaxTtl `setMaxTtl(stdClass $marker, int $maxTtl)`

Set maximum supported time-to-live

Implements a fluent interface.

getStaticTtl `getStaticTtl()`

Is the time-to-live handled static (on write), or dynamic (on read).

Returns boolean

setStaticTtl `setStaticTtl(stdClass $marker, boolean $flag)`

Set if the time-to-live is handled statically (on write) or dynamically (on read)

Implements a fluent interface.

getTtlPrecision `getTtlPrecision()`

Get time-to-live precision.

Returns float.

setTtlPrecision `setTtlPrecision(stdClass $marker, float $ttlPrecision)`

Set time-to-live precision.

Implements a fluent interface.

getUseRequestTime `getUseRequestTime()`

Get the “use request time” flag status

Returns boolean

setUseRequestTime `setUseRequestTime(stdClass $marker, boolean $flag)`

Set the “use request time” flag.

Implements a fluent interface.

getExpiredRead `getExpiredRead()`

Get flag indicating if expired items are readable.

Returns boolean

setExpiredRead `setExpiredRead(stdClass $marker, boolean $flag)`

Set if expired items are readable.

Implements a fluent interface.

getMaxKeyLength `getMaxKeyLength()`

Get maximum key lenth.

Returns int

setMaxKeyLength `setMaxKeyLength(stdClass $marker, int $maxKeyLength)`

Set maximum key lenth.

Implements a fluent interface.

getNamespaceIsPrefix `getNamespaceIsPrefix()`

Get if namespace support is implemented as a key prefix.

Returns boolean

setNamespaceIsPrefix `setNamespaceIsPrefix(stdClass $marker, boolean $flag)`

Set if namespace support is implemented as a key prefix.

Implements a fluent interface.

getNamespaceSeparator `getNamespaceSeparator()`

Get namespace separator if namespace is implemented as a key prefix.

Returns string

setNamespaceSeparator `setNamespaceSeparator(stdClass $marker, string $separator)`

Set the namespace separator if namespace is implemented as a key prefix.

Implements a fluent interface.

Examples

Get storage capabilities and do specific stuff in base of it

```
use Zend\Cache\StorageFactory;

$cache = StorageFactory::adapterFactory('filesystem');
$supportedDatatypes = $cache->getCapabilities()->getSupportedDatatypes();

// now you can run specific stuff in base of supported feature
if ($supportedDatatypes['object']) {
    $cache->set($key, $object);
} else {
    $cache->set($key, serialize($object));
}
```

Listen to change event

```php
use Zend\Cache\StorageFactory;

$cache = StorageFactory::adapterFactory('filesystem', array(
    'no_atime' => false,
));

// Catching capability changes
$cache->getEventManager()->attach('capability', function($event) {
    echo count($event->getParams()) . ' capabilities changed';
});

// change option which changes capabilities
$cache->getOptions()->setNoATime(true);
```

CHAPTER 24

Zend\Cache\Storage\Plugin

Overview

Cache storage plugins are objects to add missing functionality or to influence behavior of a storage adapter.

The plugins listen to events the adapter triggers and can change called method arguments (*.post - events), skipping and directly return a result (using `stopPropagation`), changing the result (with `setResult` of `Zend\Cache\Storage\PostEvent`) and catching exceptions (with `Zend\Cache\Storage\ExceptionEvent`).

Quick Start

Storage plugins can either be created from `Zend\Cache\StorageFactory` with the `pluginFactory`, or by simply instantiating one of the `Zend\Cache\Storage\Plugin\*`classes.

To make life easier, the `Zend\Cache\StorageFactory` comes with the method `factory` to create an adapter and all given plugins at once.

```
use Zend\Cache\StorageFactory;

// Via factory:
$cache = StorageFactory::factory(array(
    'adapter' => 'filesystem',
    'plugins' => array('serializer'),
));

// Alternately:
$cache  = StorageFactory::adapterFactory('filesystem');
$plugin = StorageFactory::pluginFactory('serializer');
$cache->addPlugin($plugin);

// Or manually:
$cache  = new Zend\Cache\Storage\Adapter\Filesystem();
```

```
16 $plugin = new Zend\Cache\Storage\Plugin\Serializer();
17 $cache->addPlugin($plugin);
```

Configuration Options

clearing_factor Set the automatic clearing factor. Used by the `ClearByFactor` plugin.

- `setClearingFactor(int $clearingFactor)` Implements a fluent interface.
- `getClearingFactor()` Returns int

clear_by_namespace Flag indicating whether or not to clear by namespace. Used by the `ClearByFactor` plugin.

- `setClearByNamespace(bool $clearByNamespace)` Implements a fluent interface.
- `getClearByNamespace()` Returns bool

exception_callback Set callback to call on intercepted exception. Used by the `ExceptionHandler` plugin.

- `setExceptionCallback(callable $exceptionCallback)` Implements a fluent interface.
- `getExceptionCallback()` Returns null|callable

optimizing_factor Set automatic optimizing factor. Used by the `OptimizeByFactor` plugin.

- `setOptimizingFactor(int $optimizingFactor)` Implements a fluent interface.
- `getOptimizingFactor()` Returns int

serializer Set serializer adapter to use. Used by `Serializer` plugin.

- `setSerializer(string|Zend\Serializer\Adapter $serializer)` Implements a fluent interface.
- `getSerializer()` Returns Zend\Serializer\Adapter

serializer_options Set configuration options for instantiating a serializer adapter. Used by the `Serializer` plugin.

- `setSerializerOptions(array $serializerOptions)` Implements a fluent interface.
- `getSerializerOptions()` Returns array

throw_exceptions Set flag indicating we should re-throw exceptions. Used by the `ExceptionHandler` plugin.

- `setThrowExceptions(bool $throwExceptions)` Implements a fluent interface.
- `getThrowExceptions()` Returns bool

Available Methods

setOptions `setOptions(Zend\Cache\Storage\Plugin\PluginOptions $options)`

Set options

Implements a fluent interface.

getOptions `getOptions()`

Get options

Returns PluginOptions

attach `attach(EventCollection $events)`

Defined by `Zend\EventManager\ListenerAggregate`, attach one or more listeners.

Returns void

detach `detach(EventCollection $events)`

Defined by `Zend\EventManager\ListenerAggregate`, detach all previously attached listeners.

Returns void

TODO: Examples

CHAPTER 25

Zend\Cache\Pattern

Overview

Cache patterns are configurable objects to solve known performance bottlenecks. Each should be used only in the specific situations they are designed to address. For example you can use one of the `CallbackCache`, `ObjectCache` or `ClassCache` patterns to cache method and function calls; to cache output generation, the `OutputCache` pattern could assist.

All cache patterns implements the same interface, `Zend\Cache\Pattern`, and most extend the abstract class `Zend\Cache\Pattern\AbstractPattern` to implement basic logic.

Configuration is provided via the `Zend\Cache\Pattern\PatternOptions` class, which can simply be instantiated with an associative array of options passed to the constructor. To configure a pattern object, you can set an instance of `Zend\Cache\Pattern\PatternOptions` with `setOptions`, or provide your options (either as an associative array or `PatternOptions` instance) as the second argument to the factory.

It's also possible to use a single instance of `Zend\Cache\Pattern\PatternOptions` and pass it to multiple pattern objects.

Quick Start

Pattern objects can either be created from the provided `Zend\Cache\PatternFactory` factory, or, by simply instantiating one of the `Zend\Cache\Pattern\*` classes.

```
use Zend\Cache\PatternFactory;
use Zend\Cache\Pattern\PatternOptions;

// Via the factory:
$callbackCache = PatternFactory::factory('callback', array(
    'storage'      => 'apc',
    'cache_output' => true,
));

```

```php
// OR, the equivalent manual instantiation:
$callbackCache = new \Zend\Cache\Pattern\CallbackCache();
$callbackCache->setOptions(new PatternOptions(array(
    'storage'      => 'apc',
    'cache_output' => true,
)));
```

Configuration Options

cache_by_default Flag indicating whether or not to cache by default. Used by the `ClassCache` and `ObjectCache` patterns.

- `setCacheByDefault(bool $cacheByDefault)` Implements a fluent interface.
- `getCacheByDefault()` Returns boolean.

cache_output Used by the `CallbackCache`, `ClassCache`, and `ObjectCache` patterns. Flag used to determine whether or not to cache output.

- `setCacheOutput(bool $cacheOutput)` Implements a fluent interface.
- `getCacheOutput()` Returns boolean

class Set the name of the class to cache. Used by the `ClassCache` pattern.

- `setclass(string $class)` Implements a fluent interface.
- `getClass()` Returns nulllstring

class_cache_methods Set list of method return values to cache. Used by `ClassCache` Pattern.

- `setClassCacheMethods(array $classCacheMethods)` Implements a fluent interface.
- `getClassCacheMethods()` Returns array

class_non_cache_methods Set list of method return values that should **not** be cached. Used by the `ClassCache` pattern.

- `setClassNonCacheMethods(array $classNonCacheMethods)` Implements a fluent interface.
- `getClassNonCacheMethods()` Returns array

dir_perm Set directory permissions; proxies to "dir_umask" property, setting the inverse of the provided value. Used by the `CaptureCache` pattern.

- `setDirPerm(string|int $dirPerm)` Implements a fluent interface.
- `getDirPerm()` Returns int

dir_umask Set the directory umask value. Used by the `CaptureCache` pattern.

- `setDirUmask(int $dirUmask)` Implements a fluent interface.
- `getDirUmask()` Returns int

file_locking Set whether or not file locking should be used. Used by the `CaptureCache` pattern.

- `setFileLocking(bool $fileLocking)` Implements a fluent interface.
- `getFileLocking()` Returns bool

file_perm Set file permissions; proxies to the "file_umask" property, setting the inverse of the value provided. Used by the `CaptureCache` pattern.

- `setFilePerm(int|string $filePerm)` Implements a fluent interface.
- `getFilePerm()` Returns int

file_umask Set file umask; used by the `CaptureCache` pattern.

- `setFileUmask(int $fileUmask)` Implements a fluent interface.
- `getFileUmask()` Returns int

index_filename Set value for index filename. Used by the `CaptureCache` pattern.

- `setIndexFilename(string $indexFilename)` Implements a fluent interface.
- `getIndexFilename()` Returns string

object Set object to cache; used by the `ObjectCache` pattern.

- `setObject(object $object)` Implements a fluent interface.
- `getObject()` Returns null|object.

object_cache_magic_properties Set flag indicating whether or not to cache magic properties. Used by the `ObjectCache` pattern.

- `setObjectCacheMagicProperties(bool $objectCacheMagicProperties)` Implements a fluent interface.
- `getObjectCacheMagicProperties()` Returns bool

object_cache_methods Set list of object methods for which to cache return values. Used by `ObjectCache` pattern.

- `setObjectCacheMethods(array $objectCacheMethods)` Implements a fluent interface.
- `getObjectCacheMethods()` Returns array

object_key Set the object key part; used to generate a callback key in order to speed up key generation. Used by the `ObjectCache` pattern.

- `setObjectKey(null|string $objectKey)` Implements a fluent interface.
- `getObjectKey()` Returns null|string

object_non_cache_methods Set list of object methods for which **not** to cache return values. Used by the `ObjectCache` pattern.

- `setObjectNonCacheMethods(array $objectNonCacheMethods)` Implements a fluent interface.
- `getObjectNonCacheMethods()` Returns array

public_dir Set location of public directory; used by the `CaptureCache` pattern.

- `setPublicDir()` Implements a fluent interface.
- `getPublicDir()` Returns null|string

storage Set the storage adapter. Required for the following Pattern classes: `CallbackCache`, `ClassCache`, `ObjectCache`, `OutputCache`.

- `setStorage(string|array|Zend\Cache\Storage\Adapter $storage)` Implements a fluent interface.
- `getStorage()` Returns null|Zend\Cache\Storage\Adapter

tag_key Set the prefix used for tag keys. Used by the `CaptureCache` pattern.

- `setTagKey(string $tagKey)` Implements a fluent interface.

- `getTagKey()` Returns string

tags Set list of tags to use for captured content. Used by the `CaptureCache` pattern.

- `setTags(array $tags)` Implements a fluent interface.
- `getTags()` Returns array

Set storage adapter to use for tags. Used by the `CaptureCache` pattern.

- `setTagStorage(string|array|Zend\Cache\Storage\Adapter $tagStorage)` Implements a fluent interface.
- `getTagStorage()` Returns null|Zend\Cache\Storage\Adapter

Available Methods

setOptions `setOptions(Zend\Cache\Pattern\PatternOptions $options)`

Set pattern options

Returns Zend\Cache\Pattern

getOptions `getOptions()`

Get all pattern options

Returns `PatternOptions` instance.

Examples

Using the callback cache pattern

```
use Zend\Cache\PatternFactory;

$callbackCache = PatternFactory::factory('callback', array(
    'storage' => 'apc'
));

// Calls and caches the function doResourceIntensiceStuff with three arguments
// and returns result
$result = $callbackCache->call('doResourceIntensiveStuff', array(
    'argument1',
    'argument2',
    'argumentN',
));
```

Using the object cache pattern

```
use Zend\Cache\PatternFactory;

$object      = new MyObject();
$objectProxy = PatternFactory::factory('object', array(
    'object'  => $object,
    'storage' => 'apc',
```

```
));

// Calls and caches $object->doResourceIntensiveStuff with three arguments
// and returns result
$result = $objectProxy->doResourceIntensiveStuff('argument1', 'argument2', 'argumentN
↪');
```

Using the class cache pattern

```
use Zend\Cache\PatternFactory;

$classProxy = PatternFactory::factory('class', array(
    'class'   => 'MyClass',
    'storage' => 'apc',
));

// Calls and caches MyClass::doResourceIntensiveStuff with three arguments
// and returns result
$result = $classProxy->doResourceIntensiveStuff('argument1', 'argument2', 'argumentN
↪');
```

Using the output cache pattern

```
use Zend\Cache\PatternFactory;

$outputCache = PatternFactory::factory('output', array(
    'storage' => 'filesystem',
));

// Start capturing all output (excluding headers) and write it to storage.
// If there is already a cached item with the same key it will be
// output and return true, else false.
if ($outputCache->start('MyUniqueKey') === false) {
    echo 'cache output since: ' . date('H:i:s') . "<br />\n";

    // end capturing output, write content to cache storage and display
    // captured content
    $outputCache->end();
}

echo 'This output is never cached.';
```

Using the capture cache pattern

You need to configure your HTTP server to redirect missing content to run your script generating it.

This example uses Apache with the following .htaccess:

```
ErrorDocument 404 /index.php
```

Within your index.php you can add the following content:

```php
use Zend\Cache\PatternFactory;

$capture = PatternFactory::factory('capture', array(
    'public_dir' => __DIR__,
));

// Start capturing all output excl. headers. and write to public directory
// If the request was already written the file will be overwritten.
$capture->start();

// do stuff to dynamically generate output
```

CHAPTER 26

Introduction

CAPTCHA stands for "Completely Automated Public Turing test to tell Computers and Humans Apart"; it is used as a challenge-response to ensure that the individual submitting information is a human and not an automated process. Typically, a captcha is used with form submissions where authenticated users are not necessary, but you want to prevent spam submissions.

Captchas can take a variety of forms, including asking logic questions, presenting skewed fonts, and presenting multiple images and asking how they relate. The `Zend\Captcha` component aims to provide a variety of back ends that may be utilized either standalone or in conjunction with the `Zend\Form` component.

CHAPTER 27

Captcha Operation

All *CAPTCHA* adapter implement `Zend\Captcha\AdapterInterface`, which looks like the following:

```
namespace Zend\Captcha;

use Zend\Validator\ValidatorInterface;

interface AdapterInterface extends ValidatorInterface
{
    public function generate();

    public function setName($name);

    public function getName();

    // Get helper name used for rendering this captcha type
    public function getHelperName();
}
```

The name setter and getter are used to specify and retrieve the *CAPTCHA* identifier. The most interesting methods are `generate()` and `render()`. `generate()` is used to create the *CAPTCHA* token. This process typically will store the token in the session so that you may compare against it in subsequent requests. `render()` is used to render the information that represents the *CAPTCHA*, be it an image, a figlet, a logic problem, or some other *CAPTCHA*.

A simple use case might look like the following:

```
// Originating request:
$captcha = new Zend\Captcha\Figlet(array(
    'name' => 'foo',
    'wordLen' => 6,
    'timeout' => 300,
));

$id = $captcha->generate();

//this will output a Figlet string
```

```php
echo $captcha->getFiglet()->render($captcha->getWord());


// On a subsequent request:
// Assume a captcha setup as before, with corresponding form fields, the value of $_POST['foo']
// would be key/value array: id => captcha ID, input => captcha value
if ($captcha->isValid($_POST['foo'], $_POST)) {
    // Validated!
}
```

Note: Under most circumstances, you probably prefer the use of `Zend\Captcha` functionality combined with the power of the `Zend\Form` component. For an example on how to use `Zend\Form\Element\Captcha`, have a look at the *Zend\Form Quick Start*.

CHAPTER 28

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 base class for most other *CAPTCHA* adapters. It provides mutators for specifying word length, session *TTL* and the session container object to use. `Zend\Captcha\Word` also encapsulates validation logic.

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

In addition to the methods required by the `Zend\Captcha\AdapterInterface` interface, `Zend\Captcha\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. `$ttl` should be specified in seconds.
- `setUseNumbers($numbers)` and `getUseNumbers()` allow you to specify if numbers will be considered as possible characters for the random work or only letters would be used.
- `setSessionClass($class)` and `getSessionClass()` allow you to specify an alternate `Zend\Session\Container` implementation to use to persist the *CAPTCHA* token and to retrieve the current value.
- `getId()` allows you to retrieve the current token identifier.
- `getWord()` allows you to retrieve the generated word to use with the *CAPTCHA*. It will generate the word for you if none has been generated yet.
- `setSession(Zend\Session\Container $session)` allows you to specify a session object to use for persisting the *CAPTCHA* token. `getSession()` allows you to retrieve the current session object.

All word *CAPTCHA*s allow you to pass an array of options or `Traversable` object to the constructor, or, alternately, pass them to `setOptions()`. By default, the **wordLen**, **timeout**, and **sessionClass** keys may all be used. Each concrete implementation may define additional keys or utilize the options in other ways.

Note: `Zend\Captcha\Word` is an abstract class and may not be instantiated directly.

Zend\Captcha\Dumb

The `Zend\Captcha\Dumb` adapter is mostly self-descriptive. It provides a random string that must be typed in reverse to validate. As such, it's not a good *CAPTCHA* solution and should only be used for testing. It extends `Zend\Captcha\Word`.

Zend\Captcha\Figlet

The `Zend\Captcha\Figlet` adapter utilizes Zend\Text\Figlet to present a figlet to the user.

Options passed to the constructor will also be passed to the Zend\Text\Figlet object. See the Zend\Text\Figlet documentation for details on what configuration options are available.

Zend\Captcha\Image

The `Zend\Captcha\Image` adapter takes the generated word and renders it as an image, performing various skewing permutations to make it difficult to automatically decipher. It requires the GD extension compiled with TrueType or Freetype support. Currently, the `Zend\Captcha\Image` adapter can only generate *PNG* images.

`Zend\Captcha\Image` extends `Zend\Captcha\Word`, and additionally exposes the following methods:

- `setExpiration($expiration)` and `getExpiration()` allow you to specify a maximum lifetime the *CAPTCHA* image may reside on the filesystem. This is typically a longer than the session lifetime. Garbage collection is run periodically each time the *CAPTCHA* object is invoked, deleting all images that have expired. Expiration values should be specified in seconds.
- `setGcFreq($gcFreq)` and `getGcFreg()` allow you to specify how frequently garbage collection should run. Garbage collection will run every `1/$gcFreq` calls. The default is 100.
- `setFont($font)` and `getFont()` allow you to specify the font you will use. `$font` should be a fully qualified path to the font file. This value is required; the *CAPTCHA* will throw an exception during generation if the font file has not been specified.
- `setFontSize($fsize)` and `getFontSize()` allow you to specify the font size in pixels for generating the *CAPTCHA*. The default is 24px.
- `setHeight($height)` and `getHeight()` allow you to specify the height in pixels of the generated *CAPTCHA* image. The default is 50px.
- `setWidth($width)` and `getWidth()` allow you to specify the width in pixels of the generated *CAPTCHA* image. The default is 200px.
- `setImgDir($imgDir)` and `getImgDir()` allow you to specify the directory for storing *CAPTCHA* images. The default is "`./images/captcha/`", relative to the bootstrap script.

- `setImgUrl($imgUrl)` and `getImgUrl()` allow you to specify the relative path to a *CAPTCHA* image to use for *HTML* markup. The default is "`/images/captcha/`".
- `setSuffix($suffix)` and `getSuffix()` allow you to specify the filename suffix for the *CAPTCHA* image. The default is "`.png`". Note: changing this value will not change the type of the generated image.
- `setDotNoiseLevel($level)` and `getDotNoiseLevel()`, along with `setLineNoiseLevel($level)` and `getLineNoiseLevel()`, allow you to control how much "noise" in the form of random dots and lines the image would contain. Each unit of `$level` produces one random dot or line. The default is 100 dots and 5 lines. The noise is added twice - before and after the image distortion transformation.

All of the above options may be passed to the constructor by simply removing the 'set' method prefix and casting the initial letter to lowercase: "suffix", "height", "imgUrl", etc.

Zend\Captcha\ReCaptcha

The `Zend\Captcha\ReCaptcha` adapter uses Zend\Service\ReCaptcha\ReCaptcha to generate and validate *CAPTCHA*s. It exposes the following methods:

- `setPrivKey($key)` and `getPrivKey()` allow you to specify the private key to use for the ReCaptcha service. This must be specified during construction, although it may be overridden at any point.
- `setPubKey($key)` and `getPubKey()` allow you to specify the public key to use with the ReCaptcha service. This must be specified during construction, although it may be overridden at any point.
- `setService(Zend\Service\ReCaptcha\ReCaptcha $service)` and `getService()` allow you to set and get the ReCaptcha service object.

CHAPTER 29

Introduction

`Zend\Config` is designed to simplify the access to, and the use of, configuration data within applications. It provides a nested object property based user interface for accessing this configuration data within application code. The configuration data may come from a variety of media supporting hierarchical data storage. Currently `Zend\Config` provides adapters for read and write configuration data that are stored in Ini or XML files.

Using Zend\Config

Normally it is expected that users would use one of the reader classes to read a configuration file using *Zend\Config\Reader\Ini* or *Zend\Config\Reader\Xml*, but if configuration data are available in a *PHP* array, one may simply pass the data to the `Zend\Config\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
$config = new Zend\Config\Config($configArray);

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

As illustrated in the example above, `Zend\Config\Config` provides nested object property syntax to access configuration data passed to its constructor.

Along with the object oriented access to the data values, `Zend\Config\Config` also has `get()` which will return the supplied default value if the data element doesn't exist. For example:

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

Using Zend\Config\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\Config(include 'config.php');

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

CHAPTER 30

Theory of Operation

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

Each configuration data array value becomes a property of the `Zend\Config\Config` object. The key is used as the property name. If a value is itself an array, then the resulting object property is created as a new `Zend\Config\Config` object, loaded with the array data. This occurs recursively, such that a hierarchy of configuration data may be created with any number of levels.

`Zend\Config\Config` implements the **Countable** and **Iterator** interfaces in order to facilitate simple access to configuration data. Thus, one may use the count() function and *PHP* constructs such as foreach with `Zend\Config\Config` objects.

By default, configuration data made available through `Zend\Config\Config` are read-only, and an assignment (e.g., `$config->database->host = 'example.com';`) results in a thrown exception. This default behavior may be overridden through the constructor, however, to allow modification of data values. Also, when modifications are allowed, `Zend\Config\Config` supports unsetting of values (i.e. `unset($config->database->host)`). The `isReadOnly()` method can be used to determine if modifications to a given `Zend\Config\Config` object are allowed and the `setReadOnly()` method can be used to stop any further modifications to a `Zend\Config\Config` object that was created allowing modifications.

Note: Modifying Config does not save changes

It is important not to confuse such in-memory modifications with saving configuration data out to specific storage media. Tools for creating and modifying configuration data for various storage media are out of scope with respect to `Zend\Config\Config`. Third-party open source solutions are readily available for the purpose of creating and modifying configuration data for various storage media.

If you have two `Zend\Config\Config` objects, you can merge them into a single object using the `merge()` function. For example, given `$config` and `$localConfig`, you can merge data from `$localConfig` to `$config`

using `$config->merge($localConfig);`. The items in `$localConfig` will override any items with the same name in `$config`.

Note: The `Zend\Config\Config` object that is performing the merge must have been constructed to allow modifications, by passing `TRUE` as the second parameter of the constructor. The `setReadOnly()` method can then be used to prevent any further modifications after the merge is complete.

CHAPTER 31

Zend\Config\Reader

`Zend\Config\Reader` gives you the ability to read a config file. It works with concrete implementations for different file format. The `Zend\Config\Reader` is only an interface, that define the two methods `fromFile()` and `fromString()`. The concrete implementations of this interface are:

- `Zend\Config\Reader\Ini`
- `Zend\Config\Reader\Xml`
- `Zend\Config\Reader\Json`
- `Zend\Config\Reader\Yaml`

The `fromFile()` and `fromString()` return a PHP array contains the data of the configuration file.

Note: Differences from ZF1

The `Zend\Config\Reader` component no longer supports the following features:

- Inheritance of sections.
- Reading of specific sections.

Zend\Config\Reader\Ini

`Zend\Config\Reader\Ini` enables developers to store configuration data in a familiar *INI* format and read them in the application by using an array syntax.

`Zend\Config\Reader\Ini` utilizes the parse_ini_file() *PHP* function. Please review this documentation to be aware of its specific behaviors, which propagate to `Zend\Config\Reader\Ini`, such as how the special values of "`TRUE`", "`FALSE`", "yes", "no", and "`NULL`" are handled.

Note: Key Separator

By default, the key separator character is the period character (“.”). This can be changed, however, using the `setNestSeparator()` method. For example:

```
$reader = new Zend\Config\Reader\Ini();
$reader->setNestSeparator('-');
```

The following example illustrates a basic use of `Zend\Config\Reader\Ini` for loading configuration data from an *INI* file. In this example there are configuration data for both a production system and for a staging system. Suppose we have the following INI configuration file:

```
webhost                  = 'www.example.com'
database.adapter         = 'pdo_mysql'
database.params.host     = 'db.example.com'
database.params.username = 'dbuser'
database.params.password = 'secret'
database.params.dbname   = 'dbproduction'
```

We can use the `Zend\Config\Reader\Ini` to read this INI file:

```
$reader = new Zend\Config\Reader\Ini();
$data   = $reader->fromFile('/path/to/config.ini');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

The `Zend\Config\Reader\Ini` supports a feature to include the content of a INI file in a specific section of another INI file. For instance, suppose we have an INI file with the database configuration:

```
database.adapter         = 'pdo_mysql'
database.params.host     = 'db.example.com'
database.params.username = 'dbuser'
database.params.password = 'secret'
database.params.dbname   = 'dbproduction'
```

We can include this configuration in another INI file, for instance:

```
webhost  = 'www.example.com'
@include = 'database.ini'
```

If we read this file using the component `Zend\Config\Reader\Ini` we will obtain the same configuration data structure of the previous example.

The `@include = 'file-to-include.ini'` can be used also in a subelement of a value. For instance we can have an INI file like that:

```
adapter         = 'pdo_mysql'
params.host     = 'db.example.com'
params.username = 'dbuser'
params.password = 'secret'
params.dbname   = 'dbproduction'
```

And assign the `@include` as sublement of the database value:

```
webhost           = 'www.example.com'
database.@include = 'database.ini'
```

Zend\Config\Reader\Xml

`Zend\Config\Reader\Xml` enables developers to read configuration data in a familiar *XML* format and read them in the application by using an array syntax. The root element of the *XML* file or string is irrelevant and may be named arbitrarily.

The following example illustrates a basic use of `Zend\Config\Reader\Xml` for loading configuration data from an *XML* file. Suppose we have the following *XML* configuration file:

```
<?xml version="1.0" encoding="utf-8"?>?>
<config>
    <webhost>www.example.com</webhost>
    <database>
        <adapter value="pdo_mysql"/>
        <params>
            <host value="db.example.com"/>
            <username value="dbuser"/>
            <password value="secret"/>
            <dbname value="dbproduction"/>
        </params>
    </database>
</config>
```

We can use the `Zend\Config\Reader\Xml` to read this XML file:

```
$reader = new Zend\Config\Reader\Xml();
$data   = $reader->fromFile('/path/to/config.xml');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

`Zend\Config\Reader\Xml` utilizes the XMLReader *PHP* class. Please review this documentation to be aware of its specific behaviors, which propagate to `Zend\Config\Reader\Xml`.

Using `Zend\Config\Reader\Xml` we can include the content of XML files in a specific XML element. This is provided using the standard function XInclude of XML. To use this function you have to add the namespace `xmlns:xi="http://www.w3.org/2001/XInclude"` to the XML file. Suppose we have an XML files that contains only the database configuration:

```
<?xml version="1.0" encoding="utf-8"?>
<config>
    <database>
        <adapter>pdo_mysql</adapter>
        <params>
            <host>db.example.com</host>
            <username>dbuser</username>
            <password>secret</password>
            <dbname>dbproduction</dbname>
        </params>
    </database>
</config>
```

We can include this configuration in another XML file, for instance:

```
<?xml version="1.0" encoding="utf-8"?>
<config xmlns:xi="http://www.w3.org/2001/XInclude">
    <webhost>www.example.com</webhost>
```

```
    <xi:include href="database.xml"/>
</config>
```

The syntax to include an XML file in a specific element is `<xi:include href="file-to-include.xml"/>`

Zend\Config\Reader\Json

`Zend\Config\Reader\Json` enables developers to read configuration data in a *JSON* format and read them in the application by using an array syntax.

The following example illustrates a basic use of `Zend\Config\Reader\Json` for loading configuration data from a *JSON* file. Suppose we have the following *JSON* configuration file:

```
{
  "webhost"  : "www.example.com",
  "database" : {
    "adapter" : "pdo_mysql",
    "params"  : {
      "host"     : "db.example.com",
      "username" : "dbuser",
      "password" : "secret",
      "dbname"   : "dbproduction"
    }
  }
}
```

We can use the `Zend\Config\Reader\>Json` to read this JSON file:

```
$reader = new Zend\Config\Reader\Json();
$data   = $reader->fromFile('/path/to/config.json');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

`Zend\Config\Reader\Json` utilizes the Zend\Json\Json class.

Using `Zend\Config\Reader\Json` we can include the content of a JSON file in a specific JSON section or element. This is provided using the special syntax `@include`. Suppose we have a JSON file that contains only the database configuration:

```
{
  "database" : {
    "adapter" : "pdo_mysql",
    "params"  : {
      "host"     : "db.example.com",
      "username" : "dbuser",
      "password" : "secret",
      "dbname"   : "dbproduction"
    }
  }
}
```

We can include this configuration in another JSON file, for instance:

```
{
    "webhost"  : "www.example.com",
```

```
    "@include" : "database.json"
}
```

Zend\Config\Reader\Yaml

`Zend\Config\Reader\Yaml` enables developers to read configuration data in a *YAML* format and read them in the application by using an array syntax. In order to use the YAML reader we need to pass a callback to an external PHP library or use the Yaml PECL extension.

The following example illustrates a basic use of `Zend\Config\Reader\Yaml` that use the Yaml PECL extension. Suppose we have the following *YAML* configuration file:

```
webhost: www.example.com
database:
    adapter: pdo_mysql
    params:
      host:     db.example.com
      username: dbuser
      password: secret
      dbname:   dbproduction
```

We can use the `Zend\Config\Reader\Yaml` to read this YAML file:

```
$reader = new Zend\Config\Reader\Yaml();
$data   = $reader->fromFile('/path/to/config.yaml');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

If you want to use an external YAML reader you have to pass the callback function in the constructor of the class. For instance, if you want to use the Spyc library:

```
// include the Spyc library
require_once ('path/to/spyc.php');

$reader = new Zend\Config\Reader\Yaml(array('Spyc','YAMLLoadString'));
$data   = $reader->fromFile('/path/to/config.yaml');

echo $data['webhost']  // prints "www.example.com"
echo $data['database']['params']['dbname'];  // prints "dbproduction"
```

You can also instantiate the `Zend\Config\Reader\Yaml` without any parameter and specify the YAML reader in a second moment using the `setYamlDecoder()` method.

Using `Zend\Config\ReaderYaml` we can include the content of a YAML file in a specific YAML section or element. This is provided using the special syntax `@include`. Suppose we have a YAML file that contains only the database configuration:

```
database:
    adapter: pdo_mysql
    params:
      host:     db.example.com
      username: dbuser
      password: secret
      dbname:   dbproduction
```

We can include this configuration in another YAML file, for instance:

```
webhost:  www.example.com
@include: database.yaml
```

CHAPTER 32

Zend\Config\Writer

`Zend\Config\Writer` gives you the ability to write config files out of array, `Zend\Config\Config` and any Traversable object. The `Zend\Config\Writer` is an interface that defines two methods: `toFile()` and `toString()`. We have three specific writers that implement this interface:

- `Zend\Config\Writer\Ini`
- `Zend\Config\Write\Xml`
- `Zend\Config\Write\PhpArray`
- `Zend\Config\Write\Json`
- `Zend\Config\Write\Yaml`

Zend\Config\Writer\Ini

The *INI* writer has two modes for rendering with regard to sections. By default the top-level configuration is always written into section names. By calling `$writer->setRenderWithoutSectionsFlags(true);` all options are written into the global namespace of the *INI* file and no sections are applied.

As an addition `Zend\Config\Writer\Ini` has an additional option parameter **nestSeparator**, which defines with which character the single nodes are separated. The default is a single dot, like it is accepted by `Zend\Config\Reader\Ini` by default.

When modifying or creating a `Zend\Config\Config` object, there are some things to know. To create or modify a value, you simply say set the parameter of the `Config` object via the parameter accessor (**->**). To create a section in the root or to create a branch, you just create a new array ("`$config->branch = array();`").

Using Zend\Config\Writer\Ini

This example illustrates the basic use of `Zend\Config\Writer\Ini` to create a new config file:

```php
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Ini();
echo $writer->toString($config);
```

The result of this code is an INI string contains the following values:

```
[production]
webhost = "www.example.com"
database.params.host = "localhost"
database.params.username = "production"
database.params.password = "secret"
database.params.dbname = "dbproduction"
```

You can use the method `toFile()` to store the INI data in a file.

Zend\Config\Writer\Xml

The `Zend\Config\Writer\Xml`can be used to generate an XML string or file starting from a `Zend\Config\Config` object.

Using Zend\Config\Writer\Ini

This example illustrates the basic use of `Zend\Config\Writer\Xml` to create a new config file:

```php
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Xml();
echo $writer->toString($config);
```

The result of this code is an XML string contains the following data:

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

```
    <production>
        <webhost>www.example.com</webhost>
        <database>
            <params>
                <host>localhost</host>
                <username>production</username>
                <password>secret</password>
                <dbname>dbproduction</dbname>
            </params>
        </database>
    </production>
</zend-config>
```

You can use the method `toFile()` to store the XML data in a file.

Zend\Config\Writer\PhpArray

The `Zend\Config\Writer\PhpArray`can be used to generate a PHP code that returns an array representation of an `Zend\Config\Config` object.

Using Zend\Config\Writer\PhpArray

This example illustrates the basic use of `Zend\Config\Writer\PhpArray` to create a new config file:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\PhpArray();
echo $writer->toString($config);
```

The result of this code is a PHP script that returns an array as follow:

```
<?php
return array (
  'production' =>
  array (
    'webhost' => 'www.example.com',
    'database' =>
    array (
      'params' =>
      array (
        'host' => 'localhost',
        'username' => 'production',
        'password' => 'secret',
        'dbname' => 'dbproduction',
```

```
        ),
    ),
  ),
);
```

You can use the method `toFile()` to store the PHP script in a file.

Zend\Config\Writer\Json

The `Zend\Config\Writer\Json`can be used to generate a PHP code that returns the JSON representation of a `Zend\Config\Config` object.

Using Zend\Config\Writer\Json

This example illustrates the basic use of `Zend\Config\Writer\Json` to create a new config file:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Json();
echo $writer->toString($config);
```

The result of this code is a JSON string contains the following values:

```
{ "webhost"  : "www.example.com",
  "database" : {
    "params"  : {
      "host"     : "localhost",
      "username" : "production",
      "password" : "secret",
      "dbname"   : "dbproduction"
    }
  }
}
```

You can use the method `toFile()` to store the JSON data in a file.

The `Zend\Config\Writer\Json` class uses the `Zend\Json\Json` component to convert the data in a JSON format.

Zend\Config\Writer\Yaml

The `Zend\Config\Writer\Yaml`can be used to generate a PHP code that returns the YAML representation of a `Zend\Config\Config` object. In order to use the YAML writer we need to pass a callback to an external PHP

library or use the Yaml PECL extension.

Using Zend\Config\Writer\Yaml

This example illustrates the basic use of `Zend\Config\Writer\Yaml` to create a new config file using the Yaml PECL extension:

```
// Create the config object
$config = new Zend\Config\Config(array(), true);
$config->production = array();

$config->production->webhost = 'www.example.com';
$config->production->database = array();
$config->production->database->params = array();
$config->production->database->params->host = 'localhost';
$config->production->database->params->username = 'production';
$config->production->database->params->password = 'secret';
$config->production->database->params->dbname = 'dbproduction';

$writer = new Zend\Config\Writer\Yaml();
echo $writer->toString($config);
```

The result of this code is a YAML string contains the following values:

```
webhost: www.example.com
database:
    params:
      host:     localhost
      username: production
      password: secret
      dbname:   dbproduction
```

You can use the method `toFile()` to store the YAML data in a file.

If you want to use an external YAML writer library you have to pass the callback function in the constructor of the class. For instance, if you want to use the Spyc library:

```
// include the Spyc library
require_once ('path/to/spyc.php');

$writer = new Zend\Config\Writer\Yaml(array('Spyc','YAMLDump'));
echo $writer->toString($config);
```

CHAPTER 33

Zend\Config\Processor

`Zend\Config\Processor` gives you the ability to perform some operations on a `Zend\Config\Config` object. The `Zend\Config\Processor` is an interface that defines two methods: `process()` and `processValue()`. These operations are provided by the following concrete implementations:

- `Zend\Config\Processor\Constant`: manage PHP constant values;
- `Zend\Config\Processor\Filter`: filter the configuration data using `Zend\Filter`;
- `Zend\Config\Processor\Queue`: manage a queue of operations to apply to configuration data;
- `Zend\Config\Processor\Token`: find and replace specific tokens;
- `Zend\Config\Processor\Translator`: translate configuration values in other languages using `Zend\I18n\Translator`;

Below we reported some examples for each type of processor.

Zend\Config\Processor\Constant

Using Zend\Config\Processor\Constant

This example illustrates the basic use of `Zend\Config\Processor\Constant`:

```
define ('TEST_CONST', 'bar');
// set true to Zend\Config\Config to allow modifications
$config = new Zend\Config\Config(array('foo' => 'TEST_CONST'), true);
$processor = new Zend\Config\Processor\Constant();

echo $config->foo . ',';
$processor->process($config);
echo $config->foo;
```

This example returns the output: `TEST_CONST, bar.`.

Zend\Config\Processor\Filter

Using Zend\Config\Processor\Filter

This example illustrates the basic use of `Zend\Config\Processor\Filter`:

```
use Zend\Filter\StringToUpper;
use Zend\Config\Processor\Filter as FilterProcessor;
use Zend\Config\Config;

$config = new Config(array ('foo' => 'bar'), true);
$upper = new StringToUpper();

$upperProcessor = new FilterProcessor($upper);

echo $config->foo . ',';
$upperProcessor->process($config);
echo $config->foo;
```

This example returns the output: `bar,BAR`.

Zend\Config\Processor\Queue

Using Zend\Config\Processor\Queue

This example illustrates the basic use of `Zend\Config\Processor\Queue`:

```
use Zend\Filter\StringToLower;
use Zend\Filter\StringToUpper;
use Zend\Config\Processor\Filter as FilterProcessor;
use Zend\Config\Processor\Queue;
use Zend\Config\Config;

$config = new Config(array ('foo' => 'bar'), true);
$upper  = new StringToUpper();
$lower  = new StringToLower();

$lowerProcessor = new FilterProcessor($lower);
$upperProcessor = new FilterProcessor($upper);

$queue = new Queue();
$queue->insert($upperProcessor);
$queue->insert($lowerProcessor);
$queue->process($config);

echo $config->foo;
```

This example returns the output: `bar`. The filters in the queue are applied with a *FIFO* logic (First In, First Out).

Zend\Config\Processor\Token

Using Zend\Config\Processor\Token

This example illustrates the basic use of `Zend\Config\Processor\Token`:

```
// set the Config to true to allow modifications
$config = new Config(array('foo' => 'Value is TOKEN'), true);
$processor = new TokenProcessor();

$processor->addToken('TOKEN', 'bar');
echo $config->foo . ',';
$processor->process($config);
echo $config->foo;
```

This example returns the output: `Value is TOKEN,Value is bar`.

Zend\Config\Processor\Translator

Using Zend\Config\Processor\Translator

This example illustrates the basic use of `Zend\Config\Processor\Translator`:

```
use Zend\Config\Config;
use Zend\Config\Processor\Translator as TranslatorProcessor;
use Zend\I18n\Translator\Translator;

$config = new Config(array('animal' => 'dog'), true);

/*
 * The following mapping would exist for the translation
 * loader you provide to the translator instance
 * $italian = array(
 *     'dog' => 'cane'
 * );
 */

$translator = new Translator();
// ... configure the translator ...
$processor = new TranslatorProcessor($translator);

echo "English: {$config->animal}, ";
$processor->process($config);
echo "Italian: {$config->animal}";
```

This example returns the output: `English:  dog, Italian:  cane`.

CHAPTER 34

Introduction

Zend\Crypt provides support of some cryptographic tools. The available features are:

- encrypt-then-authenticate using symmetric ciphers (the authentication step is provided using HMAC);
- encrypt/decrypt using symmetric and public key algorithm (e.g. RSA algorithm);
- generate digital sign using public key algorithm (e.g. RSA algorithm);
- key exchange using the Diffie-Hellman method;
- Key derivation function (e.g. using PBKDF2 algorithm);
- Secure password hash (e.g. using Bcrypt algorithm);
- generate Hash values;
- generate HMAC values;

The main scope of this component is to offer an easy and secure way to protect and authenticate sensitive data in PHP. Because the use of cryptography is not so easy we recommend to use the `Zend\Crypt` component only if you have a minimum background on this topic. For an introduction to cryptography we suggest the following references:

- Dan Boneh "Cryptography course" Stanford University, Coursera - free online course
- N.Ferguson, B.Schneier, and T.Kohno, "Cryptography Engineering", John Wiley & Sons (2010)
- B.Schneier "Applied Cryptography", John Wiley & Sons (1996)

Note: PHP-CryptLib

Most of the ideas behind the `Zend\Crypt` component have been inspired by the PHP-CryptLib project of Anthony Ferrara. PHP-CryptLib is an all-inclusive pure PHP cryptographic library for all cryptographic needs. It is meant to be easy to install and use, yet extensible and powerful enough for even the most experienced developer.

CHAPTER 35

Encrypt/decrypt using block ciphers

`Zend\Crypt\BlockCipher` implements the encrypt-then-authenticate mode using HMAC to provide authentication.

The symmetric cipher can be choose with a specific adapter that implements the `Zend\Crypt\Symmetric\SymmetricInterface`. We support the standard algorithms of the Mcrypt extension. The adapter that implements the Mcrypt is `Zend\Crypt\Symmetric\Mcrypt`.

In the following code we reported an example on how to use the BlockCipher class to encrypt-then-authenticate a string using the AES block cipher (with a key of 256 bit) and the HMAC algorithm (using the SHA-256 hash function).

```
1 use Zend\Crypt\BlockCipher;
2
3 $blockCipher = BlockCipher::factory('mcrypt', array('algo' => 'aes'));
4 $blockCipher->setKey('encryption key');
5 $result = $blockCipher->encrypt('this is a secret message');
6 echo "Encrypted text: $result \n";
```

The BlockCipher is initialized using a factory method with the name of the cipher adapter to use (mcrypt) and the parameters to pass to the adapter (the AES algorithm). In order to encrypt a string we need to specify an encryption key and we used the `setKey()` method for that scope. The encryption is provided by the `encrypt()` method.

The output of the encryption is a string, encoded in Base64 (default), that contains the HMAC value, the IV vector, and the encrypted text. The encryption mode used is the CBC (with a random IV by default) and SHA256 as default hash algorithm of the HMAC. The Mcrypt adapter encrypts using the PKCS#7 padding mechanism by default. You can specify a different padding method using a special adapter for that (Zend\Crypt\Symmetric\Padding). The encryption and authentication keys used by the `BlockCipher` are generated with the PBKDF2 algorithm, used as key derivation function from the user's key specified using the `setKey()` method.

Note: Key size

BlockCipher try to use always the longest size of the key for the specified cipher. For instance, for the AES algorithm it uses 256 bits and for the Blowfish algorithm it uses 448 bits.

You can change all the default settings passing the values to the factory parameters. For instance, if you want to use the Blowfish algorithm, with the CFB mode and the SHA512 hash function for HMAC you have to initialize the class as follow:

```
use Zend\Crypt\BlockCipher;

$blockCipher = BlockCipher::factory('mcrypt', array(
                                    'algo' => 'blowfish',
                                    'mode' => 'cfb',
                                    'hash' => 'sha512'
                                ));
```

Note: Recommendation

If you are not familiar with symmetric encryption techniques we strongly suggest to use the default values of the `BlockCipher` class. The default values are: AES algorithm, CBC mode, HMAC with SHA256, PKCS#7 padding.

To decrypt a string we can use the `decrypt()` method. In order to successfully decrypt a string we have to configure the BlockCipher with the same parameters of the encryption.

We can also initialize the BlockCipher manually without use the factory method. We can inject the symmetric cipher adapter directly to the constructor of the BlockCipher class. For instance, we can rewrite the previous example as follow:

```
use Zend\Crypt\BlockCipher;
use Zend\Crypt\Symmetric\Mcrypt;

$blockCipher = new BlockCipher(new Mcrypt(array('algo' => 'aes')));
$blockCipher->setKey('encryption key');
$result = $blockCipher->encrypt('this is a secret message');
echo "Encrypted text: $result \n";
```

CHAPTER 36

Key derivation function

In cryptography, a key derivation function (or KDF) derives one or more secret keys from a secret value such as a master key or other known information such as a password or passphrase using a pseudo-random function. For instance, a KDF function can be used to generate encryption or authentication keys from a user password. The `Zend\Crypt\Key\Derivation` implements a key derivation function using specific adapters.

User passwords are not really suitable to be used as keys in cryptographic algorithms, since users normally choose keys they can write on keyboard. These passwords use only 6 to 7 bits per character (or less). It is highly recommended to use always a KDF function of transformation a user's password to a cryptography key.

Pbkdf2 adapter

Pbkdf2 is a KDF that applies a pseudorandom function, such as a cryptographic hash, to the input password or passphrase along with a salt value and repeats the process many times to produce a derived key, which can then be used as a cryptographic key in subsequent operations. The added computational work makes password cracking much more difficult, and is known as key stretching.

In the example below we show a typical usage of the `Pbkdf2` adapter.

```
use Zend\Crypt\Key\Derivation\Pbkdf2;
use Zend\Math\Rand;

$pass = 'password';
$salt = Rand::getBytes(strlen($pass), true);
$key  = Pbkdf2::calc('sha256', $pass, $salt, 10000, strlen($pass)*2);

echo "Original password: $pass \n";
echo "Key derivation   : $key \n";
```

The `Pbkdf2` adapter takes the password (`$pass`) and generate a binary key with a size double of the password. The syntax is `calc($hash, $pass, $salt, $iterations, $length)` where `$hash` is the name of the hash function to use, `$pass` is the password, `$salt` is a pseudo random value, `$iterations` is the number of iterations of the algorithm and `$length` is the size of the key to be generated. We used the `Rand::getBytes` function of

the `Zend\Math\Rand` class to generate a random bytes using a strong generators (the `true` value means the usage of strong generators).

The number of iterations is a very important parameter for the security of the algorithm. Big values means more security. There is not a fixed value for that because the number of iterations depends on the CPU power. You should always choose a number of iteration that prevent brute force attacks. For instance, a value of 1‘000‘000 iterations, that is equal to 1 sec of elaboration for the PBKDF2 algorithm, is enough secure using an Intel Core i5-2500 CPU at 3.3 Ghz.

SaltedS2k adapter

The SaltedS2k algorithm uses an hash function and a salt to generate a key based on a user's password. This algorithm doesn't use a parameter that specify the number of iterations and for that reason it's considered less secure compared with Pbkdf2. We suggest to use the SaltedS2k algorithm only if you really need it.

Below is reported a usage example of the `SaltedS2k` adapter.

```
use Zend\Crypt\Key\Derivation\SaltedS2k;
use Zend\Math\Rand;

$pass = 'password';
$salt = Rand::getBytes(strlen($pass), true);
$key  = SaltedS2k::calc('sha256', $pass, $salt, strlen($pass)*2);

echo "Original password: $pass \n";
echo "Key derivation   : $key \n";
```

CHAPTER 37

Password secure storing

The `Zend\Crypt\Password` store a user's password in a secure way using dedicated adapters like the bcrypt algorithm.

In the example below we show hot to use the bcrypt algorithm to store a user's password:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt()
$securePass = $bcrypt->create('user password');
```

The output of the `create()` method is the encrypted password. This value can be stored in a repository, like a database instead of use alternative mechanism like MD5 or MD5 + salt that are not considered secure anymore (read this post to know why).

To verify if a given password is valid against a bcrypt value you can use the `verify()` method. Below is reported an example:

```
use Zend\Crypt\Password\Bcrypt;

$bcrypt = new Bcrypt();
$securePass = 'the bcrypt value stored somewhere';
$password = 'the password to check';

if ($bcrypt->verify($password, $bcrypt)) {
    echo "The password is correct! \n";
} else {
    echo "The password is NOT correct.\n";
}
```

By default the `Zend\Crypt\Password\Bcrypt` class uses a value of 14 for the cost parameter of the bcrypt. This is an important value for the security of the bcrypt algorithm. The cost parameter is an integer value between 4 to 33. Greater values means more execution time for the bcrypt that means more security against brute force or dictionary attacks. As for the PBKDF2 algorithm there is not a fixed value for that parameter that can be considered secure. The default value of 14 is about 1 second of computation using an Intel Core i5-2500 CPU at 3.3 Ghz that can be considered secure.

If you want to change the cost parameter of the bcrypt algorithm you can use the `setCost()` method.

Note: Bcrypt with non-ASCII passwords (8-bit characters)

The bcrypt implementation used by PHP < 5.3.7 can contains a security flaw if the password uses 8-bit characters (here the security report). The impact of this bug was that most (but not all) passwords containing non-ASCII characters with the 8th bit set were hashed incorrectly, resulting in password hashes incompatible with those of OpenBSD's original implementation of bcrypt. This security flaw has been fixed starting from PHP 5.3.7 and the prefix used in the output has changed in '$2y$' in order to put evidence on the correctness of the hash value. If you are using PHP < 5.3.7 with 8-bit passwords the `Zend\Crypt\Password\Bcrypt` throws an exception suggesting to upgrade to PHP 5.3.7+ or use only 7-bit passwords.

CHAPTER 38

Public key cryptography

Public-key cryptography refers to a cryptographic system requiring two separate keys, one of which is secret and one of which is public. Although different, the two parts of the key pair are mathematically linked. One key locks or encrypts the plaintext, and the other unlocks or decrypts the cyphertext. Neither key can perform both functions. One of these keys is published or public, while the other is kept private.

In Zend Framework we implemented two public key algorithms: Diffie-Hellman key exchange and RSA.

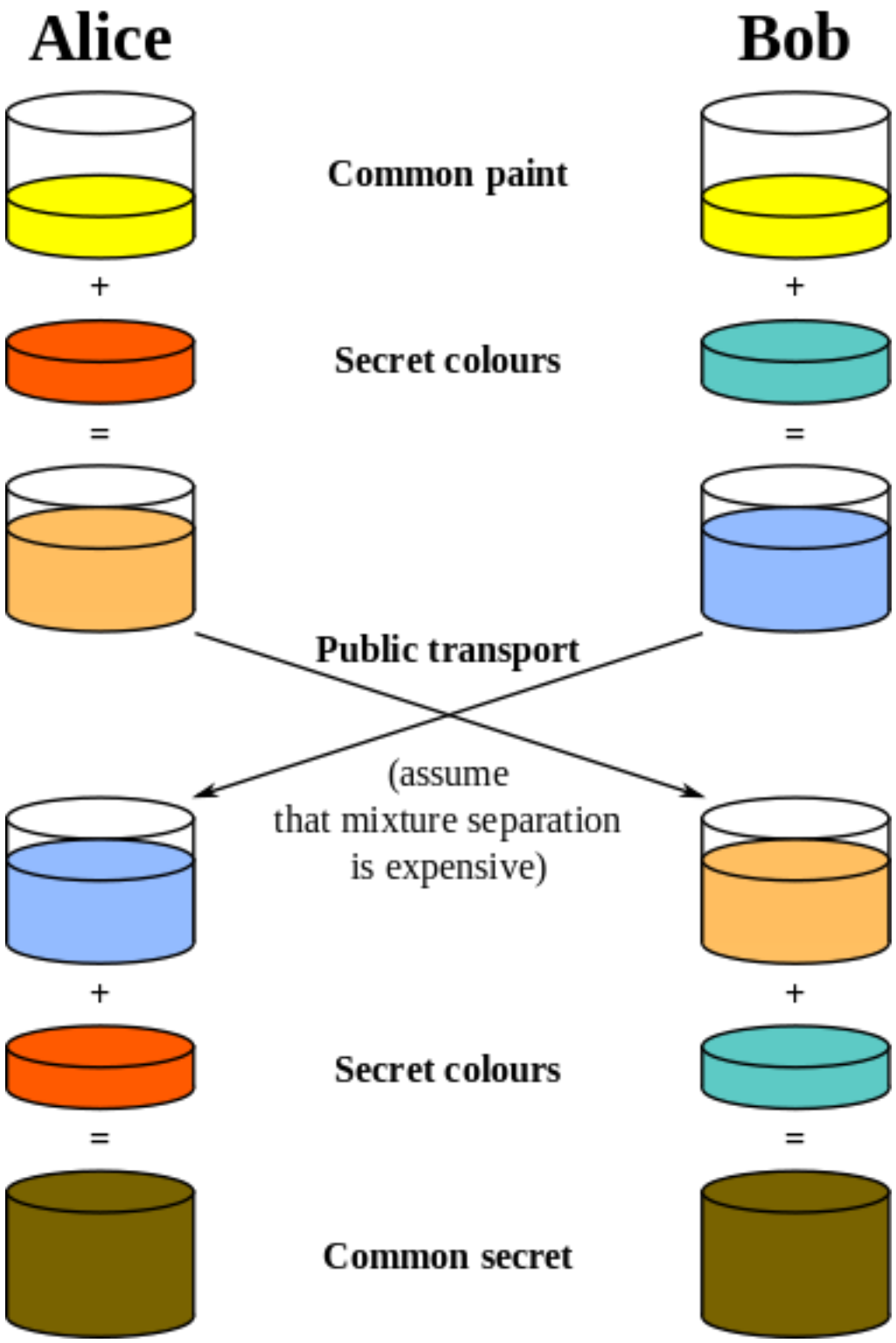
Diffie-Hellman

The Diffie-Hellman algorithm is a specific method of exchanging cryptographic keys. It is one of the earliest practical examples of key exchange implemented within the field of cryptography. The Diffie–Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

The diagram of operation of the Diffie-Hellman algorithm can be defined by the following picture (taken by the Diffie-Hellman Wikipedia page):

The schema's colors represent the parameters of the algorithm. Here is reported an example of usage using the `Zend\Crypt\PublicKey\DiffieHellman` class:

```
1 use Zend\Crypt\PublicKey\DiffieHellman;
2
3 $aliceOptions = array(
4     'prime'     =>
  ↪'155172898181473697471232257763715539915724801966915404479707795314057629378541917580651227
  ↪' .
5
  ↪'423698188993727816152646631438561595825688188889951272158842675419950341258706556549803580
  ↪' .
6
  ↪'104870537681476726513255747040765857479291291572334510643245094715007229621094194349783925
  ↪' .
```


Alice
Bob
Common paint
+
+
Secret colours
=
=
Public transport
(assume
that mixture separation
is expensive)
+
+
Secret colours
=
=
Common secret

```
                    '98476037559498584825335930558543963844 3',
    'generator'=> '2',
    'private'  =>
↪'992093140665725952364085695919679885571412495614942674862518080355353963322786201435363176
↪' .

↪'813127128916726230726309951803243888416814918577455156967890911274095150092503589658166661
↪' .

↪'463420498381785213791321533481399080168191962194483101070726325157493390557981225386151351
↪' .
                    '048287025237969518005750318710516780 91'
);

$bobOptions   = array(
    'prime'    => $aliceOptions['prime'],
    'generator'=> '2',
    'private'  =>
↪'334117357926395586257336357178925636125481806504021611510774783148414637079488997861035889
↪' .

↪'123256347304105519467727528801778689728169635518217403867000760342134081539246925625431179
↪' .

↪'634647331566005454845108330724270034742070646507148310833044977371603820970833568760781462
↪' .
                    '31616972608703322302585471319261275664'
);

$alice = new DiffieHellman($aliceOptions['prime'], $aliceOptions['generator'],
↪$aliceOptions['private']);
$bob   = new DiffieHellman($bobOptions['prime'], $bobOptions['generator'],
↪$bobOptions['private']);

$alice->generateKeys();
$bob->generateKeys();

$aliceSecretKey = $alice->computeSecretKey($bob->getPublicKey(DiffieHellman::FORMAT_
↪BINARY),
                                           DiffieHellman::FORMAT_BINARY,
                                           DiffieHellman::FORMAT_BINARY);

$bobSecretKey   = $bob->computeSecretKey($alice->getPublicKey(DiffieHellman::FORMAT_
↪BINARY),
                                         DiffieHellman::FORMAT_BINARY,
                                         DiffieHellman::FORMAT_BINARY);

if ($aliceSecretKey !== $bobSecretKey) {
    echo "ERROR!\n";
} else {
    printf("The secret key is: %s\n", base64_encode($aliceSecretKey));
}
```

The parameters of the Diffie-Hellman class are: a prime number (p), a generator (g) that is a primitive root mod p and a private integer number. The security of the Diffie-Hellman exchange algorithm is related to the choice of these parameters. To know how to choose secure numbers you can read the RFC 3526 document.

Note: The `Zend\Crypt\PublicKey\DiffieHellman` class use by default the OpenSSL extension of PHP to generate the parameters. If you want don't want to use the OpenSSL library you have to set the `useOpensslExtension` static method to `false`.

RSA

RSA is an algorithm for public-key cryptography that is based on the presumed difficulty of factoring large integers, the factoring problem. A user of RSA creates and then publishes the product of two large prime numbers, along with an auxiliary value, as their public key. The prime factors must be kept secret. Anyone can use the public key to encrypt a message, but with currently published methods, if the public key is large enough, only someone with knowledge of the prime factors can feasibly decode the message. Whether breaking RSA encryption is as hard as factoring is an open question known as the RSA problem.

The RSA algorithm can be used to encrypt/decrypt message and also to provide authenticity and integrity generating a digital signature of a message. Suppose that Alice wants to send an encrypted message to Bob. Alice must use the public key of Bob to encrypt the message. Bob can decrypt the message using his private key. Because Bob he is the only one that can access to his private key, he is the only one that can decrypt the message. If Alice wants to provide authenticity and integrity of a message to Bob she can use her private key to sign the message. Bob can check the correctness of the digital signature using the public key of Alice. Alice can provide encryption, authenticity and integrity of a message to Bob using the previous schemas in sequence, applying the encryption first and the digital signature after.

Below we reported some examples of usage of the `Zend\Crypt\PublicKey\Rsa` class in order to:

- generate a public key and a private key;
- encrypt/decrypt a string;
- generate a digital signature of a file.

Generate a public key and a private key

In order to generate a public and private key you can use the following code:

```
use Zend\Crypt\PublicKey\RsaOptions;

$rsaOptions = new RsaOptions(array(
    'pass_phrase' => 'test'
));

$rsaOptions->generateKeys(array(
    'private_key_bits' => 2048,
));

file_put_contents('private_key.pem', $rsaOptions->getPrivateKey());
file_put_contents('public_key.pub', $rsaOptions->getPublicKey());
```

This example generates a public and privat key of 2048 bit storing the keys in two separate files, the `private_key.pem` for the private key and the `public_key.pub` for the public key. You can also generate the public and private key using OpenSSL from the command line (Unix style syntax):

```
ssh-keygen -t rsa
```

Encrypt and decrypt a string

Below is reported an example on how to encrypt and decrypt a string using the RSA algorithm. You can encrypt only small strings. The maximum size of encryption is given by the length of the public/private key - 88 bits. For instance, if we use a size of 2048 bit you can encrypt string with a maximum size of 1960 bit (245 characters). This limitation is related to the OpenSSL implementation for a security reason related to the nature of the RSA algorithm.

The normal application of a public key encryption algorithm is to store a key or a hash of the data you want to respectively encrypt or sign. A hash is typically 128-256 bits (the PHP sha1() function returns a 160 bit hash). An AES encryption key is 128 to 256 bits. So either of those will comfortably fit inside a single RSA encryption.

```
use Zend\Crypt\PublicKey\Rsa;

$rsa = Rsa::factory(array(
    'public_key'    => 'public_key.pub',
    'private_key'   => 'private_key.pem',
    'pass_phrase'   => 'test',
    'binary_output' => false
));

$text = 'This is the message to encrypt';

$encrypt = $rsa->encrypt($text);
printf("Encrypted message:\n%s\n", $encrypt);

$decrypt = $rsa->decrypt($encrypt);

if ($text !== $decrypt) {
    echo "ERROR\n";
} else {
    echo "Encryption and decryption performed successfully!\n";
}
```

Generate a digital signature of a file

Below is reported an example of how to generate a digital signature of a file.

```
use Zend\Crypt\PublicKey\Rsa;

$rsa = Rsa::factory(array(
    'private_key'   => 'path/to/private_key',
    'pass_phrase'   => 'passphrase of the private key',
    'binary_output' => false
));

$file = file_get_contents('path/file/to/sign');

$signature = $rsa->sign($file, $rsa->getOptions()->getPrivateKey());
$verify    = $rsa->verify($file, $signature, $rsa->getOptions()->getPublicKey());

if ($verify) {
    echo "The signature is OK\n";
    file_put_contents($filename . '.sig', $signature);
    echo "Signature save in $filename.sig\n";
} else {
     echo "The signature is not valid!\n";
}
```

In this example we used the Base64 format to encode the digital signature of the file (`binary_output` is false).

Note: The implementation of `Zend\Crypt\PublicKey\Rsa` algorithm uses the OpenSSL extension of PHP.

CHAPTER 39

Zend\Db\Adapter

The Adapter object is the most important sub-component of Zend\Db. It is responsible for adapting any code written in or for Zend\Db to the targeted php extensions and vendor databases. In doing this, it creates an abstraction layer for the PHP extensions, which is called the "Driver" portion of the Zend\Db adapter. It also creates a lightweight abstraction layer for the various idiosyncrasies that each vendor specific platform might have in it's SQL/RDBMS implementation which is called the "Platform" portion of the adapter.

Creating an Adapter (Quickstart)

Creating an adapter can simply be done by instantiating the `Zend\Db\Adapter\Adapter` class. The most common use case, while not the most explicit, is to pass an array of information to the Adapter.

```
1 $adapter = new Zend\Db\Adapter\Adapter($driverArray);
```

This driver array is an abstraction for the extension level required parameters. Here is a table for the

Table 39.1: Connection Array Keys

Name	Required	Notes
driver	required	Mysqli, Sqlsrv, Pdo_Sqlite, Pdo_Mysql, Pdo=OtherPdoDriver
database	generally required	the name of the database (schema)
username	generally required	the connection username
password	generally required	the connection password
hostname	not generally required	the IP address or hostname to connect to
port	not generally required	the port to connect to (if applicable)
characterset	not generally required	the character set to use

* other names will work as well. Effectively, if the PHP manual uses a particular naming, this naming will be supported by our Driver. For example, dbname in most cases will also work for 'database'. Another example is that in the case of Sqlsrv, UID will work in place of username. Which format you chose is up to you, but the above table represents the official abstraction names.

So, for example, a MySQL connection using ext/mysqli:

```
$adapter = new Zend\Db\Adapter\Adapter(array(
    'driver' => 'Mysqli',
    'database' => 'zend_db_example',
    'username' => 'developer',
    'password' => 'developer-password'
));
```

Another example, of a Sqlite connection via PDO:

```
            $adapter = new Zend\Db\Adapter\Adapter(array(
                'driver' => 'Pdo_Sqlite',
                'database' => 'path/to/sqlite.db'
            ));
```

It is important to know that by using this style of adapter creation, the Adapter will attempt to create any dependencies that were not explicitly provided. A Driver object will be created from the contents of the $driver array provided in the constructor. A Platform object will be created based off the type of Driver object that was instantiated. And lastly, a default ResultSet object is created and utilized. Any of these objects can be injected, to do this, see the next section.

Creating an Adapter (By Injecting Dependencies)

The more expressive and explicit way of creating an adapter is by injecting all your dependencies up front. `Zend\Db\Adapter\Adapter` uses constructor injection, and all required dependencies are injected through the constructor, which has the following signature (in pseudo-code):

```
use Zend\Db\Adapter\Platform\PlatformInterface;
use Zend\Db\ResultSet\ResultSet;

class Zend\Db\Adapter\Adapter {
    public function __construct($driver, PlatformInterface $platform = null,␣
↪ResultSet $queryResultSetPrototype = null)
}
```

What can be injected:

$driver - an array or an instance of `Zend\Db\Adapter\Driver\DriverInterface` $platform - (optional) an instance of `Zend\Db\Platform\PlatformInterface`, the default will be created based off the driver implementation $queryResultSetPrototype - (optional) an instance of `Zend\Db\ResultSet\ResultSet`, to understand this object's role, see the section below on querying through the adapter

Query Preparation Through Zend\Db\Adapter\Adapter::query()

By default, query() prefers that you use "preparation" as a means for processing SQL statements. This generally means that you will supply a SQL statement with the values substituted by placeholders, and then the parameters for those placeholders are supplied separately. An example of this workflow with `Zend\Db\Adapter\Adapter` is:

```
$adapter->query('SELECT * FROM `artist` WHERE `id` = ?', array(5));
```

The above example will go through the following steps:

- create a new Statement object
- prepare an array into a ParameterContainer if necessary

- inject the ParameterContainer into the Statement object
- execute the Statement object, producing a Result object
- check the Result object to check if the supplied sql was a "query", or a result set producing statement
- if it is a result set producing query, clone the ResultSet prototype, inject Result as datasource, return it
- else, return the Result

Query Execution Through Zend\Db\Adapter\Adapter::query()

In some cases, you have to execute statements directly. The primary purpose for needing to execute sql instead of prepare and execute a sql statement, might be because you are attempting to execute a DDL statement (which in most extensions and vendor platforms), are un-preparable. An example of executing:

```
1           $adapter->query('ALTER TABLE ADD INDEX(`foo_index`) ON (`foo_column`))',␣
↪Adapter::QUERY_MODE_EXECUTE);
```

The primary difference to notice is that you must provide the Adapter::QUERY_MODE_EXECUTE (execute) as the second parameter.

Creating Statements

While query() is highly useful for one-off and quick querying of a database through Adapter, it generally makes more sense to create a statement and interact with it directly, so that you have greater control over the prepare-then-execute workflow. To do this, Adapter gives you a routine called createStatement() that allows you to create a Driver specific Statement to use so you can manage your own prepare-then-execute workflow.

```
1 $statement = $adapter->createStatement($sql, $optionalParameters);
2 $result = $statement->execute();
```

Using The Platform Object

The Platform object provides an API to assist in crafting queries in a way that is specific to the SQL implementation of a particular vendor. Nuances such as how identifiers or values are quoted, or what the identifier separator character is are handled by this object. To get an idea of the capabilities, the interface for a platform object looks like this:

```
1  interface Zend\Db\Adapter\Platform\PlatformInterface
2  {
3      public function getName();
4      public function getQuoteIdentifierSymbol();
5      public function quoteIdentifier($identifier);
6      public function getQuoteValueSymbol();
7      public function quoteValue($value);
8      public function getIdentifierSeparator();
9      public function quoteIdentifierInFragment($identifier, array $additionalSafeWords␣
↪= array());
10 }
```

For example, to quote a column name, specific to MySQL's way of quoting:

```php
$platform = new Zend\Db\Adapter\Platform\Mysql;
$column = $platform->quoteIdentifier('first_name'); // returns `first_name`
```

Generally speaking, it is easier to get the proper Platform instance from the adapter:

```php
$platform = $adapter->getPlatform();
// or
$platform = $adapter->platform; // magic property access
```

Using The Parameter Container

The ParameterContainer object is a container for the various parameters that need to be passed into a Statement object to fulfill all the various parameterized parts of the SQL statement. This object implements the ArrayAccess interface.

Examples

Creating a Driver and Vendor portable Query, Preparing and Iterating Result

```php
$adapter = new Zend\Db\Adapter\Adapter($driverConfig);

$qi = function($name) use ($adapter) { return $adapter->platform->quoteIdentifier(
↪$name); };
$fp = function($name) use ($adapter) { return $adapter->driver->formatParameterName(
↪$name); };

$sql = 'UPDATE ' . $qi('artist')
    . ' SET ' . $qi('name') . ' = ' . $fp('name')
    . ' WHERE ' . $qi('id') . ' = ' . $fp('id');

/* @var $statement Zend\Db\Adapter\Driver\StatementInterface */
$statement = $adapter->query($sql);

$parameters = array(
    'name' => 'Updated Artist',
    'id' => 1
);

$statement->execute($parameters);

// DATA INSERTED, NOW CHECK

/* @var $statement Zend\Db\Adapter\DriverStatementInterface */
$statement = $adapter->query('SELECT * FROM '
    . $qi('artist')
    . ' WHERE id = ' . $fp('id'));

/* @var $results Zend\Db\ResultSet\ResultSet */
$results = $statement->execute(array('id' => 1));

$row = $results->current();
$name = $row['name'];
```

CHAPTER 40

Zend\Db\ResultSet

`Zend\Db\ResultSet` is a sub-component of Zend\Db for abstracting the iteration of rowset producing queries. While data sources for this can be anything that is iterable, generally a `Zend\Db\Adpater\Driver\ResultInterface` based object is the primary source for retrieving data.

`Zend\Db\ResultSet`'s must implement the `Zend\Db\ResultSet\ResultSetInterface` and all sub-components of Zend\Db that return a ResultSet as part of their API will assume an instance of a `ResultSetInterface` should be returned. In most casts, the Prototype pattern will be used by consuming object to clone a prototype of a ResultSet and return a specialized ResultSet with a specific data source injected. The interface of ResultSetInterface looks like this:

```
interface ResultSetInterface extends \Traversable, \Countable
{
    public function initialize($dataSource);
    public function getFieldCount();
}
```

Quickstart

`Zend\Db\ResultSet\ResultSet` is the most basic form of a ResultSet object that will expose each row as either an ArrayObject-like object or an array of row data. The following workflow is based on that inside `Zend\Db\Adapter\Adapter::query()`:

```
use Zend\Db\Adapter\Driver\ResultInterface;
use Zend\Db\ResultSet\ResultSet;

$stmt = $driver->createStatement($sql);
$stmt->prepare($parameters);
$result = $stmt->execute();

if ($result instanceof ResultInterface && $result->isQueryResult()) {
    $resultSet = new ResultSet;
    $resultSet->initialize($result);
```

```
11
12      foreach ($resultSet as $row) {
13          echo $row->my_column . PHP_EOL;
14      }
15  }
```

Zend\Db\ResultSet\HydratingResultSet

`Zend\Db\ResultSet\HydratingResultSet` is a more flexible `ResultSet` object that allows the developer to choose an appropriate "hydration strategy" for getting row data into a target object. While iterating, `HydratingResultSet` will take a prototype of a target object and clone it for each successive new row it iterates. With this newly cloned row, `HydratingResultSet` will hydrate the target object with the row data.

In the example below, rows from the database will be iterated, and during iteration, `HydratingRowSet` will use the Reflection based hydrator to inject the row data directly into the protected members of the cloned UserEntity object:

```
1  use Zend\Db\Adapter\Driver\ResultInterface;
2  use Zend\Db\ResultSet\HydratingResultSet;
3  use Zend\Stdlib\Hydrator\Reflection as ReflectionHydrator;
4
5  class UserEntity {
6      protected $first_name;
7      protected $last_name;
8      public function getFirstName() { return $this->first_name; }
9      public function getLastName() { return $this->last_name; }
10 }
11
12 $stmt = $driver->createStatement($sql);
13 $stmt->prepare($parameters);
14 $result = $stmt->execute();
15
16 if ($result instanceof ResultInterface && $result->isQueryResult()) {
17     $resultSet = new HydratingResultSet(new ReflectionHydrator, new UserEntity);
18     $resultSet->initialize($result);
19
20     foreach ($resultSet as $user) {
21         echo $user->getFirstName() . ' ' . $user->getLastName() . PHP_EOL;
22     }
23 }
```

For more information, see the `Zend\Stdlib\Hydrator` documentation to get a better sense of the different strategies that can be employed in order to populate a target object.

CHAPTER 41

Zend\Db\Sql

`Zend\Db\Sql` is a SQL abstraction layer for building platform specific SQL queries via a object-oriented API. The end result of an `Zend\Db\Sql` object will be to either produce a Statement and Parameter container that represents the target query, or a full string that can be directly executed against the database platform. To achieve this, `Zend\Db\Sql` objects require a `Zend\Db\Adapter\Adapter` object in order to produce the desired results.

Zend\Db\Sql\Sql (Quickstart)

As there are for primary tasks associated with interacting with a database (from the DML, or Data Manipulation Language): selecting, inserting, updating and deleting. As such, there are four primary objects that developers can interact or building queries, `Zend\Db\Sql\Select`, `Insert`, `Update` and `Delete`.

Since these four tasks are so closely related, and generally used together within the same application, `Zend\Db\Sql\Sql` objects help you create them and produce the result you are attempting to achieve.

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select(); // @return Zend\Db\Sql\Select
$insert = $sql->insert(); // @return Zend\Db\Sql\Insert
$update = $sql->update(); // @return Zend\Db\Sql\Update
$delete = $sql->delete(); // @return Zend\Db\Sql\Delete
```

As a developer, you can now interact with these objects, as described in the sections below, to specialize each query. Once they have been populated with values, they are ready to either be prepared or executed.

To prepare (using a Select object):

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select();
$select->from('foo');
$select->where(array('id' => 2));

```

```
$statement = $sql->prepareStatementForSqlObject($select);
$results = $statement->execute();
```

To execute (using a Select object)

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter);
$select = $sql->select();
$select->from('foo');
$select->where(array('id' => 2));

$selectString = $sql->getSqlStringForSqlObject($select);
$results = $adapter->query($selectString, $adapter::QUERY_MODE_EXECUTE);
```

Zend\Db\Sql\Sql objects can also be bound to a particular table so that in getting a select, insert, update, or delete object, they are all primarily seeded with the same table when produced.

```
use Zend\Db\Sql\Sql;
$sql = new Sql($adapter, 'foo');
$select = $sql->select();
$select->where(array('id' => 2)); // $select already has the from('foo') applied
```

Zend\Db\Sql's Select, Insert, Update and Delete

Each of these objects implement the following (2) interfaces:

```
public function prepareStatement(Adapter $adapter, StatementInterface $statement);
public function getSqlString(PlatformInterface $adapterPlatform = null);
```

These are the functions you can call to either produce (a) a prepared statement, or (b) a string to be executed.

Zend\Db\Sql\Select

`Zend\Db\Sql\Select` is an object who's primary function is to present a unified API for building platform specific SQL SELECT queries. The object can be instantiated and consumed without `Zend\Db\Sql\Sql`:

```
use Zend\Db\Sql\Select;
$select = new Select();
// or, to produce a $select bound to a specific table
$select = new Select('foo');
```

If a table is provided to the Select object, then from() cannot be called later to change the name of the table.

Once you have a valid Select object, the following API can be used to further specify various select statement parts:

```
class Select extends AbstractSql implements SqlInterface, PreparableSqlInterface
{
    const JOIN_INNER = 'inner';
    const JOIN_OUTER = 'outer';
    const JOIN_LEFT = 'left';
    const JOIN_RIGHT = 'right';
    const SQL_STAR = '*';
    const ORDER_ASCENDING = 'ASC';
```

```
    const ORDER_DESENDING = 'DESC';

    public $where; // @param Where $where

    public function __construct($table = null);
    public function from($table);
    public function columns(array $columns, $prefixColumnsWithTable = true);
    public function join($name, $on, $columns = self::SQL_STAR, $type = self::JOIN_
↪INNER);
    public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
    public function group($group);
    public function having($predicate, $combination = Predicate\PredicateSet::OP_AND);
    public function order($order);
    public function limit($limit);
    public function offset($offset);
}
```

from():

```
// as a string:
$select->from('foo');

// as an array to specify an alias:
// produces SELECT "t".* FROM "table" AS "t"

$select->from(array('t' => 'table'));

// using a Sql\TableIdentifier:
// same output as above

$select->from(new TableIdentifier(array('t' => 'table')));
```

columns():

```
// as array of names
$select->columns(array('foo', 'bar'));

// as an associative array with aliases as the keys:
// produces 'bar' AS 'foo', 'bax' AS 'baz'

$select->columns(array('foo' => 'bar', 'baz' => 'bax'));
```

join():

```
$select->join(
     'foo' // table name,
     'id = bar.id', // expression to join on (will be quoted by platform object␣
↪before insertion),
     array('bar', 'baz'), // (optional) list of columns, same requiremetns as␣
↪columns() above
     $select::JOIN_OUTER // (optional), one of inner, outer, left, right also␣
↪represtned by constants in the API
);

$select->from(array('f' => 'foo'))  // base table
    ->join(array('b' => 'bar'),     // join table with alias
    'f.foo_id = b.foo_id');         // join expression
```

where(), having():

```
see Where/Having section below
```

order():

```
$select = new Select;
$select->order('id DESC'); // produces 'id' DESC

$select = new Select;
$select->order('id DESC')
     ->order('name ASC, age DESC'); // produces 'id' DESC, 'name' ASC, 'age' DESC

$select = new Select;
$select->order(array('name ASC', 'age DESC')); // produces 'name' ASC, 'age' DESC
```

limit() and offset():

```
$select = new Select;
$select->limit(5); // always takes an integer/numeric
$select->offset(10); // similarly takes an integer/numeric
```

Zend\Db\Sql\Insert

The Insert API:

```
class Insert implements SqlInterface, PreparableSqlInterface
{
     const VALUES_MERGE = 'merge';
     const VALUES_SET   = 'set';

     public function __construct($table = null);
     public function into($table);
     public function columns(array $columns);
     public function values(array $values, $flag = self::VALUES_SET);
}
```

Similarly to Select objects, the table can be set at construction time or via into().

columns():

```
$insert->columns(array('foo', 'bar')); // set the valid columns
```

values():

```
// default behavior of values is to set the values
// succesive calls will not preserve values from previous calls
$insert->values(array(
     'col_1' => 'value1',
     'col_2' => 'value2'
));
```

```
// merging values with previous calls
$insert->values(array('col_2' => 'value2'), $insert::VALUES);
```

Zend\Db\Sql\Update

```
class Update
{
    const VALUES_MERGE = 'merge';
    const VALUES_SET   = 'set';

    public $where; // @param Where $where
    public function __construct($table = null);
    public function table($table);
    public function set(array $values, $flag = self::VALUES_SET);
    public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
}
```

set():

```
$update->set(array('foo' => 'bar', 'baz' => 'bax'));
```

where():

```
See where section below.
```

Zend\Db\Sql\Delete

```
class Delete
{
    public $where; // @param Where $where
    public function __construct($table = null);
    public function from($table);
    public function where($predicate, $combination = Predicate\PredicateSet::OP_AND);
}
```

where():

```
See where section below.
```

Zend\Db\Sql\Where & Zend\Db\Sql\Having

In the following, we will talk about Where, Having is implies as being the same API.

Effectively, Where and Having extend from the same base object, a Predicate (and PredicateSet). All of the parts that make up a where or having that are and'ed or or'd together are called predicates. The full set of predicates is called a PredicateSet. This object set generally contains the values (and identifiers) separate from the fragment they belong to until the last possible moment when the statement is either used to be prepared (parameteritized), or executed. In parameterization, the parameters will be replaced with their proper placeholder (a named or positional parameter), and the values stored inside a Adapter\ParameterContainer. When executed, the values will be interpolated into the fragments they belong to and properly quoted.

It is important to know that in this API, a distinction is made between what elements are considered identifiers (TYPE_IDENTIFIER) and which of those is a value (TYPE_VALUE). There is also a special use case type for literal values (TYPE_LITERAL). These are all exposed via the `Zend\Db\Sql\ExpressionInterface` interface.

The Zend\Db\Sql\Where (Predicate/PredicateSet) API:

```php
// Where & Having:
class Predicate extends PredicateSet
{
     public $and;
     public $or;
     public $AND;
     public $OR;
     public $NEST;
     public $UNNSET;

    public function nest();
    public function setUnnest(Predicate $predicate);
    public function unnest();
    public function equalTo($left, $right, $leftType = self::TYPE_IDENTIFIER,
↪$rightType = self::TYPE_VALUE);
    public function lessThan($left, $right, $leftType = self::TYPE_IDENTIFIER,
↪$rightType = self::TYPE_VALUE);
    public function greaterThan($left, $right, $leftType = self::TYPE_IDENTIFIER,
↪$rightType = self::TYPE_VALUE);
    public function lessThanOrEqualTo($left, $right, $leftType = self::TYPE_
↪IDENTIFIER, $rightType = self::TYPE_VALUE);
    public function greaterThanOrEqualTo($left, $right, $leftType = self::TYPE_
↪IDENTIFIER, $rightType = self::TYPE_VALUE);
    public function like($identifier, $like);
    public function literal($literal, $parameter);
    public function isNull($identifier);
    public function isNotNull($identifier);
    public function in($identifier, array $valueSet = array());
    public function between($identifier, $minValue, $maxValue);


     // Inherited From PredicateSet

    public function addPredicate(PredicateInterface $predicate, $combination = null);
    public function getPredicates();
    public function orPredicate(PredicateInterface $predicate);
    public function andPredicate(PredicateInterface $predicate);
    public function getExpressionData();
    public function count();

}
```

Each method in the Where API will produce a coresponding Predicate object of a similarly named type, described below, with the full API of the object:

equalTo(), lessThan(), greaterThan(), lessThanOrEqualTo(), greaterThanOrEqualTo():

```php
$where->equalTo('id', 5);

// same as the following workflow
$where->addPredicate(
     new Predicate\Operator($left, Operator::OPERATOR_EQUAL_TO, $right, $leftType,
↪$rightType)
);

class Operator implements PredicateInterface
{
    const OPERATOR_EQUAL_TO                 = '=';
```

```
    const OP_EQ                                 = '=';
    const OPERATOR_NOT_EQUAL_TO                 = '!=';
    const OP_NE                                 = '!=';
    const OPERATOR_LESS_THAN                    = '<';
    const OP_LT                                 = '<';
    const OPERATOR_LESS_THAN_OR_EQUAL_TO        = '<=';
    const OP_LTE                                = '<=';
    const OPERATOR_GREATER_THAN                 = '>';
    const OP_GT                                 = '>';
    const OPERATOR_GREATER_THAN_OR_EQUAL_TO     = '>=';
    const OP_GTE                                = '>=';

    public function __construct($left = null, $operator = self::OPERATOR_EQUAL_TO,
↪$right = null, $leftType = self::TYPE_IDENTIFIER, $rightType = self::TYPE_VALUE);
    public function setLeft($left);
    public function getLeft();
    public function setLeftType($type);
    public function getLeftType();
    public function setOperator($operator);
    public function getOperator();
    public function setRight($value);
    public function getRight();
    public function setRightType($type);
    public function getRightType();
    public function getExpressionData();
}
```

like($identifier, $like):

```
$where->like($identifier, $like):

// same as
$where->addPredicate(
     new Predicate\Like($identifier, $like)
);

// full API

class Like implements PredicateInterface
{
    public function __construct($identifier = null, $like = null);
    public function setIdentifier($identifier);
    public function getIdentifier();
    public function setLike($like);
    public function getLike();
}
```

literal($literal, $parameter);

```
$where->literal($literal, $parameter);

// same as
$where->addPredicate(
    new Predicate\Expression($literal, $parameter)
);

// full API
class Expression implements ExpressionInterface, PredicateInterface
```

```
{
    const PLACEHOLDER = '?';
     public function __construct($expression = null, $valueParameter = null /*[,
↪$valueParameter, ... ]*/);
    public function setExpression($expression);
    public function getExpression();
    public function setParameters($parameters);
    public function getParameters();
    public function setTypes(array $types);
    public function getTypes();
}
```

isNull($identifier);

```
$where->isNull($identifier);

// same as
$where->addPredicate(
    new Predicate\IsNull($identifier)
);

// full API
class IsNull implements PredicateInterface
{
    public function __construct($identifier = null);
    public function setIdentifier($identifier);
    public function getIdentifier();
}
```

isNotNull($identifier);

```
$where->isNotNull($identifier);

// same as
$where->addPredicate(
    new Predicate\IsNotNull($identifier)
);

// full API
class IsNotNull implements PredicateInterface
{
    public function __construct($identifier = null);
    public function setIdentifier($identifier);
    public function getIdentifier();
}
```

in($identifier, array $valueSet = array());

```
$where->in($identifier, array $valueSet = array());

// same as
$where->addPredicate(
    new Predicate\In($identifier, $valueSet)
);

// full API
class In implements PredicateInterface
{
```

```
    public function __construct($identifier = null, array $valueSet = array());
    public function setIdentifier($identifier);
    public function getIdentifier();
    public function setValueSet(array $valueSet);
    public function getValueSet();
}
```

between($identifier, $minValue, $maxValue);

```
$where->between($identifier, $minValue, $maxValue);

// same as
$where->addPredicate(
    new Predicate\Between($identifier, $minValue, $maxValue)
);

// full API
class Between implements PredicateInterface
{
    public function __construct($identifier = null, $minValue = null, $maxValue =
↪null);
    public function setIdentifier($identifier);
    public function getIdentifier();
    public function setMinValue($minValue);
    public function getMinValue();
    public function setMaxValue($maxValue);
    public function getMaxValue();
    public function setSpecification($specification);
}
```

CHAPTER 42

Zend\Db\TableGateway

The Table Gateway object is intended to provide an object that represents a table in a database, and the methods of this object mirror the most common operations on a database table. In code, the interface for such an object looks like this:

```
interface Zend\Db\TableGateway\TableGatewayInterface
{
    public function getTable();
    public function select($where = null);
    public function insert($set);
    public function update($set, $where = null);
    public function delete($where);
}
```

There are two primary implementations of the `TableGatewayInterface` that are of the most useful: `AbstractTableGateway` and `TableGateway`. The `AbstractTableGateway` is an abstract basic implementation that provides functionality for `select()`, `insert()`, `update()`, `delete()`, as well as an additional API for doing these same kinds of tasks with explicit SQL objects. These methods are `selectWith()`, `insertWith()`, `updateWith()` and `deleteWith()`. In addition, AbstractTableGateway also implements a "Feature" API, that allows for expanding the behaviors of the base `TableGateway` implementation without having to extend the class with this new functionality. The `TableGateway` concrete implementation simply adds a sensible constructor to the `AbstractTableGateway` class so that out-of-the-box, `TableGateway` does not need to be extended in order to be consumed and utilized to its fullest.

Basic Usage

The quickest way to get up and running with Zend\Db\TableGateway is to configure and utilize the concrete implementation of the `TableGateway`. The API of the concrete `TableGateway` is:

```
class TableGateway extends AbstractTableGateway
{
     public $lastInsertValue;
     public $table;
```

```
    public $adapter;

    public function __construct($table, Adapter $adapter, $features = null,␣
↪ResultSet $resultSetPrototype = null, Sql $sql = null)

    /** Inherited from AbstractTableGateway */

    public function isInitialized();
    public function initialize();
    public function getTable();
    public function getAdapter();
    public function getColumns();
    public function getFeatureSet();
    public function getResultSetPrototype();
    public function getSql();
    public function select($where = null);
    public function selectWith(Select $select);
    public function insert($set);
    public function insertWith(Insert $insert);
    public function update($set, $where = null);
    public function updateWith(Update $update);
    public function delete($where);
    public function deleteWith(Delete $delete);
    public function getLastInsertValue();

}
```

The concrete `TableGateway` object practices constructor injection for getting dependencies and options into the instance. The table name and an instance of an Adapter are all that is needed to setup a working `TableGateway` object.

Out of the box, this implementation makes no assumptions about table structure or metadata, and when `select()` is executed, a simple ResultSet object with the populated Adapter's Result (the datasource) will be returned and ready for iteration.

```
use Zend\Db\TableGateway\TableGateway;
$projectTable = new TableGateway('project', $adapter);
$rowset = $projectTable->select(array('type' => 'PHP'));

echo 'Projects of type PHP: ';
foreach ($rowset as $projectRow) {
    echo $projectRow['name'] . PHP_EOL;
}

// or, when expecting a single row:
$artistTable = new TableGateway('artist', $adapter);
$rowset = $artistTable->select(array('id' => 2));
$artistRow = $rowset->current();

var_dump($artistRow);
```

The `select()` method takes the same arguments as `Zend\Db\Sql\Select::where()` with the addition of also being able to accept a closure, which in turn, will be passed the current Select object that is being used to build the SELECT query. The following usage is possible:

```
use Zend\Db\TableGateway\TableGateway;
use Zend\Db\Sql\Select;
$artistTable = new TableGateway('artist', $adapter);
```

```
4
5  // search for at most 2 artists who's name starts with Brit, ascending
6  $rowset = $artistTable->select(function (Select $select) {
7       $select->where->like('name', 'Brit%');
8       $select->order('name ASC')->limit(2);
9  });
```

TableGateway Features

The Features API allows for extending the functionality of the base `TableGateway` object without having to polymorphically extend the base class. This allows for a wider array of possible mixing and matching of features to achieve a particular behiavior that needs to be attained to make the base implementation of `TableGateway` useful for a particular problem.

With the `TableGateway` object, features should be injected though the constructor. The constructor can take Features in 3 different forms: as a single feature object, as a FeatureSet object, or as an array of Feature objects.

There are a number of features built-in and shipped with Zend\Db:

- GlobalAdapterFeature: the ability to use a global/static adapter without needing to inject it into a `TableGateway` instance. This is more useful when you are extending the `AbstractTableGateway` implementation:

```
1  class MyTableGateway extends <classname>AbstractTableGateway</classname>
2  {
3     public function __construct()
4     {
5             $this->table = 'my_table';
6             $this->featureSet = new Feature\FeatureSet();
7             $this->featureSet->addFeature(new Feature\GlobalAdapterFeature());
8             $this->initialize();
9     }
10 }
11
12 // elsewhere in code, in a bootstrap
13 Zend\Db\TableGateway\Feature\GlobalAdapterFeature::setStaticAdapter($adapter);
14
15 // in a controller, or model somewhere
16 $table = new MyTableGateway(); // adapter is statially loaded
```

- MasterSlaveFeature: the ability to use a master adapter for insert(), update(), and delete() while using a slave adapter for all select() operations.

```
1  $table = new TableGateway('artist', $adapter, new Feature\MasterSlaveFeature(
   ↪$slaveAdapter));
```

- MetadataFeature: the ability populate `TableGateway` with column information from a Metadata object. It will also store the primary key information in case RowGatewayFeature needs to consume this information.

```
1  $table = new TableGateway('artist', $adapter, new Feature\MeatadataFeature());
```

- EventFeature: the ability utilize a `TableGateway` object with Zend\EventManager and to be able to subscribe to various events in a `TableGateway` lifecycle.

```
1  $table = new TableGateway('artist', $adapter, new Feature\EventFeature(
   ↪$eventManagerInstance));
```

- RowGatewayFeature: the ability for `select()` to return a ResultSet object that upon iteration will

```
$table = new TableGateway('artist', $adapter, new Feature\RowGatewayFeature('id
↪'));
$results = $table->select(array('id' => 2));

$artistRow = $results->current();
$artistRow->name = 'New Name';
$artistRow->save();
```

CHAPTER 43

Zend\Db\RowGateway

`Zend\Db\RowGateway` is a sub-component of Zend\Db that implements the Row Gateway pattern from PoEAA. This effectively means that Row Gateway objects primarily model a row in a database, and have methods such as save() and delete() that will help persist this row-as-an-object in the database itself. Likewise, after a row from the database is retrieved, it can then be manipulated and save()'d back to the database in the same position (row), or it can be delete()'d from the table.

The interface for a Row Gateway object simply adds save() and delete() and this is the interface that should be assumed when a component has a dependency that is expected to be an instance of a RowGateway object:

```
interface RowGatewayInterface
{
    public function save();
    public function delete();
}
```

Quickstart

While most of the time, RowGateway will be used in conjucntion with other Zend\Db\ResultSet producing objects, it is possible to use it standalone. To use it standalone, you simply need an Adapter and a set of data to work with. The following use case demonstrates Zend\Db\RowGateway\RowGateway usage in its simplest form:

```
use Zend\Db\RowGateway\RowGateway;

// query the database
$resultSet = $adapter->query('SELECT * FROM `user` WHERE `id` = ?', array(2));

// get array of data
$rowData = $resultSet->current()->getArrayCopy();

// row gateway
$rowGateway = new RowGateway('id', 'my_table', $adapter);
$rowGateway->populate($rowData);
```

```php

$rowGateway->first_name = 'New Name';
$rowGateway->save();

// or delete this row:
$rowGateway->delete();
```

The workflow described above is greatly simplified when RowGateway is used in conjunction with the TableGateway feature. What this achieves is a Table Gateway object that when select()'ing from a table, will produce a ResultSet that is then capable of producing valid Row Gateway objects. Its usage looks like this:

```php
use Zend\Db\TableGateway\Feature\RowGatewayFeature;
use Zend\Db\TableGateway\TableGateway;

$table = new TableGateway('artist', $adapter, new RowGatewayFeature('id'));
$results = $table->select(array('id' => 2));

$artistRow = $results->current();
$artistRow->name = 'New Name';
$artistRow->save();
```

CHAPTER 44

Zend\Db\Metadata

`Zend\Db\Metadata` is as sub-component of Zend\Db that makes it possible to get metadata information about tables, columns, constraints, triggers, and other information from a database in a standardized way. The primary interface for the Metadata objects is:

```
interface MetadataInterface
{
    public function getSchemas();

    public function getTableNames($schema = null, $includeViews = false);
    public function getTables($schema = null, $includeViews = false);
    public function getTable($tableName, $schema = null);

    public function getViewNames($schema = null);
    public function getViews($schema = null);
    public function getView($viewName, $schema = null);

    public function getColumnNames($table, $schema = null);
    public function getColumns($table, $schema = null);
    public function getColumn($columnName, $table, $schema = null);

    public function getConstraints($table, $schema = null);
    public function getConstraint($constraintName, $table, $schema = null);
    public function getConstraintKeys($constraint, $table, $schema = null);

    public function getTriggerNames($schema = null);
    public function getTriggers($schema = null);
    public function getTrigger($triggerName, $schema = null);
}
```

Basic Usage

Usage of `Zend\Db\Metadata` is very straight forward. The top level class Zend\Db\Metadata\Metadata will, given an adapter, choose the best strategy (based on the database platform being used) for retrieving metadata. In most cases, information will come from querying the INFORMATION_SCHEMA tables generally accessible to all database connections about the currently accessible schema.

Metadata::get*Names() methods will return an array of strings, while the other methods will return specific value objects with the containing information. This is best demonstrated by the script below.

```
$metadata = new Zend\Db\Metadata\Metadata($adapter);

// get the table names
$tableNames = $metadata->getTableNames();

foreach ($tableNames as $tableName) {
    echo 'In Table ' . $tableName . PHP_EOL;

    /** @var $table Zend\Db\Metadata\Object\TableObject */
    $table = $metadata->getTable($tableName);

    echo '    With columns: ' . PHP_EOL;
    foreach ($table->getColumns() as $column) {
        /** @var $column Zend\Db\Metadata\Object\ColumnObject */
        echo '        ' . $column->getName()
            . ' -> ' . $column->getDataType()
            . PHP_EOL;
    }

    echo PHP_EOL;
    echo '    With constraints: ' . PHP_EOL;

    foreach ($metadata->getConstraints($tableName) as $constraint) {
        /** @var $constraint Zend\Db\Metadata\Object\ConstraintObject */
        echo '        ' . $constraint->getName()
            . ' -> ' . $constraint->getType()
            . PHP_EOL;
        if (!$constraint->hasColumns()) {
            continue;
        }
        echo '            column: ' . implode(', ', $constraint->getColumns());
        if ($constraint->isForeignKey()) {
            $fkCols = array();
            foreach ($constraint->getReferencedColumns() as $refColumn) {
                $fkCols[] = $constraint->getReferencedTableName() . '.' . $refColumn;
            }
            echo ' => ' . implode(', ', $fkCols);
        }
        echo PHP_EOL;

    }

    echo '----' . PHP_EOL;
}
```

Metadata returns value objects that provide an interface to help developers better explore the metadata. Below is the API for the various value objects:

The TableObject:

```
class Zend\Db\Metadata\Object\TableObject
{
    public function __construct($name);
    public function setColumns(array $columns);
    public function getColumns();
    public function setConstraints($constraints);
    public function getConstraints();
    public function setName($name);
    public function getName();
}
```

The ColumnObject:

```
class Zend\Db\Metadata\Object\ColumnObject {
    public function __construct($name, $tableName, $schemaName = null);
    public function setName($name);
    public function getName();
    public function getTableName();
    public function setTableName($tableName);
    public function setSchemaName($schemaName);
    public function getSchemaName();
    public function getOrdinalPosition();
    public function setOrdinalPosition($ordinalPosition);
    public function getColumnDefault();
    public function setColumnDefault($columnDefault);
    public function getIsNullable();
    public function setIsNullable($isNullable);
    public function isNullable();
    public function getDataType();
    public function setDataType($dataType);
    public function getCharacterMaximumLength();
    public function setCharacterMaximumLength($characterMaximumLength);
    public function getCharacterOctetLength();
    public function setCharacterOctetLength($characterOctetLength);
    public function getNumericPrecision();
    public function setNumericPrecision($numericPrecision);
    public function getNumericScale();
    public function setNumericScale($numericScale);
    public function getNumericUnsigned();
    public function setNumericUnsigned($numericUnsigned);
    public function isNumericUnsigned();
    public function getErratas();
    public function setErratas(array $erratas);
    public function getErrata($erratumName);
    public function setErrata($erratumName, $erratumValue);
}
```

The ConstraintObject:

```
class Zend\Db\Metadata\Object\ConstraintObject
{
    public function __construct($name, $tableName, $schemaName = null);
    public function setName($name);
    public function getName();
    public function setSchemaName($schemaName);
    public function getSchemaName();
    public function getTableName();
```

```
    public function setTableName($tableName);
    public function setType($type);
    public function getType();
    public function hasColumns();
    public function getColumns();
    public function setColumns(array $columns);
    public function getReferencedTableSchema();
    public function setReferencedTableSchema($referencedTableSchema);
    public function getReferencedTableName();
    public function setReferencedTableName($referencedTableName);
    public function getReferencedColumns();
    public function setReferencedColumns(array $referencedColumns);
    public function getMatchOption();
    public function setMatchOption($matchOption);
    public function getUpdateRule();
    public function setUpdateRule($updateRule);
    public function getDeleteRule();
    public function setDeleteRule($deleteRule);
    public function getCheckClause();
    public function setCheckClause($checkClause);
    public function isPrimaryKey();
    public function isUnique();
    public function isForeignKey();
    public function isCheck();

}
```

The TriggerObject:

```
class Zend\Db\Metadata\Object\TriggerObject
{
    public function getName();
    public function setName($name);
    public function getEventManipulation();
    public function setEventManipulation($eventManipulation);
    public function getEventObjectCatalog();
    public function setEventObjectCatalog($eventObjectCatalog);
    public function getEventObjectSchema();
    public function setEventObjectSchema($eventObjectSchema);
    public function getEventObjectTable();
    public function setEventObjectTable($eventObjectTable);
    public function getActionOrder();
    public function setActionOrder($actionOrder);
    public function getActionCondition();
    public function setActionCondition($actionCondition);
    public function getActionStatement();
    public function setActionStatement($actionStatement);
    public function getActionOrientation();
    public function setActionOrientation($actionOrientation);
    public function getActionTiming();
    public function setActionTiming($actionTiming);
    public function getActionReferenceOldTable();
    public function setActionReferenceOldTable($actionReferenceOldTable);
    public function getActionReferenceNewTable();
    public function setActionReferenceNewTable($actionReferenceNewTable);
    public function getActionReferenceOldRow();
    public function setActionReferenceOldRow($actionReferenceOldRow);
    public function getActionReferenceNewRow();
```

```
30      public function setActionReferenceNewRow($actionReferenceNewRow);
31      public function getCreated();
32      public function setCreated($created);
33  }
```

CHAPTER 45

Introduction to Zend\Di

Dependency Injection

Dependency Injection (here-in called DI) is a concept that has been talked about in numerous places over the web. Simply put, we'll explain the act of injecting dependencies simply with this below code:

```
1 $b = new MovieLister(new MovieFinder));
```

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

Dependency Injection Containers

When your code is written in such a way that all your dependencies are injected into consuming objects, you might find that the simple act of wiring an object has gotten more complex. When this becomes the case, and you find that this wiring is creating more boilerplate code, this makes for an excellent opportunity to utilize a Dependency Injection Container.

In it's simplest form, a Dependency Injection Container (here-in called a DiC for brevity), is an object that is capable of creating objects on request and managing the "wiring", or the injection of required dependencies, for those requested objects. Since the patterns that developers employ in writing DI capable code vary, DiC's are generally either in the form of smallish objects that suit a very specific pattern, or larger DiC frameworks.

Zend\Di is a DiC framework. While for the simplest code there is no configuration needed, and the use cases are quite simple; for more complex code, Zend\Di is capable of being configured to wire these complex use cases

CHAPTER 46

Zend\Di Quickstart

This QuickStart is intended to get developers familiar with the concepts of the Zend\Di DiC. Generally speaking, code is never as simple as it is inside this example, so working knowledge of the other sections of the manual is suggested.

Assume for a moment, you have the following code as part of your application that you feel is a good candidate for being managed by a DiC, after all, you are already injecting all your dependencies:

```
namespace MyLibrary
{
    class DbAdapter
    {
        protected $username = null;
        protected $password = null;
        public function __construct($username, $password)
        {
            $this->username = $username;
            $this->password = $password;
        }
    }
}

namespace MyMovieApp
{
    class MovieFinder
    {
        protected $dbAdapter = null;
        public function __construct(\MyLibrary\DbAdapter $dbAdapter)
        {
            $this->dbAdapter = $dbAdapter;
        }
    }

    class MovieLister
    {
        protected $movieFinder = null;
        public function __construct(MovieFinder $movieFinder)
```

```
        {
            $this->movieFinder = $movieFinder;
        }
    }
}
```

With the above code, you find yourself writing the following to wire and utilize this code:

```
// $config object is assumed

$dbAdapter = new MyLibrary\DbAdapter($config->username, $config->password);
$movieFinder = new MyMovieApp\MovieFinder($dbAdapter);
$movieLister = new MyMovieApp\MovieLister($movieFinder);
foreach ($movieLister as $movie) {
    // iterate and display $movie
}
```

If you are doing this above wiring in each controller or view that wants to list movies, not only can this become repetitive and boring to write, but also unmaintainable if for example you want to swap out one of these dependencies on a wholesale scale.

Since this example of code already practices good dependency injection, with constructor injection, it is a great candidate for using Zend\Di. The usage is as simple as:

```
    // inside a bootstrap somewhere
    $di = new Zend\Di\Di();
    $di->instanceManager()->setParameters('MyLibrary\DbAdapter', array(
        'username' => $config->username,
        'password' => $config->password
    ));

    // inside each controller
    $movieLister = $di->get('MyMovieApp\MovieLister');
    foreach ($movieLister as $movie) {
        // iterate and display $movie
    }
```

In the above example, we are obtaining a default instance of Zend\Di\Di. By 'default', we mean that Zend\Di\Di is constructed with a DefinitionList seeded with a RuntimeDefinition (uses Reflection) and an empty instance manager and no configuration. Here is the Zend\Di\Di constructor:

```
    public function __construct(DefinitionList $definitions = null, InstanceManager
↪$instanceManager = null, Configuration $config = null)
    {
        $this->definitions = ($definitions) ?: new DefinitionList(new␣
↪Definition\RuntimeDefinition());
        $this->instanceManager = ($instanceManager) ?: new InstanceManager();

        if ($config) {
            $this->configure($config);
        }
    }
```

This means that when $di->get() is called, it will be consulting the RuntimeDefinition, which uses reflection to understand the structure of the code. Once it knows the structure of the code, it can then know how the dependencies fit together and how to go about wiring your objects for you. Zend\Di\Definition\RuntimeDefinition will utilize the names of the parameters in the methods as the class parameter names. This is how both username and password key are mapped to the first and second parameter, respectively, of the constructor consuming these named parameters.

If you were to want to pass in the username and password at call time, this is achieved by passing them as the second argument of get():

```
    // inside each controller
    $di = new Zend\Di\Di();
    $movieLister = $di->get('MyMovieApp\MovieLister', array(
        'username' => $config->username,
        'password' => $config->password
    ));
    foreach ($movieLister as $movie) {
        // iterate and display $movie
    }
```

It is important to note that when using call time parameters, these parameter names will be applied to any class that accepts a parameter of such name.

By calling $di->get(), this instance of MovieLister will be automatically shared. This means subsequent calls to get() will return the same instance as previous calls. If you wish to have completely new instances of MovieLister, you can utilize $di->newInstance().

CHAPTER 47

Zend\Di Definition

Definitions are the place where Zend\Di attempts to understand the structure of the code it is attempting to wire. This means that if you've written non-ambiguous, clear and concise code; Zend\Di has a very good chance of understanding how to wire things up without much added complexity.

DefinitionList

Definitions are introduced to the Zend\Di\Di object through a definition list implemented as Zend\Di\DefinitionList (SplDoublyLinkedList). Order is important. Definitions in the front of the list will be consulted on a class before definitions at the end of the list.

Note: Regardless of what kind of Definition strategy you decide to use, it is important that your autoloaders are already setup and ready to use.

RuntimeDefinition

The default DefinitionList instantiated by Zend\Di\Di, when no other DefinitionList is provided, has as Definition\RuntimeDefinition baked-in. The RuntimeDefinition will respond to query's about classes by using Reflection. This Runtime definitions uses any available information inside methods: their signature, the names of parameters, the type-hints of the parameters, and the default values to determine if something is optional or required when making a call to that method. The more explicit you can be in your method naming and method signatures, the easier of a time Zend\Di\Definition\RuntimeDefinition will have determining the structure of your code.

This is what the constructor of a RuntimeDefinition looks like:

```
1 public function __construct(IntrospectionStrategy $introspectionStrategy = null,␣
  ↪array $explicitClasses = null)
2 {
3     $this->introspectionStrategy = ($introspectionStrategy) ?: new␣
  ↪IntrospectionStrategy();
4     if ($explicitClasses) {
```

```
        $this->setExplicitClasses($explicitClasses);
    }
}
```

The IntrospectionStrategy object is an object that determines the rules, or guidelines, for how the RuntimeDefinition will introspect information about your classes. Here are the things it knows how to do:

- Whether or not to use Annotations (Annotations are expensive and off by default, read more about these in the Annotations section)
- Which method names to include in the introspection, by default, the pattern /^set[A-Z]{1}\w*/ is registered by default, this is a list of patterns.
- Which interface names represent the interface injection pattern. By default, the pattern /\w*Aware\w*/ is registered, this is a list of patterns.

The constructor for the IntrospectionStrategy looks like this:

```
public function __construct(AnnotationManager $annotationManager = null)
{
    $this->annotationManager = ($annotationManager) ?: $this->
↪createDefaultAnnotationManager();
}
```

This goes to say that an AnnotationManager is not required, but if you wish to create a special AnnotationManager with your own annotations, and also wish to extend the RuntimeDefinition to look for these special Annotations, this is the place to do it.

The RuntimeDefinition also can be used to look up either all classes (implicitly, which is default), or explicitly look up for particular pre-defined classes. This is useful when your strategy for inspecting one set of classes might differ from those of another strategy for another set of classes. This can be achieved by using the setExplictClasses() method or by passing a list of classes as a second argument to the constructor of the RuntimeDefinition.

CompilerDefinition

The CompilerDefinition is very much similar in nature to the RuntimeDefinition with the exception that it can be seeded with more information for the purposes of "compiling" a definition. This is useful when you do not want to be making all those (sometimes expensive) calls to reflection and the annotation scanning system during the request of your application. By using the compiler, a definition can be created and written to disk to be used during a request, as opposed to the task of scanning the actual code.

For example, let's assume we want to create a script that will create definitions for some of our library code:

```
// in "package name" format
$components = array(
    'My_MovieApp',
    'My_OtherClasses',
);

foreach ($components as $component) {
    $diCompiler = new Zend\Di\Definition\CompilerDefinition;
    $diCompiler->addDirectory('/path/to/classes/' . str_replace('_', '/',
↪$component));

    $diCompiler->compile();
    file_put_contents(
        __DIR__ . '/../data/di/' . $component . '-definition.php',
```

```
        '<?php return ' . var_export($diCompiler->toArrayDefinition()->toArray(), true) . ';'
    );
}
```

This will create a couple of files that will return an array of the definition for that class. To utilize this in an application, the following code will suffice:

```
protected function setupDi(Application $app)
{
    $definitionList = new DefinitionList(array(
        new Definition\ArrayDefinition(include __DIR__ . '/path/to/data/di/My_MovieApp-definition.php'),
        new Definition\ArrayDefinition(include __DIR__ . '/path/to/data/di/My_OtherClasses-definition.php'),
        $runtime = new Definition\RuntimeDefinition(),
    ));
    $di = new Di($definitionList, null, new Configuration($this->config->di));
    $di->instanceManager()->addTypePreference('Zend\Di\LocatorInterface', $di);
    $app->setLocator($di);
}
```

The above code would more than likely go inside your application's or module's bootstrap file. This represents the simplest and most performant way of configuring your DiC for usage.

ClassDefinition

The idea behind using a ClassDefinition is two-fold. First, you may want to override some information inside of a RuntimeDefinition. Secondly, you might want to simply define your complete class's definition with an xml, ini, or php file describing the structure. This class definition can be fed in via Configuration or by directly instantiating and registering the Definition with the DefinitionList.

Todo - example

CHAPTER 48

Zend\Di InstanceManager

The InstanceManage is responsible for any runtime information associated with the Zend\Di\Di DiC. This means that the information that goes into the instance manager is specific to both how the particular consuming Application's needs and even more specifically to the environment in which the application is running.

Parameters

Parameters are simply entry points for either dependencies or instance configuration values. A class consist of a set of parameters, each uniquely named. When writing your classes, you should attempt to not use the same parameter name twice in the same class when you expect that that parameters is used for either instance configuration or an object dependency. This leads to an ambiguous parameter, and is a situation best avoided.

Our movie finder example can be further used to explain these concepts:

```
namespace MyLibrary
{
    class DbAdapter
    {
        protected $username = null;
        protected $password = null;
        public function __construct($username, $password)
        {
            $this->username = $username;
            $this->password = $password;
        }
    }
}

namespace MyMovieApp
{
    class MovieFinder
    {
        protected $dbAdapter = null;
        public function __construct(\MyLibrary\DbAdapter $dbAdapter)
```

```
        {
            $this->dbAdapter = $dbAdapter;
        }
    }

    class MovieLister
    {
        protected $movieFinder = null;
        public function __construct(MovieFinder $movieFinder)
        {
            $this->movieFinder = $movieFinder;
        }
    }
}
```

In the above example, the class DbAdapter has 2 parameters: username and password; MovieFinder has one parameter: dbAdapter, and MovieLister has one parameter: movieFinder. Any of these can be utilized for injection of either dependencies or scalar values during instance configuration or during call time.

When looking at the above code, since the dbAdapter parameter and the movieFinder parameter are both type-hinted with concrete types, the DiC can assume that it can fulfill these object tendencies by itself. On the other hand, username and password do not have type-hints and are, more than likely, scalar in nature. Since the DiC cannot reasonably know this information, it must be provided to the instance manager in the form of parameters. Not doing so will force $di->get('MyMovieApp\MovieLister') to throw an exception.

The following ways of using parameters are available:

```
// setting instance configuration into the instance manager
$di->instanceManager()->setParameters('MyLibrary\DbAdapter', array(
    'username' => 'myusername',
    'password' => 'mypassword'
));

// forcing a particular dependency to be used by the instance manager
$di->instanceManager()->setParameters('MyMovieApp\MovieFinder', array(
    'dbAdapter' => new MyLibrary\DbAdaper('myusername', 'mypassword')
));

// passing instance parameters at call time
$movieLister = $di->get('MyMovieApp\MovieLister', array(
    'username' => $config->username,
    'password' => $config->password
));

// passing a specific instance at call time
$movieLister = $di->get('MyMovieApp\MovieLister', array(
    'dbAdapter' => new MyLibrary\DbAdaper('myusername', 'mypassword')
));
```

Preferences

In some cases, you might be using interfaces as type hints as opposed to concrete types. Lets assume the movie example was modified in the following way:

```php
namespace MyMovieApp
{
    interface MovieFinderInterface
    {
        // methods required for this type
    }

    class GenericMovieFinder implements MovieFinderInterface
    {
        protected $dbAdapter = null;
        public function __construct(\MyLibrary\DbAdapter $dbAdapter)
        {
            $this->dbAdapter = $dbAdapter;
        }
    }

    class MovieLister
    {
        protected $movieFinder = null;
        public function __construct(MovieFinderInterface $movieFinder)
        {
            $this->movieFinder = $movieFinder;
        }
    }
}
```

What you'll notice above is that now the MovieLister type minimally expects that the dependency injected implements the MovieFinderInterface. This allows multiple implementations of this base interface to be used as a dependency, if that is what the consumer decides they want to do. As you can imagine, Zend\Di, by itself would not be able to determine what kind of concrete object to use fulfill this dependency, so this type of 'preference' needs to be made known to the instance manager.

To give this information to the instance manager, see the following code example:

```php
$di->instanceManager()->addTypePreference('MyMovieApp\MovieFinderInterface',
↪'MyMovieApp\GenericMovieFinder');
// assuming all instance config for username, password is setup
$di->get('MyMovieApp\MovieLister');
```

Aliases

In some situations, you'll find that you need to alias an instance. There are two main reasons to do this. First, it creates a simpler, alternative name to use when using the DiC, as opposed to using the full class name. Second, you might find that you need to have the same object type in two separate contexts. This means that when you alias a particular class, you can then attach a specific instance configuration to that alias; as opposed to attaching that configuration to the class name.

To demonstrate both of these points, we'll look at a use case where we'll have two separate DbAdapters, one will be for read-only operations, the other will be for read-write operations:

Note: Aliases can also have parameters registered at alias time

```php
// assume the MovieLister example of code from the QuickStart

$im = $di->instanceManager();

```

```php
// add alias for short naming
$im->addAlias('movielister', 'MyMovieApp\MovieLister');

// add aliases for specific instances
$im->addAlias('dbadapter-readonly', 'MyLibrary\DbAdapter', array(
    'username' => $config->db->readAdapter->username,
    'password' => $config->db->readAdapter->password,
));
$im->addAlias('dbadapter-readwrite', 'MyLibrary\DbAdapter', array(
    'username' => $config->db->readWriteAdapter>username,
    'password' => $config->db->readWriteAdapter>password,
));

// set a default type to use, pointing to an alias
$im->addTypePreference('MyLibrary\DbAdapter', 'dbadapter-readonly');

$movieListerRead = $di->get('MyMovieApp\MovieLister');
$movieListerReadWrite = $di->get('MyMovieApp\MovieLister', array('dbAdapter' =>
↪'dbadapter-readwrite'));
```

CHAPTER 49

Zend\Di Configuration

Most of the configuration for both the setup of Definitions as well as the setup of the Instance Manager can be attained by a configuration file. This file will produce an array (typically) and have a particular iterable structure.

The top two keys are 'definition' and 'instance', each specifying values for respectively, definition setup and instance manager setup.

The definition section expects the following information expressed as a PHP array:

```
$config = array(
    'definition' => array(
        'compiler' => array(/* @todo compiler information */),
        'runtime'  => array(/* @todo runtime information */),
        'class' => array(
            'instantiator' => '', // the name of the instantiator, by default this is
↪__construct
            'supertypes'   => array(), // an array of supertypes the class implements
            'methods'      => array(
                'setSomeParameter' => array( // a method name
                    'parameterName' => array(
                        'name',       // string parameter name
                        'type',       // type or null
                        'is-required' // bool
                    )
                )

            )
        )
    )
);
```

CHAPTER 50

Zend\Di Debugging & Complex Use Cases

Debugging a DiC

It is possible to dump the information contained within both the Definition and InstanceManager for a Di instance.

The easiest way is to do the following:

```
    Zend\Di\Display\Console::export($di);
```

If you are using a RuntimeDefinition where upon you expect a particular definition to be resolve at the first-call, you can see that information to the console display to force it to read that class:

```
        Zend\Di\Display\Console::export($di, array('A\ClassIWantTo\GetTheDefinitionFor
↪'));
```

Complex Use Cases

Interface Injection

```
namespace Foo\Bar {
    class Baz implements BamAwareInterface {
        public $bam;
        public function setBam(Bam $bam){
            $this->bam = $bam;
        }
    }
    class Bam {
    }
    interface BamAwareInterface
    {
        public function setBam(Bam $bam);
```

```
    }
}

namespace {
    include 'zf2bootstrap.php';
    $di = new Zend\Di\Di;
    $baz = $di->get('Foo\Bar\Baz');
}
```

Setter Injection with Class Definition

```
namespace Foo\Bar {
    class Baz {
        public $bam;
        public function setBam(Bam $bam){
            $this->bam = $bam;
        }
    }
    class Bam {
    }
}

namespace {
    $di = new Zend\Di\Di;
    $di->configure(new Zend\Di\Config(array(
        'definition' => array(
            'class' => array(
                'Foo\Bar\Baz' => array(
                    'setBam' => array('required' => true)
                )
            )
        )
    )));
    $baz = $di->get('Foo\Bar\Baz');
}
```

Multiple Injections To A Single Injection Point

```
namespace Application {
    class Page {
        public $blocks;
        public function addBlock(Block $block){
            $this->blocks[] = $block;
        }
    }
    interface Block {
    }
}

namespace MyModule {
    class BlockOne implements \Application\Block {}
    class BlockTwo implements \Application\Block {}
}

```

```
namespace {
    include 'zf2bootstrap.php';
    $di = new Zend\Di\Di;
    $di->configure(new Zend\Di\Config(array(
        'definition' => array(
            'class' => array(
                'Application\Page' => array(
                    'addBlock' => array(
                        'block' => array('type' => 'Application\Block', 'required' => true)
                    )
                )
            )
        ),
        'instance' => array(
            'Application\Page' => array(
                'injections' => array(
                    'MyModule\BlockOne',
                    'MyModule\BlockTwo'
                )
            )
        )
    )));
    $page = $di->get('Application\Page');
}
```

CHAPTER 51

Introduction

The `Zend\Dom` component provides tools for working with *DOM* documents and structures. Currently, we offer `Zend\Dom\Query`, which provides a unified interface for querying *DOM* documents utilizing both XPath and *CSS* selectors.

CHAPTER 52

Zend\Dom\Query

`Zend\Dom\Query` provides mechanisms for querying *XML* and (X) *HTML* documents utilizing either XPath or *CSS* selectors. It was developed to aid with functional testing of *MVC* applications, but could also be used for rapid development of screen scrapers.

CSS selector notation is provided as a simpler and more familiar notation for web developers to utilize when querying documents with *XML* structures. The notation should be familiar to anybody who has developed Cascading Style Sheets or who utilizes Javascript toolkits that provide functionality for selecting nodes utilizing *CSS* selectors (Prototype's $$() and Dojo's dojo.query were both inspirations for the component).

Theory of Operation

To use `Zend\Dom\Query`, you instantiate a `Zend\Dom\Query` object, optionally passing a document to query (a string). Once you have a document, you can use either the `query()` or `queryXpath()` methods; each method will return a `Zend\Dom\NodeList` object with any matching nodes.

The primary difference between `Zend\Dom\Query` and using DOMDocument + DOMXPath is the ability to select against *CSS* selectors. You can utilize any of the following, in any combination:

- **element types**: provide an element type to match: 'div', 'a', 'span', 'h2', etc.
- **style attributes**: *CSS* style attributes to match: '`.error`', '`div.error`', '`label.required`', etc. If an element defines more than one style, this will match as long as the named style is present anywhere in the style declaration.
- **id attributes**: element ID attributes to match: '#content', 'div#nav', etc.
- **arbitrary attributes**: arbitrary element attributes to match. Three different types of matching are provided:
  - **exact match**: the attribute exactly matches the string: 'div[bar="baz"]' would match a div element with a "bar" attribute that exactly matches the value "baz".
  - **word match**: the attribute contains a word matching the string: 'div[bar~="baz"]' would match a div element with a "bar" attribute that contains the word "baz". ' ' would match, but ' ' would not.

  - **substring match**: the attribute contains the string: 'div[bar*="baz"]' would match a div element with a "bar" attribute that contains the string "baz" anywhere within it.

- **direct descendents**: utilize '>' between selectors to denote direct descendents. 'div > span' would select only 'span' elements that are direct descendents of a 'div'. Can also be used with any of the selectors above.
- **descendents**: string together multiple selectors to indicate a hierarchy along which to search. '`div .foo span #one`' would select an element of id 'one' that is a descendent of arbitrary depth beneath a 'span' element, which is in turn a descendent of arbitrary depth beneath an element with a class of 'foo', that is an descendent of arbitrary depth beneath a 'div' element. For example, it would match the link to the word 'One' in the listing below:

```
<div>
<table>
    <tr>
        <td class="foo">
            <div>
                Lorem ipsum <span class="bar">
                    <a href="/foo/bar" id="one">One</a>
                    <a href="/foo/baz" id="two">Two</a>
                    <a href="/foo/bat" id="three">Three</a>
                    <a href="/foo/bla" id="four">Four</a>
                </span>
            </div>
        </td>
    </tr>
</table>
</div>
```

Once you've performed your query, you can then work with the result object to determine information about the nodes, as well as to pull them and/or their content directly for examination and manipulation. `Zend\Dom\NodeList` implements `Countable` and `Iterator`, and stores the results internally as a DOMDocument and DOMNodeList. As an example, consider the following call, that selects against the *HTML* above:

```
use Zend\Dom\Query;

$dom = new Query($html);
$results = $dom->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\NodeList` object.

Methods Available

The `Zend\Dom\Query` family of classes have the following methods available.

Zend\Dom\Query

The following methods are available to `Zend\Dom\Query`:

- `setDocumentXml($document, $encoding = null)`: specify an *XML* string to query against.
- `setDocumentXhtml($document, $encoding = null)`: specify an *XHTML* string to query against.
- `setDocumentHtml($document, $encoding = null)`: specify an *HTML* string to query against.
- `setDocument($document, $encoding = null)`: specify a string to query against; `Zend\Dom\Query` will then attempt to autodetect the document type.
- `setEncoding($encoding)`: specify an encoding string to use. This encoding will be passed to DOMDocument's constructor if specified.
- `getDocument()`: retrieve the original document string provided to the object.
- `getDocumentType()`: retrieve the document type of the document provided to the object; will be one of the `DOC_XML`, `DOC_XHTML`, or `DOC_HTML` class constants.
- `getEncoding()`: retrieves the specified encoding.
- `execute($query)`: query the document using *CSS* selector notation.
- `queryXpath($xPathQuery)`: query the document using XPath notation.

Zend\Dom\NodeList

As mentioned previously, `Zend\Dom\NodeList` implements both `Iterator` and `Countable`, and as such can be used in a `foreach()` loop as well as with the `count()` function. Additionally, it exposes the following methods:

- `getCssQuery()`: return the *CSS* selector query used to produce the result (if any).
- `getXpathQuery()`: return the XPath query used to produce the result. Internally, `Zend\Dom\Query` converts *CSS* selector queries to XPath, so this value will always be populated.
- `getDocument()`: retrieve the DOMDocument the selection was made against.

CHAPTER 53

The EventManager

Overview

The `EventManager` is a component designed for the following use cases:

- Implementing simple subject/observer patterns.
- Implementing Aspect-Oriented designs.
- Implementing event-driven architectures.

The basic architecture allows you to attach and detach listeners to named events, both on a per-instance basis as well as via shared collections; trigger events; and interrupt execution of listeners.

Quick Start

Typically, you will compose an `EventManager` instance in a class.

```
use Zend\EventManager\EventCollection;
use Zend\EventManager\EventManager;
use Zend\EventManager\EventManagerAware;

class Foo implements EventManagerAware
{
    protected $events;

    public function setEventManager(EventCollection $events)
    {
        $events->setIdentifiers(array(
            __CLASS__,
            get_called_class(),
        ));
        $this->events = $events;
        return $this;
```

```
    }

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

The above allows users to access the `EventManager` instance, or reset it with a new instance; if one does not exist, it will be lazily instantiated on-demand.

An `EventManager` is really only interesting if it triggers some events. Basic triggering takes three arguments: the event name, which is usually the current function/method name; the "context", which is usually the current object instance; and the arguments, which are usually the arguments provided to the current function/method.

```
class Foo
{
    // ... assume events definition from above

    public function bar($baz, $bat = null)
    {
        $params = compact('baz', 'bat');
        $this->getEventManager()->trigger(__FUNCTION__, $this, $params);
    }
}
```

In turn, triggering events is only interesting if something is listening for the event. Listeners attach to the `EventManager`, specifying a named event and the callback to notify. The callback receives an `Event` object, which has accessors for retrieving the event name, context, and parameters. Let's add a listener, and trigger the event.

```
use Zend\Log\Factory as LogFactory;

$log = LogFactory($someConfig);
$foo = new Foo();
$foo->getEventManager()->attach('bar', function ($e) use ($log) {
    $event  = $e->getName();
    $target = get_class($e->getTarget());
    $params = json_encode($e->getParams());

    $log->info(sprintf(
        '%s called on %s, using params %s',
        $event,
        $target,
        $params
    ));
});

// Results in log message:
$foo->bar('baz', 'bat');
// reading: bar called on Foo, using params {"baz" : "baz", "bat" : "bat"}"
```

Note that the second argument to `attach()` is any valid callback; an anonymous function is shown in the example in order to keep the example self-contained. However, you could also utilize a valid function name, a functor, a string referencing a static method, or an array callback with a named static method or instance method. Again, any PHP callback is valid.

Sometimes you may want to specify listeners without yet having an object instance of the class composing an `EventManager`. Zend Framework enables this through the concept of a `SharedEventCollection`. Simply put, you can inject individual `EventManager` instances with a well-known `SharedEventCollection`, and the `EventManager` instance will query it for additional listeners. Listeners attach to a `SharedEventCollection` in roughly the same way the do normal event managers; the call to `attach` is identical to the `EventManager`, but expects an additional parameter at the beginning: a named instance. Remember the example of composing an `EventManager`, how we passed it `__CLASS__`? That value, or any strings you provide in an array to the constructor, may be used to identify an instance when using a `SharedEventCollection`. As an example, assuming we have a `SharedEventManager` instance that we know has been injected in our `EventManager` instances (for instance, via dependency injection), we could change the above example to attach via the shared collection:

```
use Zend\Log\Factory as LogFactory;

// Assume $events is a Zend\EventManager\SharedEventManager instance

$log = LogFactory($someConfig);
$events->attach('Foo', 'bar', function ($e) use ($log) {
    $event  = $e->getName();
    $target = get_class($e->getTarget());
    $params = json_encode($e->getParams());

    $log->info(sprintf(
        '%s called on %s, using params %s',
        $event,
        $target,
        $params
    ));
});

// Later, instantiate Foo:
$foo = new Foo();
$foo->getEventManager()->setSharedEventCollection($events);

// And we can still trigger the above event:
$foo->bar('baz', 'bat');
// results in log message:
// bar called on Foo, using params {"baz" : "baz", "bat" : "bat"}"
```

Note: StaticEventManager

As of 2.0.0beta3, you can use the `StaticEventManager` singleton as a `SharedEventCollection`. As such, you do not need to worry about where and how to get access to the `SharedEventCollection`; it's globally available by simply calling *StaticEventManager::getInstance()*.

Be aware, however, that its usage is deprecated within the framework, and starting with 2.0.0beta4, you will instead configure a `SharedEventManager` instance that will be injected by the framework into individual `EventManager` instances.

The `EventManager` also provides the ability to detach listeners, short-circuit execution of an event either from within a listener or by testing return values of listeners, test and loop through the results returned by listeners, prioritize listeners, and more. Many of these features are detailed in the examples.

Wildcard Listeners

Sometimes you'll want to attach the same listener to many events or to all events of a given instance – or potentially, with a shared event collection, many contexts, and many events. The `EventManager` component allows for this.

Attaching to many events at once

```
$events = new EventManager();
$events->attach(array('these', 'are', 'event', 'names'), $callback);
```

Note that if you specify a priority, that priority will be used for all events specified.

Attaching using the wildcard

```
$events = new EventManager();
$events->attach('*', $callback);
```

Note that if you specify a priority, that priority will be used for this listener for any event triggered.

What the above specifies is that **any** event triggered will result in notification of this particular listener.

Attaching to many events at once via a SharedEventManager

```
$events = new SharedEventManager();
// Attach to many events on the context "foo"
$events->attach('foo', array('these', 'are', 'event', 'names'), $callback);

// Attach to many events on the contexts "foo" and "bar"
$events->attach(array('foo', 'bar'), array('these', 'are', 'event', 'names'),
↪$callback);
```

Note that if you specify a priority, that priority will be used for all events specified.

Attaching to many events at once via a SharedEventManager

```
$events = new SharedEventManager();
// Attach to all events on the context "foo"
$events->attach('foo', '*', $callback);

// Attach to all events on the contexts "foo" and "bar"
$events->attach(array('foo', 'bar'), '*', $callback);
```

Note that if you specify a priority, that priority will be used for all events specified.

The above is specifying that for the contexts "foo" and "bar", the specified listener should be notified for any event they trigger.

Configuration Options

EventManager Options

identifier A string or array of strings to which the given `EventManager` instance can answer when accessed via a `SharedEventManager`.

event_class The name of an alternate `Event` class to use for representing events passed to listeners.

shared_collections An instance of a `SharedEventCollection` instance to use when triggering events.

Available Methods

__construct `__construct(null|string|int $identifier)`

Constructs a new `EventManager` instance, using the given identifier, if provided, for purposes of shared collections.

setEventClass `setEventClass(string $class)`

Provide the name of an alternate `Event` class to use when creating events to pass to triggered listeners.

setSharedCollections `setSharedCollections(SharedEventCollection $collections = null)`

An instance of a `SharedEventCollection` instance to use when triggering events.

getSharedCollections `getSharedCollections()`

Returns the currently attached `SharedEventCollection` instance. Returns either a `null` if no collection is attached, or a `SharedEventCollection` instance otherwise.

trigger `trigger(string $event, mixed $target, mixed $argv, callback $callback)`

Triggers all listeners to a named event. The recommendation is to use the current function/method name for `$event`, appending it with values such as ".pre", ".post", etc. as needed. `$context` should be the current object instance, or the name of the function if not triggering within an object. `$params` should typically be an associative array or `ArrayAccess` instance; we recommend using the parameters passed to the function/method (`compact()` is often useful here). This method can also take a callback and behave in the same way as `triggerUntil()`.

The method returns an instance of `ResponseCollection`, which may be used to introspect return values of the various listeners, test for short-circuiting, and more.

triggerUntil `triggerUntil(string $event, mixed $context, mixed $argv, callback $callback)`

Triggers all listeners to a named event, just like *trigger()*, with the addition that it passes the return value from each listener to `$callback`; if `$callback` returns a boolean `true` value, execution of the listeners is interrupted. You can test for this using *$result->stopped()*.

attach `attach(string $event, callback $callback, int $priority)`

Attaches `$callback` to the `EventManager` instance, listening for the event `$event`. If a `$priority` is provided, the listener will be inserted into the internal listener stack using that priority; higher values execute earliest. (Default priority is "1", and negative priorities are allowed.)

The method returns an instance of `Zend\Stdlib\CallbackHandler`; this value can later be passed to `detach()` if desired.

attachAggregate `attachAggregate(string|ListenerAggregate $aggregate)`

If a string is passed for `$aggregate`, instantiates that class. The `$aggregate` is then passed the `EventManager` instance to its `attach()` method so that it may register listeners.

The `ListenerAggregate` instance is returned.

detach `detach(CallbackHandler $listener)`

Scans all listeners, and detaches any that match `$listener` so that they will no longer be triggered.

Returns a boolean `true` if any listeners have been identified and unsubscribed, and a boolean `false` otherwise.

detachAggregate `detachAggregate(ListenerAggregate $aggregate)`

Loops through all listeners of all events to identify listeners that are represented by the aggregate; for all matches, the listeners will be removed.

Returns a boolean `true` if any listeners have been identified and unsubscribed, and a boolean `false` otherwise.

getEvents `getEvents()`

Returns an array of all event names that have listeners attached.

getListeners `getListeners(string $event)`

Returns a `Zend\Stdlib\PriorityQueue` instance of all listeners attached to `$event`.

clearListeners `clearListeners(string $event)`

Removes all listeners attached to `$event`.

prepareArgs `prepareArgs(array $args)`

Creates an `ArrayObject` from the provided `$args`. This can be useful if you want yours listeners to be able to modify arguments such that later listeners or the triggering method can see the changes.

Examples

Modifying Arguments

Occasionally it can be useful to allow listeners to modify the arguments they receive so that later listeners or the calling method will receive those changed values.

As an example, you might want to pre-filter a date that you know will arrive as a string and convert it to a `DateTime` argument.

To do this, you can pass your arguments to `prepareArgs()`, and pass this new object when triggering an event. You will then pull that value back into your method.

```
class ValueObject
{
    // assume a composed event manager

    function inject(array $values)
    {
        $argv = compact('values');
        $argv = $this->getEventManager()->prepareArgs($argv);
        $this->getEventManager()->trigger(__FUNCTION__, $this, $argv);
        $date = isset($argv['values']['date']) ? $argv['values']['date'] : new␣
↪DateTime('now');

        // ...
```

```
    }
}

$v = new ValueObject();

$v->getEventManager()->attach('inject', function($e) {
    $values = $e->getParam('values');
    if (!$values) {
        return;
    }
    if (!isset($values['date'])) {
        $values['date'] = new DateTime('now');
        return;
    }
    $values['date'] = new Datetime($values['date']);
});

$v->inject(array(
    'date' => '2011-08-10 15:30:29',
));
```

Short Circuiting

One common use case for events is to trigger listeners until either one indicates no further processing should be done, or until a return value meets specific criteria. As examples, if an event creates a Response object, it may want execution to stop.

```
$listener = function($e) {
    // do some work

    // Stop propagation and return a response
    $e->stopPropagation(true);
    return $response;
};
```

Alternately, we could do the check from the method triggering the event.

```
class Foo implements DispatchableInterface
{
    // assume composed event manager

    public function dispatch(Request $request, Response $response = null)
    {
        $argv = compact('request', 'response');
        $results = $this->getEventManager()->triggerUntil(__FUNCTION__, $this, $argv,␣
↪function($v) {
            return ($v instanceof Response);
        });
    }
}
```

Typically, you may want to return a value that stopped execution, or use it some way. Both `trigger()` and `triggerUntil()` return a `ResponseCollection` instance; call its `stopped()` method to test if execution was stopped, and `last()` method to retrieve the return value from the last executed listener:

```php
class Foo implements DispatchableInterface
{
    // assume composed event manager

    public function dispatch(Request $request, Response $response = null)
    {
        $argv = compact('request', 'response');
        $results = $this->getEventManager()->triggerUntil(__FUNCTION__, $this, $argv, function($v) {
            return ($v instanceof Response);
        });

        // Test if execution was halted, and return last result:
        if ($results->stopped()) {
            return $results->last();
        }

        // continue...
    }
}
```

Assigning Priority to Listeners

One use case for the `EventManager` is for implementing caching systems. As such, you often want to check the cache early, and save to it late.

The third argument to `attach()` is a priority value. The higher this number, the earlier that listener will execute; the lower it is, the later it executes. The value defaults to 1, and values will trigger in the order registered within a given priority.

So, to implement a caching system, our method will need to trigger an event at method start as well as at method end. At method start, we want an event that will trigger early; at method end, an event should trigger late.

Here is the class in which we want caching:

```php
class SomeValueObject
{
    // assume it composes an event manager

    public function get($id)
    {
        $params = compact('id');
        $results = $this->getEventManager()->trigger('get.pre', $this, $params);

        // If an event stopped propagation, return the value
        if ($results->stopped()) {
            return $results->last();
        }

        // do some work...

        $params['__RESULT__'] = $someComputedContent;
        $this->getEventManager()->trigger('get.post', $this, $params);
    }
}
```

Now, let's create a `ListenerAggregateInterface` that can handle caching for us:

```php
use Zend\Cache\Cache;
use Zend\EventManager\EventCollection;
use Zend\EventManager\ListenerAggregateInterface;
use Zend\EventManager\EventInterface;

class CacheListener implements ListenerAggregateInterface
{
    protected $cache;

    protected $listeners = array();

    public function __construct(Cache $cache)
    {
        $this->cache = $cache;
    }

    public function attach(EventCollection $events)
    {
        $this->listeners[] = $events->attach('get.pre', array($this, 'load'), 100);
        $this->listeners[] = $events->attach('get.post', array($this, 'save'), -100);
    }

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

    public function load(EventInterface $e)
    {
        $id = get_class($e->getTarget()) . '-' . json_encode($e->getParams());
        if (false !== ($content = $this->cache->load($id))) {
            $e->stopPropagation(true);
            return $content;
        }
    }

    public function save(EventInterface $e)
    {
        $params  = $e->getParams();
        $content = $params['__RESULT__'];
        unset($params['__RESULT__']);

        $id = get_class($e->getTarget()) . '-' . json_encode($params);
        $this->cache->save($content, $id);
    }
}
```

We can then attach the aggregate to an instance.

```php
$value         = new SomeValueObject();
$cacheListener = new CacheListener($cache);
$value->getEventManager()->attachAggregate($cacheListener);
```

Now, as we call `get()`, if we have a cached entry, it will be returned immediately; if not, a computed entry will be

cached when we complete the method.

CHAPTER 54

Introduction to Zend\Form

`Zend\Form` is intended primarily as a bridge between your domain models and the View Layer. It composes a thin layer of objects representing form elements, an *InputFilter*, and a small number of methods for binding data to and from the form and attached objects.

The component consists of:

- `Elements`, which simply consist of a name and attributes.
- `Fieldsets`, which extend from `Elements`, but allow composing other fieldsets and elements.
- `Forms`, which extend from `Fieldsets` (and thus `Elements`), provide data and object binding, and compose `InputFilters`. Data binding is done via `Zend\Stdlib\Hydrator`.

To facilitate usage with the view layer, the `Zend\Form` component also aggregates a number of form-specific view helpers. These accept elements, fieldsets, and/or forms, and use the attributes they compose to render markup.

A small number of specialized elements are provided for accomplishing application-centric tasks. These include the `Csrf` element, used to prevent Cross Site Request Forgery attacks, and the `Captcha` element, used to display and validate *CAPTCHAs*.

A `Factory` is provided to facilitate creation of elements, fieldsets, forms, and the related input filter. The default `Form` implementation is backed by a factory to facilitate extension and ease the process of form creation.

The code related to forms can often spread between a variety of concerns: a form definition, an input filter definition, a domain model class, and one or more hydrator implementations. As such, finding the various bits of code and how they relate can become tedious. To simplify the situation, you can also annotate your domain model class, detailing the various input filter definitions, attributes, and hydrators that should all be used together. `Zend\Form\Annotation\AnnotationBuilder` can then be used to build the various objects you need.

CHAPTER 55

Form Quick Start

Forms are relatively easy to create. At the bare minimum, each element or fieldset requires a name; typically, you'll also provide some attributes to hint to the view layer how it might render the item. The form itself will also typically compose an `InputFilter`– which you can also conveniently create directly in the form via a factory. Individual elements can hint as to what defaults to use when generating a related input for the input filter.

Form validation is as easy as providing an array of data to the `setData()` method. If you want to simplify your work even more, you can bind an object to the form; on successful validation, it will be populated from the validated values.

Programmatic Form Creation

If nothing else, you can simply start creating elements, fieldsets, and forms and wiring them together.

```
use Zend\Captcha;
use Zend\Form\Element;
use Zend\Form\Fieldset;
use Zend\Form\Form;
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;

$name = new Element('name');
$name->setAttributes(array(
    'type'  => 'text',
    'label' => 'Your name',
));

$email = new Element('email');
$email->setAttributes(array(
    'type'  => 'email',
    'label' => 'Your email address',
));

$subject = new Element('subject');
```

```
$subject->setAttributes(array(
    'type'  => 'text',
    'label' => 'Subject',
));

$message = new Element('message');
$message->setAttributes(array(
    'type'  => 'textarea',
    'label' => 'Message',
));

$captcha = new Element\Captcha('captcha');
$captcha->setCaptcha(new Captcha\Dumb());
$captcha->setAttributes(array(
    'label' => 'Please verify you are human',
));

$csrf = new Element\Csrf('security');

$submit = new Element('send');
$submit->setAttributes(array(
    'type'  => 'submit',
    'label' => 'Send',
));


$form = new Form('contact');
$form->add($name);
$form->add($email);
$form->add($subject);
$form->add($message);
$form->add($captcha);
$form->add($csrf);
$form->add($send);

$nameInput = new Input('name');
// configure input... and all others
$inputFilter = new InputFilter();
// attach all inputs

$form->setInputFilter($inputFilter);
```

As a demonstration of fieldsets, let's alter the above slightly. We'll create two fieldsets, one for the sender information, and another for the message details.

```
$sender = new Fieldset('sender');
$sender->add($name);
$sender->add($email);

$details = new Fieldset('details');
$details->add($subject);
$details->add($message);

$form = new Form('contact');
$form->add($sender);
$form->add($details);
$form->add($captcha);
$form->add($csrf);
```

```php
$form->add($send);
```

Regardles of approach, as you can see, this can be tedious.

Creation via Factory

You can create the entire form, and input filter, using the `Factory`. This is particularly nice if you want to store your forms as pure configuration; you can simply pass the configuration to the factory and be done.

```php
use Zend\Form\Factory;
$factory = new Factory();
$form    = $factory->createForm(array(
    'hydrator' => 'Zend\Stdlib\Hydrator\ArraySerializable'
    'elements' => array(
        array(
            'name' => 'name',
            'attributes' => array(
                'type'  => 'text',
                'label' => 'Your name',
            ),
        ),
        array(
            'name' => 'email',
            'attributes' => array(
                'type'  => 'email',
                'label' => 'Your email address',
            ),
        ),
        array(
            'name' => 'subject',
            'attributes' => array(
                'type'  => 'text',
                'label' => 'Subject',
            ),
        ),
        array(
            'name' => 'message',
            'attributes' => array(
                'type'  => 'textarea',
                'label' => 'Message',
            ),
        ),
        array(
            'type' => 'Zend\Form\Element\Captcha',
            'name' => 'captcha',
            'attributes' => array(
                'label' => 'Please verify you are human',
                'captcha' => array(
                    'class' => 'Dumb',
                ),
            ),
        ),
        array(
            'type' => 'Zend\Form\Element\Csrf',
            'name' => 'security',
        ),
```

```
        array(
            'name' => 'send',
            'attributes' => array(
                'type'  => 'submit',
                'label' => 'Send',
            ),
        ),
    ),
    /* If we had fieldsets, they'd go here; fieldsets contain
     * "elements" and "fieldsets" keys, and potentially a "type"
     * key indicating the specific FieldsetInterface
     * implementation to use.
    'fieldsets' => array(
    ),
     */

    // Configuration to pass on to
    // Zend\InputFilter\Factory::createInputFilter()
    'input_filter' => array(
        /* ... */
    ),
));
```

If we wanted to use fieldsets, as we demonstrated in the previous example, we could do the following:

```
use Zend\Form\Factory;
$factory = new Factory();
$form    = $factory->createForm(array(
    'hydrator'  => 'Zend\Stdlib\Hydrator\ArraySerializable'
    'fieldsets' => array(
        array(
            'name' => 'sender',
            'elements' => array(
                array(
                    'name' => 'name',
                    'attributes' => array(
                        'type'  => 'text',
                        'label' => 'Your name',
                    ),
                ),
                array(
                    'name' => 'email',
                    'attributes' => array(
                        'type'  => 'email',
                        'label' => 'Your email address',
                    ),
                ),
            ),
        ),
        array(
            'name' => 'details',
            'elements' => array(
                array(
                    'name' => 'subject',
                    'attributes' => array(
                        'type'  => 'text',
                        'label' => 'Subject',
                    ),
```

```
                ),
                array(
                    'name' => 'message',
                    'attributes' => array(
                        'type'  => 'textarea',
                        'label' => 'Message',
                    ),
                ),
            ),
        ),
    ),
    'elements' => array(
        array(
            'type' => 'Zend\Form\Element\Captcha',
            'name' => 'captcha',
            'attributes' => array(
                'label' => 'Please verify you are human',
                'captcha' => array(
                    'class' => 'Dumb',
                ),
            ),
        ),
        array(
            'type' => 'Zend\Form\Element\Csrf',
            'name' => 'security',
        ),
        array(
            'name' => 'send',
            'attributes' => array(
                'type'  => 'submit',
                'label' => 'Send',
            ),
        ),
    ),

    // Configuration to pass on to
    // Zend\InputFilter\Factory::createInputFilter()
    'input_filter' => array(
        /* ... */
    ),
));
```

Note that the chief difference is nesting; otherwise, the information is basically the same.

The chief benefits to using the `Factory` are allowing you to store definitions in configuration, and usage of significant whitespace.

Factory-backed Form Extension

The default `Form` implementation is backed by the `Factory`. This allows you to extend it, and define your form internally. This has the benefit of allowing a mixture of programmatic and factory-backed creation, as well as defining a form for re-use in your application.

```
namespace Contact;

use Zend\Captcha\AdapterInterface as CaptchaAdapter;
use Zend\Form\Element;
```

```php
use Zend\Form\Form;

class ContactForm extends Form
{
    protected $captcha;

    public function setCaptcha(CaptchaAdapter $captcha)
    {
        $this->captcha = $captcha;
    }

    public function prepareElements()
    {
        // add() can take either an Element/Fieldset instance,
        // or a specification, from which the appropriate object
        // will be built.

        $this->add(array(
            'name' => 'name',
            'attributes' => array(
                'type'  => 'text',
                'label' => 'Your name',
            ),
        ));
        $this->add(array(
            'name' => 'email',
            'attributes' => array(
                'type'  => 'email',
                'label' => 'Your email address',
            ),
        ));
        $this->add(array(
            'name' => 'subject',
            'attributes' => array(
                'type'  => 'text',
                'label' => 'Subject',
            ),
        ));
        $this->add(array(
            'name' => 'message',
            'attributes' => array(
                'type'  => 'textarea',
                'label' => 'Message',
            ),
        ));
        $this->add(array(
            'type' => 'Zend\Form\Element\Captcha',
            'name' => 'captcha',
            'attributes' => array(
                'label' => 'Please verify you are human',
                'captcha' => $this->captcha,
            ),
        )),
        $this->add(new Element\Csrf('security'));
        $this->add(array(
            'name' => 'send',
            'attributes' => array(
                'type'  => 'submit',
```

```
                'label' => 'Send',
            ),
        ));

        // We could also define the input filter here, or
        // lazy-create it in the getInputFilter() method.
    }
));
```

You'll note that this example introduces a method, `prepareElements()`. This is done to allow altering and/or configuring either the form or input filter factory instances, which could then have bearing on how elements, inputs, etc. are created. In this case, it also allows injection of the CAPTCHA adapter, allowing us to configure it elsewhere in our application and inject it into the form.

Validating Forms

Validating forms requires three steps. First, the form must have an input filter attached. Second, you must inject the data to validate into the form. Third, you validate the form. If invalid, you can retrieve the error messages, if any.

```
$form = new Contact\ContactForm();

// If the form doesn't define an input filter by default, inject one.
$form->setInputFilter(new Contact\ContactFilter());

// Get the data. In an MVC application, you might try:
$data = $request->post();  // for POST data
$data = $request->query(); // for GET (or query string) data

$form->setData($data);

// Validate the form
if ($form->isValid() {
    $validatedData = $form->getData();
} else {
    $messages = $form->getMessages();
}
```

You can get the raw data if you want, by accessing the composed input filter.

```
$filter = $form->getInputFilter();

$rawValues    = $filter->getRawValues();
$nameRawValue = $filter->getRawValue('name');
```

Hinting to the Input Filter

Often, you'll create elements that you expect to behave in the same way on each usage, and for which you'll want specific filters or validation as well. Since the input filter is a separate object, how can you achieve these latter points?

Because the default form implementation composes a factory, and the default factory composes an input filter factory, you can have your elements and/or fieldsets hint to the input filter. If no input or input filter is provided in the input filter for that element, these hints will be retrieved and used to create them.

To do so, one of the following must occur. For elements, they must implement `Zend\InputFilter\InputProviderInterface`, which defines a `getInputSpecification()`

method; for fieldsets, they must implement `Zend\InputFilter\InputFilterProviderInterface`, which defines a `getInputFilterSpecification()` method.

In the case of an element, the `getInputSpecification()` method should return data to be used by the input filter factory to create an input.

```
namespace Contact\Form;

use Zend\Form\Element;
use Zend\InputFilter\InputProviderInterface;
use Zend\Validator;

class EmailElement extends Element implements InputProviderInterface
{
    protected $attributes = array(
        'type' => 'email',
    );

    public function getInputSpecification()
    {
        return array(
            'name'     => $this->getName(),
            'required' => true,
            'filters'  => array(
                array('name' => 'Zend\Filter\StringTrim'),
            ),
            'validators' => array(
                new Validator\Email(),
            ),
        );
    }
}
```

The above would hint to the input filter to create and attach an input named after the element, marking it as required, and giving it a `StringTrim` filter and an `Email` validator. Note that you can either rely on the input filter to create filters and validators, or directly instantiate them.

For fieldsets, you do very similarly; the difference is that `getInputFilterSpecification()` must return configuration for an input filter.

```
namespace Contact\Form;

use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;

class SenderFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
                'filters'  => array(
                    array('name' => 'Zend\Filter\StringTrim'),
                ),
            ),
            'email' => array(
                'required' => true,
                'filters'  => array(
```

```
                    array('name' => 'Zend\Filter\StringTrim'),
                ),
                'validators' => array(
                    new Validator\Email(),
                ),
            ),
        );
    }
}
```

Specifications are a great way to make forms, fieldsets, and elements re-usable trivially in your applications. In fact, the `Captcha` and `Csrf` elements define specifications in order to ensure they can work without additional user configuration!

Binding an object

As noted in the intro, forms in Zend Framework bridge the domain model and the view layer. Let's see that in action.

When you `bind()` an object to the form, the following happens:

- The composed `Hydrator` calls `extract()` on the object, and uses the values returned, if any, to populate the `value` attributes of all elements.
- When `isValid()` is called, if `setData()` has not been previously set, the form uses the composed `Hydrator` to extract values from the object, and uses those during validation.
- If `isValid()` is successful (and the `bindOnValidate` flag is enabled, which is true by default), then the `Hydrator` will be passed the validated values to use to hydrate the bound object. (If you do not want this behavior, call `setBindOnValidate(FormInterface::BIND_MANUAL)`).
- If the object implements `Zend\InputFilter\InputFilterAwareInterface`, the input filter it composes will be used instead of the one composed on the form.

This is easier to understand in practice.

```
$contact = new ArrayObject;
$contact['subject'] = '[Contact Form] ';
$contact['message'] = 'Type your message here';

$form    = new Contact\ContactForm;

$form->bind($contact); // form now has default values for
                       // 'subject' and 'message'

$data = array(
    'name'    => 'John Doe',
    'email'   => 'j.doe@example.tld',
    'subject' => '[Contact Form] \'sup?',
);
$form->setData($data);

if ($form->isValid()) {
    // $contact now looks like:
    // array(
    //     'name'    => 'John Doe',
    //     'email'   => 'j.doe@example.tld',
    //     'subject' => '[Contact Form] \'sup?',
    //     'message' => 'Type your message here',
```

```
    // )
    // only as an ArrayObject
}
```

When an object is bound to the form, calling `getData()` will return that object by default. If you want to return an associative array instead, you can pass the `FormInterface::VALUES_AS_ARRAY` flag to the method.

```
use Zend\Form\FormInterface;
$data = $form->getData(FormInterface::VALUES_AS_ARRAY);
```

Zend Framework ships several standard *hydrators*, and implementation is as simple as implementing `Zend\Stdlib\Hydrator\HydratorInterface`, which looks like this:

```
namespace Zend\Stdlib\Hydrator;

interface Hydrator
{
    /** @return array */
    public function extract($object);
    public function hydrate(array $data, $object);
}
```

Rendering

As noted previously, forms are meant to bridge the domain model and view layer. We've discussed the domain model binding, but what about the view?

The form component ships a set of form-specific view helpers. These accept the various form objects, and introspect them in order to generate markup. Typically, they will inspect the attributes, but in special cases, they may look at other properties and composed objects.

When preparing to render, you will likely want to call `prepare()`. This method ensures that certain injections are done, and will likely in the future munge names to allow for `scoped[array][notation]`.

The simplest view helpers available are `Form`, `FormElement`, `FormLabel`, and `FormElementErrors`. Let's use them to display the contact form.

```
<?php
// within a view script
$form = $this->form;
$form->prepare();

// Assuming the "contact/process" route exists...
$form->setAttribute('action', $this->url('contact/process'));

// Set the method attribute for the form
$form->setAttribute('method', 'post');

// Get the form label plugin
$formLabel = $this->plugin('formLabel');

// Render the opening tag
echo $this->form()->openTag($form);
?>
<div class="form_element">
<?php
```

```
    $name = $form->get('name');
    echo $formLabel->openTag() . $name->getAttribute('label');
    echo $this->formInput($name);
    echo $this->formElementErrors($name);
    echo $formLabel->closeTag();
?></div>

<div class="form_element">
<?php
    $subject = $form->get('subject');
    echo $formLabel->openTag() . $subject->getAttribute('label');
    echo $this->formInput($subject);
    echo $this->formElementErrors($subject);
    echo $formLabel->closeTag();
?></div>

<div class="form_element">
<?php
    $message = $form->get('message');
    echo $formLabel->openTag() . $message->getAttribute('label');
    echo $this->formInput($message);
    echo $this->formElementErrors($message);
    echo $formLabel->closeTag();
?></div>

<div class="form_element">
<?php
    $captcha = $form->get('captcha');
    echo $formLabel->openTag() . $captcha->getAttribute('label');
    echo $this->formInput($captcha);
    echo $this->formElementErrors($captcha);
    echo $formLabel->closeTag();
?></div>

<?php echo $this->formElement($form->get('security')) ?>
<?php echo $this->formElement($form->get('send')) ?>

<?php echo $this->form()->closeTag() ?>
```

There are a few things to note about this. First, to prevent confusion in IDEs and editors when syntax highlighting, we use helpers to both open and close the form and label tags. Second, there's a lot of repetition happening here; we could easily create a partial view script or a composite helper to reduce boilerplate. Third, note that not all elements are created equal – the CSRF and submit elements don't need labels or error messages necessarily. Finally, note that the `FormElement` helper tries to do the right thing – it delegates actual markup generation toother view helpers; however, it can only guess what specific form helper to delegate to based on the list it has. If you introduce new form view helpers, you'll need to extend the `FormElement` helper, or create your own.

However, your view files can quickly become long and repetitive to write. While we do not currently provide a single-line form view helper (as this reduces the form customization), the most simplest and recommended way to render your form is by using the `FormRow` view helper. This view helper automatically renders a label (if present), the element itself using the `FormElement` helper, as well as any errors that could arise. Here is the previous form, rewritten to take advantage of this helper :

```
<?php
// within a view script
$form = $this->form;
$form->prepare();
```

```
5
6  // Assuming the "contact/process" route exists...
7  $form->setAttribute('action', $this->url('contact/process'));
8
9  // Set the method attribute for the form
10 $form->setAttribute('method', 'post');
11
12 // Render the opening tag
13 echo $this->form()->openTag($form);
14 ?>
15 <div class="form_element">
16 <?php
17     $name = $form->get('name');
18     echo $this->formRow($name);
19 ?></div>
20
21 <div class="form_element">
22 <?php
23     $subject = $form->get('subject');
24     echo $this->formRow($subject);
25 ?></div>
26
27 <div class="form_element">
28 <?php
29     $message = $form->get('message');
30     echo $this->formRow($message);
31 ?></div>
32
33 <div class="form_element">
34 <?php
35     $captcha = $form->get('captcha');
36     echo $this->formRow($captcha);
37 ?></div>
38
39 <?php echo $this->formElement($form->get('security')) ?>
40 <?php echo $this->formElement($form->get('send')) ?>
41
42 <?php echo $this->form()->closeTag() ?>
```

Note that `FormRow` helper automatically prepends the label. If you want it to be rendered after the element itself, you can pass an optional parameter to the `FormRow` view helper :

```
1 <div class="form_element">
2 <?php
3     $name = $form->get('name');
4     echo $this->formRow($name, **'append'**);
5 ?></div>
```

Validation Groups

Sometimes you want to validate only a subset of form elements. As an example, let's say we're re-using our contact form over a web service; in this case, the `Csrf`, `Captcha`, and submit button elements are not of interest, and shouldn't be validated.

`Zend\Form` provides a proxy method to the underlying `InputFilter`'s `setValidationGroup()` method, allowing us to perform this operation.

```
$form->setValidationGroup('name', 'email', 'subject', 'message');
$form->setData($data);
if ($form->isValid()) {
    // Contains only the "name", "email", "subject", and "message" values
    $data = $form->getData();
}
```

If you later want to reset the form to validate all, simply pass the `FormInterface::VALIDATE_ALL` flag to the `setValidationGroup()` method.

```
use Zend\Form\FormInterface;
$form->setValidationGroup(FormInterface::VALIDATE_ALL);
```

When your form contains nested fieldsets, you can use an array notation to validate only a subset of the fieldsets :

```
$form->setValidationGroup(array(
            'profile' => array(
            'firstname',
            'lastname'
            )
));
$form->setData($data);
if ($form->isValid()) {
    // Contains only the "firstname" and "lastname" values from the
    // "profile" fieldset
    $data = $form->getData();
}
```

Using Annotations

Creating a complete forms solution can often be tedious: you'll create some domain model object, an input filter for validating it, a form object for providing a representation for it, and potentially a hydrator for mapping the form elements and fieldsets to the domain model. Wouldn't it be nice to have a central place to define all of these?

Annotations allow us to solve this problem. You can define the following behaviors with the shipped annotations in `Zend\Form`:

- *AllowEmpty*: mark an input as allowing an empty value. This annotation does not require a value.
- *Attributes*: specify the form, fieldset, or element attributes. This annotation requires an associative array of values, in a JSON object format: *@Attributes({"class":"zend_form","type":"text"})*.
- *ComposedObject*: specify another object with annotations to parse. Typically, this is used if a property references another object, which will then be added to your form as an additional fieldset. Expects a string value indicating the class for the object being composed.
- *ErrorMessage*: specify the error message to return for an element in the case of a failed validation. Expects a string value.
- *Exclude*: mark a property to exclude from the form or fieldset. This annotation does not require a value.
- *Filter*: provide a specification for a filter to use on a given element. Expects an associative array of values, with a "name" key pointing to a string filter name, and an "options" key pointing to an associatve array of filter options for the constructor: *@Filter({"name": "Boolean", "options": {"casting":true}})*. This annotation may be specified multiple times.
- *Flags*: flags to pass to the fieldset or form composing an element or fieldset; these are usually used to specify the name or priority. The annotation expects an associative array: *@Flags({"priority": 100})*.

- *Hydrator*: specify the hydrator class to use for this given form or fieldset. A string value is expected.
- *InputFilter*: specify the input filter class to use for this given form or fieldset. A string value is expected.
- *Input*: specify the input class to use for this given element. A string value is expected.
- *Name*: specify the name of the current element, fieldset, or form. A string value is expected.
- *Options*: options to pass to the fieldset or form that are used to inform behavior – things that are not attributes; e.g. labels, CAPTCHA adapters, etc. The annotation expects an associative array: *@Options({"label": "Username:"})*.
- *Required*: indicate whether an element is required. A boolean value is expected. By default, all elements are required, so this annotation is mainly present to allow disabling a requirement.
- *Type*: indicate the class to use for the current element, fieldset, or form. A string value is expected.
- *Validator*: provide a specification for a validator to use on a given element. Expects an associative array of values, with a "name" key pointing to a string validator name, and an "options" key pointing to an associatve array of validator options for the constructor: *@Validator({"name": "StringLength", "options": {"min":3, "max": 25}})*. This annotation may be specified multiple times.

To use annotations, you simply include them in your class and/or property docblocks. Annotation names will be resolved according to the import statements in your class; as such, you can make them as long or as short as you want depending on what you import.

Here's a simple example.

```
use Zend\Form\Annotation;

/**
 * @Annotation\Name("user")
 * @Annotation\Hydrator("Zend\Stdlib\Hydrator\ObjectProperty")
 */
class User
{
    /**
     * @Annotation\Exclude()
     */
    public $id;

    /**
     * @Annotation\Filter({"name":"StringTrim"})
     * @Annotation\Validator({"name":"StringLength", "options":{"min":1, "max":25}})
     * @Annotation\Validator({"name":"Regex", "options":{"pattern":"/^[a-zA-Z][a-zA-
↪Z0-9_-]{0,24}$/"}})
     * @Annotation\Attributes({"type":"text"})
     * @Annotation\Options({"label":"Username:"})
     */
    public $username;

    /**
     * @Annotation\Type("Zend\Form\Element\Email")
     * @Annotation\Options({"label":"Your email address:"})
     */
    public $email;
}
```

The above will hint to the annotation build to create a form with name "user", which uses the hydrator `Zend\Stdlib\Hydrator\ObjectProperty`. That form will have two elements, "username" and "email". The "username" element will have an associated input that has a `StringTrim` filter, and two validators: a

`StringLength` validator indicating the username is between 1 and 25 characters, and a `Regex` validator asserting it follows a specific accepted pattern. The form element itself will have an attribute "type" with value "text" (a text element), and a label "Username:". The "email" element will be of type `Zend\Form\Element\Email`, and have the label "Your email address:".

To use the above, we need `Zend\Form\Annotation\AnnotationBuilder`:

```
use Zend\Form\Annotation\AnnotationBuilder;

$builder = new AnnotationBuilder();
$form    = $builder->createForm('User');
```

At this point, you have a form with the appropriate hydrator attached, an input filter with the appropriate inputs, and all elements.

Note: You're not done

In all likelihood, you'll need to add some more elements to the form you construct. For example, you'll want a submit button, and likely a CSRF-protection element. We recommend creating a fieldset with common elements such as these that you can then attach to the form you build via annotations.

CHAPTER 56

Form Collections

Often, fieldsets or elements in your forms will correspond to other domain objects. In some cases, they may correspond to collections of domain objects. In this latter case, in terms of user interfaces, you may want to add items dynamically in the user interface – a great example is adding tasks to a task list.

This document is intended to demonstrate these features. To do so, we first need to define some domain objects that we'll be using.

```
namespace Application\Entity;

class Product
{
    /**
     * @var string
     \*/
    protected $name;

    /**
     * @var int
     \*/
    protected $price;

    /**
     * @var Brand
     \*/
    protected $brand;

    /**
     * @var array
     \*/
    protected $categories;

    /**
     * @param string $name
     * @return Product
     \*/
```

```
    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    /**
     * @return string
     \*/
    public function getName()
    {
        return $this->name;
    }

    /**
     * @param int $price
     * @return Product
     \*/
    public function setPrice($price)
    {
        $this->price = $price;
        return $this;
    }

    /**
     * @return int
     \*/
    public function getPrice()
    {
        return $this->price;
    }

    /**
     * @param Brand $brand
     * @return Product
     \*/
    public function setBrand(Brand $brand)
    {
        $this->brand = $brand;
        return $this;
    }

    /**
     * @return Brand
     \*/
    public function getBrand()
    {
        return $this->brand;
    }

    /**
     * @param array $categories
     * @return Product
     \*/
    public function setCategories(array $categories)
    {
        $this->categories = $categories;
        return $this;
```

```
    }

    /**
     * @return array
     \*/
    public function getCategories()
    {
        return $this->categories;
    }
}

class Brand
{
    /**
     * @var string
     \*/
    protected $name;

    /**
     * @var string
     \*/
    protected $url;

    /**
     * @param string $name
     * @return Brand
     \*/
    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    /**
     * @return string
     \*/
    public function getName()
    {
        return $this->name;
    }

    /**
     * @param string $url
     * @return Brand
     \*/
    public function setUrl($url)
    {
        $this->url = $url;
        return $this;
    }

    /**
     * @return string
     \*/
    public function getUrl()
    {
        return $this->url;
    }
```

```
}

class Category
{
    /**
     * @var string
     \*/
    protected $name;

    /**
     * @param string $name
     * @return Category
     \*/
    public function setName($name)
    {
        $this->name = $name;
        return $this;
    }

    /**
     * @return string
     \*/
    public function getName()
    {
        return $this->name;
    }
}
```

As you can see, this is really simple code. A Product has two scalar properties (name and price), a OneToOne relationship (one product has one brand), and a OneToMany relationship (one product has many categories).

Creating Fieldsets

The first step is to create three fieldsets. Each fieldset will contain all the fields and relationships for a specific entity.

Here is the `Brand` fieldset:

```
namespace Application\Form;

use Application\Entity\Brand;
use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class BrandFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function __construct()
    {
        parent::__construct('brand');
        $this->setHydrator(new ClassMethodsHydrator(false))
            ->setObject(new Brand());

        $this->setLabel('Brand');

        $this->add(array(
            'name' => 'name',
```

```
            'options' => array(
                'label' => 'Name of the brand'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));

        $this->add(array(
            'name' => 'url',
            'type' => 'Zend\Form\Element\Url',
            'options' => array(
                'label' => 'Website of the brand'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));
    }

    /**
     * @return array
     \*/
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
            )
        );
    }
}
```

We can discover some new things here. As you can see, the fieldset calls the method `setHydrator()`, giving it a `ClassMethods` hydrator, and the `setObject()` method, giving it an empty instance of a concrete `Brand` object.

When the data will be validated, the `Form` will automatically iterate through all the field sets it contains, and automatically populate the sub-objects, in order to return a complete entity.

Also notice that the `Url` element has a type of `Zend\Form\Element\Url`. This information will be used to validate the input field. You don't need to manually add filters or validators for this input as that element provides a reasonable input specification.

Finally, `getInputSpecification()` gives the specification for the remaining input ("name"), indicating that this input is required. Note that *required* in the array "attributes" (when elements are added) is only meant to add the "required" attribute to the form markup (and therefore has semantic meaning only).

Here is the `Category` fieldset:

```
namespace Application\Form;

use Application\Entity\Category;
use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class CategoryFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function __construct()
```

```
    {
        parent::__construct('category');
        $this->setHydrator(new ClassMethodsHydrator(false))
             ->setObject(new Category());

        $this->setLabel('Category');

        $this->add(array(
            'name' => 'name',
            'options' => array(
                'label' => 'Name of the category'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));
    }

    /**
     * @return array
     \*/
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
            )
        );
    }
}
```

Nothing new here.

And finally the `Product` fieldset:

```
namespace Application\Form;

use Application\Entity\Product;
use Zend\Form\Fieldset;
use Zend\InputFilter\InputFilterProviderInterface;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class ProductFieldset extends Fieldset implements InputFilterProviderInterface
{
    public function __construct()
    {
        parent::__construct('product');
        $this->setHydrator(new ClassMethodsHydrator(false))
             ->setObject(new Product());

        $this->add(array(
            'name' => 'name',
            'options' => array(
                'label' => 'Name of the product'
            ),
            'attributes' => array(
                'required' => 'required'
            )
```

```
        ));

        $this->add(array(
            'name' => 'price',
            'options' => array(
                'label' => 'Price of the product'
            ),
            'attributes' => array(
                'required' => 'required'
            )
        ));

        $this->add(array(
            'type' => 'Application\Form\BrandFieldset',
            'name' => 'brand',
            'options' => array(
                'label' => 'Brand of the product'
            )
        ));

        $this->add(array(
            'type' => 'Zend\Form\Element\Collection',
            'name' => 'categories',
            'options' => array(
                'label' => 'Please choose categories for this product',
                'count' => 2,
                'should_create_template' => true,
                'allow_add' => true,
                'target_element' => array(
                    'type' => 'Application\Form\CategoryFieldset'
                )
            )
        ));
    }

    /**
     * Should return an array specification compatible with
     * {@link Zend\InputFilter\Factory::createInputFilter()}.
     *
     * @return array
     \*/
    public function getInputFilterSpecification()
    {
        return array(
            'name' => array(
                'required' => true,
            ),
            'price' => array(
                'required' => true,
                'validators' => array(
                    array(
                        'name' => 'Float'
                    )
                )
            )
        );
    }
}
```

We have a lot of new things here!

First, notice how the brand element is added: we specify it to be of type `Application\Form\BrandFieldset`. This is how you handle a OneToOne relationship. When the form is validated, the `BrandFieldset` will first be populated, and will return a `Brand` entity (as we have specified a `ClassMethods` hydrator, and bound the fieldset to a `Brand` entity using the `setObject()` method). This `Brand` entity will then be used to populate the `Product` entity by calling the `setBrand()` method.

The next element shows you how to handle OneToMany relationship. The type is `Zend\Form\Element\Collection`, which is a specialized element to handle such cases. As you can see, the name of the element ("categories") perfectly matches the name of the property in the `Product` entity.

This element has a few interesting options:

- `count`: this is how many times the element (in this case a category) has to be rendered. We've set it to two in this examples.
- `should_create_template`: if set to `true`, it will generate a template markup in a `<span>` element, in order to simplify adding new element on the fly (we will speak about this one later).
- `allow_add`: if set to `true` (which is the default), dynamically added elements will be retrieved and validated; otherwise, they will be completely ignored. This, of course, depends on what you want to do.
- `target_element`: this is either an element or, as this is the case in this example, an array that describes the element or fieldset that will be used in the collection. In this case, the `target_element` is a `Category` fieldset.

The Form Element

So far, so good. We now have our field sets in place. But those are field sets, not forms. And only `Form` instances can be validated. So here is the form :

```
namespace Application\Form;

use Zend\Form\Form;
use Zend\InputFilter\InputFilter;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class CreateProduct extends Form
{
    public function __construct()
    {
        parent::__construct('create_product');

        $this->setAttribute('method', 'post')
             ->setHydrator(new ClassMethodsHydrator(false))
             ->setInputFilter(new InputFilter());

        $this->add(array(
            'type' => 'Application\Form\ProductFieldset',
            'options' => array(
                'use_as_base_fieldset' => true
            )
        ));

        $this->add(array(
            'type' => 'Zend\Form\Element\Csrf',
            'name' => 'csrf'
```

```
        ));

        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type' => 'submit'
            )
        ));
    }
}
```

`CreateForm` is quite simple, as it only defines a `Product` fieldset, as well as some other useful fields (`CSRF` for security, and a `Submit` button).

Notice the `use_base_fieldset` option. This option is here to say to the form: "hey, the object I bind to you is, in fact, bound to the fieldset that is the base fieldset." This will be to true most of the times.

What's cool with this approach is that each entity can have its own `Fieldset` and can be reused. You describe the elements, the filters, and validators for each entity only once, and the concrete `Form` instance will only compose those fieldsets. You no longer have to add the "username" input to every form that deals with users!

The Controller

Now, let's create the action in the controller:

```
/**
  * @return array
  \*/
 public function indexAction()
 {
     $form = new CreateProduct();
     $product = new Product();
     $form->bind($product);

     if ($this->request->isPost()) {
         $form->setData($this->request->getPost());

         if ($form->isValid()) {
             var_dump($product);
         }
     }

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

This is super easy. Nothing to do in the controllers. All the magic is done behind the scene.

The View

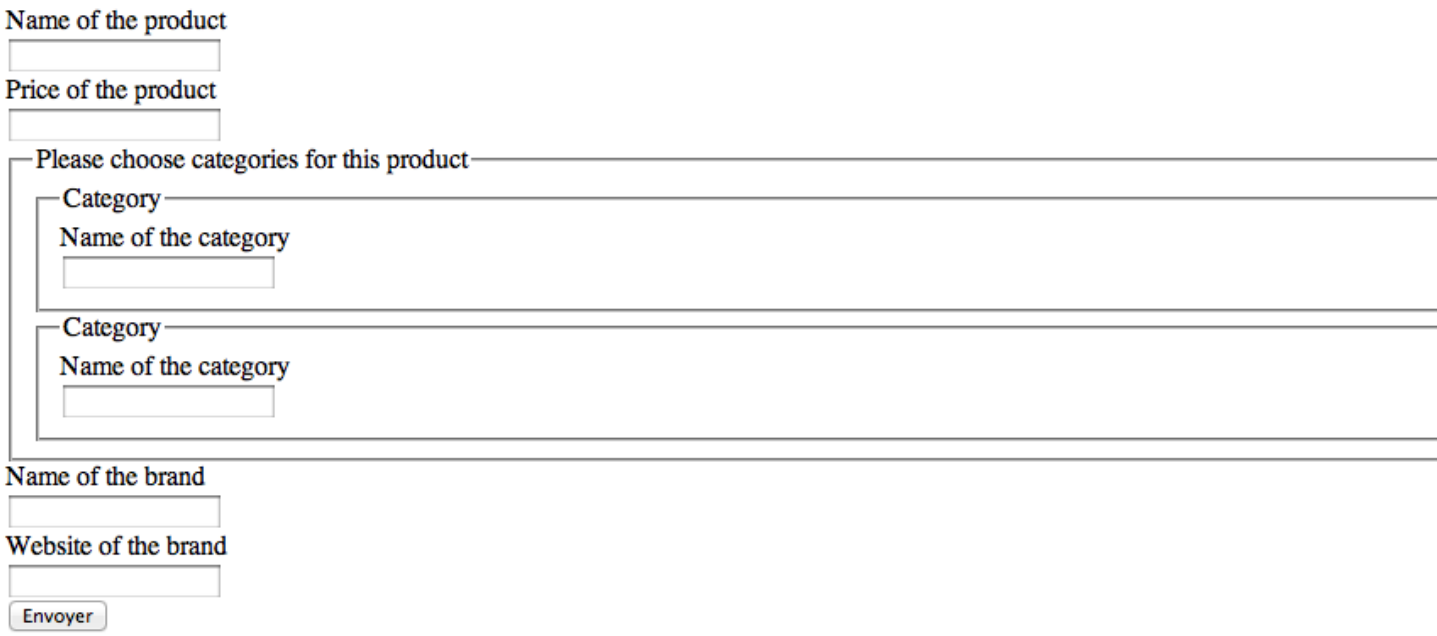
And finally, the view:

```
<?php
$form->setAttribute('action', $this->url('home'))
     ->prepare();

echo $this->form()->openTag($form);

$product = $form->get('product');

echo $this->formRow($product->get('name'));
echo $this->formRow($product->get('price'));
echo $this->formCollection($product->get('categories'));

$brand = $product->get('brand');

echo $this->formRow($brand->get('name'));
echo $this->formRow($brand->get('url'));

echo $this->formHidden($form->get('csrf'));
echo $this->formElement($form->get('submit'));

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

A few new things here :

- the `prepare()` method. You *msut* call it prior to rendering anything in the view (this function is only meant to be called in views, not in controllers).
- the `FormRow` helper renders a label (if present), the input itself, and errors.
- the `FormCollection` helper will iterate through every element in the collection, and render every element with the FormRow helper (you may specify an alternate helper if desired, using the `setElementHelper()` method on that `FormCollection` helper instance). If you need more control about the way you render your forms, you can iterate through the elements in the collection, and render them manually one by one.

Here is the result:



As you can see, collections are wrapped inside a fieldset, and every item in the collection is itself wrapped in the fieldset. In fact, the `Collection` element uses label for each item in the collection, while the label of the `Collection` element itself is used as the legend of the fieldset. If you don't want the fieldset created (just the elements within it), just add a boolean `false` as the second parameter of the the `FormCollection` view helper.

If you validate, all elements will show errors (this is normal, as we've marked them as required). As soon as the form is valid, this is what we get :

```
object(Application\Entity\Product)[622]
  protected 'name' => string 'Chair' (length=5)
  protected 'price' => string '25' (length=2)
  protected 'brand' =>
    object(Application\Entity\Brand)[597]
      protected 'name' => string 'Office Depot' (length=12)
      protected 'url' => string 'http://www.officedepot.fr' (length=25)
  protected 'categories' =>
    array (size=2)
      0 =>
        object(Application\Entity\Category)[615]
          protected 'name' => string 'Armchair' (length=8)
      1 =>
        object(Application\Entity\Category)[621]
          protected 'name' => string 'Office' (length=6)
```

As you can see, the bound object is completely filled, not with arrays, but with objects!

But that's not all.

Adding New Elements Dynamically

Remember the `should_create_template`? We are going to use it now.

Often, forms are not completely static. In our case, let's say that we don't want only two categories, but we want the user to be able to add other ones at runtime. `Zend\Form` has this capability. First, let's see what it generates when we ask it to create a template:

```
▼<fieldset>
   <legend>Please choose categories for this product</legend>
  ▶<fieldset>…</fieldset>
  ▶<fieldset>…</fieldset>
   <span data-template="<fieldset><legend>Category</legend><label>Name of the category<input name="
   product[categories][__index__][name]" required="required" type="text"></label></fieldset>"></span>
 </fieldset>
```

As you can see, the collection generates two fieldsets (the two categories) *plus* a span with a `data-template` attribute that contains the full HTML code to copy to create a new element in the collection. Of course `__index__` (this is the placeholder generated) has to be changed to a valid value. Currently, we have 2 elements (`categories[0]` and `categories[1]`, so `__index__` has to be changed to 2.

If you want, this placeholder (`__index__` is the default) can be changed using the `template_placeholder` option key:

```
$this->add(array(
    'type' => 'Zend\Form\Element\Collection',
    'name' => 'categories',
    'options' => array(
        'label' => 'Please choose categories for this product',
```

```
        'count' => 2,
        'should_create_template' => true,
        'template_placeholder' => '__placeholder__',
        'target_element' => array(
            'type' => 'Application\Form\CategoryFieldset'
        )
    )
));
```

First, let's add a small button "Add new category" anywhere in the form:

```
<button onclick="return add_category()">Add a new category</button>
```

The `add_category` function is fairly simple:

First, count the number of elements we already have. # Get the template from the `span`'s `data-template` attribute. # Change the placeholder to a valid index. # Add the element to the DOM.

Here is the code:

```
<script>
    function add_category() {
        var currentCount = $('form > fieldset > fieldset').length;
        var template = $('form > fieldset > span').data('template');
        template = template.replace('__index__', currentCount);

        $('form > fieldset').append(template);

        return false;
    }
</script>
```

(Note: the above example assumes `$()` is defined, and equivalent to jQuery's `$()` function, Dojo's `dojo.query`, etc.)

One small remark about the `template.replace`: the example uses `currentCount` and not `currentCount + 1`, as the indices are zero-based (so, if we have two elements in the collection, the third one will have the index `2`).

Now, if we validate the form, it will automatically take into account this new element by validating it, filtering it and retrieving it:

```
object(Application\Entity\Product)[622]
  protected 'name' => string 'Zend Framework' (length=14)
  protected 'price' => string '0' (length=1)
  protected 'brand' =>
    object(Application\Entity\Brand)[597]
      protected 'name' => string 'Zend' (length=4)
      protected 'url' => string 'http://www.zend.com' (length=19)
  protected 'categories' =>
    array (size=3)
      0 =>
        object(Application\Entity\Category)[615]
          protected 'name' => string 'Awesome' (length=7)
      1 =>
        object(Application\Entity\Category)[621]
          protected 'name' => string 'PHP' (length=3)
      2 =>
        object(Application\Entity\Category)[628]
          protected 'name' => string 'Framework' (length=9)
```

Of course, if you don't want to allow adding elements in a collection, you must to set the option `allow_add` to "*false*. This way, even if new elements are added, they won't be validated and, hence, not added to the entity. Here is how you do it (and, as we don't want elements to be added, we don't need the data template, either):

```
$this->add(array(
    'type' => 'Zend\Form\Element\Collection',
    'name' => 'categories',
    'options' => array(
        'label' => 'Please choose categories for this product',
        'count' => 2,
        'should_create_template' => false,
        'allow_add' => false,
        'target_element' => array(
            'type' => 'Application\Form\CategoryFieldset'
        )
    )
));
```

There are some limitations of this capability:

- Although you can add new elements and remove them, you *CANNOT* remove more elements in a collection than the initial count (for instance, if your code specifies `count == 2`, you will be able to add a third one and remove it, but you won't be able to remove any others. If the initial count is 2, you *must* have at least two elements.
- Dynamically added elements have to be added at the end of the collection. They can be added anywhere (these elements will still be validated and inserted into the entity), but if the validation fails, this newly added element will be automatically be replaced at the end of the collection.

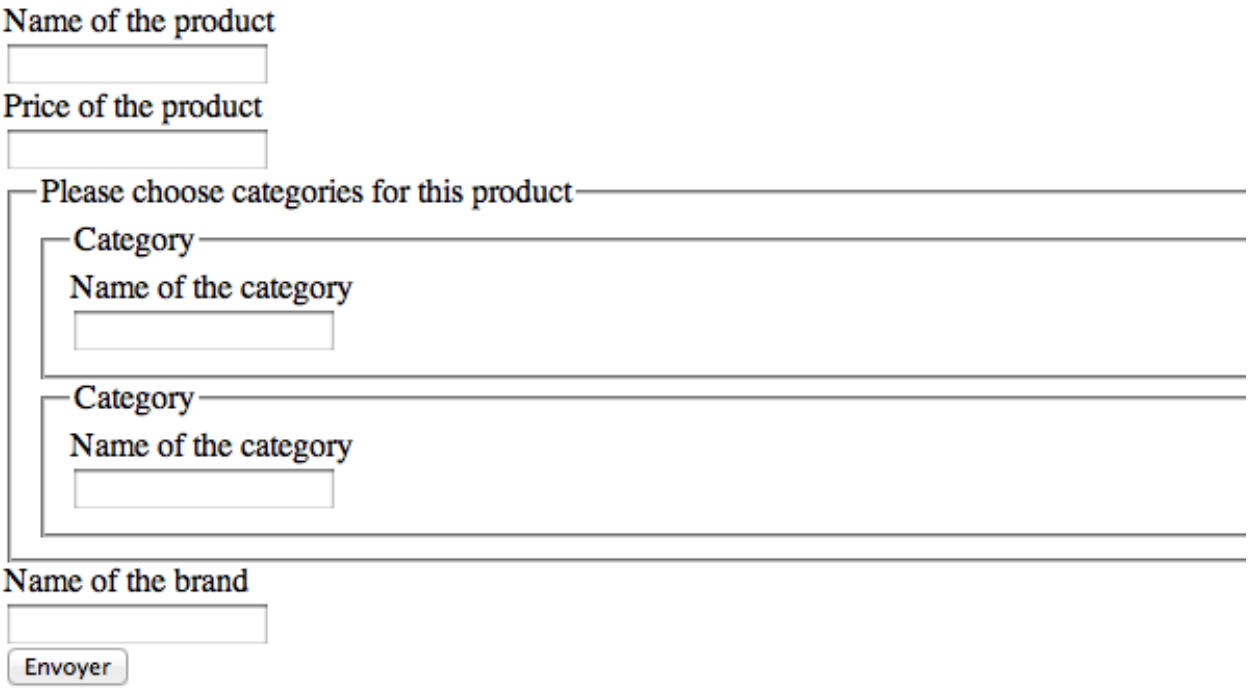
Validation groups for fieldsets and collection

Validation groups allow you to validate a subset of fields.

As an example, although the `Brand` entity has a `URL` property, we don't want to user to specify it in the creation form (but may wish to later in the "Edit Product" form, for instance). Let's update the view to remove the `URL` input:

```
<?php
$form->setAttribute('action', $this->url('home'))
     ->prepare();

echo $this->form()->openTag($form);

$product = $form->get('product');

echo $this->formRow($product->get('name'));
echo $this->formRow($product->get('price'));
echo $this->formCollection($product->get('categories'));

$brand = $product->get('brand');

echo $this->formRow($brand->get('name'));

echo $this->formHidden($form->get('csrf'));
echo $this->formElement($form->get('submit'));

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

This is what we get:



The `URL` input has disappeared, but even if we fill every input, the form won't validate. In fact, this is normal. We specified in the input filter that the `URL` is a *required* field, so if the form does not have it, it won't validate, even though we didn't add it to the view!

Of course, you could create a `BrandFieldsetWithoutURL` fieldset, but of course this is not recommended, as a lot of code will be duplicated.

The solution: validation groups. A validation group is specified in a `Form` object (hence, in our case, in the `CreateProduct` form) by giving an array of all the elements we want to validate. Our `CreateForm` now looks like this:

```
namespace Application\Form;

use Zend\Form\Form;
use Zend\InputFilter\InputFilter;
use Zend\Stdlib\Hydrator\ClassMethods as ClassMethodsHydrator;

class CreateProduct extends Form
{
    public function __construct()
    {
        parent::__construct('create_product');

        $this->setAttribute('method', 'post')
             ->setHydrator(new ClassMethodsHydrator())
             ->setInputFilter(new InputFilter());

        $this->add(array(
            'type' => 'Application\Form\ProductFieldset',
            'options' => array(
                'use_as_base_fieldset' => true
            )
        ));

        $this->add(array(
            'type' => 'Zend\Form\Element\Csrf',
            'name' => 'csrf'
        ));

        $this->add(array(
            'name' => 'submit',
            'attributes' => array(
                'type' => 'submit'
            )
        ));

        $this->setValidationGroup(array(
            'csrf',
            'product' => array(
                'name',
                'price',
                'brand' => array(
                    'name'
                ),
                'categories' => array(
                    'name'
                )
            )
        ));
    }
}
```

Of course, don't forget to add the `CSRF` element, as we want it to be validated too (but notice that I didn't write the submit element, as we don't care about it). You can recursively select the elements you want.

There is one simple limitation currently: validation groups for collections are set on a per-collection basis, not element in a collection basis. This means you cannot say, "validate the name input for the first element of the categories collection, but don't validate it for the second one." But, honestly, this is really an edge-case.

Now, the form validates (and the `URL` is set to null as we didn't specify it).

CHAPTER 57

Form Elements

Introduction

A set of specialized elements are provided for accomplishing application-centric tasks. These include several HTML5 input elements with matching server-side validators, the `Csrf` element (to prevent Cross Site Request Forgery attacks), and the `Captcha` element (to display and validate *CAPTCHAs*).

A `Factory` is provided to facilitate creation of elements, fieldsets, forms, and the related input filter. See the *Zend\Form Quick Start* for more information.

Element Base Class

`Zend\Form\Element` is a base class for all specialized elements and Zend\Form\Fieldset, but can also be used for all generic `text`, `select`, `radio`, etc. type form inputs which do not have a specialized element available.

Basic Usage

At the bare minimum, each element or fieldset requires a name. You will also typically provide some attributes to hint to the view layer how it might render the item.

```
use Zend\Form\Element;
use Zend\Form\Form;

$username = new Element('username');
$username
    ->setLabel('Username');
    ->setAttributes(array(
        'type'  => 'text',
        'class' => 'username',
        'size'  => '30',
    ));
```

```
12
13 $password = new Element('password');
14 $password
15     ->setLabel('Password');
16     ->setAttributes(array(
17         'type'  => 'password',
18         'size'  => '30',
19     ));
20
21 $form = new Form('my-form');
22 $form
23     ->add($username)
24     ->add($password);
```

Public Methods

setName(*string $name*)
Set the name for this element.

Returns `Zend\Form\Element`

getName()
Return the name for this element.

Returns string

setLabel(*string $label*)
Set the label content for this element.

Returns `Zend\Form\Element`

getLabel()
Return the label content for this element.

Returns string

setLabelAttributes(*array $labelAttributes*)
Set the attributes to use with the label.

Returns `Zend\Form\Element`

getLabelAttributes()
Return the attributes to use with the label.

Returns array

setOptions(*array $options*)
Set options for an element. Accepted options are: `"label"` and `"label_attributes"`, which call `setLabel` and `setLabelAttributes`, respectively.

Returns `Zend\Form\Element`

setAttribute(*string $key*, *mixed $value*)
Set a single element attribute.

Returns `Zend\Form\Element`

getAttribute(*string $key*)
Retrieve a single element attribute.

Returns mixed

hasAttribute(*string $key*)
Check if a specific attribute exists for this element.

Returns boolean

setAttributes(*array|Traversable $arrayOrTraversable*)
Set many attributes at once. Implementation will decide if this will overwrite or merge.

Returns `Zend\Form\Element`

getAttributes()
Retrieve all attributes at once.

Returns array|Traversable

clearAttributes()
Clear all attributes for this element.

Returns `Zend\Form\Element`

setMessages(*array|Traversable $messages*)
Set a list of messages to report when validation fails.

Returns `Zend\Form\Element`

getMessages()
Returns a list of validation failure messages, if any.

Returns array|Traversable

Captcha Element

`Zend\Form\Element\Captcha` can be used with forms where authenticated users are not necessary, but you want to prevent spam submissions. It is pairs with one of the `Zend/Form/View/Helper/Captcha/*` view helpers that matches the type of *CAPTCHA* adapter in use.

Basic Usage

A *CAPTCHA* adapter must be attached in order for validation to be included in the element's input filter specification. See the section on *Zend CAPTCHA Adapters* for more information on what adapters are available.

```
use Zend\Captcha;
use Zend\Form\Element;
use Zend\Form\Form;

$captcha = new Element\Captcha('captcha');
$captcha
    ->setCaptcha(new Captcha\Dumb())
    ->setLabel('Please verify you are human');

$form = new Form('my-form');
$form->add($captcha);
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

setCaptcha(*array|Zend\Captcha\AdapterInterface $captcha*)
Set the *CAPTCHA* adapter for this element. If `$captcha` is an array, `Zend\Captcha\Factory::factory()` will be run to create the adapter from the array configuration.

Returns `Zend\Form\Element\Captcha`

getCaptcha()
Return the *CAPTCHA* adapter for this element.

Returns `Zend\Captcha\AdapterInterface`

getInputSpecification()
Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter, and a *CAPTCHA* validator.

Returns array

Checkbox Element

`Zend\Form\Element\Checkbox` is meant to be paired with the `Zend/Form/View/Helper/FormCheckbox` for HTML inputs with type checkbox. This element adds an `InArray` validator to its input filter specification in order to validate on the server if the checkbox contains either the checked value or the unchecked value.

Basic Usage

This element automatically adds a `"type"` attribute of value `"checkbox"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$checkbox = new Element\Checkbox('checkbox');
$checkbox->setLabel('A checkbox');
$checkbox->setUseHiddenElement(true);
$checkbox->setCheckedValue("good");
$checkbox->setUncheckedValue("bad");

$form = new Form('my-form');
$form->add($checkbox);
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element* .

setOptions(*array $options*)
Set options for an element of type Checkbox. Accepted options, in addition to the inherited options of Zend\Form\Element <zend.form.element.methods.set-options>‘ , are: `"use_hidden_element"`, `"checked_value"` and `"unchecked_value"` , which call `setUseHiddenElement`, `setCheckedValue` and `setUncheckedValue` , respectively.

setUseHiddenElement(*boolean $useHiddenElement*)
If set to true (which is default), the view helper will generate a hidden element that contains the unchecked value. Therefore, when using custom unchecked value, this option have to be set to true.

useHiddenElement()
Return if a hidden element is generated.

Return type boolean

setCheckedValue(*string $checkedValue*)
Set the value to use when the checkbox is checked.

getCheckedValue()
Return the value used when the checkbox is checked.

Return type string

setUncheckedValue(*string $uncheckedValue*)
Set the value to use when the checkbox is unchecked. For this to work, you must make sure that use_hidden_element is set to true.

getUncheckedValue()
Return the value used when the checkbox is unchecked.

Return type string

getInputSpecification()
Returns a input filter specification, which includes a `Zend\Validator\InArray` to validate if the value is either checked value or unchecked value.

Return type array

Collection Element

Sometimes, you may want to add input (or a set of inputs) multiple times, either because you don't want to duplicate code, or because you does not know in advance how many elements you need (in the case of elements dynamically added to a form using JavaScript, for instance).

`Zend\Form\Element\Collection` is meant to be paired with the `Zend\Form\View\Helper\FormCollection`.

Basic Usage

```
use Zend\Form\Element;
use Zend\Form\Form;

$colors = new Element\Collection('collection');
$colors->setLabel('Colors');
$colors->setCount(2);
$colors->setTargetElement(new Element\Color());
$colors->setShouldCreateTemplate(true);

$form = new Form('my-form');
$form->add($colors);
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element* .

setOptions(*array $options*)
Set options for an element of type Collection. Accepted options, in addition to the inherited options of Zend\Form\Element <zend.form.element.methods.set-options>‘ , are: `"target_element"`, `"count"`, `"allow_add"`, `"should_create_template"` and `"template_placeholder"` , which call `setTargetElement`, `setCount`, `setAllowAdd`, `setShouldCreateTemplate` and `setTemplatePlaceholder` , respectively.

setCount(*$count*)
Defines how many times the target element will be rendered by the `Zend/Form/View/Helper/FormCollection` view helper.

getCount()
Return the number of times the target element will be initially rendered by the `Zend/Form/View/Helper/FormCollection` view helper.

setTargetElement(*$elementOrFieldset*)
This function either takes an `Zend/Form/ElementInterface`, `Zend/Form/FieldsetInterface` instance or an array to pass to the form factory. When the Collection element will be validated, the input filter will be retrieved from this target element and be used to validate each element in the collection.

getTargetElement()
Return the target element used by the collection.

setAllowAdd(*$allowAdd*)
If allowAdd is set to true (which is the default), new elements added dynamically in the form (using JavaScript, for instance) will also be validated and retrieved.

getAllowAdd()
Return if new elements can by dynamically added in the collection.

setShouldCreateTemplate(*$shouldCreateTemplate*)
If shouldCreateTemplate is set to true (defaults to false), a element will be generated by the `Zend/Form/View/Helper/FormCollection` view helper. This non-semantical span element contains a single data-template HTML5 attribute whose value is the whole HTML to copy to create a new element in the form. The template is indexed using the `templatePlaceholder` value.

getAllowAdd()
Return if a template should be created.

setTemplatePlaceholder(*$templatePlaceholder*)
Set the template placeholder (defaults to __index__) used to index element in the template.

getTemplatePlaceholder()
Returns the template placeholder used to index element in the template.

Color Element

`Zend\Form\Element\Color` is meant to be paired with the `Zend/Form/View/Helper/FormColor` for HTML5 inputs with type color. This element adds filters and a `Regex` validator to it's input filter specification in order to validate a HTML5 valid simple color value on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"color"`.

```php
use Zend\Form\Element;
use Zend\Form\Form;

$color = new Element\Color('color');
$color->setLabel('Background color');

$form = new Form('my-form');
$form->add($color);
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()
: Returns a input filter specification, which includes `Zend\Filter\StringTrim` and `Zend\Filter\StringToLower` filters, and a `Zend\Validator\Regex` to validate the RGB hex format.

  Returns array

Csrf Element

`Zend\Form\Element\Csrf` pairs with the `Zend/Form/View/Helper/FormHidden` to provide protection from *CSRF* attacks on forms, ensuring the data is submitted by the user session that generated the form and not by a rogue script. Protection is achieved by adding a hash element to a form and verifying it when the form is submitted.

Basic Usage

This element automatically adds a `"type"` attribute of value `"hidden"`.

```php
use Zend\Form\Element;
use Zend\Form\Form;

$csrf = new Element\Csrf('csrf');

$form = new Form('my-form');
$form->add($csrf);
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()
: Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter and a `Zend\Validator\Csrf` to validate the *CSRF* value.

  Returns array

Date Element

`Zend\Form\Element\Date` is meant to be paired with the `Zend/Form/View/Helper/FormDate` for HTML5 inputs with type date. This element adds filters and validators to it's input filter specification in order to validate HTML5 date input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"date"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$date = new Element\Date('appointment-date');
$date
    ->setLabel('Appointment Date')
    ->setAttributes(array(
        'min'  => '2012-01-01',
        'max'  => '2020-01-01',
        'step' => '1', // days; default step interval is 1 day
    ));

$form = new Form('my-form');
$form->add($date);
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

`getInputSpecification``()`

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

One difference from `Zend\Form\Element\DateTime` is that the `Zend\Validator\DateStep` validator will expect the `step` attribute to use an interval of days (default is 1 day).

Returns array

DateTime Element

`Zend\Form\Element\DateTime` is meant to be paired with the `Zend/Form/View/Helper/FormDateTime` for HTML5 inputs with type datetime. This element adds filters and validators to it's input filter specification in order to validate HTML5 datetime input values on the server.

Basic Usage

This element automatically adds a "type" attribute of value "datetime".

```
use Zend\Form\Element;
use Zend\Form\Form;

$dateTime = new Element\DateTime('appointment-date-time');
$dateTime
    ->setLabel('Appointment Date/Time')
    ->setAttributes(array(
        'min'  => '2010-01-01T00:00:00Z',
        'max'  => '2020-01-01T00:00:00Z',
        'step' => '1', // minutes; default step interval is 1 min
    ));

$form = new Form('my-form');
$form->add($dateTime);
```

Note: Note: the min, max, and step attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()

Returns a input filter specification, which includes Zend\Filter\StringTrim and will add the appropriate validators based on the values from the min, max, and step attributes.

If the min attribute is set, a Zend\Validator\GreaterThan validator will be added to ensure the date value is greater than the minimum value.

If the max attribute is set, a Zend\Validator\LessThanValidator validator will be added to ensure the date value is less than the maximum value.

If the step attribute is set to "any", step validations will be skipped. Otherwise, a a Zend\Validator\DateStep validator will be added to ensure the date value is within a certain interval of minutes (default is 1 minute).

Returns array

DateTimeLocal Element

Zend\Form\Element\DateTimeLocal is meant to be paired with the Zend/Form/View/Helper/FormDateTimeLocal for HTML5 inputs with type datetime-local. This element adds filters and validators to it's input filter specification in order to validate HTML5 a local datetime input values on the server.

Basic Usage

This element automatically adds a "type" attribute of value "datetime-local".

```php
use Zend\Form\Element;
use Zend\Form\Form;

$dateTimeLocal = new Element\DateTimeLocal('appointment-date-time');
$dateTimeLocal
    ->setLabel('Appointment Date')
    ->setAttributes(array(
        'min'  => '2010-01-01T00:00:00',
        'max'  => '2020-01-01T00:00:00',
        'step' => '1', // minutes; default step interval is 1 min
    ));

$form = new Form('my-form');
$form->add($dateTimeLocal);
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

getInputSpecification()
: Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

  Returns array

Email Element

`Zend\Form\Element\Email` is meant to be paired with the `Zend/Form/View/Helper/FormEmail` for HTML5 inputs with type email. This element adds filters and validators to it's input filter specification in order to validate HTML5 valid email address on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"email"`.

```php
use Zend\Form\Element;
use Zend\Form\Form;

$form = new Form('my-form');

// Single email address
$email = new Element\Email('email');
$email->setLabel('Email Address')
$form->add($email);

// Comma separated list of emails
$emails = new Element\Email('emails');
```

```php
$emails
    ->setLabel('Email Addresses')
    ->setAttribute('multiple', true);
$form->add($emails);
```

Note: Note: the `multiple` attribute should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

`getInputSpecification``()`
Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter, and a validator based on the `multiple` attribute.

If the `multiple` attribute is unset or false, a `Zend\Validator\Regex` validator will be added to validate a single email address.

If the `multiple` attribute is true, a `Zend\Validator\Explode` validator will be added to ensure the input string value is split by commas before validating each email address with `Zend\Validator\Regex`.

Returns array

Hidden Element

`Zend\Form\Element\Hidden` represents a hidden form input. It can be used with the `Zend/Form/View/Helper/FormHidden` view helper.

`Zend\Form\Element\Hidden` extends from *Zend\Form\Element*.

Basic Usage

This element automatically adds a `"type"` attribute of value `"hidden"`.

```php
use Zend\Form\Element;
use Zend\Form\Form;

$hidden = new Element\Hidden('my-hidden');
$hidden->setValue('foo');

$form = new Form('my-form');
$form->add($hidden);
```

Month Element

`Zend\Form\Element\Month` is meant to be paired with the `Zend/Form/View/Helper/FormMonth` for HTML5 inputs with type month. This element adds filters and validators to it's input filter specification in order to validate HTML5 month input values on the server.

Basic Usage

This element automatically adds a "type" attribute of value "month".

```
use Zend\Form\Element;
use Zend\Form\Form;

$month = new Element\Month('month');
$month
    ->setLabel('Month')
    ->setAttributes(array(
        'min'  => '2012-01',
        'max'  => '2020-01',
        'step' => '1', // months; default step interval is 1 month
    ));

$form = new Form('my-form');
$form->add($month);
```

Note: Note: the min, max, and step attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

getInputSpecification()

Returns a input filter specification, which includes Zend\Filter\StringTrim and will add the appropriate validators based on the values from the min, max, and step attributes. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

One difference from Zend\Form\Element\DateTime is that the Zend\Validator\DateStep validator will expect the step attribute to use an interval of months (default is 1 month).

Returns array

Number Element

Zend\Form\Element\Number is meant to be paired with the Zend/Form/View/Helper/FormNumber for HTML5 inputs with type number. This element adds filters and validators to it's input filter specification in order to validate HTML5 number input values on the server.

Basic Usage

This element automatically adds a "type" attribute of value "number".

```
use Zend\Form\Element;
use Zend\Form\Form;

$number = new Element\Number('quantity');
$number
    ->setLabel('Quantity')
```

```
    ->setAttributes(array(
        'min'  => '0',
        'max'  => '10',
        'step' => '1', // default step interval is 1
    ));

$form = new Form('my-form');
$form->add($number);
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

`getInputSpecification``()`
Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes.

If the `min` attribute is set, a `Zend\Validator\GreaterThan` validator will be added to ensure the number value is greater than the minimum value. The `min` value should be a valid floating point number.

If the `max` attribute is set, a `Zend\Validator\LessThanValidator` validator will be added to ensure the number value is less than the maximum value. The `max` value should be a valid floating point number.

If the `step` attribute is set to "any", step validations will be skipped. Otherwise, a a `Zend\Validator\Step` validator will be added to ensure the number value is within a certain interval (default is 1). The `step` value should be either "any" or a valid floating point number.

Returns array

Range Element

`Zend\Form\Element\Range` is meant to be paired with the `Zend/Form/View/Helper/FormRange` for HTML5 inputs with type range. This element adds filters and validators to it's input filter specification in order to validate HTML5 range values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"range"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$range = new Element\Range('range');
$range
    ->setLabel('Minimum and Maximum Amount')
    ->setAttributes(array(
        'min'  => '0',   // default minimum is 0
        'max'  => '100', // default maximum is 100
        'step' => '1',   // default interval is 1
```

```
    ));

$form = new Form('my-form');
$form->add($range);
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\Number*.

`getInputSpecification()```
Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\Number* for more information.

The `Range` element differs from `Zend\Form\Element\Number` in that the `Zend\Validator\GreaterThan` and `Zend\Validator\LessThan` validators will always be present. The default minimum is 1, and the default maximum is 100.

Returns array

Time Element

`Zend\Form\Element\Time` is meant to be paired with the `Zend/Form/View/Helper/FormTime` for HTML5 inputs with type time. This element adds filters and validators to it's input filter specification in order to validate HTML5 time input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"time"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$time = new Element\Month('time');
$time
    ->setLabel('Time')
    ->setAttributes(array(
        'min'  => '00:00:00',
        'max'  => '23:59:59',
        'step' => '60', // seconds; default step interval is 60 seconds
    ));

$form = new Form('my-form');
$form->add($time);
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

getInputSpecification()
: Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

  One difference from `Zend\Form\Element\DateTime` is that the `Zend\Validator\DateStep` validator will expect the `step` attribute to use an interval of seconds (default is 60 seconds).

  Returns array

Url Element

`Zend\Form\Element\Url` is meant to be paired with the `Zend/Form/View/Helper/FormUrl` for HTML5 inputs with type url. This element adds filters and a `Zend\Validator\Uri` validator to it's input filter specification for validating HTML5 URL input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"url"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$url = new Element\Url('webpage-url');
$url->setLabel('Webpage URL');

$form = new Form('my-form');
$form->add($url);
```

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element*.

getInputSpecification()
: Returns a input filter specification, which includes a `Zend\Filter\StringTrim` filter, and a `Zend\Validator\Uri` to validate the URI string.

  Returns array

Week Element

`Zend\Form\Element\Week` is meant to be paired with the `Zend/Form/View/Helper/FormWeek` for HTML5 inputs with type week. This element adds filters and validators to it's input filter specification in order to validate HTML5 week input values on the server.

Basic Usage

This element automatically adds a `"type"` attribute of value `"week"`.

```
use Zend\Form\Element;
use Zend\Form\Form;

$week = new Element\Week('week');
$week
    ->setLabel('Week')
    ->setAttributes(array(
        'min'  => '2012-W01',
        'max'  => '2020-W01',
        'step' => '1', // weeks; default step interval is 1 week
    ));

$form = new Form('my-form');
$form->add($week);
```

Note: Note: the `min`, `max`, and `step` attributes should be set prior to calling Zend\Form::prepare(). Otherwise, the default input specification for the element may not contain the correct validation rules.

Public Methods

The following methods are in addition to the inherited *methods of Zend\Form\Element\DateTime*.

`getInputSpecification``()`

Returns a input filter specification, which includes `Zend\Filter\StringTrim` and will add the appropriate validators based on the values from the `min`, `max`, and `step` attributes. See *getInputSpecification in Zend\Form\Element\DateTime* for more information.

One difference from `Zend\Form\Element\DateTime` is that the `Zend\Validator\DateStep` validator will expect the `step` attribute to use an interval of weeks (default is 1 week).

Returns array

CHAPTER 58

Form View Helpers

Introduction

Zend Framework comes with an initial set of helper classes related to Forms: e.g., rendering a text input, selection box, or form labels. You can use helper, or plugin, classes to perform these behaviors for you.

See the section on *view helpers* for more information.

Standard Helpers

Form

The `Form` view helper is used to render a `<form>` HTML element and its attributes.

Basic usage:

```
use Zend\Form\Form;
use Zend\Form\Element;

// Within your view...

$form = new Form();
// ...add elements and input filter to form...

// Set attributes
$form->setAttribute('action', $this->url('contact/process'));
$form->setAttribute('method', 'post');

// Prepare the form elements
$form->prepare();

// Render the opening tag
echo $this->form()->openTag($form);
```

```
// <form action="/contact/process" method="post">

// ...render the form elements...

// Render the closing tag
echo $this->form()->closeTag();
// </form>
```

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

openTag(*FormInterface $form = null*)
Renders the `<form>` open tag for the `$form` instance.

Return type string

closeTag()
Renders a `</form>` closing tag.

Return type string

FormButton

The `FormButton` view helper is used to render a `<button>` HTML element and its attributes.

Basic usage:

```
use Zend\Form\Element;

$element = new Element\Button('my-button');
$element->setLabel("Reset");

// Within your view...

/**
 * Example #1: Render entire button in one shot...
 */
echo $this->formButton($element);
// <button name="my-button" type="button">Reset</button>

/**
 * Example #2: Render button in 3 steps
 */
// Render the opening tag
echo $this->formButton()->openTag($element);
// <button name="my-button" type="button">

echo '<span class="inner">' . $element->getLabel() . '</span>';

// Render the closing tag
echo $this->formButton()->closeTag();
// </button>

/**
 * Example #3: Override the element label
 */
echo $this->formButton()->render($element, 'My Content');
// <button name="my-button" type="button">My Content</button>
```

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\FormInput*.

openTag(*$element = null*)
Renders the `<button>` open tag for the `$element` instance.

Return type string

closeTag()
Renders a `</button>` closing tag.

Return type string

render(*ElementInterface $element*[, *$buttonContent = null*])
Renders a button's opening tag, inner content, and closing tag.

Parameters

- **$element** – The button element.
- **$buttonContent** – (optional) The inner content to render. If `null`, will default to the `$element`'s label.

Return type string

FormCheckbox

The `FormCheckbox` view helper can be used to render a `<input type="checkbox">` HTML form input. It is meant to work with the *Zend\Form\Element\Checkbox* element, which provides a default input specification for validating the checkbox values.

`FormCheckbox` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Checkbox('my-checkbox');

// Within your view...

/**
 * Example #1: Default options
 */
echo $this->formCheckbox($element);
// <input type="hidden" name="my-checkbox" value="0">
// <input type="checkbox" name="my-checkbox" value="1">

/**
 * Example #2: Disable hidden element
 */
$element->setUseHiddenElement(false);
echo $this->formCheckbox($element);
// <input type="checkbox" name="my-checkbox" value="1">

/**
 * Example #3: Change checked/unchecked values
 */
$element->setUseHiddenElement(true)
        ->setUncheckedValue('no')
        ->setCheckedValue('yes');
echo $this->formCheckbox($element);
```

```
// <input type="hidden" name="my-checkbox" value="no">
// <input type="checkbox" name="my-checkbox" value="yes">
```

FormElement

The `FormElement` view helper proxies the rendering to specific form view helpers depending on the type of the `Zend\\Form\\Element` that is passed in. For instance, if the passed in element had a type of "text", the `FormElement` helper will retrieve and use the `FormText` helper to render the element.

Basic usage:

```
use Zend\Form\Form;
use Zend\Form\Element;

// Within your view...

/**
 * Example #1: Render different types of form elements
 */
$textElement     = new Element\Text('my-text');
$checkboxElement = new Element\Checkbox('my-checkbox');

echo $this->formElement($textElement);
// <input type="text" name="my-text" value="">

echo $this->formElement($checkboxElement);
// <input type="hidden" name="my-checkbox" value="0">
// <input type="checkbox" name="my-checkbox" value="1">

/**
 * Example #2: Loop through form elements and render them
 */
$form = new Form();
// ...add elements and input filter to form...
$form->prepare();

// Render the opening tag
echo $this->form()->openTag($form);

// ...loop through and render the form elements...
foreach ($form as $element) {
    echo $this->formElement($element);       // <-- Magic!
    echo $this->formElementErrors($element);
}

// Render the closing tag
echo $this->form()->closeTag();
```

FormElementErrors

The `FormElementErrors` view helper is used to render the validation error messages of an element.

Basic usage:

```php
use Zend\Form\Form;
use Zend\Form\Element;
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Input;

// Create a form
$form    = new Form();
$element = new Element\Text('my-text');
$form->add($element);

// Create a input
$input = new Input('my-text');
$input->setRequired(true);

$inputFilter = new InputFilter();
$inputFilter->add($input);
$form->setInputFilter($inputFilter);

// Force a failure
$form->setData(array()); // Empty data
$form->isValid();        // Not valid

// Within your view...

/**
 * Example #1: Default options
 */
echo $this->formElementErrors($element);
// <ul><li>Value is required and can&#039;t be empty</li></ul>

/**
 * Example #2: Add attributes to open format
 */
echo $this->formElementErrors($element, array('class' => 'help-inline'));
// <ul class="help-inline"><li>Value is required and can&#039;t be empty</li></ul>

/**
 * Example #3: Custom format
 */
echo $this->formElementErrors()
                ->setMessageOpenFormat('<div class="help-inline">')
                ->setMessageSeparatorString('</div><div class="help-inline">')
                ->setMessageCloseString('</div>')
                ->render($element);
// <div class="help-inline">Value is required and can&#039;t be empty</div>
```

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

setMessageOpenFormat (*string $messageOpenFormat*)
Set the formatted string used to open message representation.

Parameters **$messageOpenFormat** – The formatted string to use to open the messages. Uses `'<ul%s><li>'` by default. Attributes are inserted here.

getMessageOpenFormat ()
Returns the formatted string used to open message representation.

Return type string

setMessageSeparatorString (*string $messageSeparatorString*)

Sets the string used to separate messages.

Parameters **$messageSeparatorString** – The string to use to separate the messages. Uses `'</li><li>'` by default.

getMessageSeparatorString()
Returns the string used to separate messages.

Return type string

setMessageCloseString(*string $messageCloseString*)
Sets the string used to close message representation.

Parameters **$messageCloseString** – The string to use to close the messages. Uses `'</li></ul>'` by default.

getMessageCloseString()
Returns the string used to close message representation.

Return type string

setAttributes(*array $attributes*)
Set the attributes that will go on the message open format.

Parameters **$attributes** – Key value pairs of attributes.

getAttributes()
Returns the attributes that will go on the message open format.

Return type array

render(*ElementInterface $element*[, *array $attributes = array()*])
Renders validation errors for the provided `$element`.

Parameters

- **$element** – The element.
- **$attributes** – Additional attributes that will go on the message open format. These are merged with those set via `setAttributes()`.

Return type string

FormHidden

The `FormHidden` view helper can be used to render a `<input type="hidden">` HTML form input. It is meant to work with the *Zend\Form\Element\Hidden* element.

`FormHidden` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Hidden('my-hidden');
$element->setValue('foo');

// Within your view...

echo $this->formHidden($element);
// <input type="hidden" name="my-hidden" value="foo">
```

FormImage

The `FormImage` view helper can be used to render a `<input type="image">` HTML form input. It is meant to work with the Zend\Form\Element\Image element.

`FormImage` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Image('my-image');
$element->setAttribute('src', '/img/my-pic.png');

// Within your view...

echo $this->formImage($element);
// <input type="image" name="my-image" src="/img/my-pic.png">
```

FormInput

The `FormInput` view helper is used to render a `<input>` HTML form input tag. It acts as a base class for all of the specifically typed form input helpers (FormText, FormCheckbox, FormSubmit, etc.), and is not suggested for direct use.

It contains a general map of valid tag attributes and types for attribute filtering. Each subclass of `FormInput` implements it's own specific map of valid tag attributes. The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

render(*ElementInterface $element*)
Renders the `<input>` tag for the `$element`.

Return type string

FormLabel

The `FormLabel` view helper is used to render a `<label>` HTML element and its attributes. If you have a `Zend\\I18n\\Translator\\Translator` attached, `FormLabel` will translate the label contents during it's rendering.

Basic usage:

```
use Zend\Form\Element;

$element = new Element\Text('my-text');
$element->setLabel('Label')
        ->setAttribute('id', 'text-id')
        ->setLabelAttributes(array('class' => 'control-label'));

// Within your view...

/**
 * Example #1: Render label in one shot
 */
echo $this->formLabel($element);
// <label class="control-label" for="text-id">Label</label>

echo $this->formLabel($element, $this->formText($element));
```

```php
// <label class="control-label" for="text-id">Label<input type="text" name="my-text">
↪</label>

echo $this->formLabel($element, $this->formText($element), 'append');
// <label class="control-label" for="text-id"><input type="text" name="my-text">Label
↪</label>

/**
 * Example #2: Render label in separate steps
 */
// Render the opening tag
echo $this->formLabel()->openTag($element);
// <label class="control-label" for="text-id">

// Render the closing tag
echo $this->formLabel()->closeTag();
// </label>
```

Attaching a translator and setting a text domain:

```php
// Setting a translator
$this->formLabel()->setTranslator($translator);

// Setting a text domain
$this->formLabel()->setTranslatorTextDomain('my-text-domain');

// Setting both
$this->formLabel()->setTranslator($translator, 'my-text-domain');
```

Note: Note: If you have a translator in the Service Manager under the key, 'translator', the view helper plugin manager will automatically attach the translator to the FormLabel view helper. See `Zend\\View\\HelperPluginManager::injectTranslator()` for more information.

The following public methods are in addition to those inherited from *Zend\Form\View\Helper\AbstractHelper*.

__invoke(*ElementInterface $element = null*, *string $labelContent = null*, *string $position = null*)
Render a form label, optionally with content.

Always generates a "for" statement, as we cannot assume the form input will be provided in the `$labelContent`.

Parameters

- **$element** – A form element.
- **$labelContent** – If null, will attempt to use the element's label value.
- **$position** – Append or prepend the element's label value to the `$labelContent`. One of `FormLabel::APPEND` or `FormLabel::PREPEND` (default)

Return type string

openTag(*array|ElementInterface $attributesOrElement = null*)
Renders the `<label>` open tag and attributes.

Parameters **$attributesOrElement** – An array of key value attributes or a `ElementInterface` instance.

Return type string

closeTag()
Renders a `</label>` closing tag.

Return type string

AbstractHelper

The `AbstractHelper` is used as a base abstract class for Form view helpers, providing methods for validating form HTML attributes, as well as controlling the doctype and character encoding. `AbstractHelper` also extends from `Zend\I18n\View\Helper\AbstractTranslatorHelper` which provides an implementation for the `Zend\I18n\Translator\TranslatorAwareInterface` that allows setting a translator and text domain. The following public methods are in addition to the inherited *methods of Zend\I18n\View\Helper\AbstractTranslatorHelper*.

setDoctype(*string $doctype*)
Sets a doctype to use in the helper.

getDoctype()
Returns the doctype used in the helper.

Return type string

setEncoding(*string $encoding*)
Set the translation text domain to use in helper when translating.

getEncoding()
Returns the character encoding used in the helper.

Return type string

getId()
Returns the element id. If no ID attribute present, attempts to use the name attribute. If name attribute is also not present, returns null.

Return type string or null

HTML5 Helpers

FormColor

The `FormColor` view helper can be used to render a `<input type="color">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Color* element, which provides a default input specification for validating HTML5 color values.

`FormColor` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Color('my-color');

// Within your view...

echo $this->formColor($element);
// <input type="color" name="my-color" value="">
```

FormDate

The `FormDate` view helper can be used to render a `<input type="date">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Date* element, which provides a default input specification for validating HTML5 date values.

`FormDate` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Date('my-date');

// Within your view...

echo $this->formDate($element);
// <input type="date" name="my-date" value="">
```

FormDateTime

The `FormDateTime` view helper can be used to render a `<input type="datetime">` HTML5 form input. It is meant to work with the *Zend\Form\Element\DateTime* element, which provides a default input specification for validating HTML5 datetime values.

`FormDateTime` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\DateTime('my-datetime');

// Within your view...

echo $this->formDateTime($element);
// <input type="datetime" name="my-datetime" value="">
```

FormDateTimeLocal

The `FormDateTimeLocal` view helper can be used to render a `<input type="datetime-local">` HTML5 form input. It is meant to work with the *Zend\Form\Element\DateTimeLocal* element, which provides a default input specification for validating HTML5 datetime values.

`FormDateTimeLocal` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\DateTimeLocal('my-datetime');

// Within your view...

echo $this->formDateTimeLocal($element);
// <input type="datetime-local" name="my-datetime" value="">
```

FormEmail

The `FormEmail` view helper can be used to render a `<input type="email">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Email* element, which provides a default input specification with an email validator.

`FormEmail` extends from *Zend\Form\View\Helper\FormInput*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Email('my-email');

// Within your view...

echo $this->formEmail($element);
// <input type="email" name="my-email" value="">
```

FormMonth

The `FormMonth` view helper can be used to render a `<input type="month">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Month* element, which provides a default input specification for validating HTML5 date values.

`FormMonth` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Month('my-month');

// Within your view...

echo $this->formMonth($element);
// <input type="month" name="my-month" value="">
```

FormTime

The `FormTime` view helper can be used to render a `<input type="time">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Time* element, which provides a default input specification for validating HTML5 time values.

`FormTime` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Time('my-time');

// Within your view...

echo $this->formTime($element);
// <input type="time" name="my-time" value="">
```

FormWeek

The `FormWeek` view helper can be used to render a `<input type="week">` HTML5 form input. It is meant to work with the *Zend\Form\Element\Week* element, which provides a default input specification for validating HTML5

week values.

`FormWeek` extends from *Zend\Form\View\Helper\FormDateTime*. Basic usage:

```
use Zend\Form\Element;

$element = new Element\Week('my-week');

// Within your view...

echo $this->formWeek($element);
// <input type="week" name="my-week" value="">
```

CHAPTER 59

Zend\Http

Overview

`Zend\Http` is a primary foundational component of Zend Framework. Since much of what PHP does is web-based, specifically HTTP, it makes sense to have a performant, extensible, concise and consistent API to do all things HTTP. In nutshell, there are several parts of `Zend\Http`:

- Context-less `Request` and `Response` classes that expose a fluent API for introspecting several aspects of HTTP messages:
  - Request line information and response status information
  - Parameters, such as those found in *POST* and *GET*
  - Message Body
  - Headers
- A Client implementation with various adapters that allow for sending requests and introspecting responses.

Zend\Http Request, Response and Headers

The Request, Response and Headers portion of the `Zend\Http` component provides a fluent, object-oriented interface for introspecting information from all the various parts of an HTTP request or HTTP response. The two main objects are `Zend\Http\Request` and `Zend\Http\Response`. These two classes are "context-less", meaning that they model a request or response in the same way whether it is presented by a client (to **send** a request and **receive** a response) or by a server (to **receive** a request and **send** a response). In other words, regardless of the context, the API remains the same for introspecting their various respective parts. Each attempts to fully model a request or response so that a developer can create these objects from a factory, or create and populate them manually.

CHAPTER 60

Zend\Http\Request

Overview

The `Zend\Http\Request` object is responsible for providing a fluent API that allows a developer to interact with all the various parts of an HTTP request.

A typical HTTP request looks like this:

```
--------------------------
| METHOD | URI | VERSION |
--------------------------
|         HEADERS         |
--------------------------
|          BODY           |
--------------------------
```

In simplified terms, the request consist of a method, *URI* and the HTTP version number which all make up the "Request Line." Next is a set of headers; there can be 0 or an unlimited number of headers. After that is the request body, which is typically used when a client wishes to send data to the server in the form of an encoded file, or include a set of POST parameters, for example. More information on the structure and specification of an HTTP request can be found in RFC-2616 on the W3.org site.

Quick Start

Request objects can either be created from the provided `fromString()` factory, or, if you wish to have a completely empty object to start with, by simply instantiating the `Zend\Http\Request` class.

```
use Zend\Http\Request;
$request = Request::fromString(<<<EOS
POST /foo HTTP/1.1
HeaderField1: header-field-value
HeaderField2: header-field-value2
```

```
6
7   foo=bar&
8   EOS);
9
10  // OR, the completely equivalent
11
12  $request = new Request();
13  $request->setMethod(Request::METHOD_POST);
14  $request->setUri('/foo');
15  $request->header()->addHeaders(array(
16      'HeaderField1' => 'header-field-value',
17      'HeaderField2' => 'header-field-value2',
18  );
19  $request->post()->set('foo', 'bar');
```

Configuration Options

None currently

Available Methods

Request::fromString `Request::fromString(string $string)`

A factory that produces a Request object from a well-formed Http Request string

Returns `Zend\Http\Request`

setMethod `setMethod(string $method)`

Set the method for this request.

Returns `Zend\Http\Request`

getMethod `getMethod()`

Return the method for this request.

Returns string.

setUri `setUri(string|\Zend\Stdlib\RequestInterface|\Zend\Stdlib\Message|\Zend\Stdlib\Paramet`
`$uri)`

Set the URI/URL for this request; this can be a string or an instance of `Zend\Uri\Http`.

Returns Zend\Http\Request

getUri `getUri()`

Return the URI for this request object.

Returns string.

uri `uri()`

Return the URI for this request object as an instance of `Zend\Uri\Http`.

Returns `Zend\Uri\Http`.

setVersion `setVersion(string $version)`

Set the HTTP version for this object, one of 1.0 or 1.1 (`Request::VERSION_10`, `Request::VERSION_11`).

Returns `Zend\Http\Request`.

setVersion `getVersion()`

Return the HTTP version for this request

Returns string

setQuery `setQuery(Zend\Stdlib\ParametersInterface $query)`

Provide an alternate Parameter Container implementation for query parameters in this object. (This is NOT the primary API for value setting; for that, see `query()`.)

Returns Zend\Http\Request

setQuery `query()`

Return the parameter container responsible for query parameters.

Returns `Zend\Stdlib\ParametersInterface`

setPost `setPost(Zend\Stdlib\ParametersInterface $post)`

Provide an alternate Parameter Container implementation for post parameters in this object. (This is NOT the primary API for value setting; for that, see `post()`.)

Returns `Zend\Http\Request`

post `post()`

Return the parameter container responsible for post parameters.

Returns `Zend\Stdlib\ParametersInterface`

cookie `cookie()`

Return the Cookie header, this is the same as calling *$request->header()->get('Cookie');*.

Returns `Zend\Http\Header\Cookie`

setFile `setFile(Zend\Stdlib\ParametersInterface $files)`

Provide an alternate Parameter Container implementation for file parameters in this object. (This is NOT the primary API for value setting; for that, see `file()`.)

Returns `Zend\Http\Request`

file `file()`

Return the parameter container responsible for file parameters

Returns `Zend\Stdlib\ParametersInterface`

setServer `setServer(Zend\Stdlib\ParametersInterface $server)`

Provide an alternate Parameter Container implementation for server parameters in this object. (This is NOT the primary API for value setting; for that, see `server()`.)

Returns `Zend\Http\Request`

server `server()`

Return the parameter container responsible for server parameters

Returns `Zend\Stdlib\ParametersInterface`

setEnv `setEnv(Zend\Stdlib\ParametersInterface $env)`

Provide an alternate Parameter Container implementation for env parameters in this object. (This is NOT the primary API for value setting; for that, see `env()`.)

Returns `Zend\Http\Request`

env `env()`

Return the parameter container responsible for env parameters

Returns `Zend\Stdlib\ParametersInterface`

setHeader `setHeader(Zend\Http\Headers $headers)`

Provide an alternate Parameter Container implementation for headers in this object. (This is NOT the primary API for value setting; for that, see `header()`.)

Returns `Zend\Http\Request`

header `header()`

Return the header container responsible for headers

Returns `Zend\Http\Headers`

setRawBody `setRawBody(string $string)`

Set the raw body for the request

Returns `Zend\Http\Request`

getRawBody `getRawBody()`

Get the raw body for the request

Returns string

isOptions `isOptions()`

Is this an OPTIONS method request?

Returns bool

isGet `isGet()`

Is this a GET method request?

Returns bool

isHead `isHead()`

Is this a HEAD method request?

Returns bool

isPost `isPost()`

Is this a POST method request?

Returns bool

isPut `isPut()`

Is this a PUT method request?

Returns bool

isDelete `isDelete()`

Is this a DELETE method request?

Returns bool

isTrace `isTrace()`

Is this a TRACE method request?

Returns bool

isConnect `isConnect()`

Is this a CONNECT method request?

Returns bool

renderRequestLine `renderRequestLine()`

Return the formatted request line (first line) for this HTTP request

Returns string

toString `toString()`

Returns string

__toString `__toString()`

Allow PHP casting of this object

Returns string

setMetadata `setMetadata(string|int|array|Traversable $spec, mixed $value)`

Set message metadata

Non-destructive setting of message metadata; always adds to the metadata, never overwrites the entire metadata container.

Returns `Zend\Stdlib\Message`

getMetadata `getMetadata(null|string|int $key, null|mixed $default)`

Retrieve all metadata or a single metadatum as specified by key

Returns mixed

setContent `setContent(mixed $value)`

Set message content

Returns `Zend\Stdlib\Message`

getContent `getContent()`

Get message content

Returns mixed

Examples

Generating a Request object from a string

```
use Zend\Http\Request;
$string = "GET /foo HTTP/1.1\r\n\r\nSome Content";
$request = Request::fromString($string);

$request->getMethod();  // returns Request::METHOD_GET
$request->getUri();     // returns '/foo'
$request->getVersion(); // returns Request::VERSION_11 or '1.1'
$request->getRawBody(); // returns 'Some Content'
```

Generating a Request object from an array

```
N/A
```

Retrieving and setting headers

```
use Zend\Http\Request;
$request = new Request();
$request->getHeaders()->get('Content-Type'); // return content type
$request->getHeaders()->addHeader(new Cookie('foo' => 'bar'));
foreach ($request->getHeaders() as $header) {
    echo $header->getFieldName() . ' with value ' . $header->getFieldValue();
}
```

Retrieving and setting GET and POST values

```
use Zend\Http\Request;
$request = new Request();

// post() and get() both return, by default, a Parameters object, which extends␣
↪ArrayObject
$request->post()->foo = 'value';
echo $request->get()->myVar;
echo $request->get()->offsetGet('myVar');
```

Generating a formatted HTTP Request from a Request object

```
use Zend\Http\Request;
$request = new Request();
$request->setMethod(Request::METHOD_POST);
$request->setUri('/foo');
$request->header()->addHeaders(array(
    'HeaderField1' => 'header-field-value',
    'HeaderField2' => 'header-field-value2',
));
$request->post()->set('foo', 'bar');
echo $request->toString();

```

```
/** Will produce:
POST /foo HTTP/1.1
HeaderField1: header-field-value
HeaderField2: header-field-value2

foo=bar
*/
```

CHAPTER 61

Zend\Http\Response

Overview

The `Zend\Http\Response` class is responsible for providing a fluent API that allows a developer to interact with all the various parts of an HTTP response.

A typical HTTP Response looks like this:

```
---------------------------
| VERSION | CODE | REASON |
---------------------------
|         HEADERS         |
---------------------------
|          BODY           |
---------------------------
```

The first line of the response consists of the HTTP version, status code, and the reason string for the provided status code; this is called the Response Line. Next is a set of headers; there can be 0 or an unlimited number of headers. The remainder of the response is the response body, which is typically a string of HTML that will render on the client's browser, but which can also be a place for request/response payload data typical of an AJAX request. More information on the structure and specification of an HTTP response can be found in RFC-2616 on the W3.org site.

Quick Start

Response objects can either be created from the provided `fromString()` factory, or, if you wish to have a completely empty object to start with, by simply instantiating the `Zend\Http\Response` class.

```
use Zend\Http\Response;
$response = Response::fromString(<<<EOS
HTTP/1.0 200 OK
HeaderField1: header-field-value
HeaderField2: header-field-value2
```

```
6
7  <html>
8  <body>
9      Hello World
10 </body>
11 </html>
12 EOS);
13
14 // OR
15
16 $response = new Response();
17 $response->setStatusCode(Response::STATUS_CODE_200);
18 $response->getHeaders()->addHeaders(array(
19     'HeaderField1' => 'header-field-value',
20     'HeaderField2' => 'header-field-value2',
21 );
22 $response->setRawBody(<<<EOS
23 <html>
24 <body>
25     Hello World
26 </body>
27 </html>
28 EOS);
```

Configuration Options

None currently available

Available Methods

Response::fromString `Response::fromString(string $string)`

Populate object from string

Returns `Zend\Http\Response`

renderStatusLine `renderStatusLine()`

Render the status line header

Returns string

setHeaders `setHeaders(Zend\Http\Headers $headers)`

Set response headers

Returns `Zend\Http\Response`

headers `headers()`

Get response headers

Returns `Zend\Http\Headers`

setVersion `setVersion(string $version)`

Returns `Zend\Http\Response`

getVersion `getVersion()`

Returns string

getStatusCode `getStatusCode()`

Retrieve HTTP status code

Returns int

setReasonPhrase `setReasonPhrase(string $reasonPhrase)`

Returns `Zend\Http\Response`

getReasonPhrase `getReasonPhrase()`

Get HTTP status message

Returns string

setStatusCode `setStatusCode(numeric $code)`

Set HTTP status code and (optionally) message

Returns `Zend\Http\Response`

isClientError `isClientError()`

Does the status code indicate a client error?

Returns bool

isForbidden `isForbidden()`

Is the request forbidden due to ACLs?

Returns bool

isInformational `isInformational()`

Is the current status "informational"?

Returns bool

isNotFound `isNotFound()`

Does the status code indicate the resource is not found?

Returns bool

isOk `isOk()`

Do we have a normal, OK response?

Returns bool

isServerError `isServerError()`

Does the status code reflect a server error?

Returns bool

isRedirect `isRedirect()`

Do we have a redirect?

Returns bool

isSuccess `isSuccess()`

Was the response successful?

Returns bool

decodeChunkedBody `decodeChunkedBody(string $body)`

Decode a "chunked" transfer-encoded body and return the decoded text

Returns string

decodeGzip `decodeGzip(string $body)`

Decode a gzip encoded message (when Content-encoding = gzip)

Currently requires PHP with zlib support

Returns string

decodeGzip `decodeDeflate(string $body)`

Decode a zlib deflated message (when Content-encoding = deflate)

Currently requires PHP with zlib support

Returns string

setMetadata `setMetadata(string|int|array|Traversable $spec, mixed $value)`

Set message metadata

Non-destructive setting of message metadata; always adds to the metadata, never overwrites the entire metadata container.

Returns `Zend\Stdlib\Message`

getMetadata `getMetadata(null|string|int $key, null|mixed $default)`

Retrieve all metadata or a single metadatum as specified by key

Returns mixed

setContent `setContent(mixed $value)`

Set message content

Returns `Zend\Stdlib\Message`

getContent `getContent()`

Get message content

Returns mixed

toString `toString()`

Returns string

Examples

Generating a Response object from a string

```
use Zend\Http\Response;
$request = Response::fromString(<<<EOS
HTTP/1.0 200 OK
HeaderField1: header-field-value
HeaderField2: header-field-value2

<html>
<body>
    Hello World
</body>
</html>
EOS);
```

Generating a Response object from a string

```
use Zend\Http\Response;
$response = new Response();
$response->setStatusCode(Response::STATUS_CODE_200);
$response->getHeaders()->addHeaders(array(
    'HeaderField1' => 'header-field-value',
    'HeaderField2' => 'header-field-value2',
));
$response->setRawBody(<<<EOS
<html>
<body>
    Hello World
</body>
</html>
EOS);
```

CHAPTER 62

Zend\Http\Headers And The Various Header Classes

Overview

The `Zend\Http\Headers` class is a container for HTTP headers. It is typically accessed as part of a `Zend\Http\Request` or `Zend\Http\Response` `header()` call. The Headers container will lazily load actual Header objects as to reduce the overhead of header specific parsing.

The `Zend\Http\Header\*` classes are the domain specific implementations for the various types of Headers that one might encounter during the typical HTTP request. If a header of unknown type is encountered, it will be implemented as a `Zend\Http\Header\GenericHeader` instance. See the below table for a list of the various HTTP headers and the API that is specific to each header type.

Quick Start

The quickest way to get started interacting with header objects is by getting an already populated Headers container from a request or response object.

Configuration Options

None currently available.

Available Methods

Headers::fromString `Headers::fromString(string $string)`

Populates headers from string representation

Parses a string for headers, and aggregates them, in order, in the current instance, primarily as strings until they are needed (they will be lazy loaded).

Returns `Zend\Http\Headers`

setPluginClassLoader `setPluginClassLoader(Zend\Loader\PluginClassLocator $pluginClassLoader)`

Set an alternate implementation for the plugin class loader

Returns `Zend\Http\Headers`

getPluginClassLoader `getPluginClassLoader()`

Return an instance of a `Zend\Loader\PluginClassLocator`, lazyload and inject map if necessary.

Returns `Zend\Loader\PluginClassLocator`

addHeaders `addHeaders(array|Traversable $headers)`

Add many headers at once

Expects an array (or `Traversable` object) of type/value pairs.

Returns `Zend\Http\Headers`

addHeaders `addHeaderLine(string $headerFieldNameOrLine, string $fieldValue)`

Add a raw header line, either in name => value, or as a single string 'name: value'

This method allows for lazy-loading in that the parsing and instantiation of Header object will be delayed until they are retrieved by either `get()` or `current()`.

Returns `Zend\Http\Headers`

addHeader `addHeader(Zend\Http\Header\HeaderInterface $header)`

Add a Header to this container, for raw values see `addHeaderLine()` and `addHeaders()`.

Returns `Zend\Http\Headers`

removeHeader `removeHeader(Zend\Http\Header\HeaderInterface $header)`

Remove a Header from the container

Returns bool

clearHeaders `clearHeaders()`

Clear all headers

Removes all headers from queue

Returns `Zend\Http\Headers`

get `get(string $name)`

Get all headers of a certain name/type

Returns false| `Zend\Http\Header\HeaderInterface`| `ArrayIterator`

has `has(string $name)`

Test for existence of a type of header

Returns bool

next `next()`

Advance the pointer for this object as an interator

Returns void

key `key()`

Return the current key for this object as an interator

Returns mixed

valid `valid()`

Is this iterator still valid?

Returns bool

rewind `rewind()`

Reset the internal pointer for this object as an iterator

Returns void

current `current()`

Return the current value for this iterator, lazy loading it if need be

Returns `Zend\Http\Header\HeaderInterface`

count `count()`

Return the number of headers in this container. If all headers have not been parsed, actual count could increase if MultipleHeader objects exist in the Request/Response. If you need an exact count, iterate.

Returns int

toString `toString()`

Render all headers at once

This method handles the normal iteration of headers; it is up to the concrete classes to prepend with the appropriate status/request line.

Returns string

toArray `toArray()`

Return the headers container as an array

Returns array

forceLoading `forceLoading()`

By calling this, it will force parsing and loading of all headers, after this `count()` will be accurate

Returns bool

Examples

Zend\Http\Header* Base Methods

fromString `fromString(string $headerLine)`

Factory to generate a header object from a string

Returns `Zend\Http\Header\GenericHeader`

getFieldName `getFieldName()`

Retrieve header field name

Returns string

getFieldValue `getFieldValue()`

Retrieve header field value

Returns string

toString `toString()`

Cast to string as a well formed HTTP header line

Returns in form of "NAME: VALUE\r\n"

Returns string

List of Http Header Types

Class Name	Additional Methods
Accept	N/A
AcceptCharset	N/A
AcceptEncoding	N/A
AcceptLanguage	N/A
AcceptRanges	getRangeUnit() / setRangeUnit() - The range unit of the accept ranges header
Age	getDeltaSeconds() / setDeltaSeconds() - The age in delta seconds
Allow	getAllowedMethods() / setAllowedMethods() - An array of allowed methods
AuthenticationInfo	N/A
Authorization	N/A
CacheControl	N/A
Connection	N/A
ContentDisposition	N/A
ContentEncoding	N/A
ContentLanguage	N/A
ContentLength	N/A
ContentLocation	N/A
ContentMD5	N/A
ContentRange	N/A
ContentType	N/A
Cookie	Extends \ArrayObjectsetEncodeValue() / getEncodeValue() - Whether or not to encode values
Date	N/A
Etag	N/A
Expect	N/A
Expires	N/A
From	N/A
Host	N/A
IfMatch	N/A
IfModifiedSince	N/A
IfNoneMatch	N/A

Class Name	Additional Methods
IfRange	N/A
IfUnmodifiedSince	N/A
KeepAlive	N/A
LastModified	N/A
Location	N/A
MaxForwards	N/A
Pragma	N/A
ProxyAuthenticate	N/A
ProxyAuthorization	N/A
Range	N/A
Referer	N/A
Refresh	N/A
RetryAfter	N/A
Server	N/A
SetCookie	getName() / setName() - The cookies namegetValue() / setValue() - The cookie valuegetDomain() / setDomain(
TE	N/A
Trailer	N/A
TransferEncoding	N/A
Upgrade	N/A
UserAgent	N/A
Vary	N/A
Via	N/A
Warning	N/A
WWWAuthenticate	N/A

CHAPTER 63

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($cookieStr, [$refUri, [$encodeValue]]) 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.

  The `fromString()` method accepts the following parameters:

  - `$cookieStr`: a cookie string as represented in the 'Set-Cookie'*HTTP* response header or 'Cookie'*HTTP* request header (required)
  - `$refUri`: a reference *URI* according to which the cookie's domain and path will be set. (optional, defaults to parsing the value from the $cookieStr)
  - `$encodeValue`: If the value should be passed through urldecode. Also effects the cookie's behavior when being converted back to a cookie string. (optional, defaults to true)

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:

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:

- `getName()`: Get the name of the cookie
- `getValue()`: Get the real, decoded value of the cookie
- `getDomain()`: Get the cookie's domain
- `getPath()`: Get the cookie's path, which defaults to '/'
- `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:

- `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*.
- `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.
- `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.

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');
```

```php

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: `Zend_Http_Cookie->match(mixed $uri, [boolean $matchSessionCookies, [int $now]]);`

- `$uri`: A `Zend_Uri_Http` object with a domain name and path to be checked. Optionally, a string representing a valid *HTTP URL* can be passed instead. The cookie will match if the *URL*'s scheme (HTTP or *HTTPS*), domain and path all match.
- `$matchSessionCookies`: Whether session cookies should be matched or not. Defaults to `TRUE`. If set to `FALSE`, cookies with no expiration time will never match.
- `$now`: Time (represented as UNIX time stamp) to check a cookie against for expiration. If not specified, will default to the current time.

Matching cookies

```php
// 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
```

```
18
19 $cookie->match('https://www.example.com/somedir/foo.php', false);
20 // Will return false, because session cookies are not matched
21
22 $cookie->match('https://sub.domain.example.com/somedir/otherdir/foo.php');
23 // Will return true
24
25 // Create another cookie object - now, not secure, with expiration time
26 // in two hours
27 $cookie = Zend_Http_Cookie::fromString('foo=two+words; ' +
28                                        'domain=www.example.com; ' +
29                                        'expires='
30                                        . date(DATE_COOKIE, time() +␣
   ↪7200));
31
32 $cookie->match('http://www.example.com/');
33 // Will return true
34
35 $cookie->match('https://www.example.com/');
36 // Will return true - non secure cookies can go over secure connections
37 // as well!
38
39 $cookie->match('http://subdomain.example.com/');
40 // Will return false, because the domain is wrong
41
42 $cookie->match('http://www.example.com/', true, time() + (3 * 3600));
43 // Will return false, because we added a time offset of +3 hours to
44 // 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()`. Additionnaly, iterating over the CookieJar will let you retrieve all the `Zend_Http_Cookie` objects from it.

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 its *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: *this section*.

CHAPTER 64

Zend\Http\Client

Overview

`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 *this section*).

Quick Start

The class constructor optionally accepts a URL as its first parameter (can be either a string or a `Zend\Uri\Http` object), and an array or `Zend\Config\Config` object containing configuration options. Both can be left out, and set later using the setUri() and setConfig() methods.

```
use Zend\Http\Client;
$client = new Client('http://example.org', array(
    'maxredirects' => 0,
    'timeout'      => 30
));

// This is actually exactly the same:
$client = new Client();
$client->setUri('http://example.org');
$client->setConfig(array(
    'maxredirects' => 0,
    'timeout'      => 30
));

// You can also use a Zend\Config\Ini object to set the client's configuration
$config = new Zend\Config\Ini('httpclient.ini', 'secure');
$client->setConfig($config);
```

Note: `Zend\Http\Client` uses `Zend\Uri\Http` to validate URLs. This means that some special characters like the pipe symbol ('|') or the caret symbol ('^') will not be accepted in the URL by default. This can be modified by setting the 'allowunwise' option of `Zend\Uri` to '`TRUE`'. See this section for more information.

Configuration Options

The constructor and setConfig() method accept an associative array of configuration parameters, or a `Zend\Config\Config` object. Setting these parameters is optional, as they all have default values.

Table 64.1: 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 this section)	boolean	FALSE
useragent	User agent identifier string (sent in request headers)	string	'Zend\Http\Client'
timeout	Connection timeout (seconds)	integer	10
httpversion	HTTP protocol version (usually '1.1' or '1.0')	string	'1.1'
adapter	Connection adapter class to use (see this section)	mixed	'Zend\Http\Client\Adapter\Socket'
keepalive	Whether to enable keep-alive connections with the server. Useful and might improve performance if several consecutive requests to the same server are performed.	boolean	FALSE
storeresponse	Whether to store last response for later retrieval with getLastResponse(). If set to FALSEgetLastResponse() will return NULL.	boolean	TRUE
encodecookies	Whether to pass the cookie value through urlencode/urldecode. Enabling this breaks support with some web servers. Disabling this limits the range of values the cookies can contain.	boolean	TRUE

Available Methods

__construct `__construct(string $uri, array $config)`

Constructor

Returns void

setConfig `setConfig(Config|array $config = array ( ))`

Set configuration parameters for this HTTP client

Returns Zend\Http\Client

setAdapter `setAdapter(Zend\Http\Client\Adapter|string $adapter)`

Load the connection adapter

While this method is not called more than one for a client, it is seperated from ->send() to preserve logic and readability

Returns null

getAdapter `getAdapter()`

Load the connection adapter

Returns Zend\Http\Client\Adapter

getRequest `getRequest()`

Get Request

Returns Request

getResponse `getResponse()`

Get Response

Returns Response

setRequest `setRequest(Zend\Http\Zend\Http\Request $request)`

Set request

Returns void

setResponse `setResponse(Zend\Http\Zend\Http\Response $response)`

Set response

Returns void

getLastRequest `getLastRequest()`

Get the last request (as a string)

Returns string

getLastResponse `getLastResponse()`

Get the last response (as a string)

Returns string

getRedirectionsCount `getRedirectionsCount()`

Get the redirections count

Returns integer

setUri `setUri(string|Zend\Http\Zend\Uri\Http $uri)`

Set Uri (to the request)

Returns void

getUri `getUri()`

Get uri (from the request)

Returns Zend\Http\Zend\Uri\Http

setMethod `setMethod(string $method)`

Set the HTTP method (to the request)

Returns Zend\Http\Client

getMethod `getMethod()`

Get the HTTP method

Returns string

setEncType `setEncType(string $encType, string $boundary)`

Set the encoding type and the boundary (if any)

Returns void

getEncType `getEncType()`

Get the encoding type

Returns type

setRawBody `setRawBody(string $body)`

Set raw body (for advanced use cases)

Returns Zend\Http\Client

setParameterPost `setParameterPost(array $post)`

Set the POST parameters

Returns Zend\Http\Client

setParameterGet `setParameterGet(array $query)`

Set the GET parameters

Returns Zend\Http\Client

getCookies `getCookies()`

Return the current cookies

Returns array

addCookie `addCookie(ArrayIterator|SetCookie|string $cookie, string $value, string $domain, string $expire, string $path, boolean $secure = false, boolean $httponly = true)`

Add a cookie

Returns Zend\Http\Client

setCookies `setCookies(array $cookies)`

Set an array of cookies

Returns Zend\Http\Client

clearCookies `clearCookies()`

Clear all the cookies

Returns void

setHeaders `setHeaders(Headers|array $headers)`

Set the headers (for the request)

Returns Zend\Http\Client

hasHeader `hasHeader(string $name)`

Check if exists the header type specified

Returns boolean

getHeader `getHeader(string $name)`

Get the header value of the request

Returns string|boolean

setStream `setStream(string|boolean $streamfile = true)`

Set streaming for received data

Returns Zend\Http\Client

getStream `getStream()`

Get status of streaming for received data

Returns boolean|string

setAuth `setAuth(string $user, string $password, string $type = 'basic')`

Create a HTTP authentication "Authorization:" header according to the specified user, password and authentication method.

Returns Zend\Http\Client

resetParameters `resetParameters()`

Reset all the HTTP parameters (auth,cookies,request, response, etc)

Returns void

send `send(Request $request)`

Send HTTP request

Returns Response

setFileUpload `setFileUpload(string $filename, string $formname, string $data, string $ctype)`

Set a file to upload (using a POST request)

Can be used in two ways: 1. $data is null (default): $filename is treated as the name if a local file which will be read and sent. Will try to guess the content type using mime_content_type(). 2. $data is set - $filename is sent as the file name, but $data is sent as the file contents and no file is read from the file system. In this case, you need to manually set the Content-Type ($ctype) or it will default to application/octet-stream.

Returns Zend\Http\Client

removeFileUpload `removeFileUpload(string $filename)`

Remove a file to upload

Returns boolean

encodeFormData `encodeFormData(string $boundary, string $name, mixed $value, string $filename, array $headers = array ( ))`

Encode data to a multipart/form-data part suitable for a POST request.

Returns string

Examples

Performing a Simple GET Request

Performing simple *HTTP* requests is very easily done using the request() method, and rarely needs more than three lines of code:

```
use Zend\Config\Client;
$client = new Client('http://example.org');
$response = $client->send();
```

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].

Using Request Methods Other Than GET

For convenience, these are all defined as class constants: Zend\Http\Client::GET, Zend\Http\Client::POST and so on.

If no method is specified, the method set by the last `setMethod()` call is used. If `setMethod()` was never called, the default request method is `GET` (see the above example).

```
use Zend\Http\Client;
$client = new Client();
// Preforming a POST request
$response = $client->send('POST');

// Yet another way of preforming a POST request
$client->setMethod(Client::POST);
$response = $client->send();
```

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 its first parameter, and the `GET` parameter's value as its 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.

```
use Zend\Http\Client;
$client = new Client();

// Setting a get parameter using the setParameterGet method
$client->setParameterGet('knight', 'lancelot');

```

[1] See RFC 2616 -http://www.w3.org/Protocols/rfc2616/rfc2616.html.

```php
// 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',
));
```

Setting POST Parameters

While `GET` parameters can be sent with every request method, POST parameters are only sent in the body of POST requests. Adding POST parameters to a request is very similar to adding `GET` parameters, and can be done with the setParameterPost() method, which is similar to the setParameterGet() method in structure.

```php
use Zend\Http\Client;
$client = new Client();
// 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.

Using A Request Object With The Client

```php
use Zend\Http\Request;
use Zend\Http\Client;
$request = new Request();
$request->setUri('http://www.test.com');
$request->setMethod('POST');
$request->setParameterPost(array('foo' => 'bar'));

$client = new Client();
$response = $client->dispatch($request);

if ($response->isSuccess()) {
    //  the POST was successful
}
```

CHAPTER 65

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 four built-in connection adapters:

- `Zend_Http_Client_Adapter_Socket` (default)
- `Zend_Http_Client_Adapter_Proxy`
- `Zend_Http_Client_Adapter_Curl`
- `Zend_Http_Client_Adapter_Test`

The `Zend_Http_Client` object's adapter connection adapter is set using the 'adapter' configuration option. When instantiating the client object, you can set the 'adapter' configuration option to a string containing the adapter's name (eg. 'Zend_Http_Client_Adapter_Socket') or to a variable holding an adapter object (eg. `new Zend_Http_Client_Adapter_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 65.1: Zend_Http_Client_Adapter_Socket configuration parameters

Parameter	Description	Expected Type	Default Value
persistent	Whether to use persistent TCP connections	boolean	FALSE
ssltransport	SSL transport layer (eg. 'sslv2', 'tls')	string	ssl
sslcert	Path to a PEM encoded SSL certificate	string	NULL
sslpassphrase	Passphrase for the SSL certificate file	string	NULL
sslusecontext	Enables proxied connections to use SSL even if the proxy connection itself does not.	boolean	FALSE

Note: Persistent TCP Connections

Using persistent *TCP* connections can potentially speed up *HTTP* requests - but in most use cases, will have little positive effect and might overload the *HTTP* server you are connecting to.

It is recommended to use persistent *TCP* connections only if you connect to the same server very frequently, and are sure that the server is capable of handling a large number of concurrent connections. In any case you are encouraged to benchmark the effect of persistent connections on both the client speed and server load before using this option.

Additionally, when using persistent connections it is recommended to enable Keep-Alive *HTTP* requests as described in the configuration section- otherwise persistent connections might have little or no effect.

Note: HTTPS SSL Stream Parameters

`ssltransport`, `sslcert` and `sslpassphrase` are only relevant when connecting using *HTTPS*.

While the default *SSL* settings should work for most applications, you might need to change them if the server you are connecting to requires special client setup. If so, you should read the sections about *SSL* transport layers and options here.

Changing the HTTPS transport layer

```
// Set the configuration parameters
$config = array(
    'adapter'      => 'Zend_Http_Client_Adapter_Socket',
    'ssltransport' => 'tls'
);

// Instantiate a client object
$client = new Zend_Http_Client('https://www.example.com', $config);

// The following request will be sent over a TLS secure connection.
$response = $client->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)
```

Customizing and accessing the Socket adapter stream context

Starting from Zend Framework 1.9, `Zend_Http_Client_Adapter_Socket` provides direct access to the underlying stream context used to connect to the remote server. This allows the user to pass specific options and parameters to the *TCP* stream, and to the *SSL* wrapper in case of *HTTPS* connections.

You can access the stream context using the following methods of `Zend_Http_Client_Adapter_Socket`:

- **setStreamContext($context)** Sets the stream context to be used by the adapter. Can accept either a stream context resource created using the stream_context_create() *PHP* function, or an array of stream context options, in the same format provided to this function. Providing an array will create a new stream context using these options, and set it.
- **getStreamContext()** Get the stream context of the adapter. If no stream context was set, will create a default stream context and return it. You can then set or get the value of different context options using regular *PHP* stream context functions.

Setting stream context options for the Socket adapter

```
// Array of options
$options = array(
    'socket' => array(
        // Bind local socket side to a specific interface
        'bindto' => '10.1.2.3:50505'
    ),
    'ssl' => array(
        // Verify server side certificate,
        // do not accept invalid or self-signed SSL certificates
        'verify_peer' => true,
        'allow_self_signed' => false,

        // Capture the peer's certificate
        'capture_peer_cert' => true
    )
);

// Create an adapter object and attach it to the HTTP client
$adapter = new Zend_Http_Client_Adapter_Socket();
$client = new Zend_Http_Client();
$client->setAdapter($adapter);

// Method 1: pass the options array to setStreamContext()
$adapter->setStreamContext($options);

// Method 2: create a stream context and pass it to setStreamContext()
$context = stream_context_create($options);
$adapter->setStreamContext($context);

// Method 3: get the default stream context and set the options on it
$context = $adapter->getStreamContext();
stream_context_set_option($context, $options);

// Now, preform the request
$response = $client->request();

// If everything went well, you can now access the context again
```

```
$opts = stream_context_get_options($adapter->getStreamContext());
echo $opts['ssl']['peer_certificate'];
```

Note: Note that you must set any stream context options before using the adapter to preform actual requests. If no context is set before preforming *HTTP* requests with the Socket adapter, a default stream context will be created. This context resource could be accessed after preforming any requests using the `getStreamContext()` method.

The Proxy Adapter

The `Zend_Http_Client_Adapter_Proxy` adapter is similar to the default Socket adapter - only the connection is made through an *HTTP* proxy server instead of a direct connection to the target server. This allows usage of `Zend_Http_Client` behind proxy servers - which is sometimes needed for security or performance reasons.

Using the Proxy adapter requires several additional configuration parameters to be set, in addition to the default 'adapter' option:

Table 65.2: 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.

Using Zend_Http_Client behind a proxy server

```
// Set the configuration parameters
$config = array(
    'adapter'    => 'Zend_Http_Client_Adapter_Proxy',
    'proxy_host' => 'proxy.int.zend.com',
    'proxy_port' => 8000,
    'proxy_user' => 'shahar.e',
    'proxy_pass' => 'bananashaped'
);

// Instantiate a client object
```

```
$client = new Zend_Http_Client('http://www.example.com', $config);

// Continue working...
```

As mentioned, if proxy_host is not set or is set to a blank string, the connection will fall back to a regular direct connection. This allows you to easily write your application in a way that allows a proxy to be used optionally, according to a configuration parameter.

Note: Since the proxy adapter inherits from `Zend_Http_Client_Adapter_Socket`, you can use the stream context access method (see *this section*) to set stream context options on Proxy connections as demonstrated above.

The cURL Adapter

cURL is a standard *HTTP* client library that is distributed with many operating systems and can be used in *PHP* via the cURL extension. It offers functionality for many special cases which can occur for a *HTTP* client and make it a perfect choice for a *HTTP* adapter. It supports secure connections, proxy, all sorts of authentication mechanisms and shines in applications that move large files around between servers.

Setting cURL options

```
$config = array(
    'adapter'   => 'Zend_Http_Client_Adapter_Curl',
    'curloptions' => array(CURLOPT_FOLLOWLOCATION => true),
);
$client = new Zend_Http_Client($uri, $config);
```

By default the cURL adapter is configured to behave exactly like the Socket Adapter and it also accepts the same configuration parameters as the Socket and Proxy adapters. You can also change the cURL options by either specifying the 'curloptions' key in the constructor of the adapter or by calling `setCurlOption($name, $value)`. The `$name` key corresponds to the CURL_* constants of the cURL extension. You can get access to the Curl handle by calling `$adapter->getHandle();`

Transfering Files by Handle

You can use cURL to transfer very large files over *HTTP* by filehandle.

```
$putFileSize   = filesize("filepath");
$putFileHandle = fopen("filepath", "r");

$adapter = new Zend_Http_Client_Adapter_Curl();
$client = new Zend_Http_Client();
$client->setAdapter($adapter);
$adapter->setConfig(array(
    'curloptions' => array(
        CURLOPT_INFILE => $putFileHandle,
        CURLOPT_INFILESIZE => $putFileSize
    )
));
$client->request("PUT");
```

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.

Testing Against a Single HTTP Response Stub

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = new Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Set the expected response
$adapter->setResponse(
    "HTTP/1.1 200 OK"         . "\r\n" .
    "Content-type: text/xml" . "\r\n" .
                               "\r\n" .
    '<?xml version="1.0" encoding="UTF-8"?>' .
    '<rss version="2.0" ' .
    '     xmlns:content="http://purl.org/rss/1.0/modules/content/"' .
    '     xmlns:wfw="http://wellformedweb.org/CommentAPI/"' .
    '     xmlns:dc="http://purl.org/dc/elements/1.1/">' .
    '  <channel>' .
    '    <title>Premature Optimization</title>' .
    // and so on...
    '</rss>');

$response = $client->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.

Testing Against Multiple HTTP Response Stubs

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = new Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
```

```
));

// Set the first expected response
$adapter->setResponse(
    "HTTP/1.1 302 Found"      . "\r\n" .
    "Location: /"             . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
    '  <head><title>Moved</title></head>' .
    '  <body><p>This page has moved.</p></body>' .
    '</html>');

// Set the next successive response
$adapter->addResponse(
    "HTTP/1.1 200 OK"         . "\r\n" .
    "Content-Type: text/html" . "\r\n" .
                                "\r\n" .
    '<html>' .
    '  <head><title>My Pet Store Home Page</title></head>' .
    '  <body><p>...</p></body>' .
    '</html>');

// inject the http client object ($client) into your object
// being tested and then test your object's behavior below
```

The setResponse() method clears any responses in the `Zend_Http_Client_Adapter_Test`'s buffer and sets the first response that will be returned. The addResponse() method will add successive responses.

The responses will be replayed in the order that they were added. If more requests are made than the number of responses stored, the responses will cycle again in order.

In the example above, the adapter is configured to test your object's behavior when it encounters a 302 redirect. Depending on your application, following a redirect may or may not be desired behavior. In our example, we expect that the redirect will be followed and we configure the test adapter to help us test this. The initial 302 response is set up with the setResponse() method and the 200 response to be returned next is added with the addResponse() method. After configuring the test adapter, inject the *HTTP* client containing the adapter into your object under test and test its behavior.

If you need the adapter to fail on demand you can use `setNextRequestWillFail($flag)`. The method will cause the next call to `connect()` to throw an `Zend_Http_Client_Adapter_Exception` exception. This can be useful when your application caches content from an external site (in case the site goes down) and you want to test this feature.

Forcing the adapter to fail

```
// Instantiate a new adapter and client
$adapter = new Zend_Http_Client_Adapter_Test();
$client = new Zend_Http_Client('http://www.example.com', array(
    'adapter' => $adapter
));

// Force the next request to fail with an exception
$adapter->setNextRequestWillFail(true);

try {
```

```
    // This call will result in a Zend_Http_Client_Adapter_Exception
    $client->request();
} catch (Zend_Http_Client_Adapter_Exception $e) {
    // ...
}

// Further requests will work as expected until
// you call setNextRequestWillFail(true) again
```

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:

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'
));
```

CHAPTER 66

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`:

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:

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 *this section*.

`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.

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 *this section*.

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:

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');
```

```php

// 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:

Setting Multiple Custom Request Headers

```php
// 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.

Using setFileUpload to Upload Files

```php
// 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->request('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'.

Note: Uploading files

When uploading files, the *HTTP* request content-type is automatically set to multipart/form-data. Keep in mind that you must send a POST or PUT request in order to upload files. Most servers will ignore the 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().

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.

Note: Using raw POST data

Setting raw POST data for a request will override any POST parameters or file uploads. You should not try to use both on the same request. Keep in mind that most servers will ignore the request body unless you send a POST request.

HTTP Authentication

Currently, `Zend_Http_Client` only supports basic *HTTP* authentication. This feature is utilized using the `setAuth()` method, or by specifying a username and a password in the URI. The `setAuth()` method takes 3 parameters: The user name, the password and an optional authentication type parameter. As mentioned, currently only basic authentication is supported (digest authentication support is planned).

Setting HTTP Authentication User and Password

```
// Using basic authentication
$client->setAuth('shahar', 'myPassword!', Zend_Http_Client::AUTH_BASIC);

// Since basic auth is default, you can just do this:
$client->setAuth('shahar', 'myPassword!');

// You can also specify username and password in the URI
$client->setUri('http://christer:secret@example.com');
```

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.

Note: Resetting parameters

Note that non-request specific headers are not reset by default when the `resetParameters()` method is used. Only the 'Content-length' and 'Content-type' headers are reset. This allows you to set-and-forget headers like 'Accept-language' and 'Accept-encoding'

To clean all headers and other data except for URI and method, use `resetParameters(true)`.

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.

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) {
```

```
9
10      $client->setCookieJar($_SESSION['cookiejar']);
11  } else {
12      // If we don't, authenticate and store cookies
13      $client->setCookieJar();
14      $client->setUri('http://www.example.com/login.php');
15      $client->setParameterPost(array(
16          'user' => 'shahar',
17          'pass' => 'somesecret'
18      ));
19      $client->request(Zend_Http_Client::POST);
20
21      // Now, clear parameters and set the URI to the original one
22      // (note that the cookies that were set by the server are now
23      // stored in the jar)
24      $client->resetParameters();
25      $client->setUri('http://www.example.com/fetchdata.php');
26  }
27
28  $response = $client->request(Zend_Http_Client::GET);
29
30  // Store cookies in session, for next page
31  $_SESSION['cookiejar'] = $client->getCookieJar();
```

Data Streaming

By default, `Zend_Http_Client` accepts and returns data as *PHP* strings. However, in many cases there are big files to be sent or received, thus keeping them in memory might be unnecessary or too expensive. For these cases, `Zend_Http_Client` supports reading data from files (and in general, *PHP* streams) and writing data to files (streams).

In order to use stream to pass data to `Zend_Http_Client`, use `setRawData()` method with data argument being stream resource (e.g., result of `fopen()`).

Sending file to HTTP server with streaming

```
1  $fp = fopen("mybigfile.zip", "r");
2  $client->setRawData($fp, 'application/zip')->request('PUT');
```

Only PUT requests currently support sending streams to *HTTP* server.

In order to receive data from the server as stream, use `setStream()`. Optional argument specifies the filename where the data will be stored. If the argument is just `TRUE` (default), temporary file will be used and will be deleted once response object is destroyed. Setting argument to `FALSE` disables the streaming functionality.

When using streaming, `request()` method will return object of class `Zend_Http_Client_Response_Stream`, which has two useful methods: `getStreamName()` will return the name of the file where the response is stored, and `getStream()` will return stream from which the response could be read.

You can either write the response to pre-defined file, or use temporary file for storing it and send it out or write it to another file using regular stream functions.

Receiving file from HTTP server with streaming

```
$client->setStream(); // will use temp file
$response = $client->request('GET');
// copy file
copy($response->getStreamName(), "my/downloads/file");
// use stream
$fp = fopen("my/downloads/file2", "w");
stream_copy_to_stream($response->getStream(), $fp);
// Also can write to known file
$client->setStream("my/downloads/myfile)->request('GET');
```

CHAPTER 67

Translating

Zend_I18n comes with a complete translation suite which supports all major formats and includes popular features like plural translations and text domains. The Translator component is mostly dependency free, except for the fallback to a default locale, where it relies on the Intl PHP extension.

The translator itself is initialized without any parameters, as any configuration to it is optional. A translator without any translations will actually do nothing but just return the given message IDs.

Adding translations

To add translations to the translator, there are two options. You can either add every translation file individually, which is the best way if you use translation formats which store multiple locales in the same file, or you can add translations via a pattern, which works best for formats which contain one locale per file.

To add a single file to the translator, use the `addTranslationFile()` method:

```
use Zend\I18n\Translator\Translator;

$translator = new Translator();
$translator->addTranslationFile($type, $filename, $textDomain, $locale);
```

The type given there is a name of one of the format loaders listed in the next section. Filename points to the file containing the file containing the translations and the text domain specifies a category name for the translations. If the text domain is omitted, it will default to the "default" value. The locale specifies which language the translated strings are from and is only required for formats which contain translations for a single locale.

Note: For each text domain and locale combination, there can only be one file loaded. Every successive file would override the translations which were loaded prior.

When storing one locale per file, you should specify those files via a pattern. This allows you to add new translations to the file system, without touching your code. Patterns are added with the `addTranslationPattern()` method:

```
use Zend\I18n\Translator\Translator;

$translator = new Translator();
$translator->addTranslationPattern($type, $pattern, $textDomain);
```

The parameters for adding patterns is pretty similar to adding individual files, except that don't specify a locale and give the file location as sprtinf pattern. The locale is passed to the sprintf call, so you can either use %s oder %1$s where it should be substituted. So when youf translation files are located in /var/messages/LOCALE/messages.mo, you would specify your pattern as /var/messages/%s/messages.mo.

Supported formats

The translator supports the following major translation formats:

- PHP arrays
- Gettext
- Tmx
- Xliff

Setting a locale

By default, the translator will get the locale to use from the Intl extension's `Locale` class. If you want to set an alternative locale explicitly, you can do so by passing it to the `setLocale()` method.

When there is not translation for a specific message ID in a locale, the message ID itself will be returned by default. Alternatively you can set a fallback locale which is used to retrieve a fallback translation. To do so, pass it to the `setFallbackLocale()` method.

Translating messages

Translating messages can accomplished by calling the `translate()` method of the translator:

```
$translator->translate($message, $textDomain, $locale);
```

The message is the ID of your message to translate. If it does not exist in the loader translations or is empty, the original message ID will be returned. The text domain parameter is the one you specified when adding translations. If omitted, the default text domain will be used. The locale parameter will usually not be used in this context, as by default the locale is taken from the locale set in the translator.

To translate plural messages, you can use the `translatePlural()` method. It works similar to `translate()`, but instead of a single messages it takes a singluar and a plural value and an additional integer number on which the returned plural form is based on:

```
$translator->translatePlural($singular, $plural, $number, $textDomain, $locale);
```

Plural translations are only available if the underlying format supports the transport of plural messages and plural rule definitions.

Caching

In production it makes sense to cache your translations. This not only saves you from loading and parsing the individual formats each time, but also guarantees an optimized loading procedure. To enable caching, simply pass a `Zend\Cache\Storage\Adapter` to the `setCache()` method. To disable the cache, you can just pass a null value to it.

CHAPTER 68

I18n View Helpers

Introduction

Zend Framework comes with an initial set of helper classes related to Internationalization: e.g., formatting a date, formatting currency, or displaying translated content. You can use helper, or plugin, classes to perform these behaviors for you.

See the section on *view helpers* for more information.

CurrencyFormat Helper

The `CurrencyFormat` view helper can be used to simplify rendering of localized currency values. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl). **Basic Usage**

```
// Within your view

echo $this->currencyFormat(1234.56, "USD", "en_US");
// This returns: "$1,234.56"

echo $this->currencyFormat(1234.56, "EUR", "de_DE");
// This returns: "1.234,56 €"
```

`currencyFormat` (*float $number*, *string $currencyCode*[, *string $locale*])

Parameters

- **`$number`** – The numeric currency value.
- **`$currencyCode`** – The 3-letter ISO 4217 currency code indicating the currency to use.
- **`$locale`** – (Optional) Locale in which the currency would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

The `$currencyCode` and `$locale` options can be set prior to formatting and will be applied each time the helper is used:

```
// Within your view
$this->plugin("currencyformat")->setCurrencyCode("USD")->setLocale("en_US");

echo $this->currencyFormat(1234.56);  // "$1,234.56"
echo $this->currencyFormat(5678.90);  // "$5,678.90"
```

DateFormat Helper

The `DateFormat` view helper can be used to simplify rendering of localized date/time values. It acts as a wrapper for the `IntlDateFormatter` class within the Internationalization extension (Intl). **Basic Usage**

```
// Within your view

// Date and Time
echo $this->dateFormat(
    new DateTime(),
    IntlDateFormatter::MEDIUM, // date
    IntlDateFormatter::MEDIUM, // time
    "en_US"
);
// This returns: "Jul 2, 2012 6:44:03 PM"

// Date Only
echo $this->dateFormat(
    new DateTime(),
    IntlDateFormatter::LONG, // date
    IntlDateFormatter::NONE, // time
    "en_US"
);
// This returns: "July 2, 2012"

// Time Only
echo $this->dateFormat(
    new DateTime(),
    IntlDateFormatter::NONE,  // date
    IntlDateFormatter::SHORT, // time
    "en_US"
);
// This returns: "6:44 PM"
```

dateFormat (*mixed $date*[, *int $dateType*[, *int $timeType*[, *string $locale*]]])

Parameters

- **$date** – The value to format. This may be a `DateTime` object, an integer representing a Unix timestamp value or an array in the format output by `localtime()`.
- **$dateType** – (Optional) Date type to use (none, short, medium, long, full). This is one of the IntlDateFormatter constants. Defaults to `IntlDateFormatter::NONE`.
- **$timeType** – (Optional) Time type to use (none, short, medium, long, full). This is one of the IntlDateFormatter constants. Defaults to `IntlDateFormatter::NONE`.

- **$locale** – (Optional) Locale in which the date would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

The `$locale` option can be set prior to formatting with the `setLocale()` method and will be applied each time the helper is used.

By default, the system's default timezone will be used when formatting. This overrides any timezone that may be set inside a DateTime object. To change the timezone when formatting, use the `setTimezone` method.

```
// Within your view
$this->plugin("dateFormat")->setTimezone("America/New_York")->setLocale("en_US");

echo $this->dateFormat(new DateTime(), IntlDateFormatter::MEDIUM);  // "Jul 2, 2012"
echo $this->dateFormat(new DateTime(), IntlDateFormatter::SHORT);   // "7/2/12"
```

NumberFormat Helper

The `NumberFormat` view helper can be used to simplify rendering of locale-specific number and percentage strings. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl). **Basic Usage**

```
// Within your view

// Example of Decimal formatting:
echo $this->numberFormat(
    1234567.891234567890000,
    NumberFormatter::DECIMAL,
    NumberFormatter::TYPE_DEFAULT,
    "de_DE"
);
// This returns: "1.234.567,891"

// Example of Percent formatting:
echo $this->numberFormat(
    0.80,
    NumberFormatter::PERCENT,
    NumberFormatter::TYPE_DEFAULT,
    "en_US"
);
// This returns: "80%"

// Example of Scientific notation formatting:
echo $this->numberFormat(
    0.00123456789,
    NumberFormatter::SCIENTIFIC,
    NumberFormatter::TYPE_DEFAULT,
    "fr_FR"
);
// This returns: "1,23456789E-3"
```

numberFormat(*number $number*[, *int $formatStyle*[, *int $formatType*[, *string $locale*]]])

Parameters

- **$number** – The numeric value.

- **$formatStyle** – (Optional) Style of the formatting, one of the format style constants. If unset, it will use `NumberFormatter::DECIMAL` as the default style.
- **$formatType** – (Optional) The formatting type to use. If unset, it will use `NumberFormatter::TYPE_DEFAULT` as the default type.
- **$locale** – (Optional) Locale in which the number would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

The `$formatStyle`, `$formatType`, and `$locale` options can be set prior to formatting and will be applied each time the helper is used.

```
// Within your view
$this->plugin("numberformat")
             ->setFormatStyle(NumberFormatter::PERCENT)
             ->setFormatType(NumberFormatter::TYPE_DOUBLE)
             ->setLocale("en_US");

echo $this->numberFormat(0.56);  // "56%"
echo $this->numberFormat(0.90);  // "90%"
```

Translate Helper

The `Translate` view helper can be used to translate content. It acts as a wrapper for the `Zend\I18n\Translator\Translator` class. **Setup**

Before using the `Translate` view helper, you must have first created a `Translator` object and have attached it to the view helper. If you use the `Zend\View\HelperPluginManager` to invoke the view helper, this will be done automatically for you. **Basic Usage**

```
// Within your view

echo $this->translate("Some translated text.");

echo $this->translate("Translated text from a custom text domain.", "customDomain");

echo sprintf($this->translate("The current time is %s."), $currentTime);

echo $this->translate("Translate in a specific locale", "default", "de_DE");
```

translate(*string $message*[, *string $textDomain*[, *string $locale*]])

Parameters

- **$message** – The message to be translated.
- **$textDomain** – (Optional) The text domain where this translation lives. Defaults to the value "default".
- **$locale** – (Optional) Locale in which the message would be translated (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Gettext

The `xgettext` utility can be used to compile *.po files from PHP source files containing the translate view helper.

```
xgettext --language=php --add-location --keyword=translate my-view-file.phtml
```

See the Gettext Wikipedia page for more information. **Public Methods**

Public methods for setting a `Zend\I18n\Translator\Translator` and a default text domain are inherited from *Zend\I18n\View\Helper\AbstractTranslatorHelper*.

TranslatePlural Helper

The `TranslatePlural` view helper can be used to translate words which take into account numeric meanings. English, for example, has a singular definition of "car", for one car. And has the plural definition, "cars", meaning zero "cars" or more than one car. Other languages like Russian or Polish have more plurals with different rules.

The viewhelper acts as a wrapper for the `Zend\I18n\Translator\Translator` class. **Setup**

Before using the `TranslatePlural` view helper, you must have first created a `Translator` object and have attached it to the view helper. If you use the `Zend\View\HelperPluginManager` to invoke the view helper, this will be done automatically for you. **Basic Usage**

```
1 // Within your view
2 echo $this->translatePlural("car", "cars", $num);
3
4 // Use a custom domain
5 echo $this->translatePlural("monitor", "monitors", $num, "customDomain");
6
7 // Change locale
8 echo $this->translate("locale", "locales", $num, "default", "de_DE");
```

`translatePlural` (*string $singular*, *string $plural*, *int $number*[, *string $textDomain*[, *string $locale*]])

Parameters

- **`$singular`** – The singular message to be translated.
- **`$plural`** – The plural message to be translated.
- **`$number`** – The number to evaluate and determine which message to use.
- **`$textDomain`** – (Optional) The text domain where this translation lives. Defaults to the value "default".
- **`$locale`** – (Optional) Locale in which the message would be translated (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

Public Methods

Public methods for setting a `Zend\I18n\Translator\Translator` and a default text domain are inherited from *Zend\I18n\View\Helper\AbstractTranslatorHelper*.

Abstract Translator Helper

The `AbstractTranslatorHelper` view helper is used as a base abstract class for any helpers that need to translate content. It provides an implementation for the `Zend\I18n\Translator\TranslatorAwareInterface` which allows injecting a translator and setting a text domain. **Public Methods**

setTranslator(*Translator $translator*[, *string $textDomain = null*])
Sets `Zend\I18n\Translator\Translator` to use in helper. The `$textDomain` argument is optional. It is provided as a convenience for setting both the translator and textDomain at the same time.

getTranslator()
Returns the `Zend\I18n\Translator\Translator` used in the helper.

Return type `Zend\I18n\Translator\Translator`

hasTranslator()
Returns a true if a `Zend\I18n\Translator\Translator` is set in the helper, and false if otherwise.

Return type boolean

setTranslatorEnabled(*boolean $enabled*)
Sets whether translations should be enabled or disabled.

isTranslatorEnabled()
Returns true if translations are enabled, and false if disabled.

Return type boolean

setTranslatorTextDomain(*string $textDomain*)
Set the translation text domain to use in helper when translating.

getTranslatorTextDomain()
Returns the translation text domain used in the helper.

Return type string

CHAPTER 69

I18n Filters

Zend Framework comes with a set of filters related to Internationalization.

CHAPTER 70

Alnum Filter

The `Alnum` filter can be used to return only alphabetic characters and digits in the unicode "letter" and "number" categories, respectively. All other characters are supressed.

Supported options for Alnum Filter

The following options are supported for `Alnum`:

`Alnum([ boolean $allowWhiteSpace [, string $locale ]])`

- `$allowWhiteSpace`: If set to true then whitespace characters are allowed. Otherwise they are suppressed. Default is "false" (whitespace is not allowed).

  Methods for getting/setting the allowWhiteSpace option are also available: `getAllowWhiteSpace()` and `setAllowWhiteSpace()`

- `$locale`: The locale string used in identifying the characters to filter (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`).

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

Alnum Filter Usage

```
1 // Default settings, deny whitespace
2 $filter = \Zend\I18n\Filter\Alnum();
3 echo $filter->filter("This is (my) content: 123");
4 // Returns "Thisismycontent123"
5
6 // First param in constructor is $allowWhiteSpace
7 $filter = \Zend\I18n\Filter\Alnum(true);
8 echo $filter->filter("This is (my) content: 123");
9 // Returns "This is my content 123"
```

Note: Note: `Alnum` works on almost all languages, except: Chinese, Japanese and Korean. Within these languages the english alphabet is used instead of the characters from these languages. The language itself is detected using the `Locale`.

CHAPTER 71

Alpha Filter

The `Alpha` filter can be used to return only alphabetic characters in the unicode "letter" category. All other characters are supressed.

Supported options for Alpha Filter

The following options are supported for `Alpha`:

`Alpha([ boolean $allowWhiteSpace [, string $locale ]])`

- `$allowWhiteSpace`: If set to true then whitespace characters are allowed. Otherwise they are suppressed. Default is "false" (whitespace is not allowed).

  Methods for getting/setting the allowWhiteSpace option are also available: `getAllowWhiteSpace()` and `setAllowWhiteSpace()`

- `$locale`: The locale string used in identifying the characters to filter (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`).

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

Alpha Filter Usage

```
1 // Default settings, deny whitespace
2 $filter = \Zend\I18n\Filter\Alpha();
3 echo $filter->filter("This is (my) content: 123");
4 // Returns "Thisismycontent"
5
6 // Allow whitespace
7 $filter = \Zend\I18n\Filter\Alpha(true);
8 echo $filter->filter("This is (my) content: 123");
9 // Returns "This is my content "
```

Note: Note: `Alpha` works on almost all languages, except: Chinese, Japanese and Korean. Within these languages the english alphabet is used instead of the characters from these languages. The language itself is detected using the `Locale`.

CHAPTER 72

NumberFormat Filter

The `NumberFormat` filter can be used to return locale-specific number and percentage strings. It acts as a wrapper for the `NumberFormatter` class within the Internationalization extension (Intl).

Supported options for NumberFormat Filter

The following options are supported for `NumberFormat`:

`NumberFormat([ string $locale [, int $style [, int $type ]]])`

- `$locale`: (Optional) Locale in which the number would be formatted (locale name, e.g. en_US). If unset, it will use the default locale (`Locale::getDefault()`)

  Methods for getting/setting the locale are also available: `getLocale()` and `setLocale()`

- `$style`: (Optional) Style of the formatting, one of the format style constants. If unset, it will use `NumberFormatter::DEFAULT_STYLE` as the default style.

  Methods for getting/setting the format style are also available: `getStyle()` and `setStyle()`

- `$type`: (Optional) The formatting type to use. If unset, it will use `NumberFormatter::TYPE_DOUBLE` as the default type.

  Methods for getting/setting the format type are also available: `getType()` and `setType()`

NumberFormat Filter Usage

```
$filter = \Zend\I18n\Filter\NumberFormat("de_DE");
echo $filter->filter(1234567.8912346);
// Returns "1.234.567,891"

$filter = \Zend\I18n\Filter\NumberFormat("en_US", NumberFormatter::PERCENT);
echo $filter->filter(0.80);
// Returns "80%"

```

```php
$filter = \Zend\I18n\Filter\NumberFormat("fr_FR", NumberFormatter::SCIENTIFIC);
echo $filter->filter(0.00123456789);
// Returns "1,23456789E-3"
```

CHAPTER 73

Introduction

The `Zend\InputFilter` component can be used to filter and validate generic sets of input data. For instance, you could use it to filter `$_GET` or `$_POST` values, CLI arguments, etc.

To pass input data to the `InputFilter`, you can use the `setData()` method. The data must be specified using an associative array. Below is an example on how to validate the data coming from a form using the *POST* method.

```
use Zend\InputFilter\InputFilter;
use Zend\InputFilter\Input;
use Zend\Validator;

$email = new Input('email');
$email->getValidatorChain()
      ->addValidator(new Validator\EmailAddress());

$password = new Input('password');
$password->getValidatorChain()
         ->addValidator(new Validator\StringLength(8));

$inputFilter = new InputFilter();
$inputFilter->add($email)
            ->add($password)
            ->setData($_POST);

if ($inputFilter->isValid()) {
    echo "The form is valid\n";
} else {
    echo "The form is not valid\n";
    foreach ($inputFilter->getInvalidInput() as $error) {
        print_r ($error->getMessages());
    }
}
```

In this example we validated the email and password values. The email must be a valid address and the password must be composed with at least 8 characters. If the input data are not valid, we report the list of invalid input using the `getInvalidInput()` method.

You can add one or more validators to each input using the `addValidator()` method for each validator. It is also possible to specify a "validation group", a subset of the data to be validated; this may be done using the `setValidationGroup()` method. You can specify the list of the input names as an array or as individual parameters.

```
// As individual parameters
$filterInput->setValidationGroup('email', 'password');

// or as an array of names
$filterInput->setValidationGroup(array('email', 'password'));
```

You can validate and/or filter the data using the `InputFilter`. To filter data, use the `getFilterChain()` method of individual `Input` instances, and attach filters to the returned filter chain. Below is an example that uses filtering without validation.

```
use Zend\InputFilter\Input;
use Zend\InputFilter\InputFilter;

$input = new Input('foo');
$input->getFilterChain()
      ->attachByName('stringtrim')
      ->attachByName('alpha');

$inputfilter = new InputFilter();
$inputfilter->add($input, 'foo')
            ->setData(array(
                'foo' => ' Bar3 ';
            ));

echo "Before:\n";
echo $inputFilter->getRawValue('foo') . "\n"; // the output is ' Bar3 '
echo "After:\n";
echo $inputFilter->getValue('foo') . "\n"; // the output is 'Bar'
```

The `getValue()` method returns the filtered value of the 'foo' input, while `getRawValue()` returns the original value of the input.

We provide also `Zend\InputFilter\Factory`, to allow initialization of the `InputFilter` based on a configuration array (or `Traversable` object). Below is an example where we create a password input value with the same constraints proposed before (a string with at least 8 characters):

```
use Zend\InputFilter\Factory;

$factory = new Factory();
$inputFilter = $factory->createInputFilter(array(
    'password' => array(
        'name'       => 'password',
        'required'   => true,
        'validators' => array(
            array(
                'name' => 'not_empty',
            ),
            array(
                'name' => 'string_length',
                'options' => array(
                    'min' => 8
                ),
            ),
```

```
        ),
    ),
));

$inputFilter->setData($_POST);
echo $inputFilter->isValid() ? "Valid form" : "Invalid form";
```

The factory may be used to create not only `Input` instances, but also nested `InputFilter`s, allowing you to create validation and filtering rules for hierarchical data sets.

Finally, the default `InputFilter` implementation is backed by a `Factory`. This means that when calling `add()`, you can provide a specification that the `Factory` would understand, and it will create the appropriate object. You may create either `Input` or `InputFilter` objects in this fashion.

```
use Zend\InputFilter\InputFilter;

$filter = new InputFilter();

// Adding a single input
$filter->add(array(
    'name' => 'password',
    'required' => true,
    'validators' => array(
        array(
            'name' => 'not_empty',
        ),
        array(
            'name' => 'string_length',
            'options' => array(
                'min' => 8
            ),
        ),
    ),
));

// Adding an input filter composing a single input to the current filter
$filter->add(array(
    'type' => 'Zend\Filter\InputFilter',
    'password' => array(
        'name' => 'password',
        'required' => true,
        'validators' => array(
            array(
                'name' => 'not_empty',
            ),
            array(
                'name' => 'string_length',
                'options' => array(
                    'min' => 8
                ),
            ),
        ),
    ),
));
```

CHAPTER 74

Introduction

`Zend\Ldap\Ldap` is a class for performing *LDAP* operations including but not limited to binding, searching and modifying entries in an *LDAP* directory.

Theory of operation

This component currently consists of the main `Zend\Ldap\Ldap` class, that conceptually represents a binding to a single *LDAP* server and allows for executing operations against a *LDAP* server such as OpenLDAP or ActiveDirectory (AD) servers. The parameters for binding may be provided explicitly or in the form of an options array. `Zend\Ldap\Node` provides an object-oriented interface for single *LDAP* nodes and can be used to form a basis for an active-record-like interface for a *LDAP*-based domain model.

The component provides several helper classes to perform operations on *LDAP* entries (`Zend\Ldap\Attribute`) such as setting and retrieving attributes (date values, passwords, boolean values, ...), to create and modify *LDAP* filter strings (`Zend\Ldap\Filter`) and to manipulate *LDAP* distinguished names (DN) (`Zend\Ldap\Dn`).

Additionally the component abstracts *LDAP* schema browsing for OpenLDAP and ActiveDirectory servers `Zend\Ldap\Node\Schema` and server information retrieval for OpenLDAP-, ActiveDirectory- and Novell eDirectory servers (`Zend\Ldap\Node\RootDse`).

Using the `Zend\Ldap\Ldap` class depends on the type of *LDAP* server and is best summarized with some simple examples.

If you are using OpenLDAP, a simple example looks like the following (note that the **bindRequiresDn** option is important if you are **not** using AD):

```
$options = array(
    'host'              => 's0.foo.net',
    'username'          => 'CN=user1,DC=foo,DC=net',
    'password'          => 'pass1',
    'bindRequiresDn'    => true,
    'accountDomainName' => 'foo.net',
    'baseDn'            => 'OU=Sales,DC=foo,DC=net',
);
```

```
$ldap = new Zend\Ldap\Ldap($options);
$acctname = $ldap->getCanonicalAccountName('abaker',
                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

If you are using Microsoft AD a simple example is:

```
$options = array(
    'host'                   => 'dc1.w.net',
    'useStartTls'            => true,
    'username'               => 'user1@w.net',
    'password'               => 'pass1',
    'accountDomainName'      => 'w.net',
    'accountDomainNameShort' => 'W',
    'baseDn'                 => 'CN=Users,DC=w,DC=net',
);
$ldap = new Zend\Ldap\Ldap($options);
$acctname = $ldap->getCanonicalAccountName('bcarter',
                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

Note that we use the `getCanonicalAccountName()` method to retrieve the account DN here only because that is what exercises the most of what little code is currently present in this class.

Automatic Username Canonicalization When Binding

If `bind()` is called with a non-DN username but **bindRequiresDN** is `TRUE` and no username in DN form was supplied as an option, the bind will fail. However, if a username in DN form is supplied in the options array, `Zend\Ldap\Ldap` will first bind with that username, retrieve the account DN for the username supplied to `bind()` and then re-bind with that DN.

This behavior is critical to *Zend\Authentication\Adapter\Ldap*, which passes the username supplied by the user directly to `bind()`.

The following example illustrates how the non-DN username '**abaker**' can be used with `bind()`:

```
$options = array(
        'host'              => 's0.foo.net',
        'username'          => 'CN=user1,DC=foo,DC=net',
        'password'          => 'pass1',
        'bindRequiresDn'    => true,
        'accountDomainName' => 'foo.net',
        'baseDn'            => 'OU=Sales,DC=foo,DC=net',
);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind('abaker', 'moonbike55');
$acctname = $ldap->getCanonicalAccountName('abaker',
                                           Zend\Ldap\Ldap::ACCTNAME_FORM_DN);
echo "$acctname\n";
```

The `bind()` call in this example sees that the username '**abaker**' is not in DN form, finds **bindRequiresDn** is `TRUE`, uses '`CN=user1,DC=foo,DC=net`' and '**pass1**' to bind, retrieves the DN for '**abaker**', unbinds and then rebinds with the newly discovered '`CN=Alice Baker,OU=Sales,DC=foo,DC=net`'.

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 74.1: Options for accountCanonicalForm

Name	Value	Example
ACCTNAME_FORM_DN	1	CN=Alice Baker,CN=Users,DC=example,DC=com
ACCTNAME_FORM_USERNAME	2	abaker
ACCTNAME_FORM_BACKSLASH	3	EXAMPLE\abaker
ACCTNAME_FORM_PRINCIPAL	4	abaker@example.com

The default canonicalization depends on what account domain name options were supplied. If **accountDomainNameShort** was supplied, the default **accountCanonicalForm** value is `ACCTNAME_FORM_BACKSLASH`. Otherwise, if **accountDomainName** was supplied, the default is `ACCTNAME_FORM_PRINCIPAL`.

Account name canonicalization ensures that the string used to identify an account is consistent regardless of what was supplied to `bind()`. For example, if the user supplies an account name of `abaker@example.com` or just **abaker** and the **accountCanonicalForm** is set to 3, the resulting canonicalized name would be **EXAMPLEabaker**.

Multi-domain Authentication and Failover

The `Zend\Ldap\Ldap` component by itself makes no attempt to authenticate with multiple servers. However, `Zend\Ldap\Ldap` is specifically designed to handle this scenario gracefully. The required technique is to simply iterate over an array of arrays of serve options and attempt to bind with each server. As described above `bind()` will automatically canonicalize each name, so it does not matter if the user passes `abaker@foo.net` or **Wbcarter** or **cdavis**- the `bind()` method will only succeed if the credentials were successfully used in the bind.

Consider the following example that illustrates the technique required to implement multi-domain authentication and failover:

```
$acctname = 'W\\user2';
$password = 'pass2';

$multiOptions = array(
    'server1' => array(
        'host'                   => 's0.foo.net',
        'username'               => 'CN=user1,DC=foo,DC=net',
        'password'               => 'pass1',
        'bindRequiresDn'         => true,
        'accountDomainName'      => 'foo.net',
        'accountDomainNameShort' => 'FOO',
        'accountCanonicalForm'   => 4, // ACCT_FORM_PRINCIPAL
        'baseDn'                 => 'OU=Sales,DC=foo,DC=net',
    ),
    'server2' => array(
        'host'                   => 'dc1.w.net',
        'useSsl'                 => true,
        'username'               => 'user1@w.net',
        'password'               => 'pass1',
        'accountDomainName'      => 'w.net',
        'accountDomainNameShort' => 'W',
        'accountCanonicalForm'   => 4, // ACCT_FORM_PRINCIPAL
```

```
        'baseDn'                    => 'CN=Users,DC=w,DC=net',
    ),
);

$ldap = new Zend\Ldap\Ldap();

foreach ($multiOptions as $name => $options) {

    echo "Trying to bind using server options for '$name'\n";

    $ldap->setOptions($options);
    try {
        $ldap->bind($acctname, $password);
        $acctname = $ldap->getCanonicalAccountName($acctname);
        echo "SUCCESS: authenticated $acctname\n";
        return;
    } catch (Zend\Ldap\Exception\LdapException $zle) {
        echo '  ' . $zle->getMessage() . "\n";
        if ($zle->getCode() === Zend\Ldap\Exception\LdapException::LDAP_X_DOMAIN_
↪MISMATCH) {
            continue;
        }
    }
}
```

If the bind fails for any reason, the next set of server options is tried.

The `getCanonicalAccountName()` call gets the canonical account name that the application would presumably use to associate data with such as preferences. The **accountCanonicalForm = 4** in all server options ensures that the canonical form is consistent regardless of which server was ultimately used.

The special `LDAP_X_DOMAIN_MISMATCH` exception occurs when an account name with a domain component was supplied (e.g., `abaker@foo.net` or **FOOabaker** and not just **abaker**) but the domain component did not match either domain in the currently selected server options. This exception indicates that the server is not an authority for the account. In this case, the bind will not be performed, thereby eliminating unnecessary communication with the server. Note that the **continue** instruction has no effect in this example, but in practice for error handling and debugging purposes, you will probably want to check for `LDAP_X_DOMAIN_MISMATCH` as well as `LDAP_NO_SUCH_OBJECT` and `LDAP_INVALID_CREDENTIALS`.

The above code is very similar to code used within *Zend\Authentication\Adapter\Ldap*. In fact, we recommend that you simply use that authentication adapter for multi-domain + failover *LDAP* based authentication (or copy the code).

CHAPTER 75

API overview

Configuration / options

The `Zend\Ldap\Ldap` component accepts an array of options either supplied to the constructor or through the `setOptions()` method. The permitted options are as follows:

Table 75.1: Zend\Ldap\Ldap Options

Name	Description
host	The default hostname of LDAP server if not supplied to connect() (also may be used when trying to canonicalize usernames in bind()).
port	Default port of LDAP server if not supplied to connect().
useStartTls	Whether or not the LDAP client should use TLS (aka SSLv2) encrypted transport. A value of TRUE is strongly favored in production environments to prevent passwords from be transmitted in clear text. The default value is FALSE, as servers frequently require that a certificate be installed separately after installation. The useSsl and useStartTls options are mutually exclusive. The useStartTls option should be favored over useSsl but not all servers support this newer mechanism.
useSsl	Whether or not the LDAP client should use SSL encrypted transport. The useSsl and useStartTls options are mutually exclusive.
username	The default credentials username. Some servers require that this be in DN form. This must be given in DN form if the LDAP server requires a DN to bind and binding should be possible with simple usernames.
password	The default credentials password (used only with username above).
bindRequiresDn	If TRUE, this instructs Zend\Ldap\Ldap to retrieve the DN for the account used to bind if the username is not already in DN form. The default value is FALSE.
baseDn	The default base DN used for searching (e.g., for accounts). This option is required for most account related operations and should indicate the DN under which accounts are located.
accountCanonicalForm	A small integer indicating the form to which account names should be canonicalized. See the Account Name Canonicalization section below.
accountDomainName	The FQDN domain for which the target LDAP server is an authority (e.g., example.com).
accountDomainNameShort	The 'short' domain for which the target LDAP server is an authority. This is usually used to specify the NetBIOS domain name for Windows networks but may also be used by non-AD servers.
accountFilterFormat	The LDAP search filter used to search for accounts. This string is a sprintf() style expression that must contain one '%s' to accommodate the username. The default value is '(&(objectClass=user)(sAMAccountName=%s))' unless bindRequiresDn is set to TRUE, in which case the default is '(&(objectClass=posixAccount)(uid=%s))'. Users of custom schemas may need to change this option.
allowEmptyPassword	Some LDAP servers can be configured to accept an empty string password as an anonymous bind. This behavior is almost always undesirable. For this reason, empty passwords are explicitly disallowed. Set this value to TRUE to allow an empty string password to be submitted during the bind.
optReferrals	If set to TRUE, this option indicates to the LDAP client that referrals should be followed. The default value is FALSE.
tryUsernameSplit	If set to FALSE, this option indicates that the given username should not be split at the first @ or \ character to separate the username from the domain during the binding-procedure. This allows the user to use usernames that contain an @ or \ character that do not inherit some domain-information, e.g. using email-addresses for binding. The default value is TRUE.
networkTimeout	Number of seconds to wait for LDAP connection before fail. If not set the default value is the system value.

API Reference

Note: Method names in **italics** are static methods.

CHAPTER 76

Zend\Ldap\Ldap

`Zend\Ldap\Ldap` is the base interface into a *LDAP* server. It provides connection and binding methods as well as methods to operate on the *LDAP* tree.

Method
__construct($options)
resource getResource()
integer getLastErrorCode()
string getLastError(integer &$errorCode, array &$errorMessages)
Zend\Ldap\Ldap setOptions($options)
array getOptions()
string getBaseDn()
string getCanonicalAccountName(string $acctname, integer $form)
Zend\Ldap\Ldap disconnect()
Zend\Ldap\Ldap connect(string $host, integer $port, boolean $useSsl, boolean $useStartTls, integer $networkTimeout)
Zend\Ldap\Ldap bind(string $username, string $password)
Zend\Ldap\Collection search(string|Zend\Ldap\Filter\AbstractFilter $filter, string|Zend\Ldap\Dn $basedn, integer $scope, array $attrib
integer count(string|Zend\Ldap\Filter\AbstractFilter $filter, string|Zend\Ldap\Dn $basedn, integer $scope)
integer countChildren(string|Zend\Ldap\Dn $dn)
boolean exists(string|Zend\Ldap\Dn $dn)
array searchEntries(string|Zend\Ldap\Filter\AbstractFilter $filter, string|Zend\Ldap\Dn $basedn, integer $scope, array $attributes, strin
array getEntry(string|Zend\Ldap\Dn $dn, array $attributes, boolean $throwOnNotFound)
void prepareLdapEntryArray(array &$entry)
Zend\Ldap\Ldap add(string|Zend\Ldap\Dn $dn, array $entry)
Zend\Ldap\Ldap update(string|Zend\Ldap\Dn $dn, array $entry)
Zend\Ldap\Ldap save(string|Zend\Ldap\Dn $dn, array $entry)
Zend\Ldap\Ldap delete(string|Zend\Ldap\Dn $dn, boolean $recursively)
Zend\Ldap\Ldap moveToSubtree(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively, boolean $alwaysEmula
Zend\Ldap\Ldap move(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively, boolean $alwaysEmulate)

Method
Zend\Ldap\Ldap rename(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively, boolean $alwaysEmulate)
Zend\Ldap\Ldap copyToSubtree(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively)
Zend\Ldap\Ldap copy(string|Zend\Ldap\Dn $from, string|Zend\Ldap\Dn $to, boolean $recursively)
Zend\Ldap\Node getNode(string|Zend\Ldap\Dn $dn)
Zend\Ldap\Node getBaseNode()
Zend\Ldap\Node\RootDse getRootDse()
Zend\Ldap\Node\Schema getSchema()

Zend\Ldap\Collection

`Zend\Ldap\Collection` implements *Iterator* to allow for item traversal using `foreach()` and *Countable* to be able to respond to `count()`. With its protected `createEntry()` method it provides a simple extension point for developers needing custom result objects.

Table 76.2: Zend\Ldap\Collection API

Method	Description
__construct(Zend\Ldap\Collection\Iterator\Interface $iterator)	Constructor. The constructor must be provided by a Zend\Ldap\Collection\Iterator\Interface which does the real result iteration. Zend\Ldap\Collection\Iterator\Default is the default implementation for iterating ext/ldap results.
boolean close()	Closes the internal iterator. This is also called in the destructor.
array toArray()	Returns all entries as an array.
array getFirst()	Returns the first entry in the collection or NULL if the collection is empty.

CHAPTER 77

Zend\Ldap\Attribute

`Zend\Ldap\Attribute` is a helper class providing only static methods to manipulate arrays suitable to the structure used in `Zend\Ldap\Ldap` data modification methods and to the data format required by the *LDAP* server. *PHP* data types are converted using `Zend\Ldap\Converter\Converter` methods.

Table 77.1: Zend\Ldap\Attribute API

Method	Description
void setAttribute(array &$data, string $attribName, mixed $value, boolean $append)	Sets the attribute $attribName in $data to the value $value. If $append is TRUE (FALSE by default) $value will be appended to the attribute. $value can be a scalar value or an array of scalar values. Conversion will take place.
array|mixed getAttribute(array $data, string $attribName, integer|null $index)	Returns the attribute $attribName from $data. If $index is NULL (default) an array will be returned containing all the values for the given attribute. An empty array will be returned if the attribute does not exist in the given array. If an integer index is specified the corresponding value at the given index will be returned. If the index is out of bounds, NULL will be returned. Conversion will take place.
boolean attributeHasValue(array &$data, string $attribName, mixed|array $value)	Checks if the attribute $attribName in $data has the value(s) given in $value. The method returns TRUE only if all values in $value are present in the attribute. Comparison is done strictly (respecting the data type).
void removeDuplicatesFromAttribute(array &$data, string $attribName)	Removes all duplicates from the attribute $attribName in $data.
void removeFromAttribute(array &$data, string $attribName, mixed|array $value)	Removes the value(s) given in $value from the attribute $attribName in $data.
void setPassword(array &$data, string $password, string $hashType, string $attribName)	Sets a LDAP password for the attribute $attribName in $data. $attribName defaults to 'userPassword' which is the standard password attribute. The password hash can be specified with $hashType. The default value here is Zend\Ldap\Attribute::PASSWORD_HASH_MD5 with Zend\Ldap\Attribute::PASSWORD_HASH_SHA as the other possibility.
string createPassword(string $password, string $hashType)	Creates a LDAP password. The password hash can be specified with $hashType. The default value here is Zend\Ldap\Attribute::PASSWORD_HASH_MD5 with Zend\Ldap\Attribute::PASSWORD_HASH_SHA as the other possibility.
void setDateTimeAttribute(array &$data, string $attribName, integer|array $value, boolean $utc, boolean $append)	Sets the attribute $attribName in $data to the date/time value $value. if $append is TRUE (FALSE by default) $value will be appended to the attribute. $value can be an integer value or an array of integers. Date-time-conversion according to Zend\Ldap\Converter\Converter::toLdapDateTime() will take place.
array|integer getDateTimeAttribute(array $data, string $attribName, integer|null $index)	Returns the date/time attribute $attribName from $data. If $index is NULL (default) an array will be returned containing all the date/time values for the given attribute. An empty array will be returned if the attribute does not exist in the given array. If an integer index is specified the corresponding date/time value at the given index will be returned. If the index is out of bounds, NULL will be returned. Date-time-conversion according to Zend\Ldap\Converter\Converter::fromLdapDateTime() will take place.

CHAPTER 78

Zend\Ldap\Converter\Converter

`Zend\Ldap\Converter\Converter` is a helper class providing only static methods to manipulate arrays suitable to the data format required by the *LDAP* server. *PHP* data types are converted the following way:

string No conversion will be done.

integer and float The value will be converted to a string.

boolean `TRUE` will be converted to **'TRUE'** and `FALSE` to **'FALSE'**

object and array The value will be converted to a string by using `serialize()`.

Date/Time The value will be converted to a string with the following `date()` format *YmdHisO*, UTC timezone (+0000) will be replaced with a *Z*. For example *01-30-2011 01:17:32 PM GMT-6* will be *20113001131732-0600* and *30-01-2012 15:17:32 UTC* will be *20120130151732Z*

resource If a *stream* resource is given, the data will be fetched by calling `stream_get_contents()`.

others All other data types (namely non-stream resources) will be omitted.

On reading values the following conversion will take place:

'TRUE' Converted to `TRUE`.

'FALSE' Converted to `FALSE`.

others All other strings won't be automatically converted and are passed as they are.

Table 78.1: Zend\Ldap\Converter\Converter API

Method	Description
string ascToHex32(string $string)	Convert all Ascii characters with decimal value less than 32 to hexadecimal value.
string hex32ToAsc(string $string)	Convert all hexadecimal characters by his Ascii value.
string|null toLdap(mixed $value, int $type)	Converts a PHP data type into its LDAP representation. $type argument is used to set the conversion method by default Converter::STANDARD where the function will try to guess the conversion method to use, others possibilities are Converter::BOOLEAN and Converter::GENERALIZED_TIME See introduction for details.
mixed fromLdap(string $value, int $type, boolean $dateTimeAsUtc)	Converts an LDAP value into its PHP data type. See introduction and toLdap() and toLdapDateTime() for details.
string|null toLdapDate-Time(integer|string|DateTime $date, boolean $asUtc)	Converts a timestamp, a DateTime Object, a string that is parseable by strtotime() or a DateTime into its LDAP date/time representation. If $asUtc is TRUE (FALSE by default) the resulting LDAP date/time string will be inUTC, otherwise a local date/time string will be returned.
DateTime fromLdapDateTime(string $date, boolean $asUtc)	Converts LDAP date/time representation into a PHP DateTime object.
string toLdap-Boolean(boolean|integer|string $value)	Converts a PHP data type into its LDAP boolean representation. By default always return 'FALSE' except if the value is true , 'true' or 1
boolean fromLdapBoolean(string $value)	Converts LDAP boolean representation into a PHP boolean data type.
string toLdapSerialize(mixed $value)	The value will be converted to a string by using serialize().
mixed fromLdapUnserialize(string $value)	The value will be converted from a string by using unserialize().

CHAPTER 79

Zend\Ldap\Dn

`Zend\Ldap\Dn` provides an object-oriented interface to manipulating *LDAP* distinguished names (DN). The parameter `$caseFold` that is used in several methods determines the way DN attributes are handled regarding their case. Allowed values for this parameter are:

ZendLdapDn::ATTR_CASEFOLD_NONE No case-folding will be done.

ZendLdapDn::ATTR_CASEFOLD_UPPER All attributes will be converted to upper-case.

ZendLdapDn::ATTR_CASEFOLD_LOWER All attributes will be converted to lower-case.

The default case-folding is `Zend\Ldap\Dn::ATTR_CASEFOLD_NONE` and can be set with `Zend\Ldap\Dn::setDefaultCaseFold()`. Each instance of `Zend\Ldap\Dn` can have its own case-folding-setting. If the `$caseFold` parameter is omitted in method-calls it defaults to the instance's case-folding setting.

The class implements *ArrayAccess* to allow indexer-access to the different parts of the DN. The *ArrayAccess*-methods proxy to `Zend\Ldap\Dn::get($offset, 1, null)` for *offsetGet(integer $offset)*, to `Zend\Ldap\Dn::set($offset, $value)` for `offsetSet()` and to `Zend\Ldap\Dn::remove($offset, 1)` for `offsetUnset()`. `offsetExists()` simply checks if the index is within the bounds.

Table 79.1: Zend\Ldap\Dn API

Method	Description
Zend\Ldap\Dn factory(string|array $dn, string|null $caseFold)	Creates a Zend\Ldap\Dn instance from an array or a string. The array must conform to the array structure detailed under Zend\Ldap\Dn::implodeDn().
Zend\Ldap\Dn fromString(string $dn, string|null $caseFold)	Creates a Zend\Ldap\Dn instance from a string.
Zend\Ldap\Dn fromArray(array $dn, string|null $caseFold)	Creates a Zend\Ldap\Dn instance from an array. The array must conform to the array structure detailed under Zend\Ldap\Dn::implodeDn().
array getRdn(string|null $caseFold)	Gets the RDN of the current DN. The return value is an array with the RDN attribute names its keys and the RDN attribute values.
string getRdnString(string|null $caseFold)	Gets the RDN of the current DN. The return value is a string.
Zend\Ldap\Dn getParentDn(integer $levelUp)	Gets the DN of the current DN's ancestor $levelUp levels up the tree. $levelUp defaults to 1.
array get(integer $index, integer $length, string|null $caseFold)	Returns a slice of the current DN determined by $index and $length. $index starts with 0 on the DN part from the left.
Zend\Ldap\Dn set(integer $index, array $value)	Replaces a DN part in the current DN. This operation manipulates the current instance.
Zend\Ldap\Dn remove(integer $index, integer $length)	Removes a DN part from the current DN. This operation manipulates the current instance. $length defaults to 1
Zend\Ldap\Dn append(array $value)	Appends a DN part to the current DN. This operation manipulates the current instance.
Zend\Ldap\Dn prepend(array $value)	Prepends a DN part to the current DN. This operation manipulates the current instance.
Zend\Ldap\Dn insert(integer $index, array $value)	Inserts a DN part after the index $index to the current DN. This operation manipulates the current instance.
void setCaseFold(string|null $caseFold)	Sets the case-folding option to the current DN instance. If $caseFold is NULL the default case-folding setting (Zend\Ldap\Dn::ATTR_CASEFOLD_NONE by default or set via Zend\Ldap\Dn::setDefaultCaseFold() will be set for the current instance.
string toString(string|null $caseFold)	Returns DN as a string.
array toArray(string|null $caseFold)	Returns DN as an array.
string __toString()	Returns DN as a string - proxies to Zend\Ldap\Dn::toString(null).
void setDefaultCaseFold(string $caseFold)	Sets the default case-folding option used by all instances on creation by default. Already existing instances are not affected by this setting.
array escapeValue(string|array $values)	Escapes a DN value according to RFC 2253.
array unescapeValue(string|array $values)	Undoes the conversion done by Zend\Ldap\Dn::escapeValue().
array explodeDn(string $dn, array &$keys, array &$vals, string|null $caseFold)	Explodes the DN $dn into an array containing all parts of the given DN. $keys optionally receive DN keys (e.g. CN, OU, DC, ...). $vals optionally receive DN values. The resulting array will be of type array(array("cn" => "name1", "uid" => "user"), array("cn" => "name2"), array("dc" => "example"), array("dc" => "org")) for a DN of cn=name1+uid=user,cn=name2,dc=example,dc=org.
boolean checkDn(string $dn, array &$keys, array &$vals, string|null $caseFold)	Checks if a given DN $dn is malformed. If $keys or $keys and $vals are given, these arrays will be filled with the appropriate DN keys and values.
string implodeRdn(array $part, string|null $caseFold)	Returns a DN part in the form $attribute=$value
string implodeDn(array $dnArray, string|null $caseFold, string $separator)	Implodes an array in the form delivered by Zend\Ldap\Dn::explodeDn() to a DN string. $separator defaults to ',' but some LDAP servers also understand ';'. $dnArray must of type array(array("cn" => "name1", "uid" => "user"), array("cn" => "name2"), array("dc" => "example"), array("dc" => "org"))
boolean isChildOf(string|Zend\Ldap\Dn	Checks if given $childDn is beneath $parentDn subtree.

CHAPTER 80

Zend\Ldap\Filter

Table 80.1: Zend\Ldap\Filter API

Method	Description
Zend\Ldap\Filter equals(string $attr, string $value)	Creates an ‘equals’ filter: (attr=value).
Zend\Ldap\Filter begins(string $attr, string $value)	Creates an ‘begins with’ filter: (attr=value*).
Zend\Ldap\Filter ends(string $attr, string $value)	Creates an ‘ends with’ filter: (attr=*value).
Zend\Ldap\Filter contains(string $attr, string $value)	Creates an ‘contains’ filter: (attr=*value*).
Zend\Ldap\Filter greater(string $attr, string $value)	Creates an ‘greater’ filter: (attr>value).
Zend\Ldap\Filter greaterOrEqual(string $attr, string $value)	Creates an ‘greater or equal’ filter: (attr>=value).
Zend\Ldap\Filter less(string $attr, string $value)	Creates an ‘less’ filter: (attr<value).
Zend\Ldap\Filter lessOrEqual(string $attr, string $value)	Creates an ‘less or equal’ filter: (attr<=value).
Zend\Ldap\Filter approx(string $attr, string $value)	Creates an ‘approx’ filter: (attr~=value).
Zend\Ldap\Filter any(string $attr)	Creates an ‘any’ filter: (attr=*).
Zend\Ldap\Filter string(string $filter)	Creates a simple custom string filter. The user is responsible for all value-escaping as the filter is used as is.
Zend\Ldap\Filter mask(string $mask, string $value,...)	Creates a filter from a string mask. All $value parameters will be escaped and substituted into $mask by using sprintf()
Zend\Ldap\Filter andFilter(Zend\Ldap\Filter\AbstractFilter $filter,...)	Creates an ‘and’ filter from all arguments given.
Zend\Ldap\Filter orFilter(Zend\Ldap\Filter\AbstractFilter $filter,...)	Creates an ‘or’ filter from all arguments given.
__construct(string $attr, string $value, string $filtertype, string|null $prepend, string|null $append)	Constructor. Creates an arbitrary filter according to the parameters supplied. The resulting filter will be a concatenation of $attr . $filtertype . $prepend . $value . $append. Normally this constructor is not needed as all filters can be created by using the appropriate factory methods.
string toString()	Returns a string representation of the filter.

CHAPTER 81

Zend\Ldap\Node

`Zend\Ldap\Node` includes the magic property accessors `__set()`, `__get()`, `__unset()` and `__isset()` to access the attributes by their name. They proxy to `Zend\Ldap\Node::setAttribute()`, `Zend\Ldap\Node::getAttribute()`, `Zend\Ldap\Node::deleteAttribute()` and `Zend\Ldap\Node::existsAttribute()` respectively. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. `Zend\Ldap\Node` also implements *Iterator* and *RecursiveIterator* to allow for recursive tree-traversal.

Method	Description
Zend\Ldap\Ldap getLdap()	Returns the cu
Zend\Ldap\Node attachLdap(Zend\Ldap\Ldap $ldap)	Attach the cur
Zend\Ldap\Node detachLdap()	Detach node f
boolean isAttached()	Checks if the
Zend\Ldap\Node create(string|array|Zend\Ldap\Dn $dn, array $objectClass)	Factory metho
Zend\Ldap\Node fromLdap(string|array|Zend\Ldap\Dn $dn, Zend\Ldap\Ldap $ldap)	Factory metho
Zend\Ldap\Node fromArray((array $data, boolean $fromDataSource)	Factory metho
boolean isNew()	Tells if the no
boolean willBeDeleted()	Tells if this no
Zend\Ldap\Node delete()	Marks this no
boolean willBeMoved()	Tells if this no
Zend\Ldap\Node update(Zend\Ldap\Ldap $ldap)	Sends all pend
Zend\Ldap\Dn getCurrentDn()	Gets the curre
Zend\Ldap\Dn getDn()	Gets the origi
string getDnString(string $caseFold)	Gets the origi
array getDnArray(string $caseFold)	Gets the origi
string getRdnString(string $caseFold)	Gets the RDN
array getRdnArray(string $caseFold)	Gets the RDN
Zend\Ldap\Node setDn(Zend\Ldap\Dn|string|array $newDn)	Sets the new
Zend\Ldap\Node move(Zend\Ldap\Dn|string|array $newDn)	This is an alia
Zend\Ldap\Node rename(Zend\Ldap\Dn|string|array $newDn)	This is an alia

Method	Description
array getObjectClass()	Returns the ob
Zend\Ldap\Node setObjectClass(array|string $value)	Sets the objec
Zend\Ldap\Node appendObjectClass(array|string $value)	Appends to th
string toLdif(array $options)	Returns a LD
array getChangedData()	Gets changed
array getChanges()	Returns all ch
string toString()	Returns the D
string __toString()	Casts to string
array toArray(boolean $includeSystemAttributes)	Returns an ar
string toJson(boolean $includeSystemAttributes)	Returns a JSO
array getData(boolean $includeSystemAttributes)	Returns the no
boolean existsAttribute(string $name, boolean $emptyExists)	Checks wheth
boolean attributeHasValue(string $name, mixed|array $value)	Checks if the
integer count()	Returns the nu
mixed getAttribute(string $name, integer|null $index)	Gets a LDAP
array getAttributes(boolean $includeSystemAttributes)	Gets all attrib
Zend\Ldap\Node setAttribute(string $name, mixed $value)	Sets a LDAP
Zend\Ldap\Node appendToAttribute(string $name, mixed $value)	Appends to a
array|integer getDateTimeAttribute(string $name, integer|null $index)	Gets a LDAP
Zend\Ldap\Node setDateTimeAttribute(string $name, integer|array $value, boolean $utc)	Sets a LDAP
Zend\Ldap\Node appendToDateTimeAttribute(string $name, integer|array $value, boolean $utc)	Appends to a
Zend\Ldap\Node setPasswordAttribute(string $password, string $hashType, string $attribName)	Sets a LDAP
Zend\Ldap\Node deleteAttribute(string $name)	Deletes a LDA
void removeDuplicatesFromAttribute(string$name)	Removes dup
void removeFromAttribute(string $attribName, mixed|array $value)	Removes the
boolean exists(Zend\Ldap\Ldap $ldap)	Checks if the
Zend\Ldap\Node reload(Zend\Ldap\Ldap $ldap)	Reloads the c
Zend\Ldap\Node\Collection searchSubtree(string|Zend\Ldap\Filter\AbstractFilter $filter, integer $scope, string $sort)	Searches the n
integer countSubtree(string|Zend\Ldap\Filter\AbstractFilter $filter, integer $scope)	Count the noc
integer countChildren()	Count the noc
Zend\Ldap\Node\Collection searchChildren(string|Zend\Ldap\Filter\AbstractFilter $filter, string $sort)	Searches the n
boolean hasChildren()	Returns whet
Zend\Ldap\Node\ChildrenIterator getChildren()	Returns all ch
Zend\Ldap\Node getParent(Zend\Ldap\Ldap $ldap)	Returns the pa

CHAPTER 82

Zend\Ldap\Node\RootDse

The following methods are available on all vendor-specific subclasses.

`Zend\Ldap\Node\RootDse` includes the magic property accessors `__get()` and `__isset()` to access the attributes by their name. They proxy to `Zend\Ldap\Node\RootDse::getAttribute()` and `Zend\Ldap\Node\RootDse::existsAttribute()` respectively. `__set()` and `__unset()` are also implemented but they throw a *BadMethodCallException* as modifications are not allowed on RootDSE nodes. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. `offsetSet()` and `offsetUnset()` also throw a *BadMethodCallException* due ro obvious reasons.

Table 82.1: Zend\Ldap\Node\RootDse API

Method	Description
Zend\Ldap\Dn getDn()	Gets the DN of the current node as a Zend\Ldap\Dn.
string getDnString(string $caseFold)	Gets the DN of the current node as a string.
array getDnArray(string $caseFold)	Gets the DN of the current node as an array.
string getRdnString(string $caseFold)	Gets the RDN of the current node as a string.
array getRdnArray(string $caseFold)	Gets the RDN of the current node as an array.
array getObjectClass()	Returns the objectClass of the node.
string toString()	Returns the DN of the current node - proxies to Zend\Ldap\Dn::getDnString().
string __toString()	Casts to string representation - proxies to Zend\Ldap\Dn::toString().
array toArray(boolean $includeSystemAttributes)	Returns an array representation of the current node. If $includeSystemAttributes is FALSE (defaults to TRUE) the system specific attributes are stripped from the array. Unlike Zend\Ldap\Node\RootDse::getAttributes() the resulting array contains the DN with key 'dn'.
string toJson(boolean $includeSystemAttributes)	Returns a JSON representation of the current node using Zend\Ldap\Node\RootDse::toArray().
array getData(boolean $includeSystemAttributes)	Returns the node's attributes. The array contains all attributes in its internal format (no conversion).
boolean existsAttribute(string $name, boolean $emptyExists)	Checks whether a given attribute exists. If $emptyExists is FALSE, empty attributes (containing only array()) are treated as non-existent returning FALSE. If $emptyExists is TRUE, empty attributes are treated as existent returning TRUE. In this case the method returns FALSE only if the attribute name is missing in the key-collection.
boolean attributeHasValue(string $name, mixed|array $value)	Checks if the given value(s) exist in the attribute. The method returns TRUE only if all values in $value are present in the attribute. Comparison is done strictly (respecting the data type).
integer count()	Returns the number of attributes in the node. Implements Countable.
mixed getAttribute(string $name, integer|null $index)	Gets a LDAP attribute. Data conversion is applied using Zend\Ldap\Attribute::getAttribute().
array getAttributes(boolean $includeSystemAttributes)	Gets all attributes of node. If $includeSystemAttributes is FALSE (defaults to TRUE) the system specific attributes are stripped from the array.
array|integer getDateTimeAttribute(string $name, integer|null $index)	Gets a LDAP date/time attribute. Data conversion is applied using Zend\Ldap\Attribute::getDateTimeAttribute().
Zend\Ldap\Node\RootDse reload(Zend\Ldap\Ldap $ldap)	Reloads the current node's attributes from the given LDAP server.
Zend\Ldap\Node\RootDse create(Zend\Ldap\Ldap $ldap)	Factory method to create the RootDSE.
array getNamingContexts()	Gets the namingContexts.
string|null getSubschemaSubentry()	Gets the subschemaSubentry.
boolean supportsVersion(string|int|array $versions)	Determines if the LDAP version is supported.
boolean supportsSaslMechanism(string|array $mechlist)	Determines if the sasl mechanism is supported.
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_GENERICfor unknown LDAP serversZend\Ldap\Node\RootDse::SERVER_TYPE_OPENLDAPfor OpenLDAP servers Zend\Ldap\Node\RootDse::SERVER_TYPE_ACTIVEDIRECTORYfor Microsoft ActiveDirectory

OpenLDAP

Additionally the common methods above apply to instances of `Zend\Ldap\Node\RootDse\OpenLdap`.

Note: Refer to LDAP Operational Attributes and Objects for information on the attributes of OpenLDAP RootDSE.

Table 82.2: Zend\Ldap\Node\RootDse\OpenLdap API

Method	Description
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_OPENLDAP
string|null getConfigContext()	Gets the configContext.
string|null getMonitorContext()	Gets the monitorContext.
boolean supportsControl(string|array $oids)	Determines if the control is supported.
boolean supportsExtension(string|array $oids)	Determines if the extension is supported.
boolean supportsFeature(string|array $oids)	Determines if the feature is supported.

ActiveDirectory

Additionally the common methods above apply to instances of `Zend\Ldap\Node\RootDse\ActiveDirectory`.

Note: Refer to RootDSE for information on the attributes of Microsoft ActiveDirectory RootDSE.

Table 82.3: Zend\Ldap\Node\RootDse\ActiveDirectory API

Method	Description
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_ACTIVEDIRECTORY
string|null getConfigurationNamingContext()	Gets the configurationNamingContext.
string|null getCurrentTime()	Gets the currentTime.
string|null getDefaultNamingContext()	Gets the defaultNamingContext.
string|null getDnsHostName()	Gets the dnsHostName.
string|null getDomainController-Functionality()	Gets the domainControllerFunctionality.
string|null getDomainFunctionality()	Gets the domainFunctionality.
string|null getDsServiceName()	Gets the dsServiceName.
string|null getForestFunctionality()	Gets the forestFunctionality.
string|null getHighestCommittedUSN()	Gets the highestCommittedUSN.
string|null getIsGlobalCatalogReady()	Gets the isGlobalCatalogReady.
string|null getIsSynchronized()	Gets the isSynchronized.
string|null getLdapServiceName()	Gets the ldapServiceName.
string|null getRootDomainNamingContext()	Gets the rootDomainNamingContext.
string|null getSchemaNamingContext()	Gets the schemaNamingContext.
string|null getServerName()	Gets the serverName.
boolean supportsCapability(string|array $oids)	Determines if the capability is supported.
boolean supportsControl(string|array $oids)	Determines if the control is supported.
boolean supportsPolicy(string|array $policies)	Determines if the version is supported.

eDirectory

Additionally the common methods above apply to instances of *ZendLdapNodeRootDseeDirectory*.

Note: Refer to Getting Information about the LDAP Server for information on the attributes of Novell eDirectory RootDSE.

Table 82.4: Zend\Ldap\Node\RootDse\eDirectory API

Method	Description
integer getServerType()	Gets the server type. Returns Zend\Ldap\Node\RootDse::SERVER_TYPE_EDIRECTORY
boolean supportsExtension(string|array $oids)	Determines if the extension is supported.
string|null getVendorName()	Gets the vendorName.
string|null getVendorVersion()	Gets the vendorVersion.
string|null getDsaName()	Gets the dsaName.
string|null getStatisticsErrors()	Gets the server statistics “errors”.
string|null getStatisticsSecurityErrors()	Gets the server statistics “securityErrors”.
string|null getStatisticsChainings()	Gets the server statistics “chainings”.
string|null getStatisticsReferralsReturned()	Gets the server statistics “referralsReturned”.
string|null getStatisticsExtendedOps()	Gets the server statistics “extendedOps”.
string|null getStatisticsAbandonOps()	Gets the server statistics “abandonOps”.
string|null getStatisticsWholeSubtreeSearchOps()	Gets the server statistics “wholeSubtreeSearchOps”.

CHAPTER 83

Zend\Ldap\Node\Schema

The following methods are available on all vendor-specific subclasses.

ZendLdapNodeSchema includes the magic property accessors *__get()* and *__isset()* to access the attributes by their name. They proxy to *ZendLdapNodeSchema::getAttribute()* and *ZendLdapNodeSchema::existsAttribute()* respectively. *__set()* and *__unset()* are also implemented, but they throw a *BadMethodCallException* as modifications are not allowed on RootDSE nodes. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. *offsetSet()* and *offsetUnset()* also throw a *BadMethodCallException* due to obvious reasons.

Table 83.1: Zend\Ldap\Node\Schema API

Method	Description
Zend\Ldap\Dn getDn()	Gets the DN of the current node as a Zend\Ldap\Dn.
string getDnString(string $caseFold)	Gets the DN of the current node as a string.
array getDnArray(string $caseFold)	Gets the DN of the current node as an array.
string getRdnString(string $caseFold)	Gets the RDN of the current node as a string.
array getRdnArray(string $caseFold)	Gets the RDN of the current node as an array.
array getObjectClass()	Returns the objectClass of the node.
string toString()	Returns the DN of the current node - proxies to Zend\Ldap\Dn::getDnString().
string __toString()	Casts to string representation - proxies to Zend\Ldap\Dn::toString().
array toArray(boolean $includeSystemAttributes)	Returns an array representation of the current node. If $includeSystemAttributes is FALSE (defaults to TRUE), the system specific attributes are stripped from the array. Unlike Zend\Ldap\Node\Schema::getAttributes(), the resulting array contains the DN with key 'dn'.
string toJson(boolean $includeSystemAttributes)	Returns a JSON representation of the current node using Zend\Ldap\Node\Schema::toArray().
array getData(boolean $includeSystemAttributes)	Returns the node's attributes. The array contains all attributes in its internal format (no conversion).
boolean existsAttribute(string $name, boolean $emptyExists)	Checks whether a given attribute exists. If $emptyExists is FALSE, empty attributes (containing only array()) are treated as non-existent returning FALSE. If $emptyExists is TRUE, empty attributes are treated as existent returning TRUE. In this case the method returns FALSE only if the attribute name is missing in the key-collection.
boolean attributeHasValue(string $name, mixed|array $value)	Checks if the given value(s) exist in the attribute. The method returns TRUE only if all values in $value are present in the attribute. Comparison is done strictly (respecting the data type).
integer count()	Returns the number of attributes in the node. Implements Countable.
mixed getAttribute(string $name, integer|null $index)	Gets a LDAP attribute. Data conversion is applied using Zend\Ldap\Attribute::getAttribute().
array getAttributes(boolean $includeSystemAttributes)	Gets all attributes of node. If $includeSystemAttributes is FALSE (defaults to TRUE) the system specific attributes are stripped from the array.
array|integer getDateTimeAttribute(string $name, integer|null $index)	Gets a LDAP date/time attribute. Data conversion is applied using Zend\Ldap\Attribute::getDateTimeAttribute().
Zend\Ldap\Node\Schema reload(Zend\Ldap\Ldap $ldap)	Reloads the current node's attributes from the given LDAP server.
Zend\Ldap\Node\Schema create(Zend\Ldap\Ldap $ldap)	Factory method to create the Schema node.
array getAttributeTypes()	Gets the attribute types as an array of .
array getObjectClasses()	Gets the object classes as an array of Zend\Ldap\Node\Schema\ObjectClass\Interface.

Table 83.2: Zend\Ldap\Node\Schema\AttributeType\Interface API

Method	Description
string getName()	Gets the attribute name.
string getOid()	Gets the attribute OID.
string getSyntax()	Gets the attribute syntax.
int|null getMaxLength()	Gets the attribute maximum length.
boolean isSingleValued()	Returns if the attribute is single-valued.
string getDescription()	Gets the attribute description

Table 83.3: Zend\Ldap\Node\Schema\ObjectClass\Interface API

Method	Description
string getName()	Returns the objectClass name.
string getOid()	Returns the objectClass OID.
array getMustContain()	Returns the attributes that this objectClass must contain.
array getMayContain()	Returns the attributes that this objectClass may contain.
string getDescription()	Returns the attribute description
integer getType()	Returns the objectClass type. The method returns one of the following values: Zend\Ldap\Node\Schema::OBJECTCLASS_TYPE_UNKNOWNfor unknown class typesZend\Ldap\Node\Schema::OBJECTCLASS_TYPE_STRUCTURALfor structural classesZend\Ldap\Node\Schema::OBJECTCLASS_TYPE_ABSTRACTfor abstract classesZend\Ldap\Node\Schema::OBJECTCLASS_TYPE_AUXILIARYfor auxiliary classes
array getParentClasses()	Returns the parent objectClasses of this class. This includes structural, abstract and auxiliary objectClasses.

Classes representing attribute types and object classes extend *ZendLdapNodeSchemaAbstractItem* which provides some core methods to access arbitrary attributes on the underlying *LDAP* node. *ZendLdapNodeSchemaAbstractItem* includes the magic property accessors *__get()* and *__isset()* to access the attributes by their name. Furthermore the class implements *ArrayAccess* for array-style-access to the attributes. *offsetSet()* and *offsetUnset()* throw a *BadMethodCallException* as modifications are not allowed on schema information nodes.

Table 83.4: Zend\Ldap\Node\Schema\AbstractItem API

Method	Description
array getData()	Gets all the underlying data from the schema information node.
integer count()	Returns the number of attributes in this schema information node. Implements Countable.

OpenLDAP

Additionally the common methods above apply to instances of *ZendLdapNodeSchemaOpenLDAP*.

Table 83.5: Zend\Ldap\Node\Schema\OpenLDAP API

Method	Description
array getLdapSyntaxes()	Gets the LDAP syntaxes.
array getMatchingRules()	Gets the matching rules.
array getMatchingRuleUse()	Gets the matching rule use.

Table 83.6: Zend\Ldap\Node\Schema\AttributeType\OpenLDAP API

Method	Description
Zend\Ldap\Node\Schema\AttributeType\OpenLdap|null getParent()	Returns the parent attribute type in the inheritance tree if one exists.

Table 83.7: Zend\Ldap\Node\Schema\ObjectClass\OpenLDAP API

Method	Description
array getParents()	Returns the parent object classes in the inheritance tree if one exists. The returned array is an array of Zend\Ldap\Node\Schema\ObjectClass\OpenLdap.

ActiveDirectory

Note: Schema browsing on ActiveDirectory servers

Due to restrictions on Microsoft ActiveDirectory servers regarding the number of entries returned by generic search routines and due to the structure of the ActiveDirectory schema repository, schema browsing is currently **not** available for Microsoft ActiveDirectory servers.

ZendLdapNodeSchemaActiveDirectory does not provide any additional methods.

Table 83.8: Zend\Ldap\Node\Schema\AttributeType\ActiveDirectory API

Zend\Ldap\Node\Schema\AttributeType\ActiveDirectory does not provide any additional methods.

Table 83.9: Zend\Ldap\Node\Schema\ObjectClass\ActiveDirectory API

Zend\Ldap\Node\Schema\ObjectClass\ActiveDirectory does not provide any additional methods.

CHAPTER 84

Zend\Ldap\Ldif\Encoder

Table 84.1: Zend\Ldap\Ldif\Encoder API

Method	Description
array decode(string $string)	Decodes the string $string into an array of LDIF items.
string encode(scalar|array|Zend\Ldap\Node $value, array $options)	Encode $value into a LDIF representation. $options is an array that may contain the following keys: ‘sort’ Sort the given attributes with dn following objectClass and following all other attributes sorted alphabetically. TRUE by default. ‘version’ The LDIF format version. 1 by default. ‘wrap’ The line-length. 78 by default to conform to the LDIF specification.

CHAPTER 85

Usage Scenarios

Authentication scenarios

OpenLDAP

ActiveDirectory

Basic CRUD operations

Retrieving data from the LDAP

Getting an entry by its DN

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
/*
$hm is an array of the following structure
array(
    'dn'          => 'cn=Hugo Müller,ou=People,dc=my,dc=local',
    'cn'          => array('Hugo Müller'),
    'sn'          => array('Müller'),
    'objectclass' => array('inetOrgPerson', 'top'),
    ...
)
*/
```

Check for the existence of a given DN

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$isThere = $ldap->exists('cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Count children of a given DN

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$childrenCount = $ldap->countChildren(
                            'cn=Hugo Müller,ou=People,dc=my,dc=local');
```

Searching the LDAP tree

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$result = $ldap->search('(objectclass=*)',
                        'ou=People,dc=my,dc=local',
                        Zend\Ldap\Ldap::SEARCH_SCOPE_ONE);
foreach ($result as $item) {
    echo $item["dn"] . ': ' . $item['cn'][0] . PHP_EOL;
}
```

Adding data to the LDAP

Add a new entry to the LDAP

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$entry = array();
Zend\Ldap\Attribute::setAttribute($entry, 'cn', 'Hans Meier');
Zend\Ldap\Attribute::setAttribute($entry, 'sn', 'Meier');
Zend\Ldap\Attribute::setAttribute($entry, 'objectClass', 'inetOrgPerson');
$ldap->add('cn=Hans Meier,ou=People,dc=my,dc=local', $entry);
```

Deleting from the LDAP

Delete an existing entry from the LDAP

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ldap->delete('cn=Hans Meier,ou=People,dc=my,dc=local');
```

Updating the LDAP

Update an existing entry on the LDAP

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$hm = $ldap->getEntry('cn=Hugo Müller,ou=People,dc=my,dc=local');
Zend\Ldap\Attribute::setAttribute($hm, 'mail', 'mueller@my.local');
Zend\Ldap\Attribute::setPassword($hm,
                                 'newPa$$w0rd',
                                 Zend\Ldap\Attribute::PASSWORD_HASH_SHA1);
$ldap->update('cn=Hugo Müller,ou=People,dc=my,dc=local', $hm);
```

Extended operations

Copy and move entries in the LDAP

Copy a LDAP entry recursively with all its descendants

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ldap->copy('cn=Hugo Müller,ou=People,dc=my,dc=local',
            'cn=Hans Meier,ou=People,dc=my,dc=local',
            true);
```

Move a LDAP entry recursively with all its descendants to a different subtree

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ldap->moveToSubtree('cn=Hugo Müller,ou=People,dc=my,dc=local',
                     'ou=Dismissed,dc=my,dc=local',
                     true);
```

CHAPTER 86

Tools

Creation and modification of DN strings

Using the filter API to create search filters

Create simple LDAP filters

```
$f1  = Zend\Ldap\Filter::equals('name', 'value');          // (name=value)
$f2  = Zend\Ldap\Filter::begins('name', 'value');          // (name=value*)
$f3  = Zend\Ldap\Filter::ends('name', 'value');            // (name=*value)
$f4  = Zend\Ldap\Filter::contains('name', 'value');        // (name=*value*)
$f5  = Zend\Ldap\Filter::greater('name', 'value');         // (name>value)
$f6  = Zend\Ldap\Filter::greaterOrEqual('name', 'value'); // (name>=value)
$f7  = Zend\Ldap\Filter::less('name', 'value');            // (name<value)
$f8  = Zend\Ldap\Filter::lessOrEqual('name', 'value');     // (name<=value)
$f9  = Zend\Ldap\Filter::approx('name', 'value');          // (name~=value)
$f10 = Zend\Ldap\Filter::any('name');                      // (name=*)
```

Create more complex LDAP filters

```
$f1 = Zend\Ldap\Filter::ends('name', 'value')->negate(); // (!(name=*value))

$f2 = Zend\Ldap\Filter::equals('name', 'value');
$f3 = Zend\Ldap\Filter::begins('name', 'value');
$f4 = Zend\Ldap\Filter::ends('name', 'value');

// (&(name=value)(name=value*)(name=*value))
$f5 = Zend\Ldap\Filter::andFilter($f2, $f3, $f4);

```

```
// (|(name=value)(name=value*)(name=*value))
$f6 = Zend\Ldap\Filter::orFilter($f2, $f3, $f4);
```

Modify LDAP entries using the Attribute API

CHAPTER 87

Object oriented access to the LDAP tree using Zend\Ldap\Node

Basic CRUD operations

Retrieving data from the LDAP

Getting a node by its DN

Searching a node's subtree

Adding a new node to the LDAP

Deleting a node from the LDAP

Updating a node on the LDAP

Extended operations

Copy and move nodes in the LDAP

Tree traversal

Traverse LDAP tree recursively

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$ldap->bind();
$ri = new RecursiveIteratorIterator($ldap->getBaseNode(),
                                    RecursiveIteratorIterator::SELF_FIRST);
```

```
6 foreach ($ri as $rdn => $n) {
7     var_dump($n);
8 }
```

CHAPTER 88

Getting information from the LDAP server

RootDSE

See the following documents for more information on the attributes contained within the RootDSE for a given *LDAP* server.

- OpenLDAP
- Microsoft ActiveDirectory
- Novell eDirectory

Getting hands on the RootDSE

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$rootdse = $ldap->getRootDse();
$serverType = $rootdse->getServerType();
```

Schema Browsing

Getting hands on the server schema

```
$options = array(/* ... */);
$ldap = new Zend\Ldap\Ldap($options);
$schema = $ldap->getSchema();
$classes = $schema->getObjectClasses();
```

OpenLDAP

ActiveDirectory

Note: Schema browsing on ActiveDirectory servers

Due to restrictions on Microsoft ActiveDirectory servers regarding the number of entries returned by generic search routines and due to the structure of the ActiveDirectory schema repository, schema browsing is currently **not** available for Microsoft ActiveDirectory servers.

CHAPTER 89

Serializing LDAP data to and from LDIF

Serialize a LDAP entry to LDIF

```
$data = array(
    'dn'                          => 'uid=rogasawara,ou=,o=Airius',
    'objectclass'                 => array('top',
                                           'person',
                                           'organizationalPerson',
                                           'inetOrgPerson'),
    'uid'                         => array('rogasawara'),
    'mail'                        => array('rogasawara@airius.co.jp'),
    'givenname;lang-ja'           => array(''),
    'sn;lang-ja'                  => array(''),
    'cn;lang-ja'                  => array(' '),
    'title;lang-ja'               => array(' '),
    'preferredlanguage'           => array('ja'),
    'givenname'                   => array(''),
    'sn'                          => array(''),
    'cn'                          => array(' '),
    'title'                       => array(' '),
    'givenname;lang-ja;phonetic' => array(''),
    'sn;lang-ja;phonetic'         => array(''),
    'cn;lang-ja;phonetic'         => array(' '),
    'title;lang-ja;phonetic'      => array(' '),
    'givenname;lang-en'           => array('Rodney'),
    'sn;lang-en'                  => array('Ogasawara'),
    'cn;lang-en'                  => array('Rodney Ogasawara'),
    'title;lang-en'               => array('Sales, Director'),
);
$ldif = Zend\Ldap\Ldif\Encoder::encode($data, array('sort' => false,
                                                    'version' => null));
/*
$ldif contains:
dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6YOoLG89QWlyaXVz
```

```
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: rogasawara
mail: rogasawara@airius.co.jp
givenname;lang-ja:: 44Ot44OJ44OL44O8
sn;lang-ja:: 5bCP56yg5Y6f
cn;lang-ja:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title;lang-ja:: 5Za25qWt6YOoIOmDqOmVtw==
preferredlanguage: ja
givenname:: 44Ot44OJ44OL44O8
sn:: 5bCP56yg5Y6f
cn:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title:: 5Za25qWt6YOoIOmDqOmVtw==
givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
givenname;lang-en: Rodney
sn;lang-en: Ogasawara
cn;lang-en: Rodney Ogasawara
title;lang-en: Sales, Director
*/
```

Deserialize a LDIF string into a LDAP entry

```
$ldif = "dn:: dWlkPXJvZ2FzYXdhcmEsb3U95Za25qWt6YOoLG89QWlyaXVz
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
uid: rogasawara
mail: rogasawara@airius.co.jp
givenname;lang-ja:: 44Ot44OJ44OL44O8
sn;lang-ja:: 5bCP56yg5Y6f
cn;lang-ja:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title;lang-ja:: 5Za25qWt6YOoIOmDqOmVtw==
preferredlanguage: ja
givenname:: 44Ot44OJ44OL44O8
sn:: 5bCP56yg5Y6f
cn:: 5bCP56yg5Y6fIOODreODieODi+ODvA==
title:: 5Za25qWt6YOoIOmDqOmVtw==
givenname;lang-ja;phonetic:: 44KN44Gp44Gr44O8
sn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJ
cn;lang-ja;phonetic:: 44GK44GM44GV44KP44KJIOOCjeOBqeOBq+ODvA==
title;lang-ja;phonetic:: 44GI44GE44GO44KH44GG44G2IOOBtuOBoeOCh+OBhg==
givenname;lang-en: Rodney
sn;lang-en: Ogasawara
cn;lang-en: Rodney Ogasawara
title;lang-en: Sales, Director";
$data = Zend\Ldap\Ldif\Encoder::decode($ldif);
/*
$data = array(
    'dn'                          => 'uid=rogasawara,ou=,o=Airius',
```

```
    'objectclass'                => array('top',
                                          'person',
                                          'organizationalPerson',
                                          'inetOrgPerson'),
    'uid'                        => array('rogasawara'),
    'mail'                       => array('rogasawara@airius.co.jp'),
    'givenname;lang-ja'          => array(''),
    'sn;lang-ja'                 => array(''),
    'cn;lang-ja'                 => array(' '),
    'title;lang-ja'              => array(' '),
    'preferredlanguage'          => array('ja'),
    'givenname'                  => array(''),
    'sn'                         => array(''),
    'cn'                         => array(' '),
    'title'                      => array(' '),
    'givenname;lang-ja;phonetic' => array(''),
    'sn;lang-ja;phonetic'        => array(''),
    'cn;lang-ja;phonetic'        => array(' '),
    'title;lang-ja;phonetic'     => array(' '),
    'givenname;lang-en'          => array('Rodney'),
    'sn;lang-en'                 => array('Ogasawara'),
    'cn;lang-en'                 => array('Rodney Ogasawara'),
    'title;lang-en'              => array('Sales, Director'),
);
*/
```

CHAPTER 90

The AutoloaderFactory

Overview

Starting with version 2.0, Zend Framework now offers multiple autoloader strategies. Often, it will be useful to employ multiple autoloading strategies; as an example, you may have a class map for your most used classes, but want to use a PSR-0 style autoloader for 3rd party libraries.

While you could potentially manually configure these, it may be more useful to define the autoloader configuration somewhere and cache it. For these cases, the `AutoloaderFactory` will be useful.

Quick Start

Configuration may be stored as a PHP array, or in some form of configuration file. As an example, consider the following PHP array:

```
$config = array(
    'Zend\Loader\ClassMapAutoloader' => array(
        'application' => APPLICATION_PATH . '/.classmap.php',
        'zf'          => APPLICATION_PATH . '/../library/Zend/.classmap.php',
    ),
    'Zend\Loader\StandardAutoloader' => array(
        'namespaces' => array(
            'Phly\Mustache' => APPLICATION_PATH . '/../library/Phly/Mustache',
            'Doctrine'      => APPLICATION_PATH . '/../library/Doctrine',
        ),
    ),
);
```

An equivalent INI-style configuration might look like the following:

```
Zend\Loader\ClassMapAutoloader.application = APPLICATION_PATH "/.classmap.php"
Zend\Loader\ClassMapAutoloader.zf          = APPLICATION_PATH "/../library/Zend/.
↪classmap.php"
```

```
Zend\Loader\StandardAutoloader.namespaces.Phly\Mustache = APPLICATION_PATH "/../
↪library/Phly/Mustache"
Zend\Loader\StandardAutoloader.namespaces.Doctrine       = APPLICATION_PATH "/../
↪library/Doctrine"
```

Once you have your configuration in a PHP array, you simply pass it to the `AutoloaderFactory`.

```
// This example assumes ZF is on your include_path.
// You could also load the factory class from a path relative to the
// current script, or via an absolute path.
require_once 'Zend/Loader/AutoloaderFactory.php';
Zend\Loader\AutoloaderFactory::factory($config);
```

The `AutoloaderFactory` will instantiate each autoloader with the given options, and also call its `register()` method to register it with the SPL autoloader.

Configuration Options

AutoloaderFactory Options

$options The `AutoloaderFactory` expects an associative array or `Traversable` object. Keys should be valid autoloader class names, and the values should be the options that should be passed to the class constructor.

Internally, the `AutoloaderFactory` checks to see if the autoloader class referenced exists. If not, it will use the *StandardAutoloader* to attempt to load the class via the `include_path` (or, in the case of "Zend"-namespaced classes, using the Zend Framework library path). If the class is not found, or does not implement the *SplAutoloader* interface, an exception will be raised.

Available Methods

factory Instantiate and register autoloaders `factory($options)`

factory() This method is **static**, and is used to instantiate autoloaders and register them with the SPL autoloader. It expects either an array or `Traversable` object as denoted in the *Options section*.

getRegisteredAutoloaders Retrieve a list of all autoloaders registered using the factory `getRegisteredAutoloaders()`

getRegisteredAutoloaders() This method is **static**, and may be used to retrieve a list of all autoloaders registered via the `factory()` method. It returns simply an array of autoloader instances.

Examples

Please see the *Quick Start* for a detailed example.

CHAPTER 91

The PluginClassLoader

Overview

Resolving plugin names to class names is a common requirement within Zend Framework applications. The `PluginClassLoader` implements the interfaces *PluginClassLocator*, *ShortNameLocator*, and `IteratorAggregate`, providing a simple mechanism for aliasing plugin names to classnames for later retrieval.

While it can act as a standalone class, it is intended that developers will extend the class to provide a per-component plugin map. This allows seeding the map with the most often-used plugins, while simultaneously allowing the end-user to overwrite existing or register new plugins.

Additionally, `PluginClassLoader` provides the ability to statically seed all new instances of a given `PluginClassLoader` or one of its extensions (via Late Static Binding). If your application will always call for defining or overriding particular plugin maps on given `PluginClassLoader` extensions, this is a powerful capability.

Quick Start

Typical use cases involve simply instantiating a `PluginClassLoader`, seeding it with one or more plugin/class name associations, and then using it to retrieve the class name associated with a given plugin name.

```
use Zend\View\HelperLoader;

// Provide a global map, or override defaults:
HelperLoader::addStaticMap(array(
    'url' => 'My\Custom\UrlHelper',
));

// Instantiate the loader:
$loader = new Zend\View\HelperLoader();

// Register a new plugin:
$loader->registerPlugin('bugUrl', 'My\Custom\BugUrlHelper');
```

```
13
14 // Load/retrieve the associated plugin class:
15 $class = $loader->load('url'); // 'My\Custom\UrlHelper'
```

Note: Case Sensitivity

The `PluginClassLoader` is designed to do case-insensitive plugin name lookups. While the above example defines a "bugUrl" plugin name, internally, this will be stored as simply "bugurl". If another plugin is registered with simply a different word case, it will overwrite this entry.

Configuration Options

PluginClassLoader Options

$map The constructor may take a single option, an array or `Traversable` object of key/value pairs corresponding to a plugin name and class name, respectively.

Available Methods

__construct Instantiate and initialize the loader `__construct($map = null)`

__construct() The constructor is used to instantiate and intialize the plugin class loader. If passed a string, an array, or a `Traversable` object, it will pass this to the *registerPlugins()* method in order to seed (or overwrite) the plugin class map.

addStaticMap Statically seed the plugin loader map `addStaticMap($map)`

addStaticMap() Static method for globally pre-seeding the loader with a class map. It accepts either an array or `Traversable` object of plugin name/class name pairs.

When using this method, be certain you understand the precedence in which maps will be merged; in decreasing order of preference:

- Manually registered plugin/class name pairs (e.g., via *registerPlugin()* or *registerPlugins()*).
- A map passed to the constructor .
- The static map.
- The map defined within the class itself.

Also, please note that calling the method will **not** affect any instances already created.

registerPlugin Register a plugin/class association `registerPlugin($shortName, $className)`

registerPlugin() Defined by the *PluginClassLocator* interface. Expects two string arguments, the plugin `$shortName`, and the class `$className` which it represents.

registerPlugins Register many plugin/class associations at once `registerPlugins($map)`

registerPlugins() Expects a string, an array or `Traversable` object of plugin name/class name pairs representing a plugin class map.

If a string argument is provided, `registerPlugins()` assumes this is a class name. If the class does not exist, an exception will be thrown. If it does, it then instantiates the class and checks to see whether or not it implements `Traversable`.

unregisterPlugin Remove a plugin/class association from the map `unregisterPlugin($shortName)`

unregisterPlugin() Defined by the `PluginClassLocator` interface; remove a plugin/class association from the plugin class map.

getRegisteredPlugins Return the complete plugin class map `getRegisteredPlugins()`

getRegisteredPlugins() Defined by the `PluginClassLocator` interface; return the entire plugin class map as an array.

isLoaded Determine if a given plugin name resolves `isLoaded($name)`

isLoaded() Defined by the `ShortNameLocator` interface; determine if the given plugin has been resolved to a class name.

getClassName Return the class name to which a plugin resolves `getClassName($name)`

getClassName() Defined by the `ShortNameLocator` interface; return the class name to which a plugin name resolves.

load Resolve a plugin name `load($name)`

load() Defined by the `ShortNameLocator` interface; attempt to resolve a plugin name to a class name. If successful, returns the class name; otherwise, returns a boolean `false`.

getIterator Return iterator capable of looping over plugin class map `getIterator()`

getIterator() Defined by the `IteratorAggregate` interface; allows iteration over the plugin class map. This can come in useful for using `PluginClassLoader` instances to other `PluginClassLoader` instances in order to merge maps.

Examples

Using Static Maps

It's often convenient to provide global overrides or additions to the maps in a `PluginClassLoader` instance. This can be done using the `addStaticMap()` method:

```
use Zend\Loader\PluginClassLoader;

PluginClassLoader::addStaticMap(array(
    'url' => 'Zend\View\Helper\Url',
));
```

Any later instances created will now have this map defined, allowing you to load that plugin.

```
use Zend\Loader\PluginClassLoader;

$loader = new PluginClassLoader();
$helper = $loader->load('url'); // Zend\View\Helper\Url
```

Creating a pre-loaded map

In many cases, you know exactly which plugins you may be drawing upon on a regular basis, and which classes they will refer to. In this case, simply extend the `PluginClassLoader` and define the map within the extending class.

```
namespace My\Plugins;

use Zend\Loader\PluginClassLoader;

class PluginLoader extends PluginClassLoader
{
    /**
     * @var array Plugin map
     */
    protected $plugins = array(
        'foo'    => 'My\Plugins\Foo',
        'bar'    => 'My\Plugins\Bar',
        'foobar' => 'My\Plugins\FooBar',
    );
}
```

At this point, you can simply instantiate the map and use it.

```
$loader = new My\Plugins\PluginLoader();
$class  = $loader->load('foobar'); // My\Plugins\FooBar
```

`PluginClassLoader` makes use of late static binding, allowing per-class static maps. If you want to allow defining a *static map* specific to this extending class, simply declare a protected static `$staticMap` property:

```
namespace My\Plugins;

use Zend\Loader\PluginClassLoader;

class PluginLoader extends PluginClassLoader
{
    protected static $staticMap = array();

    // ...
}
```

To inject the static map, use the extending class' name to call the static `addStaticMap()` method.

```
PluginLoader::addStaticMap(array(
    'url' => 'Zend\View\Helper\Url',
));
```

Extending a plugin map using another plugin map

In some cases, a general map class may already exist; as an example, most components in Zend Framework that utilize a plugin broker have an associated `PluginClassLoader` extension defining the plugins available for that component within the framework. What if you want to define some additions to these? Where should that code go?

One possibility is to define the map in a configuration file, and then inject the configuration into an instance of the plugin loader. This is certainly trivial to implement, but removes the code defining the plugin map from the library.

An alternate solution is to define a new plugin map class. The class name or an instance of the class may then be passed to the constructor or `registerPlugins()`.

```php
namespace My\Plugins;

use Zend\Loader\PluginClassLoader;
use Zend\View\Helper\HelperLoader;

class PluginLoader extends PluginClassLoader
{
    /**
     * @var array Plugin map
     */
    protected $plugins = array(
        'foo'    => 'My\Plugins\Foo',
        'bar'    => 'My\Plugins\Bar',
        'foobar' => 'My\Plugins\FooBar',
    );
}

// Inject in constructor:
$loader = new HelperLoader('My\Plugins\PluginLoader');
$loader = new HelperLoader(new PluginLoader());

// Or via registerPlugins():
$loader->registerPlugins('My\Plugins\PluginLoader');
$loader->registerPlugins(new PluginLoader());
```

CHAPTER 92

The ShortNameLocator Interface

Overview

Within Zend Framework applications, it's often expedient to provide a mechanism for using class aliases instead of full class names to load adapters and plugins, or to allow using aliases for the purposes of slipstreaming alternate implementations into the framework.

In the first case, consider the adapter pattern. It's often unwieldy to utilize a full class name (e.g., `Zend\Cloud\DocumentService\Adapter\SimpleDb`); using the short name of the adapter, `SimpleDb`, would be much simpler.

In the second case, consider the case of helpers. Let us assume we have a "url" helper; you may find that while the shipped helper does 90% of what you need, you'd like to extend it or provide an alternate implementation. At the same time, you don't want to change your code to reflect the new helper. In this case, a short name allows you to alias an alternate class to utilize.

Classes implementing the `ShortNameLocator` interface provide a mechanism for resolving a short name to a fully qualified class name; how they do so is left to the implementers, and may combine strategies defined by other interfaces, such as *PluginClassLocator* or *PrefixPathMapper*.

Quick Start

Implementing a `ShortNameLocator` is trivial, and requires only three methods, as shown below.

```
1  namespace Zend\Loader;
2
3  interface ShortNameLocator
4  {
5      public function isLoaded($name);
6      public function getClassName($name);
7      public function load($name);
8  }
```

Configuration Options

This component defines no configuration options, as it is an interface.

Available Methods

isLoaded Is the requested plugin loaded? `isLoaded($name)`

> **isLoaded()** Implement this method to return a boolean indicating whether or not the class has been able to resolve the plugin name to a class.

getClassName Get the class name associated with a plugin name `getClassName($name)`

> **getClassName()** Implement this method to return the class name associated with a plugin name.

load Resolve a plugin to a class name `load($name)`

> **load()** This method should resolve a plugin name to a class name.

Examples

Please see the *Quick Start* for the interface specification.

CHAPTER 93

The PluginClassLocator interface

Overview

The `PluginClassLocator` interface describes a component capable of maintaining an internal map of plugin names to actual class names. Classes implementing this interface can register and unregister plugin/class associations, and return the entire map.

Quick Start

Classes implementing the `PluginClassLocator` need to implement only three methods, as illustrated below.

```
namespace Zend\Loader;

interface PluginClassLocator
{
    public function registerPlugin($shortName, $className);
    public function unregisterPlugin($shortName);
    public function getRegisteredPlugins();
}
```

Configuration Options

This component defines no configuration options, as it is an interface.

Available Methods

registerPlugin Register a mapping of plugin name to class name `registerPlugin($shortName, $className)`

registerPlugin() Implement this method to add or overwrite plugin name/class name associations in the internal plugin map. `$shortName` will be aliased to `$className`.

unregisterPlugin Remove a plugin/class name association `unregisterPlugin($shortName)`

unregisterPlugin() Implement this to allow removing an existing plugin mapping corresponding to `$shortName`.

getRegisteredPlugins Retrieve the map of plugin name/class name associations `getRegisteredPlugins()`

getRegisteredPlugins() Implement this to allow returning the plugin name/class name map.

Examples

Please see the *Quick Start* for the interface specification.

CHAPTER 94

The SplAutoloader Interface

Overview

While any valid PHP callback may be registered with `spl_autoload_register()`, Zend Framework autoloaders often provide more flexibility by being stateful and allowing configuration. To provide a common interface, Zend Framework provides the `SplAutoloader` interface.

Objects implementing this interface provide a standard mechanism for configuration, a method that may be invoked to attempt to load a class, and a method for registering with the SPL autoloading mechanism.

Quick Start

To create your own autoloading mechanism, simply create a class implementing the `SplAutoloader` interface (you may review the methods defined in the *Methods section*). As a simple example, consider the following autoloader, which will look for a class file named after the class within a list of registered directories.

```
1  namespace Custom;
2
3  use Zend\Loader\SplAutoloader;
4
5  class ModifiedIncludePathAutoloader implements SplAutoloader
6  {
7      protected $paths = array();
8
9      public function __construct($options = null)
10     {
11         if (null !== $options) {
12             $this->setOptions($options);
13         }
14     }
15
16     public function setOptions($options)
```

```
    {
        if (!is_array($options) && !($options instanceof \Traversable)) {
            throw new \InvalidArgumentException();
        }

        foreach ($options as $path) {
            if (!in_array($path, $this->paths)) {
                $this->paths[] = $path;
            }
        }
        return $this;
    }

    public function autoload($classname)
    {
        $filename = $classname . '.php';
        foreach ($this->paths as $path) {
            $test = $path . DIRECTORY_SEPARATOR . $filename;
            if (file_exists($test)) {
                return include($test);
            }
        }
        return false;
    }

    public function register()
    {
        spl_autoload_register(array($this, 'autoload'));
    }
}
```

Configuration Options

This component defines no configuration options, as it is an interface.

Available Methods

__construct Initialize and configure an autoloader `__construct($options = null)`

> **Constructor** Autoloader constructors should optionally receive configuration options. Typically, if received, these will be passed to the `setOptions()` method to process.

setOptions Configure the autoloader state `setOptions($options)`

> **setOptions()** Used to configure the autoloader. Typically, it should expect either an array or a `Traversable` object, though validation of the options is left to implementation. Additionally, it is recommended that the method return the autoloader instance in order to implement a fluent interface.

autoload Attempt to resolve a class name to the file defining it `autoload($classname)`

> **autoload()** This method should be used to resolve a class name to the file defining it. When a positive match is found, return the class name; otherwise, return a boolean false.

register Register the autoloader with the SPL autoloader `register()`

register() Should be used to register the autoloader instance with `spl_autoload_register()`. Invariably, the method should look like the following:

```
public function register()
{
    spl_autoload_register(array($this, 'autoload'));
}
```

Examples

Please see the *Quick Start* for a complete example.

CHAPTER 95

The ClassMapAutoloader

Overview

The `ClassMapAutoloader` is designed with performance in mind. The idea behind it is simple: when asked to load a class, see if it's in the map, and, if so, load the file associated with the class in the map. This avoids unnecessary filesystem operations, and can also ensure the autoloader "plays nice" with opcode caches and PHP's realpath cache.

In order to use the `ClassMapAutoloader`, you first need class maps. Zend Framework ships with a class map per component or, if you grabbed the entire ZF distribution, a class map for the entire Zend Framework. These maps are typically in a file named `.classmap.php` within either the "Zend" directory, or an individual component's source directory.

Zend Framework also provides a tool for generating these class maps; you can find it in `bin/classmap_generator.php` of the distribution. Full documentation of this too is provided in :ref:' <zend.loader.classmap-generator>'.

Quick Start

The first step is to generate a class map file. You may run this over any directory containing source code anywhere underneath it.

```
php classmap_generator.php Some/Directory/
```

This will create a file named `Some/Directory/.classmap.php`, which is a PHP file returning an associative array that represents the class map.

Within your code, you will now instantiate the `ClassMapAutoloader`, and provide it the location of the map.

```
// This example assumes ZF is on your include_path.
// You could also load the autoloader class from a path relative to the
// current script, or via an absolute path.
require_once 'Zend/Loader/ClassMapAutoloader.php';
$loader = new Zend\Loader\ClassMapAutoloader();
```

```
6
7  // Register the class map:
8  $loader->registerAutoloadMap('Some/Directory/.classmap.php');
9
10 // Register with spl_autoload:
11 $loader->register();
```

At this point, you may now use any classes referenced in your class map.

Configuration Options

The `ClassMapAutoloader` defines the following options.

ClassMapAutoloader Options

$options The `ClassMapAutoloader` expects an array of options, where each option is either a filename referencing a class map, or an associative array of class name/filename pairs.

As an example:

```
1  // Configuration defining both a file-based class map, and an array map
2  $config = array(
3      __DIR__ . '/library/.classmap.php', // file-based class map
4      array(                              // array class map
5          'Application\Bootstrap' => __DIR__ . '/application/Bootstrap.php',
6          'Test\Bootstrap'        => __DIR__ . '/tests/Bootstrap.php',
7      ),
8  );
```

Available Methods

__construct Initialize and configure the object `__construct($options = null)`

Constructor Used during instantiation of the object. Optionally, pass options, which may be either an array or `Traversable` object; this argument will be passed to *setOptions()*.

setOptions Configure the autoloader `setOptions($options)`

setOptions() Configures the state of the autoloader, including registering class maps. Expects an array or `Traversable` object; the argument will be passed to *registerAutoloadMaps()*.

registerAutoloadMap Register a class map `registerAutoloadMap($map)`

registerAutoloadMap() Registers a class map with the autoloader. `$map` may be either a string referencing a PHP script that returns a class map, or an array defining a class map.

More than one class map may be registered; each will be merged with the previous, meaning it's possible for a later class map to overwrite entries from a previously registered map.

registerAutoloadMaps Register multiple class maps at once `registerAutoloadMaps($maps)`

registerAutoloadMaps() Register multiple class maps with the autoloader. Expects either an array or `Traversable` object; it then iterates over the argument and passes each value to *registerAutoloadMap()*.

getAutoloadMap Retrieve the current class map `getAutoloadMap()`

getAutoloadMap() Retrieves the state of the current class map; the return value is simply an array.

autoload Attempt to load a class. `autoload($class)`

autoload() Attempts to load the class specified. Returns a boolean `false` on failure, or a string indicating the class loaded on success.

register Register with spl_autoload. `register()`

register() Registers the `autoload()` method of the current instance with `spl_autoload_register()`.

Examples

Using configuration to seed ClassMapAutoloader

Often, you will want to configure your `ClassMapAutoloader`. These values may come from a configuration file, a cache (such as ShMem or memcached), or a simple PHP array. The following is an example of a PHP array that could be used to configure the autoloader:

```
// Configuration defining both a file-based class map, and an array map
$config = array(
APPLICATION_PATH . '/../library/.classmap.php', // file-based class map
    array(                                      // array class map
        'Application\Bootstrap' => APPLICATION_PATH . '/Bootstrap.php',
        'Test\Bootstrap'        => APPLICATION_PATH . '/../tests/Bootstrap.php',
    ),
);
```

An eqivalent INI style configuration might look like this:

```
classmap.library = APPLICATION_PATH "/../library/.classmap.php"
classmap.resources.Application\Bootstrap = APPLICATION_PATH "/Bootstrap.php"
classmap.resources.Test\Bootstrap = APPLICATION_PATH "/../tests/Bootstrap.php"
```

Once you have your configuration, you can pass it either to the constructor of the `ClassMapAutoloader`, to its `setOptions()` method, or to `registerAutoloadMaps()`.

```
/* The following are all equivalent */

// To the constructor:
$loader = new Zend\Loader\ClassMapAutoloader($config);

// To setOptions():
$loader = new Zend\Loader\ClassMapAutoloader();
$loader->setOptions($config);

// To registerAutoloadMaps():
$loader = new Zend\Loader\ClassMapAutoloader();
$loader->registerAutoloadMaps($config);
```

CHAPTER 96

The StandardAutoloader

Overview

`Zend\Loader\StandardAutoloader` is designed as a PSR-0-compliant autoloader. It assumes a 1:1 mapping of the namespace+classname to the filesystem, wherein namespace separators and underscores are translated to directory separators. A simple statement that illustrates how resolution works is as follows:

```
1 $filename = str_replace(array('_', '\\'), DIRECTORY_SEPARATOR, $classname)
2           . '.php';
```

Previous incarnations of PSR-0-compliant autoloaders in Zend Framework have relied upon the `include_path` for file lookups. This has led to a number of issues:

- Due to the use of `include`, if the file is not found, a warning is raised – even if another autoloader is capable of resolving the class later.
- Documenting how to setup the `include_path` has proven to be a difficult concept to convey.
- If multiple Zend Framework installations exist on the `include_path`, the first one on the path wins – even if that was not the one the developer intended.

To solve these problems, the `StandardAutoloader` by default requires that you explicitly register namespace/path pairs (or vendor prefix/path pairs), and will only load a file if it exists within the given path. Multiple pairs may be provided.

As a measure of last resort, you may also use the `StandardAutoloader` as a "fallback" autoloader – one that will look for classes of any namespace or vendor prefix on the `include_path`. This practice is not recommended, however, due to performance implications.

Finally, as with all autoloaders in Zend Framework, the `StandardAutoloader` is capable of registering itself with PHP's SPL autoloader registry.

Note: Vocabulary: Namespaces vs. Vendor Prefixes

In terms of autloading, a "namespace" corresponds to PHP's own definition of namespaces in PHP versions 5.3 and above.

A "vendor prefix" refers to the practice, popularized in PHP versions prior to 5.3, of providing a pseudo-namespace in the form of underscore-separated words in class names. As an example, the class `Phly_Couch_Document` uses a vendor prefix of "Phly", and a component prefix of "Phly_Couch" – but it is a class sitting in the global namespace within PHP 5.3.

The `StandardAutoloader` is capable of loading either namespaced or vendor prefixed class names, but treats them separately when attempting to match them to an appropriate path.

Quick Start

Basic use of the `StandardAutoloader` requires simply registering namespace/path pairs. This can either be done at instantiation, or via explicit method calls after the object has been initialized. Calling `register()` will register the autoloader with the SPL autoloader registry.

If the option key 'autoregister_zf' is set to true then the class will register the "Zend" namespace to the directory above where its own classfile is located on the filesystem.

Manual Configuration

```
// This example assumes ZF is on your include_path.
// You could also load the autoloader class from a path relative to the
// current script, or via an absolute path.
require_once 'Zend/Loader/StandardAutoloader.php';
$loader = new Zend\Loader\StandardAutoloader(array('autoregister_zf' => true));

// Register the "Phly" namespace:
$loader->registerNamespace('Phly', APPLICATION_PATH . '/../library/Phly');

// Register the "Scapi" vendor prefix:
$loader->registerPrefix('Scapi', APPLICATION_PATH . '/../library/Scapi');

// Optionally, specify the autoloader as a "fallback" autoloader;
// this is not recommended.
$loader->setFallbackAutoloader(true);

// Register with spl_autoload:
$loader->register();
```

Configuration at Instantiation

The `StandardAutoloader` may also be configured at instantiation. Please note:

- The argument passed may be either an array or a `Traversable` object (such as a `Zend\Config` object.
- The argument passed is also a valid argument for passing to the `setOptions()` method.

The following is equivalent to the previous example.

```
require_once 'Zend/Loader/StandardAutoloader.php';
$loader = new Zend\Loader\StandardAutoloader(array(
    'autoregister_zf' => true,
    'namespaces' => array(
        'Phly' => APPLICATION_PATH . '/../library/Phly',
```

```
    ),
    'prefixes' => array(
        'Scapi' => APPLICATION_PATH . '/../library/Scapi',
    ),
    'fallback_autoloader' => true,
));

// Register with spl_autoload:
$loader->register();
```

Configuration Options

The `StandardAutoloader` defines the following options.

StandardAutoloader Options

namespaces An associative array of namespace/path pairs. The path should be an absolute path or path relative to the calling script, and contain only classes that live in that namespace (or its subnamespaces). By default, the "Zend" namespace is registered, pointing to the arent directory of the file defining the `StandardAutoloader`.

prefixes An associative array of vendor prefix/path pairs. The path should be an absolute path or path relative to the calling script, and contain only classes that begin with the provided vendor prefix.

fallback_autoloader A boolean value indicating whether or not this instance should act as a "fallback" autoloader (i.e., look for classes of any namespace or vendor prefix on the `include_path`). By default, `false`.

autoregister_zf An boolean value indicating that the class should register the "Zend" namespace to the directory above where its own classfile is located on the filesystem.

Available Methods

__construct Initialize a new instance of the object `__construct($options = null)`

Constructor Takes an optional `$options` argument. This argument may be an associative array or `Traversable` object. If not null, the argument is passed to *setOptions()*.

setOptions Set object state based on provided options. `setOptions($options)`

setOptions() Takes an argument of either an associative array or `Traversable` object. Recognized keys are detailed under :ref:' <zend.loader.standard-autoloader.options>', with the following behaviors:

- The `namespaces` value will be passed to *registerNamespaces()*.
- The `prefixes` value will be passed to *registerPrefixes()*.
- The `fallback_autoloader` value will be passed to *setFallbackAutoloader()*.

setFallbackAutoloader Enable/disable fallback autoloader status `setFallbackAutoloader($flag)`

setFallbackAutoloader() Takes a boolean flag indicating whether or not to act as a fallback autoloader when registered with the SPL autoloader.

isFallbackAutoloader Query fallback autoloader status `isFallbackAutoloader()`

isFallbackAutoloader() Indicates whether or not this instance is flagged as a fallback autoloader.

registerNamespace Register a namespace with the autoloader `registerNamespace($namespace, $directory)`

registerNamespace() Register a namespace with the autoloader, pointing it to a specific directory on the filesystem for class resolution. For classes matching that initial namespace, the autoloader will then perform lookups within that directory.

registerNamespaces Register multiple namespaces with the autoloader `registerNamespaces($namespaces)`

registerNamespaces() Accepts either an array or `Traversable` object. It will then iterate through the argument, and pass each item to *registerNamespace()*.

registerPrefix Register a vendor prefix with the autoloader. `registerPrefix($prefix, $directory)`

registerPrefix() Register a vendor prefix with the autoloader, pointing it to a specific directory on the filesystem for class resolution. For classes matching that initial vendor prefix, the autoloader will then perform lookups within that directory.

registerPrefixes Register many vendor prefixes with the autoloader `registerPrefixes($prefixes)`

registerPrefixes() Accepts either an array or `Traversable` object. It will then iterate through the argument, and pass each item to *registerPrefix()*.

autoload Attempt to load a class. `autoload($class)`

autoload() Attempts to load the class specified. Returns a boolean `false` on failure, or a string indicating the class loaded on success.

register Register with spl_autoload. `register()`

register() Registers the `autoload()` method of the current instance with `spl_autoload_register()`.

Examples

Please review the *examples in the quick start* for usage.

CHAPTER 97

The Class Map Generator utility: bin/classmap_generator.php

Overview

The script `bin/classmap_generator.php` can be used to generate class map files for use with *the ClassMapAutoloader*.

Internally, it consumes both Zend\Console\Getopt (for parsing command-line options) and Zend\File\ClassFileLocator for recursively finding all PHP class files in a given tree.

Quick Start

You may run the script over any directory containing source code. By default, it will look in the current directory, and will write the script to `.classmap.php` in the directory you specify.

```
1 php classmap_generator.php Some/Directory/
```

Configuration Options

Class Map Generator Options

–help or -h Returns the usage message. If any other options are provided, they will be ignored.

–library or -l Expects a single argument, a string specifying the library directory to parse. If this option is not specified, it will assume the current working directory.

–output or -o Where to write the autoload class map file. If not provided, assumes ".classmap.php" in the library directory.

–overwrite or -w If an autoload class map file already exists with the name as specified via the `--output` option, you can overwrite it by specifying this flag. Otherwise, the script will not write the class map and return a warning.

CHAPTER 98

The PrefixPathLoader

Overview

Zend Framework's 1.X series introduced a plugin methodology surrounding associations of vendor/component prefixes and filesystem paths in the `Zend_Loader_PluginLoader` class. Zend Framework 2 provides equivalent functionality with the `PrefixPathLoader` class, and expands it to take advantage of PHP 5.3 namespaces.

The concept is relatively simple: a given vendor prefix or namespace is mapped to one or more paths, and multiple prefix/path maps may be provided. To resolve a plugin name, the prefixes are searched as a stack (i.e., last in, first out, or LIFO), and each path associated with the prefix is also searched as a stack. As soon as a file is found matching the plugin name, the class will be returned.

Since searching through the filesystem can lead to performance degradation, the `PrefixPathLoader` provides several optimizations. First, it will attempt to autoload a plugin before scanning the filesystem. This allows you to benefit from your autoloader and/or an opcode cache. Second, it aggregates the class name and class file associated with each discovered plugin. You can then retrieve this information and cache it for later seeding a *ClassMapAutoloader* and *PluginClassLoader*.

`PrefixPathLoader` implements the `ShortNameLocator` and `PrefixPathMapper` interfaces.

Note: Case Sensitivity

Unlike the *PluginClassLoader*, plugins resolved via the `PrefixPathLoader` are considered case sensitive. This is due to the fact that the lookup is done on the filesystem, and thus a file exactly matching the plugin name must exist.

Note: Preference is for Namespaces

Unlike the Zend Framework 1 variant, the `PrefixPathLoader` assumes that "prefixes" are PHP 5.3 namespaces by default. You can override this behavior, however, per prefix/path you map. Please see the documentation and examples below for details.

Quick Start

The `PrefixPathLoader` invariably requires some configuration – it needs to know what namespaces and/or vendor prefixes it should try, as well as the paths associated with each. You can inform the class of these at instantiation, or later by calling either the `addPrefixPath()` or `addPrefixPaths()` methods.

```
use Zend\Loader\PrefixPathLoader;

// Configure at instantiation:
$loader = new PrefixPathLoader(array(
    array('prefix' => 'Foo', 'path' => '../library/Foo'),
    array('prefix' => 'Bar', 'path' => '../vendor/Bar'),
));

// Or configure manually using methods:
$loader = new PrefixPathLoader();
$loader->addPrefixPath('Foo', '../library/Foo');

$loader->addPrefixPaths(array(
    array('prefix' => 'Foo', 'path' => '../library/Foo'),
    array('prefix' => 'Bar', 'path' => '../vendor/Bar'),
));
```

Once configured, you may then attempt to lookup a plugin.

```
if (false === ($class = $loader->load('bar'))) {
    throw new Exception("Plugin class matching 'bar' not found!");
}
$plugin = new $class();
```

Configuration Options

PrefixPathLoader Options

$options The constructor accepts either an array or a `Traversable` object of prefix paths. For the format allowed, please see the *addPrefixPaths()* method documentation.

Available Methods

__construct Instantiate and initialize loader `__construct($options = null)`

> **__construct()** Instantiates and initializes a `PrefixPathLoader` instance. If the `$prefixPaths` protected member is defined, it re-initializes it to an `Zend\Stdlib\ArrayStack` instance, and passes the original value to *the addPrefixPaths() method*. It then checks to see if `$staticPaths` has been populated, and, if so, passes that on to the `addPrefixPaths()` method to merge the values. Finally, if `$options` is non-null, it passes that to `addPrefixPaths()`.

addStaticPaths Add paths statically `addStaticPaths($paths)`

> **addStaticPaths()** Expects an array or `Traversable` object compatible with the `addPrefixPaths()` method. This method is static, and populates the protected `$staticPaths` member, which is used during instantiation to either override default paths or add additional prefix/path pairs to search.

setOptions Configure object state `setOptions($options)`

setOptions() Proxies to *addPrefixPaths()*.

addPrefixPath Map a namespace/vendor prefix to the given filesystem path `addPrefixPath($prefix, $path, $namespaced = true)`

addPrefixPath() Use this method to map a single filesystem path to a given namespace or vendor prefix. By default, the `$prefix` will be considered a PHP 5.3 namespace; you may specify that it is a vendor prefix by passing a boolean `false` value to the `$namespaced` argument.

If the `$prefix` has been previously mapped, this method adds another `$path` to a stack – meaning the new path will be searched first when attempting to resolve a plugin name to this `$prefix`.

addPrefixPaths Add many prefix/path pairs at once `addPrefixPaths($prefixPaths)`

addPrefixPaths() This method expects an array or `Traversable` object. Each item in the array or object must be one of the following:

- An array, with the keys "prefix" and "path", and optionally "namespaced"; the keys correspond to the arguments to *addPrefixPath()*. The "prefix" and "path" keys should point to string values, while the "namespaced" key should be a boolean.
- An object, with the attributes "prefix" and "path", and optionally "namespaced"; the attributes correspond to the arguments to *addPrefixPath()*. The "prefix" and "path" attributes should point to string values, while the "namespaced" attribute should be a boolean.

The method will loop over arguments, and pass values to *addPrefixPath()* to process.

getPaths Retrieve all paths associated with a prefix, or all paths `getPaths($prefix = null)`

getPaths() Use this method to obtain the prefix/paths map. If no `$prefix` is provided, the return value is an `Zend\Stdlib\ArrayStack`, where the keys are namespaces or vendor prefixes, and the values are `Zend\Stdlib\SplStack` instances containing all paths associated with the given namespace or prefix.

If the `$prefix` argument is provided, two outcomes are possible. If the prefix is not found, a boolean `false` value is returned. If the prefix is found, a `Zend\Stdlib\SplStack` instance containing all paths associated with that prefix is returned.

clearPaths Clear all maps, or all paths for a given prefix `clearPaths($prefix = null)`

clearPaths() If no `$prefix` is provided, all prefix/path pairs are removed. If a `$prefix` is provided and found within the map, only that prefix is removed. Finally, if a `$prefix` is provided, but not found, a boolean `false` is returned.

removePrefixPath

`removePrefixPath($prefix, $path)`

removePrefixPath() Removes a single path from a given prefix.

isLoaded Has the given plugin been loaded? `isLoaded($name)`

isLoaded() Use this method to determine if the given plugin has been resolved to a class and file. Unlike `PluginClassLoader`, this method can return a boolean `false` even if the loader is capable of loading the plugin; it simply indicates whether or not the current instance has yet resolved the plugin via the `load()` method.

getClassName Retrieve the class name to which a plugin resolves `getClassName($name)`

getClassName() Given a plugin name, this method will attempt to return the associated class name. The method completes successfully if, and only if, the plugin has been successfully loaded via `load()`. Otherwise, it will return a boolean `false`.

load Attempt to resolve a plugin to a class `load($name)`

> **load()** Given a plugin name, the `load()` method will loop through the internal `ArrayStack`. The plugin name is first normalized using `ucwords()`, and then appended to the current vendor prefix or namespace. If the resulting class name resolves via autoloading, the class name is immediately returned. Otherwise, it then loops through the associated `SplStack` of paths for the prefix, looking for a file matching the plugin name (i.e., for plugin `Foo`, file name `Foo.php`) in the given path. If a match is found, the class name is returned.
>
> If no match is found, a boolean false is returned.

getPluginMap Get a list of plugin/class name pairs `getPluginMap()`

> **getPluginMap()** Returns an array of resolved plugin name/class name pairs. This value may be used to seed a `PluginClassLoader` instance.

getClassMap Get a list of class name/file name pairs `getClassMap()`

> **getClassMap()** Returns an array of resolved class name/file name pairs. This value may be used to seed a `ClassMapAutoloader` instance.

Examples

Using multiple paths for the same prefix

Sometimes you may have code containing the same namespace or vendor prefix in two different locations. Potentially, the same class may be defined in different locations, but with slightly different functionality. (We do not recommend this, but sometimes it happens.)

The `PrefixPathLoader` easily allows for these situations; simply register the path you want to take precedence last.

Consider the following directory structures:

```
project
|-- library
|   |-- Foo
|   |   |-- Bar.php
|   |   `-- Baz.php
|-- vendor
|   |-- Foo
|   |   |-- Bar.php
|   |   `-- Foobar.php
```

For purposes of this example, we'll assume that the common namespace is "Foo", and that the "Bar" plugin from the vendor branch is preferred. To make this possible, simply register the "vendor" directory last.

```
use Zend\Loader\PrefixPathLoader;

$loader = new PrefixPathLoader();

// Multiple calls to addPrefixPath():
$loader->addPrefixPath('Foo', PROJECT_ROOT . '/library/Foo')
       ->addPrefixPath('Foo', PROJECT_ROOT . '/vendor/Foo');

// Or use a single call to addPrefixPaths():
$loader->addPrefixPaths(array(
    array('prefix' => 'Foo', 'path' => PROJECT_ROOT . '/library/Foo'),
    array('prefix' => 'Foo', 'path' => PROJECT_ROOT . '/vendor/Foo'),
```

```
));

// And then resolve plugins:
$bar    = $loader->load('bar');    // Foo\Bar from vendor/Foo/Bar.php
$baz    = $loader->load('baz');    // Foo\Baz from library/Foo/Baz.php
$foobar = $loader->load('foobar'); // Foo\Foobar from vendor/Foo/Baz.php
```

Prototyping with PrefixPathLoader

`PrefixPathLoader` is quite useful for prototyping applications. With minimal configuration, you can access a full directory of plugins, without needing to update maps as new plugins are added. However, this comes with a price: performance. Since plugins are resolved typically using by searching the filesystem, you are introducing I/O calls every time you request a new plugin.

With this in mind, `PrefixPathLoader` provides two methods for assisting in migrating to more performant solutions. The first is `getClassMap()`. This method returns an array of class name/file name pairs suitable for use with *ClassMapAutoloader*. Injecting your autoloader with that map will ensure that on subsequent calls, `load()` should be able to find the appropriate class via autoloading – assuming that the match is on the first prefix checked.

The second solution is the `getPluginMap()` method, which creates a plugin name/class name map suitable for injecting into a *PluginClassLoader* instance. Combine this with class map-based autoloading, and you can actually eliminate I/O calls altogether when using an opcode cache.

Usage of these methods is quite simple.

```
// After a number of load() operations, or at the end of the request:
$classMap  = $loader->getClassMap();
$pluginMap = $loader->getPluginMap();
```

From here, you will need to do a little work. First, you need to serialize this information somehow for later use. For that, there are two options: `Zend\Serializer` or `Zend\Cache`.

```
// Using Zend\Serializer:
use Zend\Serializer\Serializer;

$adapter = Serializer::factory('PhpCode');
$content = "<?php\nreturn " . $adapter->serialize($classMap) . ";";
file_put_contents(APPLICATION_PATH . '/.classmap.php', $content);

// Using Zend\Cache:
use Zend\Cache\Cache;

$cache = Cache::factory(
    'Core', 'File',
    array('lifetime' => null, 'automatic_serialization' => true),
    array('cache_dir' => APPLICATION_PATH . '/../cache/classmaps')
);
$cache->save($pluginMap, 'pluginmap');
```

Note: the examples alternate between the class map and plugin map; however, either technique applies to either map.

Once the data is cached, you can retrieve it late to populate. In the example of the class map above, you would simply pass the filename to the `ClassMapAutoloader` instance:

```
$autoloader = new Zend\Loader\ClassMapAutoloader();
$autoloader->registerAutoloadMap(APPLICATION_PATH . '/.classmap.php');
```

If using `Zend\Cache`, you would retrieve the cached data, and pass it to the appropriate component; in this case, we pass the value to a `PluginClassLoader` instance.

```
$map = $cache->load('pluginmap');

$loader = new Zend\Loader\PluginClassLoader($map);
```

With some creative and well disciplined architecture, you can likely automate these processes to ensure that development can benefit from the dynamic nature of the `PrefixPathLoader`, and production can benefit from the performance optimizations of the `ClassMapAutoloader` and `PluginClassLoader`.

CHAPTER 99

The PrefixPathMapper Interface

Overview

One approach to resolving plugin names to class names utilizes prefix/path pairs. In this methodology, the developer specifies one or more directories containing plugins that have a common namespace or prefix. When resolving a plugin, the mapper will loop through these prefixes, and look for a class file matching the requested plugin; if found, that plugin class is loaded from the file and used. The `PrefixPathMapper` interface defines a common interface for specifying and modifying a map of prefix/path pairs.

Quick Start

The `PrefixPathMapper` provides simply two methods: one for registering a prefix path, and another for removing one.

```
namespace Zend\Loader;

interface PrefixPathMapper
{
    public function addPrefixPath($prefix, $path);
    public function removePrefixPath($prefix, $path);
}
```

Configuration Options

This component defines no configuration options, as it is an interface.

Available Methods

addPrefixPath Register a prefix/path association `addPrefixPath($prefix, $path)`

> **addPrefixPath()** Implement this method to allow registering a prefix/path pair. The prefix may be either an older, PHP 5.2-style vendor prefix or a true PHP 5.3 namespace; the path should be a path to a directory of files using the given prefix or namespace. The implemenation should determine whether or not to aggregate paths for each namespace, or simply maintain a 1:1 association.

removePrefixPath Remove a prefix/path association `removePrefixPath($prefix, $path)`

> **removePrefixPath()** Implement this method to remove a prefix/path association from the internal map.

Examples

Please see the *Quick Start* for the interface specification.

CHAPTER 100

Overview

`Zend\Log\Logger` is a component for general purpose logging. It supports multiple log backends, formatting messages sent to the log, and filtering messages from being logged. These functions are divided into the following objects:

- A Logger (instance of `Zend\Log\Logger`) is the object that your application uses the most. You can have as many Logger objects as you like; they do not interact. A Logger object must contain at least one Writer, and can optionally contain one or more Filters.
- A Writer (inherits from `Zend\Log\Writer\AbstractWriter`) is responsible for saving data to storage.
- A Filter (implements `Zend\Log\Filter`) blocks log data from being saved. A filter is applied to an individual writer. Filters can be chained.
- A Formatter (inheriting from `Zend\Log\Formatter\AbstractFormatter`) can format the log data before it is written by a Writer. Each Writer has exactly one Formatter.

Creating a Log

To get started logging, instantiate a Writer and then pass it to a Logger instance:

```
1 $logger = new Zend\Log\Logger;
2 $writer = new Zend\Log\Writer\Stream('php://output');
3
4 $logger->addWriter($writer);
```

It is important to note that the Logger must have at least one Writer. You can add any number of Writers using the Log's `addWriter()` method.

You can also add a priority to each writer. The priority is specified as number and passed as second argument in the `addWriter()` method.

Another way to add a writer to a Logger is to use the name of the writer as follow:

```php
$logger = new Zend\Log\Logger;

$logger->addWriter('stream', null, array('stream' => 'php://output'));
```

In this example we passed the stream `php://output` as parameter (as array).

Logging Messages

To log a message, call the `log()` method of a Log instance and pass it the message with a corresponding priority:

```php
$logger->log(Zend\Log\Logger::INFO, 'Informational message');
```

The first parameter of the `log()` method is an integer `priority` and the second parameter is a string `message`. The priority must be one of the priorities recognized by the Logger instance. This is explained in the next section. There is also an optional third parameter used to pass extra informations to the writer's log.

A shortcut is also available. Instead of calling the `log()` method, you can call a method by the same name as the priority:

```php
$logger->log(Zend\Log\Logger::INFO, 'Informational message');
$logger->info('Informational message');

$logger->log(Zend\Log\Logger::EMERG, 'Emergency message');
$logger->emerg('Emergency message');
```

Destroying a Log

If the Logger object is no longer needed, set the variable containing it to `NULL` to destroy it. This will automatically call the `shutdown()` instance method of each attached Writer before the Log object is destroyed:

```php
$logger = null;
```

Explicitly destroying the log in this way is optional and is performed automatically at *PHP* shutdown.

Using Built-in Priorities

The `Zend\Log\Logger` class defines the following priorities:

```
EMERG   = 0;  // Emergency: system is unusable
ALERT   = 1;  // Alert: action must be taken immediately
CRIT    = 2;  // Critical: critical conditions
ERR     = 3;  // Error: error conditions
WARN    = 4;  // Warning: warning conditions
NOTICE  = 5;  // Notice: normal but significant condition
INFO    = 6;  // Informational: informational messages
DEBUG   = 7;  // Debug: debug messages
```

These priorities are always available, and a convenience method of the same name is available for each one.

The priorities are not arbitrary. They come from the BSD syslog protocol, which is described in RFC-3164. The names and corresponding priority numbers are also compatible with another *PHP* logging system, PEAR Log, which perhaps promotes interoperability between it and `Zend\Log\Logger`.

Priority numbers descend in order of importance. `EMERG` (0) is the most important priority. `DEBUG` (7) is the least important priority of the built-in priorities. You may define priorities of lower importance than `DEBUG`. When selecting the priority for your log message, be aware of this priority hierarchy and choose appropriately.

Understanding Log Events

When you call the `log()` method or one of its shortcuts, a log event is created. This is simply an associative array with data describing the event that is passed to the writers. The following keys are always created in this array: `timestamp`, `message`, `priority`, and `priorityName`.

The creation of the `event` array is completely transparent.

Log PHP Errors

`Zend\Log\Logger` can also be used to log *PHP* errors and intercept Exceptions. Calling the static method `registerErrorHandler($logger)` will add the $logger object before the current PHP error handler, and will pass the error along as well.

```
use Zend\Log\Logger;
use Zend\Log\Writer\Stream as StreamWriter;

$logger = new Logger;
$writer = new StreamWriter('php://output');
$logger->addWriter($writer);

Logger::registerErrorHandler($this->logger);
```

If you want to unregister the error handler you can use the `unregisterErrorHandler()` static method.

Table 100.1: Zend\Log\Logger events from PHP errors fields matching handler (int $errno , string $errstr [, string $errfile [, int $errline [, array $errcontext]]]) from set_error_handler

Name	Error Handler Parameter	Description
message	errstr	Contains the error message, as a string.
errno	errno	Contains the level of the error raised, as an integer.
file	errfile	Contains the filename that the error was raised in, as a string.
line	errline	Contains the line number the error was raised at, as an integer.
context	errcontext	(optional) An array that points to the active symbol table at the point the error occurred. In other words, errcontext will contain an array of every variable that existed in the scope the error was triggered in. User error handler must not modify error context.

You can also configure a Logger to intercept Exceptions using the static method `registerExceptionHandler($logger)`.

CHAPTER 101

Writers

A Writer is an object that inherits from `Zend\Log\Writer\AbstractWriter`. A Writer's responsibility is to record log data to a storage backend.

Writing to Streams

`Zend\Log\Writer\Stream` sends log data to a PHP stream.

To write log data to the *PHP* output buffer, use the URL `php://output`. Alternatively, you can send log data directly to a stream like `STDERR` (`php://stderr`).

```
$writer = new Zend\Log\Writer\Stream('php://output');
$logger = new Zend\Log\Logger($writer);

$logger->info('Informational message');
```

To write data to a file, use one of the Filesystem URLs:

```
$writer = new Zend\Log\Writer\Stream('/path/to/logfile');
$logger = new Zend\Log\Logger($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);
```

```php
$logger = new Zend\Log\Logger($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\Adapter\Adapter`. The constructor of `Zend\Log\Writer\Db` receives a `Zend\Db\Adapter\Adapter` instance, a table name, an optional mapping of event data to database columns, and an optional string contains the character separator for the log array:

```php
$dbconfig = array(
    // Sqlite Configuration
    'driver' => 'Pdo',
    'dsn' => 'sqlite:' . __DIR__ . '/tmp/sqlite.db',
);
$db = new Zend\Db\Adapter\Adapter($dbconfig);

$writer = new Zend\Log\Writer\Db($db, 'log_table_name');
$logger = new Zend\Log\Logger($writer);

$logger->info('Informational message');
```

The example above writes a single row of log data to the database table named 'log_table_name' table. The database column will be created according to the event array generated by the `Zend\Log\Logger` instance.

If we specify the mapping of the events with the database columns the log will store in the database only the selected fields.

```php
$dbconfig = array(
    // Sqlite Configuration
    'driver' => 'Pdo',
    'dsn' => 'sqlite:' . __DIR__ . '/tmp/sqlite.db',
);
$db = new Zend\Db\Adapter\Adapter($dbconfig);

$mapping = array(
    'timestamp' => 'date',
    'priority'  => 'type',
    'message'   => 'event'
);
$writer = new Zend\Log\Writer\Db($db, 'log_table_name', $mapping);
$logger = new Zend\Log\Logger($writer);

$logger->info('Informational message');
```

The previous example will store only the log information timestamp, priority and message in the database fields date, type and event.

The `Zend\Log\Writer\Db` has a second optional parameter in the constructor. This parameter is the character separator for the log events managed by an array. For instance, if we have a log that contains an array extra fields, this will be translated in 'extra-field', where '-' is the character separator (default) and field is the subname of the specific extra field.

Stubbing Out the Writer

The `Zend\Log\Writer\Null` is a stub that does not write log data to anything. It is useful for disabling logging or stubbing out logging during tests:

```
$writer = new Zend\Log\Writer\Null;
$logger = new Zend\Log\Logger($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;
$logger = new Zend\Log\Logger($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();
$logger->addWriter($writer1);
$logger->addWriter($writer2);

// goes to both writers
$logger->info('Informational message');
```

You can also specify the priority number for each writer to change the order of writing. The priority number is an integer number (greater or equal to 1) passed as second parameter in the `addWriter()` method.

CHAPTER 102

Filters

A Filter object blocks a message from being written to the log.

You can add a filter to a specific Writer using `addFilter()` method of that Writer:

```
use Zend\Log\Logger;

$logger = new Logger();

$writer1 = new Zend\Log\Writer\Stream('/path/to/first/logfile');
$logger->addWriter($writer1);

$writer2 = new Zend\Log\Writer\Stream('/path/to/second/logfile');
$logger->addWriter($writer2);

// add a filter only to writer2
$filter = new Zend\Log\Filter\Priority(Logger::CRIT);
$writer2->addFilter($filter);

// logged to writer1, blocked from writer2
$logger->info('Informational message');

// logged by both writers
$logger->emerg('Emergency message');
```

Available filters

The Zend\Log\Filter available are:

- **Priority**, filter logging by $priority. By default, it will accept any log event whose priority value is less than or equal to $priority.
- **Regex**, filter out any log messages not matching the regex pattern. This filter use the preg_match() function of PHP.

- **SuppressFilter**, this is a simple boolean filter. Call suppress(true) to suppress all log events. Call suppress(false) to accept all log events.
- **Validator**, filter out any log messages not matching the Zend\Validator\Validator object passed to the filter.

CHAPTER 103

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();
$logger->addWriter($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();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above outputs the following *XML* (space added for clarity):

```
<logEntry>
  <timestamp>2007-04-06T07:24:37-07:00</timestamp>
  <message>informational message</message>
  <priority>6</priority>
  <priorityName>INFO</priorityName>
</logEntry>
```

It's possible to customize the root element as well as specify a mapping of *XML* elements to the items in the event data array. The constructor of `Zend\Log\Formatter\Xml` accepts a string with the name of the root element as the first parameter and an associative array with the element mapping as the second parameter:

```
$writer = new Zend\Log\Writer\Stream('php://output');
$formatter = new Zend\Log\Formatter\Xml('log',
                                        array('msg' => 'message',
                                              'level' => 'priorityName')
                                        );
$writer->setFormatter($formatter);

$logger = new Zend\Log\Logger();
$logger->addWriter($writer);

$logger->info('informational message');
```

The code above changes the root element from its default of `logEntry` to `log`. It also maps the element `msg` to the event data item `message`. This results in the following output:

```
<log>
  <msg>informational message</msg>
  <level>INFO</level>
</log>
```

Formatting to FirePhp

`Zend\Log\Formatter\FirePhp` formats log data for the Firebug extension for Firefox.

CHAPTER 104

Zend\Mail\Message

Overview

The `Message` class encapsulates a single email message as described in RFCs 822 and 2822. It acts basically as a value object for setting mail headers and content.

If desired, multi-part email messages may also be created. This is as trivial as creating the message body using the Zend\Mime component, assigning it to the mail message body.

The `Message` class is simply a value object. It is not capable of sending or storing itself; for those purposes, you will need to use, respectively, a Storage adapter or *Transport adapter*.

Quick Start

Creating a `Message` is simple: simply instantiate it.

```
use Zend\Mail\Message;

$message = new Message();
```

Once you have your `Message` instance, you can start adding content or headers. Let's set who the mail is from, who it's addressed to, a subject, and some content:

```
$message->addFrom("matthew@zend.com", "Matthew Weier O'Phinney")
        ->addTo("foobar@example.com")
        ->setSubject("Sending an email from Zend\Mail!");
$message->setBody("This is the message body.");
```

You can also add recipients to carbon-copy ("Cc:") or blind carbon-copy ("Bcc:").

```
$message->addCc("ralph.schindler@zend.com")
        ->addBcc("enrico.z@zend.com");
```

If you want to specify an alternate address to which replies may be sent, that can be done, too.

```
$message->addReplyTo("matthew@weierophinney.net", "Matthew");
```

Interestingly, RFC822 allows for multiple "From:" addresses. When you do this, the first one will be used as the sender, **unless** you specify a "Sender:" header. The `Message` class allows for this.

```
/*
 * Mail headers created:
 * From: Ralph Schindler <ralph.schindler@zend.com>, Enrico Zimuel <enrico.z@zend.com>
 * Sender: Matthew Weier O'Phinney <matthew@zend.com></matthew>
 */
$message->addFrom("ralph.schindler@zend.com", "Ralph Schindler")
        ->addFrom("enrico.z@zend.com", "Enrico Zimuel")
        ->setSender("matthew@zend.com", "Matthew Weier O'Phinney");
```

By default, the `Message` class assumes ASCII encoding for your email. If you wish to use another encoding, you can do so; setting this will ensure all headers and body content are properly encoded using quoted-printable encoding.

```
$message->setEncoding("UTF-8");
```

If you wish to set other headers, you can do that as well.

```
/*
 * Mail headers created:
 * X-API-Key: FOO-BAR-BAZ-BAT
 */
$message->getHeaders()->addHeaderLine('X-API-Key', 'FOO-BAR-BAZ-BAT');
```

Sometimes you may want to provide HTML content, or multi-part content. To do that, you'll first create a MIME message object, and then set it as the body of your mail message object. When you do so, the `Message` class will automatically set a "MIME-Version" header, as well as an appropriate "Content-Type" header.

```
use Zend\Mail\Message;
use Zend\Mime\Message as MimeMessage;
use Zend\Mime\Part as MimePart;

$text = new MimePart($textContent);
$text->type = "text/plain";

$html = new MimePart($htmlMarkup);
$html->type = "text/html";

$image = new MimePart(fopen($pathToImage));
$image->type = "image/jpeg";

$body = new MimeMessage();
$body->setParts(array($text, $html, $image));

$message = new Message();
$message->setBody($body);
```

If you want a string representation of your email, you can get that:

```
echo $message->toString();
```

Finally, you can fully introspect the message – including getting all addresses of recipients and senders, all ehaders, and the message body.

```php
// Headers
// Note: this will also grab all headers for which accessors/mutators exist in
// the Message object itself.
foreach ($message->getHeaders() as $header) {
    echo $header->toString();
    // or grab values: $header->getFieldName(), $header->getFieldValue()
}

// The logic below also works for the methods cc(), bcc(), to(), and replyTo()
foreach ($message->from() as $address) {
    printf("%s: %s\n", $address->getEmail(), $address->getName());
}

// Sender
$address = $message->getSender();
printf("%s: %s\n", $address->getEmail(), $address->getName());

// Subject
echo "Subject: ", $message->getSubject(), "\n";

// Encoding
echo "Encoding: ", $message->getEncoding(), "\n";

// Message body:
echo $message->getBody();     // raw body, or MIME object
echo $message->getBodyText(); // body as it will be sent
```

Once your message is shaped to your liking, pass it to a *mail transport* in order to send it!

```php
$transport->send($message);
```

Configuration Options

The `Message` class has no configuration options, and is instead a value object.

Available Methods

isValid `isValid()`

Is the message valid?

If we don't have any From addresses, we're invalid, according to RFC2822.

Returns bool

setEncoding `setEncoding(string $encoding)`

Set the message encoding.

Implements a fluent interface.

getEncoding `getEncoding()`

Get the message encoding.

Returns string.

setHeaders `setHeaders(Zend\Mail\Headers $headers)`

Compose headers.

Implements a fluent interface.

getHeaders `getHeaders()`

Access headers collection.

Lazy-loads a Zend\Mail\Headers instance if none is already attached.

Returns a Zend\Mail\Headers instance.

setFrom `setFrom(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, string|null $name)`

Set (overwrite) From addresses.

Implements a fluent interface.

addFrom `addFrom(string|Zend\Mail\Address|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, string|null $name)`

Add a "From" address.

Implements a fluent interface.

from `from()`

Retrieve list of From senders

Returns Zend\Mail\AddressList instance.

setTo `setTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, null|string $name)`

Overwrite the address list in the To recipients.

Implements a fluent interface.

addTo `addTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, null|string $name)`

Add one or more addresses to the To recipients.

Appends to the list.

Implements a fluent interface.

to `to()`

Access the address list of the To header.

Lazy-loads a Zend\Mail\AddressList and populates the To header if not previously done.

Returns a Zend\Mail\AddressList instance.

setCc `setCc(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, string|null $name)`

Set (overwrite) CC addresses.

Implements a fluent interface.

addCc `addCc(string|Zend\Mail\Address|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, string|null $name)`

Add a "Cc" address.

Implements a fluent interface.

cc `cc()`

Retrieve list of CC recipients

Lazy-loads a Zend\Mail\AddressList and populates the Cc header if not previously done.

Returns a Zend\Mail\AddressList instance.

setBcc `setBcc(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, string|null $name)`

Set (overwrite) BCC addresses.

Implements a fluent interface.

addBcc `addBcc(string|Zend\Mail\Address|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, string|null $name)`

Add a "Bcc" address.

Implements a fluent interface.

bcc `bcc()`

Retrieve list of BCC recipients.

Lazy-loads a Zend\Mail\AddressList and populates the Bcc header if not previously done.

Returns a Zend\Mail\AddressList instance.

setReplyTo `setReplyTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressList, null|string $name)`

Overwrite the address list in the Reply-To recipients.

Implements a fluent interface.

addReplyTo `addReplyTo(string|AddressDescription|array|Zend\Mail\AddressList|Traversable $emailOrAddressOrList, null|string $name)`

Add one or more addresses to the Reply-To recipients.

Implements a fluent interface.

replyTo `replyTo()`

Access the address list of the Reply-To header

Lazy-loads a Zend\Mail\AddressList and populates the Reply-To header if not previously done.

Returns a Zend\Mail\AddressList instance.

setSender `setSender(mixed $emailOrAddress, mixed $name)`

Set the message envelope Sender header.

Implements a fluent interface.

getSender `getSender()`

Retrieve the sender address, if any.

Returns null or a Zend\Mail\AddressDescription instance.

setSubject `setSubject(string $subject)`

Set the message subject header value.

Implements a fluent interface.

getSubject `getSubject()`

Get the message subject header value.

Returns null or a string.

setBody `setBody(null|string|Zend\Mime\Message|object $body)`

Set the message body.

Implements a fluent interface.

getBody `getBody()`

Return the currently set message body.

Returns null, a string, or an object.

getBodyText `getBodyText()`

Get the string-serialized message body text.

Returns null or a string.

toString `toString()`

Serialize to string.

Returns string.

Examples

Please *see the Quick Start section*.

CHAPTER 105

Zend\Mail\Transport

Overview

Transports take care of the actual delivery of mail. Typically, you only need to worry about two possibilities: using PHP's native `mail()` functionality, which uses system resources to deliver mail, or using the *SMTP* protocol for delivering mail via a remote server. Zend Framework also includes a "File" transport, which creates a mail file for each message sent; these can later be introspected as logs or consumed for the purposes of sending via an alternate transport mechanism later.

The `Zend\Mail\Transport` interface defines exactly one method, `send()`. This method accepts a `Zend\Mail\Message` instance, which it then introspects and serializes in order to send.

Quick Start

Using a mail transport is typically as simple as instantiating it, optionally configuring it, and then passing a message to it.

Sendmail Transport Usage

```
use Zend\Mail\Message;
use Zend\Mail\Transport\Sendmail as SendmailTransport;

$message = new Message();
$message->addTo('matthew@zend.com')
        ->addFrom('ralph.schindler@zend.com')
        ->setSubject('Greetings and Salutations!')
        ->setBody("Sorry, I'm going to be late today!");

$transport = new SendmailTransport();
$transport->send($message);
```

SMTP Transport Usage

```php
use Zend\Mail\Message;
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

$message = new Message();
$message->addTo('matthew@zend.com')
        ->addFrom('ralph.schindler@zend.com')
        ->setSubject('Greetings and Salutations!')
        ->setBody("Sorry, I'm going to be late today!");

// Setup SMTP transport using LOGIN authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'login',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
$transport->send($message);
```

File Transport Usage

```php
use Zend\Mail\Message;
use Zend\Mail\Transport\File as FileTransport;
use Zend\Mail\Transport\FileOptions;

$message = new Message();
$message->addTo('matthew@zend.com')
        ->addFrom('ralph.schindler@zend.com')
        ->setSubject('Greetings and Salutations!')
        ->setBody("Sorry, I'm going to be late today!");

// Setup SMTP transport using LOGIN authentication
$transport = new FileTransport();
$options   = new FileOptions(array(
    'path'              => 'data/mail/',
    'callback'  => function (FileTransport $transport) {
        return 'Message_' . microtime(true) . '_' . mt_rand() . '.txt';
    },
));
$transport->setOptions($options);
$transport->send($message);
```

Configuration Options

Configuration options are per transport. Please follow the links below for transport-specific options.

- *SMTP Transport Options*

- *File Transport Options*

Available Methods

send `send(Zend\Mail\Message $message)`

Send a mail message.

Returns void

Examples

Please see the *Quick Start section* for examples.

CHAPTER 106

Zend\Mail\Transport\SmtpOptions

Overview

This document details the various options available to the `Zend\Mail\Transport\Smtp` mail transport.

Quick Start

Basic SMTP Transport Usage

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name' => 'localhost.localdomain',
    'host' => '127.0.0.1',
    'port' => 25,
));
$transport->setOptions($options);
```

SMTP Transport Usage with PLAIN AUTH

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport using LOGIN authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
```

```
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'plain',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
```

SMTP Transport Usage with LOGIN AUTH

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport using LOGIN authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'login',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
```

SMTP Transport Usage with CRAM-MD5 AUTH

```
use Zend\Mail\Transport\Smtp as SmtpTransport;
use Zend\Mail\Transport\SmtpOptions;

// Setup SMTP transport using LOGIN authentication
$transport = new SmtpTransport();
$options   = new SmtpOptions(array(
    'name'              => 'localhost.localdomain',
    'host'              => '127.0.0.1',
    'connection_class'  => 'crammd5',
    'connection_config' => array(
        'username' => 'user',
        'password' => 'pass',
    ),
));
$transport->setOptions($options);
```

Configuration Options

Configuration Options

name Name of the SMTP host; defaults to “localhost”.

host Remote hostname or IP address; defaults to "127.0.0.1".

port Port on which the remote host is listening; defaults to "25".

connection_class Fully-qualified classname or short name resolvable via `Zend\Mail\Protocol\SmtpLoader`. Typically, this will be one of "smtp", "plain", "login", or "crammd5", and defaults to "smtp".

Typically, the connection class should extend the `Zend\Mail\Protocol\AbstractProtocol` class, and specifically the SMTP variant.

connection_config Optional associative array of parameters to pass to the *connection class* in order to configure it. By default this is empty. For connection classes other than the default, you will typically need to define the "username" and "password" options.

Available Methods

getName `getName()`

Returns the string name of the local client hostname.

setName `setName(string $name)`

Set the string name of the local client hostname.

Implements a fluent interface.

getConnectionClass `getConnectionClass()`

Returns a string indicating the connection class name to use.

setConnectionClass `setConnectionClass(string $connectionClass)`

Set the connection class to use.

Implements a fluent interface.

getConnectionConfig `getConnectionConfig()`

Get configuration for the connection class.

Returns array.

setConnectionConfig `setConnectionConfig(array $config)`

Set configuration for the connection class. Typically, if using anything other than the default connection class, this will be an associative array with the keys "username" and "password".

Implements a fluent interface.

getHost `getHost()`

Returns a string indicating the IP address or host name of the SMTP server via which to send messages.

setHost `setHost(string $host)`

Set the SMTP host name or IP address.

Implements a fluent interface.

getPort `getPort()`

Retrieve the integer port on which the SMTP host is listening.

setPort `setPort(int $port)`

Set the port on which the SMTP host is listening.

Implements a fluent interface.

__construct `__construct(null|array|Traversable $config)`

Instantiate the class, and optionally configure it with values provided.

Examples

Please see the *Quick Start* for examples.

CHAPTER 107

Zend\Mail\Transport\FileOptions

Overview

This document details the various options available to the `Zend\Mail\Transport\File` mail transport.

Quick Start

File Transport Usage

```php
use Zend\Mail\Transport\File as FileTransport;
use Zend\Mail\Transport\FileOptions;

// Setup SMTP transport using LOGIN authentication
$transport = new FileTransport();
$options   = new FileOptions(array(
    'path'              => 'data/mail/',
    'callback'  => function (FileTransport $transport) {
        return 'Message_' . microtime(true) . '_' . mt_rand() . '.txt';
    },
));
$transport->setOptions($options);
```

Configuration Options

Configuration Options

path The path under which mail files will be written.

callback A PHP callable to be invoked in order to generate a unique name for a message file. By default, the following is used:

```
function (Zend\Mail\FileTransport $transport) {
    return 'ZendMail_' . time() . '_' . mt_rand() . '.tmp';
}
```

Available Methods

`Zend\Mail\Transport\FileOptions` extends `Zend\Stdlib\Options`, and inherits all functionality from that class; this includes `ArrayAccess` and property overloading. Additionally, the following explicit setters and getters are provided.

__construct `setPath(string $path)`

Set the path under which mail files will be written.

Implements fluent interface.

getPath `getPath()`

Get the path under which mail files will be written.

Returns string

setCallback `setCallback(Callable $callback)`

Set the callback used to generate unique filenames for messages.

Implements fluent interface.

getCallback `getCallback()`

Get the callback used to generate unique filenames for messages.

Returns PHP callable argument.

__construct `__construct(null|array|Traversable $config)`

Initialize the object. Allows passing a PHP array or `Traversable` object with which to populate the instance.

Examples

Please see the *Quick Start* for examples.

CHAPTER 108

Introduction

`Zend\Math` namespace provides general mathematical functions. So far the supported functionalities are:

- `Zend\Math\Rand`, a random number generator;
- `Zend\Math\BigInteger`, a library to manage big integers.

We expect to add more functionalities in the future.

Random number generator

`Zend\Math\Rand` implements a random number generator that is able to generate random numbers for general purpose usage and for cryptographic scopes. To generate good random numbers this component uses the OpenSSL and the Mcrypt extension of PHP. If you don't have the OpenSSL or the Mcrypt extension installed in your environment the component will use the mt_rand function of PHP as fallback. The `mt_rand` is not considered secure for cryptographic purpose, that means if you will try to use it to generate secure random number the class will throw an exception.

In particular, the algorithm that generates random bytes in `Zend\Math\Rand` tries to call the openssl_random_pseudo_bytes function of the OpenSSL extension if installed. If the OpenSSL extension is not present in the system the algorithm tries to use the the mcrypt_create_iv function of the Mcrypt extension (using the `MCRYPT_DEV_URANDOM` parameter). Finally, if the OpenSSL and Mcrypt are not installed the generator uses the `mt_rand` function of PHP.

The `Zend\Math\Rand` class offers the following methods to generate random values:

- `getBytes($length, $strong = false)` to generate a random set of `$length` bytes;
- `getBoolean($strong = false)` to generate a random boolean value (true or false);
- `getInteger($min, $max, $strong = false)` to generate a random integer between `$min` and `$max`;
- `getFloat($strong = false)` to generate a random float number between 0 and 1;
- `getString($length, $charlist = null, $strong = false)` to generate a random string of $length characters using the alphabet $chalist (if not provided the default alphabet is the Base64).

In all these methods the parameter `$strong` specify the usage of a strong random number generator. We suggest to set the $strong to true if you need to generate random number for cryptographic and security implementation.

If `$strong` is set to true and you try to generate random values in a PHP environment without the OpenSSL and the Mcrypt extensions the component will throw an Exception.

Below we reported an example on how to generate random data using `Zend\Math\Rand`.

```
use Zend\Math\Rand;

$bytes = Rand::getBytes(32, true);
printf("Random bytes (in Base64): %s\n", base64_encode($bytes));

$boolean = Rand::getBoolean();
printf("Random boolean: %s\n", $boolean ? 'true' : 'false');

$integer = Rand::getInteger(0,1000);
printf("Random integer in [0-1000]: %d\n", $integer);

$float = Rand::getFloat();
printf("Random float in [0-1): %f\n", $float);

$string = Rand::getString(32, 'abcdefghijklmnopqrstuvwxyz', true);
printf("Random string in latin alphabet: %s\n", $string);
```

Big integers

`Zend\Math\BigInteger\BigInteger` offers a class to manage arbitrary length integer. PHP supports integer numbers with a maximum value of `PHP_INT_MAX`. If you need to manage integers bigger than `PHP_INT_MAX` you have to use external libraries or PHP extensions like GMP or BC Math.

`Zend\Math\BigInteger\BigInteger` is able to manage big integers using the GMP or the BC Math extensions as adapters.

The mathematical functions implemented in `Zend\Math\BigInteger\BigInteger` are:

- `add($leftOperand, $rightOperand)`, add two big integers;
- `sub($leftOperand, $rightOperand)`, subtract two big integers;
- `mul($leftOperand, $rightOperand)`, multiply two big integers;
- `div($leftOperand, $rightOperand)`, divide two big integers (this method returns only integer part of result);
- `pow($operand, $exp)`, raise a big integers to another;
- `sqrt($operand)`, get the square root of a big integer;
- `abs($operand)`, get the absolute value of a big integer;
- `mod($leftOperand, $modulus)`, get modulus of a big integer;
- `powmod($leftOperand, $rightOperand, $modulus)`, raise a big integer to another, reduced by a specified modulus;
- `comp($leftOperand, $rightOperand)`, compare two big integers, returns < 0 if leftOperand is less than rightOperand; > 0 if leftOperand is greater than rightOperand, and 0 if they are equal;
- `intToBin($int, $twoc = false)`, convert big integer into it's binary number representation;

- `binToInt($bytes, $twoc = false)`, convert binary number into big integer;
- `baseConvert($operand, $fromBase, $toBase = 10)`, convert a number between arbitrary bases;

Below is reported an example using the BC Math adapter to calculate the sum of two integer random numbers with 100 digits.

```
use Zend\Math\BigInteger\BigInteger;
use Zend\Math\Rand;

$bigInt = BigInteger::factory('bcmath');

$x = Rand::getString(100,'0123456789');
$y = Rand::getString(100,'0123456789');

$sum = $bigInt->add($x, $y);
$len = strlen($sum);

printf("%{$len}s +\n%{$len}s =\n%s\n%s\n", $x, $y, str_repeat('-', $len), $sum);
```

As you can see in the code the big integers are managed using strings. Even the result of the sum is represented as a string.

Below is reported another example using the BC Math adapter to generate the binary representation of a negative big integer of 100 digits.

```
use Zend\Math\BigInteger\BigInteger;
use Zend\Math\Rand;

$bigInt = BigInteger::factory('bcmath');

$digit = 100;
$x = '-' . Rand::getString($digit,'0123456789');

$byte = $bigInt->intToBin($x);

printf("The binary representation of the big integer with $digit digit:\n%s\nis (in␣
↪Base64 format): %s\n",
       $x, base64_encode($byte));
printf("Length in bytes: %d\n", strlen($byte));

$byte = $bigInt->intToBin($x, true);

printf("The two's complement binary representation of the big integer with $digit␣
↪digit:\n%s\nis (in Base64 format): %s\n",
       $x, base64_encode($byte));
printf("Length in bytes: %d\n", strlen($byte));
```

We generated the binary representation of the big integer number using the default binary format and the two's complement representation (specified with the `true` parameter in the `intToBin` function).

CHAPTER 109

Introduction to the Module System

Zend Framework 2.0 introduces a new and powerful approach to modules. This new module system is designed with flexibility, simplicity, and re-usability in mind. A module may contain just about anything: PHP code, including MVC functionality; library code; view scripts; and/or public assets such as images, CSS, and JavaScript. The possibilities are endless.

Note: The module system in ZF2 has been designed to be useful as a generic and powerful foundation from which developers and other projects can build their own module or plugin systems.

For a better understanding of the event-driven concepts behind the ZF2 module system, it may be helpful to read the *EventManager documentation.*.

The module system is made up of the following:

- **The Module Autoloader**-`Zend\Loader\ModuleAutoloader` is a specialized autoloader that is responsible for the locating and loading of modules'`Module` classes from a variety of sources.
- *The Module Manager*-`Zend\ModuleManager\ModuleManager` simply takes an array of module names and fires a sequence of events for each one, allowing the behavior of the module system to be defined entirely by the listeners which are attached to the module manager.
- **ModuleManager Listeners**- Event listeners can be attached to the module manager's various events. These listeners can do everything from resolving and loading modules to performing complex initialization tasks and introspection into each returned module object.

Note: The name of a module in a typical Zend Framework 2 application is simply a PHP namespace and must follow all of the same rules for naming.

The recommended structure of a typical MVC-oriented ZF2 module is as follows:

```
module_root/
    Module.php
    autoload_classmap.php
```

```
autoload_function.php
autoload_register.php
config/
    module.config.php
public/
    images/
    css/
    js/
src/
    <module_namespace>/
        <code files>
tests/
    phpunit.xml
    bootstrap.php
    <module_namespace>/
        <test code files>
views/
    <dir-named-after-module-namespace>/
        <dir-named-after-a-controller>/
            <.phtml files>
```

The autoload_*.php Files

The three `autoload_*.php` files are not required, but recommended. They provide the following:

- `autoload_classmap.php` should return an array classmap of class name/filename pairs (with the filenames resolved via the `__DIR__` magic constant).
- `autoload_function.php` should return a PHP callback that can be passed to `spl_autoload_register()`. Typically, this callback should utilize the map returned by `autoload_classmap.php`.
- `autoload_register.php` should register a PHP callback (typically that returned by `autoload_function.php` with `spl_autoload_register()`.

The purpose of these three files is to provide reasonable default mechanisms for autoloading the classes contained in the module, thus providing a trivial way to consume the module without requiring `Zend\ModuleManager` (e.g., for use outside a ZF2 application).

CHAPTER 110

The Module Manager

The module manager, `Zend\ModuleManager\ModuleManager`, is a very simple class which is responsible for iterating over an array of module names and triggering a sequence of events for each. Instantiation of module classes, initialization tasks, and configuration are all performed by attached event listeners.

Module Manager Events

Events triggered by `Zend\ModuleManager\ModuleManager`

loadModules This event is primarily used internally to help encapsulate the work of loading modules in event listeners, and allow the loadModules.post event to be more user-friendly. Internal listeners will attach to this event with a negative priority instead of loadModules.post so that users can safely assume things like config merging have been done once loadModules.post is triggered, without having to worry about priorities at all.

loadModule.resolve Triggered for each module that is to be loaded. The listener(s) to this event are responsible for taking a module name and resolving it to an instance of some class. The default module resolver shipped with ZF2 simply looks for the class `{modulename}\Module`, instantiating and returning it if it exists.

The name of the module may be retrieved by listeners using the `getModuleName()` method of the `Event` object; a listener should then take that name and resolve it to an object instance representing the given module. Multiple listeners can be attached to this event, and the module manager will trigger them in order of their priority until one returns an object. This allows you to attach additional listeners which have alternative methods of resolving modules from a given module name.

loadModule Once a module resolver listener has resolved the module name to an object, the module manager then triggers this event, passing the newly created object to all listeners.

loadModules.post This event is triggered by the module manager to allow any listeners to perform work after every module has finished loading. For example, the default configuration listener, `Zend\ModuleManager\Listener\ConfigListener` (covered later), attaches to this event to merge additional user-supplied configuration which is meant to override the default supplied configurations of installed modules.

Module Manager Listeners

By default, Zend Framework provides several useful module manager listeners.

Provided Module Manager Listeners

ZendModuleManagerListenerDefaultListenerAggregate To help simplify the most common use case of the module manager, ZF2 provides this default aggregate listener. In most cases, this will be the only listener you will need to attach to use the module manager, as it will take care of properly attaching the requisite listeners (those listed below) for the module system to function properly.

ZendModuleManagerListenerAutoloaderListener This listener checks each module to see if it has implemented `Zend\ModuleManager\Feature\AutoloaderProviderInterface` or simply defined the `getAutoloaderConfig()` method. If so, it calls the `getAutoloaderConfig()` method on the module class and passes the returned array to `Zend\Loader\AutoloaderFactory`.

ZendModuleManagerListenerConfigListener If a module class has a `getConfig()` method, this listener will call it and merge the returned array (or `Traversable` object) into the main application configuration.

ZendModuleManagerListenerInitTrigger If a module class either implements `Zend\ModuleManager\Feature\InitProviderInterface`, or simply defines an `init()` method, this listener will call `init()` and pass the current instance of `Zend\ModuleManager\ModuleManager` as the sole parameter. The `init()` method is called for **every** module implementing this feature, on **every** page request and should **only** be used for performing **lightweight** tasks such as registering event listeners.

ZendModuleManagerListenerLocatorRegistrationListener If a module class implements `Zend\ModuleManager\Feature\LocatorRegisteredInterface`, this listener will inject the module class instance into the `ServiceManager` using the module class name as the service name. This allows you to later retrieve the module class from the `ServiceManager`.

ZendModuleManagerListenerModuleResolverListener This is the default module resolver. It attaches to the "loadModule.resolve" event and simply returns an instance of `{moduleName}\Module`.

ZendModuleManagerListenerOnBootstrapListener If a module class implements `Zend\ModuleManager\Feature\BootstrapListenerInterface`, or simply defines an `onBootstrap()` method, this listener will register the `onBootstrap()` method with the `Zend\Mvc\Application` `bootstrap` event. This method will then be triggered during the `bootstrap` event (and passed an `MvcEvent` instance).

Like the `InitTrigger`, the `onBootstrap()` method is called for **every** module implementing this feature, on **every** page request, and should **only** be used for performing **lightweight** tasks such as registering event listeners.

ZendModuleManagerListenerServiceListener If a module class implements `Zend\ModuleManager\Feature\ServiceProviderInterface`, or simply defines an `getServiceConfig()` method, this listener will call that method and aggregate the return values for use in configuring the `ServiceManager`.

The `getServiceConfig()` method may return either an array of configuration compatible with `Zend\ServiceManager\Config`, an instance of that class, or the string name of a class that extends it. Values are merged and aggregated on completion, and then merged with any configuration from the `ConfigListener` falling under the `service_manager` key. For more information, see the `ServiceManager` documentation.

Unlike the other listeners, this listener is not managed by the `DefaultListenerAggregate`; instead, it is created and instantiated within the `Zend\Mvc\Service\ModuleManagerFactory`, where it is injected with the current `ServiceManager` instance before being registered with the `ModuleManager` events.

CHAPTER 111

The Module Class

By default, ZF2 module system simply expects each module name to be able to be resolved to an object instance. The default module resolver, `Zend\ModuleManager\Listener\ModuleResolverListener`, simply instantiates an instance of `{moduleName}\Module` for each enabled module.

A Minimal Module

As an example, provided the module name "MyModule", `Zend\ModuleManager\Listener\ModuleResolverListener` will simply expect the class `MyModule\Module` to be available. It relies on a registered autoloader, (typically `Zend\Loader\ModuleAutoloader`) to find and include the `MyModule\Module` class if it is not already available.

A module named "MyModule" module might start out looking something like this:

```
MyModule/
    Module.php
```

Within `Module.php`, you define your `MyModule\Module` class:

```
1  namespace MyModule;
2
3  class Module
4  {
5  }
```

Though it will not serve any purpose at this point, this "MyModule" module now has everything it needs to be considered a valid module and be loaded by the module system!

This `Module` class serves as the single entry point for module manager listeners to interact with a module. From within this simple, yet powerful class, modules can override or provide additional application configuration, perform initialization tasks such as registering autoloader(s) and event listeners, declaring dependencies, and much more.

A Typical Module Class

The following example shows a more typical usage of the `Module` class:

```
namespace MyModule;

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

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

For a list of the provided module manager listeners and the interfaces and methods that `Module` classes may implement in order to interact with the module manager and application, see the *module manager listeners documentation* and the *module mananger events documentation*.

The "loadModules.post" Event

It is not safe for a module to assume that any other modules have already been loaded at the time `init()` method is called. If your module needs to perform any actions after all other modules have been loaded, the module manager's "loadModules.post" event makes this easy.

Note: For more information on methods like `init()` and `getConfig()`, refer to the *module manager listeners documentation*.

Sample Usage of "loadModules.post" Event

```
use Zend\EventManager\EventInterface as Event;
use Zend\ModuleManager\ModuleManager;

class Module
{
    public function init(ModuleManager $moduleManger)
    {
        // Remember to keep the init() method as lightweight as possible
        $events = $moduleManager->getEventManager();
```

```
        $events->attach('loadModules.post', array($this, 'modulesLoaded'));
    }

    public function modulesLoaded(Event $e)
    {
        // This method is called once all modules are loaded.
        $moduleManager = $e->getTarget();
        $loadedModules = $moduleManager->getLoadedModules();
        $config        = $moduleManager->getConfig();
    }
}
```

The MVC "bootstrap" Event

If you are writing an MVC-oriented module for ZF2, you may need access to additional parts of the application in your `Module` class such as the instance of `Zend\Mvc\Application` or its registered service manager instance. For this, you may utilize the MVC "bootstrap" event. The bootstrap event is triggered after the "loadModule.post" event, once *$application->bootstrap()* is called.

Sample Usage of the MVC "bootstrap" Event

```
use Zend\EventManager\EventInterface as Event;

class Module
{
    public function onBootstrap(Event $e)
    {
        // This method is called once the MVC bootstrapping is complete
        $application = $e->getApplication();
        $services    = $application->getServiceManager();
    }
}
```

CHAPTER 112

The Module Autoloader

Zend Framework 2 ships with a default module autoloader. `Zend\Loader\ModuleAutoloader` is a specialized autoloader that is responsible for location of, and on-demand loading of, the `Module` classes from a variety of sources.

Module Autoloader Usage

If you are using the provided `Zend\ModuleManager\Listener\DefaultListenerAggregate`, then it is very simple to set up the module autoloader. You simply need to provide an array of module paths, either absolute or relative to the application's root, for the module autoloader to check when loading modules. The default listener aggregate will take care of instantiating and registering the module autoloader for you.

Keep in mind that in order for paths relative to your application directory to work, you must have the directive *chdir(dirname(__DIR__));* in your `public/index.php`.

Registering module paths with the default listener aggregate

The following example will search for modules in three different paths. Two are local directories for this application, and the third is a system-wide shared directory.

```
// public/index.php
use Zend\ModuleManager\Listener;
use Zend\ModuleManager\ModuleManager;

chdir(dirname(__DIR__));

// Instantiate and configure the default listener aggregate
$listenerOptions = new Listener\ListenerOptions(array(
    'module_paths' => array(
        './module',
        './vendor',
        '/usr/share/zfmodules',
    )
```

```
));
$defaultListeners = new Listener\DefaultListenerAggregate($listenerOptions);

// Instantiate the module manager
$moduleManager = new ModuleManager(array(
    'Application',
    'FooModule',
    'BarModule',
));

// Attach the default listener aggregate and load the modules
$moduleManager->getEventManager()->attachAggregate($defaultListeners);
$moduleManager->loadModules();
```

Note: Module paths behave very similar to the PHP include path, and are searched in the order they are defined. If you have modules with the same name in more than one registered module path, the module autoloader will return the first one it finds.

Non-Standard / Explicit Module Paths

Sometimes you may want to specify exactly where a module is instead of having `Zend\Loader\ModuleAutoloader` try to find it in the registered paths.

Registering a Non-Standard / Explicit Module Path

In this example, the autoloader will first check for `MyModule\Module` in `/path/to/mymoduledir-v1.2/Module.php`. If it's not found, then it will fall back to searching any other registered module paths.

```
// ./public/index.php
use Zend\Loader\ModuleAutoloader;
use Zend\ModuleManager\Listener;
use Zend\ModuleManager\ModuleManager;

chdir(dirname(__DIR__));

// Instantiate and configure the default listener aggregate
$listenerOptions = new Listener\ListenerOptions(array(
    'module_paths' => array(
        './module',
        './vendor',
        '/usr/share/zfmodules',
        'MyModule' => '/path/to/mymoduledir-v1.2',
    )
));
$defaultListeners = new Listener\DefaultListenerAggregate($listenerOptions);

/**
 * Without DefaultListenerAggregate:
 *
 * $moduleAutoloader = new ModuleAutoloader(array(
 *     './module',
 *     './vendor',
```

```
 *     '/usr/share/zfmodules',
 *     'MyModule' => '/path/to/mymoduledir-v1.2',
 * ));
 * $moduleAutoloader->register();
 *
 */

// Instantiate the module manager
$moduleManager = new ModuleManager(array(
    'MyModule',
    'FooModule',
    'BarModule',
));

// Attach the default listener aggregate and load the modules
$moduleManager->getEventManager()->attachAggregate($defaultListeners);
$moduleManager->loadModules();
```

This same method works if you provide the path to a phar archive.

Packaging Modules with Phar

If you prefer, you may easily package your module as a phar archive. The module autoloader is able to autoload modules in the following archive formats: .phar, .phar.gz, .phar.bz2, .phar.tar, .phar.tar.gz, .phar.tar.bz2, .phar.zip, .tar, .tar.gz, .tar.bz2, and .zip.

The easiest way to package your module is to simply tar the module directory. You can then replace the `MyModule/` directory with `MyModule.tar`, and it should still be autoloaded without any additional changes!

Note: If possible, avoid using any type of compression (bz2, gz, zip) on your phar archives, as it introduces unnecessary CPU overhead to each request.

CHAPTER 113

Best Practices when Creating Modules

When creating a ZF2 module, there are some best practices you should keep in mind.

- **Keep the init() method lightweight.** Be conservative with the actions you perform in the `init()` and `onBootstrap()` methods of your `Module` class. These methods are run for **every** page request, and should not perform anything heavy. As a rule of thumb, registering event listeners is an appropriate task to perform in these methods. Such lightweight tasks will generally not have a measurable impact on the performance of your application, even with many modules enabled. It is considered bad practice to utilize these methods for setting up or configuring instances of application resources such as a database connection, application logger, or mailer. Tasks such as these are better served through the service manager capabilities of Zend Framework 2.
- **Do not perform writes within a module.** You should **never** code your module to perform or expect any writes within the module's directory. Once installed, the files within a module's directory should always match the distribution verbatim. Any user-provided configuration should be performed via overrides in the Application module or via application-level configuration files. Any other required filesystem writes should be performed in some writeable path that is outside of the module's directory.

  There are two primary advantages to following this rule. First, any modules which attempt to write within themselves will not be compatible with phar packaging. Second, by keeping the module in sync with the upstream distribution, updates via mechanisms such as Git will be simple and trouble-free. Of course, the Application module is a special exception to this rule, as there is typically no upstream distribution for this module, and it's unlikely you would want to run this package from within a phar archive.
- **Utilize a vendor prefix for module names.** To avoid module naming conflicts, you are encouraged to prefix your module namespace with a vendor prefix. As an example, the (incomplete) developer tools module distributed by Zend is named "ZendDeveloperTools" instead of simply "DeveloperTools".

CHAPTER 114

Introduction to the MVC Layer

`Zend\Mvc` is a brand new MVC implementation designed from the ground up for Zend Framework 2.0. The focus of this implementation is performance and flexibility.

The MVC layer is built on top of the following components:

- `Zend\ServiceManager`. Zend Framework provides a set of default service definitions to use in order to create and configure your application instance and workflow.
- `Zend\EventManager`, which is used everywhere from initial bootstrapping of the application to returning the response; the MVC is event driven.
- `Zend\Http`, specifically the request and response objects, which are used with:
- `Zend\Stdlib\DispatchableInterface`; all "controllers" are simply dispatchable objects

Within the MVC layer, several subcomponents are exposed:

- `Zend\Mvc\Router` contains classes pertaining to routing a request (the act of matching a request to a controller, or dispatchable)
- `Zend\Mvc\PhpEnvironment`, a set of decorators for the HTTP `Request` and `Response` objects that ensure the request is injected with the current environment (including query parameters, POST parameters, HTTP headers, etc.)
- `Zend\Mvc\Controller`, a set of abstract "controller" classes with basic responsibilities such as event wiring, action dispatching, etc.
- `Zend\Mvc\Service`, which provides a set of `ServiceManager` factories and definitions for the default application workflow.
- `Zend\Mvc\View`, which provides the default wiring for renderer selection, view script resolution, helper registration, and more; additionally, it provides a number of listeners that tie into the MVC workflow to provide features such as automated template name resolution, automated view model creation and injection, and more.

The gateway to the MVC is the `Zend\Mvc\Application` object (referred to simply as `Application` from this point forward). Its primary responsibilities are to **bootstrap** resources, **route** the request, and to retrieve and **dispatch** the controller discovered. Once accomplished, it returns a response, which can then be **sent**.

Basic Application Structure

The basic structure of an application is as follows:

```
application_root/
    config/
        application.php
        autoload/
            global.php
            local.php
            // etc.
    data/
    module/
    vendor/
    public/
        .htaccess
        index.php
```

The `public/index.php` performs the basic work of martialling configuration and configuring the `Application`. Once done, it `run()`s the `Application` and `send()`s the response returned.

The `config` directory will typically contain configuration used by `Zend\Module\Manager` in order to load modules and merge configuration; we will detail this more later.

The `vendor` subdirectory should contain any third-party modules or libraries on which your application depends. This might include Zend Framework, custom libraries from your organization, or other third-party libraries from other projects. Libraries and modules placed in the `vendor` subdirectory should not be modified from their original, distributed state.

Finally, the `module` directory will contain one or more modules delivering your application's functionality.

Let's now turn to modules, as they are the basic units of a web application.

Basic Module Structure

A module may contain just about anything: PHP code, including MVC functionality; library code; view scripts; and/or or public assets such as images, CSS, and JavaScript. The one requirement – and even this is optional – is that a module acts as a PHP namespace and that it contains a `Module` class under that namespace. This class will then be consumed by `Zend\Module\Manager` in order to perform a number of tasks.

The recommended structure of a module is as follows:

```
module_root/
    Module.php
    autoload_classmap.php
    autoload_function.php
    autoload_register.php
    config/
        module.config.php
    public/
        images/
        css/
        js/
    src/
        <module_namespace>/
            <code files>
    test/
```

```
        phpunit.xml
        bootstrap.php
        <module_namespace>/
            <test code files>
    view/
        <dir-named-after-module-namespace>/
            <dir-named-after-a-controller>/
                <.phtml files>
```

Since a module acts as a namespace, the module root directory should be that namespace. Typically, this namespace will also include a vendor prefix of sorts. As an example a module centered around "User" functionality delivered by Zend might be named "ZendUser", and this is also what the module root directory will be named.

The `Module.php` file directly under the module root directory will be in the module namespace.

```
1 namespace ZendUser;
2
3 class Module
4 {
5 }
```

By default, if an `init()` method is defined, this method will be triggered by a `Zend\Module\Manager` listener when it loads the module class, and passed an instance of the manager. This allows you to perform tasks such as setting up module-specific event listeners. The `init()` method is called for **every** module on **every** page request and should **only** be used for performing **lightweight** tasks such as registering event listeners. Similarly, an `onBootstrap()` method (which accepts an `MvcEvent` instance) may be defined; it will be triggered for every page request, and should be used for lightweight tasks only.

The three `autoload_*.php` files are not required, but recommended. They provide the following:

- `autoload_classmap.php` should return an array classmap of class name/filename pairs (with the filenames resolved via the `__DIR__` magic constant).
- `autoload_function.php` should return a PHP callback that can be passed to `spl_autoload_register()`. Typically, this callback should utilize the map returned by `autoload_filemap.php`.
- `autoload_register.php` should register a PHP callback (typically that returned by `autoload_function.php` with `spl_autoload_register()`.

The point of these three files is to provide reasonable default mechanisms for autoloading the classes contained in the module, thus providing a trivial way to consume the module without requiring `Zend\Module` (e.g., for use outside a ZF2 application).

The `config` directory should contain any module-specific configuration. These files may be in any format `Zend\Config` supports. We recommend naming the main configuration "module.format", and for PHP-based configuration, "module.config.php". Typically, you will create configuration for the router as well as for the dependency injector.

The `src` directory should be a PSR-0 compliant directory structure with your module's source code. Typically, you should at least have one subdirectory named after your module namespace; however, you can ship code from multiple namespaces if desired.

The `test` directory should contain your unit tests. Typically, these will be written using PHPUnit, and contain artifacts related to its configuration (e.g., `phpunit.xml`, `bootstrap.php`).

The `public` directory can be used for assets that you may want to expose in your application's document root. These might include images, CSS files, JavaScript files, etc. How these are exposed is left to the developer.

The `view` directory contains view scripts related to your controllers.

Bootstrapping an Application

The `Application` has six basic dependencies.

- **configuration**, usually an array or object implementing `ArrayAccess`.
- **ServiceManager** instance.
- **EventManager** instance, which, by default, is pulled from the `ServiceManager`, by the service name "EventManager".
- **ModuleManager** instance, which, by default, is pulled from the `ServiceManager`, by the service name "ModuleManager".
- **Request** instance, which, by default, is pulled from the `ServiceManager`, by the service name "Request".
- **Response** instance, which, by default, is pulled from the `ServiceManager`, by the service name "Response".

These may be satisfied at instantiation:

```
use Zend\EventManager\EventManager;
use Zend\Http\PhpEnvironment;
use Zend\ModuleManager\ModuleManager;
use Zend\Mvc\Application;
use Zend\ServiceManager\ServiceManager;

$config = include 'config/application.php';

$serviceManager = new ServiceManager();
$serviceManager->setService('EventManager', new EventManager());
$serviceManager->setService('ModuleManager', new ModuleManager());
$serviceManager->setService('Request', new PhpEnvironment\Request());
$serviceManager->setService('Response', new PhpEnvironment\Response());

$application = new Application($config, $serviceManager);
```

Once you've done this, there are two additional actions you can take. The first is to "bootstrap" the application. In the default implementation, this does the following:

- Attaches the default route listener (`Zend\Mvc\RouteListener`).
- Attaches the default dispatch listener (`Zend\Mvc\DispatchListener`).
- Attaches the `ViewManager` listener (`Zend\Mvc\View\ViewManager`).
- Creates the `MvcEvent`, and injects it with the application, request, and response; it also retrieves the router (`Zend\Mvc\Router\Http\TreeRouteStack`) at this time and attaches it to the event.
- Triggers the "bootstrap" event.

If you do not want these actions, or want to provide alternatives, you can do so by extending the `Application` class and/or simply coding what actions you want to occur.

The second action you can take with the configured `Application` is to `run()` it. Calling this method simply does the following: it triggers the "route" event, followed by the "dispatch" event, and, depending on execution, the "render" event; when done, it triggers the "finish" event, and then returns the response instance. If an error occurs during either the "route" or "dispatch" event, a "dispatch.error" event is triggered as well.

This is a lot to remember in order to bootstrap the application; in fact, we haven't covered all the services available by default yet. You can greatly simplify things by using the default `ServiceManager` configuration shipped with the MVC.

```php
use Zend\Loader\AutoloaderFactory;
use Zend\Mvc\Service\ServiceManagerConfig;
use Zend\ServiceManager\ServiceManager;

// setup autoloader
AutoloaderFactory::factory();

// get application stack configuration
$configuration = include 'config/application.config.php';

// setup service manager
$serviceManager = new ServiceManager(new ServiceManagerConfig());
$serviceManager->setService('ApplicationConfig', $configuration);

// load modules -- which will provide services, configuration, and more
$serviceManager->get('ModuleManager')->loadModules();

// bootstrap and run application
$application = $serviceManager->get('Application');
$application->bootstrap();
$response = $application->run();
$response->send();
```

You'll note that you have a great amount of control over the workflow. Using the `ServiceManager`, you have fine-grained control over what services are available, how they are instantiated, and what dependencies are injected into them. Using the `EventManager`'s priority system, you can intercept any of the application events ("bootstrap", "route", "dispatch", "dispatch.error", "render", and "finish") anywhere during execution, allowing you to craft your own application workflows as needed.

Bootstrapping a Modular Application

While the previous approach largely works, where does the configuration come from? When we create a modular application, the assumption will be that it's from the modules themselves. How do we get that information and aggregate it, then?

The answer is via `Zend\ModuleManager\ModuleManager`. This component allows you to specify where modules exist, and it will then locate each module and initialize it. Module classes can tie into various listeners on the `ModuleManager` in order to provide configuration, services, listeners, and more to the application. Sound complicated? It's not.

Configuring the Module Manager

The first step is configuring the module manager. You simply inform the module manager which modules to load, and potentially provide configuration for the module listeners.

Remember the `application.php` from earlier? We're going to provide some configuration.

```php
<?php
// config/application.php
return array(
    'modules' => array(
        /* ... */
    ),
    'module_listener_options' => array(
```

```
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

As we add modules to the system, we'll add items to the `modules` array.

Each `Module` class that has configuration it wants the `Application` to know about should define a `getConfig()` method. That method should return an array or `Traversable` object such as `Zend\Config\Config`. As an example:

```
namespace ZendUser;

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

There are a number of other methods you can define for tasks ranging from providing autoloader configuration, to providing services to the `ServiceManager`, to listening to the bootstrap event. The ModuleManager documentation goes into more detail on these.

Conclusion

The ZF2 MVC layer is incredibly flexible, offering an opt-in, easy to create modular infrastructure, as well as the ability to craft your own application workflows via the `ServiceManager` and `EventManager`. The module manager is a lightweight and simple approach to enforcing a modular architecture that encourages clean separation of concerns and code re-use.

CHAPTER 115

Quick Start

Now that you know the basics of how applications and modules are structured, we'll show you the easy way to get started.

Install the Zend Skeleton Application

The easiest way to get started is to grab the sample application and module repositories. This can be done in the following ways.

Using Composer

Simply clone the `ZendSkeletonApplication` repository:

```
prompt> git clone git://github.com/zendframework/ZendSkeletonApplication.git my-
↪application
```

Then run Composer's `install` command to install the ZF library and any other configured dependencies:

```
prompt> php ./composer.phar install
```

Using Git

Simply clone the `ZendSkeletonApplication` repository, using the `--recursive` option, which will also grab ZF.

```
prompt> git clone --recursive git://github.com/zendframework/ZendSkeletonApplication.
↪git my-application
```

Manual installation

- Download a tarball of the `ZendSkeletonApplication` repository:
  - Zip: https://github.com/zendframework/ZendSkeletonApplication/zipball/master
  - Tarball: https://github.com/zendframework/ZendSkeletonApplication/tarball/master
- Deflate the archive you selected and rename the parent directory according to your project needs; we use "my-application" throughout this document.
- Install Zend Framework, and either have its library on your PHP `include_path`, symlink the library into your project's "library", or install it directly into your application using Pyrus.

Create a new module

By default, one module is provided with the `ZendSkeletonApplication`, named "Application". It provides simply a controller to handle the "home" page of the application, the layout template, and templates for 404 and error pages.

Typically, you will not need to touch this other than to provide an alternate entry page for your site and/or alternate error page.

Additional functionality will be provided by creating new modules.

To get you started with modules, we recommend using the `ZendSkeletonModule` as a base. Download it from here:

- Zip: https://github.com/zendframework/ZendSkeletonModule/zipball/master
- Tarball: https://github.com/zendframework/ZendSkeletonModule/tarball/master

Deflate the package, and rename the directory "ZendSkeletonModule" to reflect the name of the new module you want to create; when done, move the module into your new project's `modules/` directory.

At this point, it's time to create some functionality.

Update the Module class

Let's update the module class. We'll want to make sure the namespace is correct, configuration is enabled and returned, and that we setup autoloading on initialization. Since we're actively working on this module, the class list will be in flux, we probably want to be pretty lenient in our autoloading approach, so let's keep it flexible by using the `StandardAutoloader`. Let's begin.

First, let's have `autoload_classmap.php` return an empty array:

```
<?php
// autoload_classmap.php
return array();
```

We'll also edit our `config/module.config.php` file to read as follows:

```
return array(
    'view_manager' => array(
        'template_path_stack' => array(
            '<module-name>' => __DIR__ . '/../view'
        ),
```

```
    ),
);
```

Fill in "module-name" with a lowercased, dash-separated version of your module name – e.g., "ZendUser" would become "zend-user".

Next, edit the `Module.php` file to read as follows:

```
namespace <your module name here>;

use Zend\ModuleManager\Feature\AutoloaderProviderInterface;
use Zend\ModuleManager\Feature\ConfigProviderInterface;

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

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

At this point, you now have your module configured properly. Let's create a controller!

Create a Controller

Controllers are simply objects that implement `Zend\Stdlib\DispatchableInterface`. This means they simply need to implement a `dispatch()` method that takes minimally a `Response` object as an argument.

In practice, though, this would mean writing logic to branch based on matched routing within every controller. As such, we've created two base controller classes for you to start with:

- `Zend\Mvc\Controller\AbstractActionController` allows routes to match an "action". When matched, a method named after the action will be called by the controller. As an example, if you had a route that returned "foo" for the "action" key, the "fooAction" method would be invoked.
- `Zend\Mvc\Controller\AbstractRestfulController` introspects the Request to determine what HTTP method was used, and calls a method based on that accordingly.
  - `GET` will call either the `getList()` method, or, if an "id" was matched during routing, the `get()` method (with that identifer value).
  - `POST` will call the `create()` method, passing in the `$_POST` values.

  - `PUT` expects an "id" to be matched during routing, and will call the `update()` method, passing in the identifier, and any data found in the raw post body.
  - `DELETE` expects an "id" to be matched during routing, and will call the `delete()` method.

To get started, we'll simply create a "hello world" style controller, with a single action. First, create the directory `src/<module name>/Controller`, and then create the file `HelloController.php` inside it. Edit it in your favorite text editor or IDE, and insert the following contents:

```
<?php
namespace <module name>\Controller;

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

class HelloController extends AbstractActionController
{
    public function worldAction()
    {
        $message = $this->params()->fromQuery('message', 'foo');
        return new ViewModel(array('message' => $message));
    }
}
```

So, what are we doing here?

- We're creating an action controller.
- We're defining an action, "world".
- We're pulling a message from the query parameters (yes, this is a superbly bad idea in production! Always sanitize your inputs!).
- We're returning a ViewModel with an array of values that will get processed later.

We return a `ViewModel`. The view layer will use this when rendering the view, pulling variables and the template name from it. By default, you can omit the template name, and it will resolve to "lowercase-controller-name/lowercase-action-name". However, you can override this to specify something different by calling `setTemplate()` on the `ViewModel` instance. Typically, templates will resolve to files with a ".phtml" suffix in your module's `view` directory.

So, with that in mind, let's create a view script.

Create a view script

Create the directory `view/<module-name>hello`. Inside that directory, create a file named `world.phtml`. Inside that, paste in the following:

```
<h1>Greetings!</h1>

<p>You said "<?php echo $this->escapeHtml($message) ?>".</p>
```

That's it. Save the file.

Note: What is the method `escapeHtml()`? It's actually a *view helper*, and it's designed to help mitigate *XSS* attacks. Never trust user input; if you are at all uncertain about the source of a given variable in your view script, escape it using one of the *provided escape view helper* depending on the type of data you have.

Create a route

Now that we have a controller and a view script, we need to create a route to it.

Note: `ZendSkeletonApplication` ships with a "default route" that will likely get you to this action. That route basically expects "/{module}/{controller}/{action}", which allows you to specify this: "/zend-user/hello/world". We're going to create a route here mainly for illustration purposes, as creating explicit routes is a recommended practice. The application will look for a `Zend\Mvc\Router\RouteStack` instance to setup routing. The default generated router is a `Zend\Mvc\Router\Http\TreeRouteStack`.

To use the "default route" functionality, you will need to add the following route definition to your module. Replace

```
return array(
    '<module-name>' => array(
        'type'    => 'Literal',
        'options' => array(
            'route'    => '/<module-name>',
            'defaults' => array(
                '__NAMESPACE__' => '<module-namespace>\Controller',
                'controller'    => '<module-name>-Index',
                'action'        => 'index',
            ),
        ),
        'may_terminate' => true,
        'child_routes' => array(
            'default' => array(
                'type'    => 'Segment',
                'options' => array(
                    'route'    => '/[:controller[/:action]]',
                    'constraints' => array(
                        'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                        'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
                    ),
                    'defaults' => array(
                    ),
                ),
            ),
        ),
    ),
    'controller' => array(
        'classes' => array(
            '<module-name>-Index' => '<module-namespace>\Controller\IndexController',
            // Do similar for each other controller in your module
        ),
    ),
    // ... other configuration ...
);
```

Additionally, we need to tell the application we have a controller.

Note: We inform the application about controllers we expect to have in the application. This is to prevent somebody requesting any service the `ServiceManager` knows about in an attempt to break the application. The dispatcher uses a special, scoped container that will only pull controllers that are specifically registered with it, either as invokable classes or via factories.

Open your `config/module.config.php` file, and modify it to add to the "routes" and "controller" parameters so it reads as follows:

```
return array(
    'routes' => array(
        '<module name>-hello-world' => array(
            'type'    => 'Literal',
                'options' => array(
                'route' => '/hello/world',
                'defaults' => array(
                    'controller' => '<module namespace>-Hello',
                    'action'     => 'world',
                ),
            ),
        ),
    ),
    'controller' => array(
        'classes' => array(
            '<module namespace>-Hello' => '<module namespace>
↪\Controller\HelloController',
        ),
    ),
    // ... other configuration ...
);
```

Tell the application about our module

One problem: we haven't told our application about our new module!

By default, modules are not parsed unless we tell the module manager about them. As such, we need to notify the application about them.

Remember the `config/application.php` file? Let's modify it to add our new module. Once done, it should read as follows:

```
<?php
return array(
    'modules' => array(
        'Application',
        '<module namespace>',
    ),
    'module_listener_options' => array(
        'module_paths' => array(
            './module',
            './vendor',
        ),
    ),
);
```

Replace `<module namespace>` with the namespace of your module.

Test it out!

Now we can test things out! Create a new vhost pointing its document root to the `public` directory of your application, and fire it up in a browser. You should see the default homepage template of ZendSkeletonApplication.

Now alter the location in your URL to append the path "hello/world", and load the page. You should now get the following content:

```
<h1>Greetings!</h1>

<p>You said "foo".</p>
```

Now alter the location to append "?message=bar" and load the page. You should now get:

```
<h1>Greetings!</h1>

<p>You said "bar".</p>
```

Congratulations! You've created your first ZF2 MVC module!

CHAPTER 116

Default Services

The default and recommended way to write Zend Framework applications uses a set of services defined in the `Zend\Mvc\Service` namespace. This chapter details what each of those services are, the classes they represent, and the configuration options available.

ServiceManager

This is the one service class referenced directly in the bootstrapping. It provides the following:

- **Invokable services**
  - `DispatchListener`, mapping to `Zend\Mvc\DispatchListener`.
  - `Request`, mapping to `Zend\Http\PhpEnvironment\Request`.
  - `Response`, mapping to `Zend\Http\PhpEnvironment\Response`.
  - `RouteListener`, mapping to `Zend\Mvc\RouteListener`.
  - `ViewManager`, mapping to `Zend\Mvc\View\ViewManager`.
- **Factories**
  - `Application`, mapping to `Zend\Mvc\Service\ApplicationFactory`.
  - `Configuration`, mapping to `Zend\Mvc\Service\ConfigFactory`. Internally, this pulls the `ModuleManager` service, and calls its `loadModules()` method, and retrieves the merged configuration from the module event. As such, this service contains the entire, merged application configuration.
  - `ControllerLoader`, mapping to `Zend\Mvc\Service\ControllerLoaderFactory`. Internally, this pulls the `Configuration` service, and, if it contains a `controller` key, inspects that for `classes` and `factories` subkeys. These are used to configure a scoped service manager container, from which controllers will be retrieved.

    Additionally, the scoped container is configured to use the `Di` service as an abstract service factory – effectively allowing you to fall back to DI in order to retrieve your controllers. If you want to use

Zend\Di to retrieve your controllers, you must white-list them in your DI configuration under the allowed_controllers key (otherwise, they will just be ignored).

Finally, if the loaded controller is Pluggable, an initializer will inject it with the ControllerPluginBroker service.

- ControllerPluginBroker, mapping to Zend\Mvc\Service\ControllerPluginBrokerFactory. This instantiates the Zend\Mvc\Controller\PluginBroker instance, passing it the ControllerPluginLoader service as well as the service manager instance.
- ControllerPluginLoader, mapping to Zend\Mvc\Service\ControllerPluginLoaderFactory. This grabs the Configuration service, and looks for a controller key with a map subkey. If found, this value is passed to the constructor of Zend\Mvc\Controller\PluginLoader (otherwise, an empty array is passed).
- DependencyInjector, mapping to Zend\Mvc\Service\DiFactory. This pulls the Configuration service, and looks for a "di" key; if found, that value is used to configure a new Zend\Di\Di instance. Additionally, the Di instance is used to seed a Zend\ServiceManager\Di\DiAbstractServiceFactory instance which is then attached to the service manager as an abstract factory – effectively enabling DI as a fallback for providing services.
- EventManager, mapping to Zend\Mvc\Service\EventManagerFactory. This factory composes a static reference to a SharedEventManager, which is injected in a new EventManager instance. This service is not shared by default, allowing the ability to have an EventManager per service, with a shared SharedEventManager injected in each.
- ModuleManager, mapping to Zend\Mvc\Service\ModuleManagerFactory.

  This is perhaps the most complex factory in the MVC stack. It expects that an ApplicationConfiguration service has been injected, with keys for module_listener_options and modules; see the quick start for samples.

  It instantiates an instance of Zend\ModuleManager\Listener\DefaultListenerAggregate, using the "module_listener_options" retrieved. It also instantiates an instance of Zend\ModuleManager\Listener\ServiceListener, providing it the service manager.

  Next, it retrieves the EventManager service, and attaches the above listeners.

  It instantiates a Zend\ModuleManager\ModuleEvent instance, setting the "ServiceManager" parameter to the service manager object.

  Finally, it instantiates a Zend\ModuleManager\ModuleManager instance, and injects the EventManager and ModuleEvent.
- Router, mapping to Zend\Mvc\Service\RouterFactory. This grabs the Configuration service, and pulls from the router key, passing it to Zend\Mvc\Router\Http\TreeRouteStack::factory in order to get a configured router instance.
- ViewFeedRenderer, mapping to Zend\Mvc\Service\ViewFeedRendererFactory, which simply returns a Zend\View\Renderer\FeedRenderer instance.
- ViewFeedStrategy, mapping to Zend\Mvc\Service\ViewFeedStrategyFactory. This instantiates a Zend\View\Strategy\FeedStrategy instance with the ViewFeedRenderer service.
- ViewJsonRenderer, mapping to Zend\Mvc\Service\ViewJsonRendererFactory, which simply returns a Zend\View\Renderer\JsonRenderer instance.
- ViewJsonStrategy, mapping to Zend\Mvc\Service\ViewJsonStrategyFactory. This instantiates a Zend\View\Strategy\JsonStrategy instance with the ViewJsonRenderer service.

- **Aliases**
  - `Config`, mapping to the `Configuration` service.
  - `Di`, mapping to the `DependencyInjector` service.
  - `Zend\EventManager\EventManagerInterface`, mapping to the `EventManager` service. This is mainly to ensure that when falling through to DI, classes are still injected via the `ServiceManager`.
  - `Zend\Mvc\Controller\PluginBroker`, mapping to the `ControllerPluginBroker` service. This is mainly to ensure that when falling through to DI, classes are still injected via the `ServiceManager`.
  - `Zend\Mvc\Controller\PluginLoader`, mapping to the `ControllerPluginLoader` service. This is mainly to ensure that when falling through to DI, classes are still injected via the `ServiceManager`.

Additionally, two initializers are registered. Initializers are run on created instances, and may be used to further configure them. The two initializers the `ServiceManagerConfig` class creates and registers do the following:

- For objects that implement `Zend\EventManager\EventManagerAwareInterface`, the `EventManager` service will be retrieved and injected. This service is **not** shared, though each instance it creates is injected with a shared instance of `SharedEventManager`.
- For objects that implement `Zend\ServiceManager\ServiceManagerAwareInterface`, the `ServiceManager` will inject itself into the object.

Finally, the `ServiceManager` registers itself as the `ServiceManager` service, and aliases itself to the class names `Zend\ServiceManager\ServiceManagerInterface` and `Zend\ServiceManager\ServiceManager`.

ViewManager

The View layer within `Zend\Mvc` consists of a large number of collaborators and event listeners. As such, `Zend\Mvc\View\ViewManager` was created to handle creation of the various objects, as well as wiring them together and establishing event listeners.

The `ViewManager` itself is an event listener on the `bootstrap` event. It retrieves the `ServiceManager` from the `Application` object, as well as its composed `EventManager`.

Configuration for all members of the `ViewManager` fall under the `view_manager` configuration key, and expect values as noted below. The following services are created and managed by the `ViewManager`:

- `ViewHelperLoader`, representing and aliased to `Zend\View\HelperLoader`. If a `helper_map` subkey is provided, its value will be used as a map to seed the helper loader.
- `ViewHelperBroker`, representing and aliased to `Zend\View\HelperBroker`. It is seeded with the `ViewHelperLoader` service, as well as the `ServiceManager` itself.

  The `Router` service is retrieved, and injected into the `Url` helper.

  If the `base_path` key is present, it is used to inject the `BasePath` view helper; otherwise, the `Request` service is retrieved, and the value of its `getBasePath()` method is used.

  If the `doctype` key is present, it will be used to set the value of the `Doctype` view helper.
- `ViewTemplateMapResolver`, representing and aliased to `Zend\View\Resolver\TemplateMapResolver`. If a `template_map` key is present, it will be used to seed the template map.

- `ViewTemplatePathStack`, representing and aliased to `Zend\View\Resolver\TemplatePathStack`. If a `template_path_stack` key is prsent, it will be used to seed the stack.
- `ViewResolver`, representing and aliased to `Zend\View\Resolver\AggregateResolver` and `Zend\View\Resolver\ResolverInterface`. It is seeded with the `ViewTemplateMapResolver` and `ViewTemplatePathStack` services as resolvers.
- `ViewRenderer`, representing and aliased to `Zend\View\Renderer\PhpRenderer` and `Zend\View\Renderer\RendererInterface`. It is seeded with the `ViewResolver` and `ViewHelperBroker` services. Additionally, the `ViewModel` helper gets seeded with the `ViewModel` as its root (layout) model.
- `ViewPhpRendererStrategy`, representing and aliased to `Zend\View\Strategy\PhpRendererStrategy`. It gets seeded with the `ViewRenderer` service.
- `View`, representing and aliased to `Zend\View\View`. It gets seeded with the `EventManager` service, and attaches the `ViewPhpRendererStrategy` as an aggregate listener.
- `DefaultRenderingStrategy`, representing and aliased to `Zend\Mvc\View\DefaultRenderingStrategy`. If the `layout` key is prsent, it is used to seed the strategy's layout template. It is seeded with the `View` service.
- `ExceptionStrategy`, representing and aliased to `Zend\Mvc\View\ExceptionStrategy`. If the `dislay_exceptions` or `exception_template` keys are present, they are usd to configure the strategy.
- `RouteNotFoundStrategy`, representing and aliased to `Zend\Mvc\View\RouteNotFoundStrategy` and `404Stategy`. If the `display_not_found_reason` or `not_found_template` keys are present, they are used to configure the strategy.
- `ViewModel`. In this case, no service is registered; the `ViewModel` is simply retrieved from the `MvcEvent` and injected with the layout template name. template

The `ViewManager` also creates several other listeners, but does not expose them as services; these include `Zend\Mvc\View\CreateViewModelListener`, `Zend\Mvc\View\InjectTemplateListener`, and `Zend\Mvc\View\InjectViewModelListener`. These, along with `RouteNotFoundStrategy`, `ExceptionStrategy`, and `DefaultRenderingStrategy` are attached as listeners either to the application `EventManager` instance or the `SharedEventManager` instance.

Finally, if you have a `strategies` key in your configuration, the `ViewManager` will loop over these and attach them in order to the `View` service as listeners, at a priority of 100 (allowing them to execute before the `DefaultRenderingStrategy`).

Application Configuration Options

The following options may be used to provide initial configuration for the `ServiceManager`, `ModuleManager`, and `Application` instances, allowing them to then find and aggregate the configuration used for the `Configuration` service, which is intended for configuring all other objects in the system.

```
<?php
return array(
    // This should be an array of module namespaces used in the application.
    'modules' => array(
    ),

    // These are various options for the listeners attached to the ModuleManager
    'module_listener_options' => array(
        // This should be an array of paths in which modules reside.
        // If a string key is provided, the listener will consider that a module
```

```
        // namespace, the value of that key the specific path to that module's
        // Module class.
        'module_paths' => array(
        ),

        // An array of paths from which to glob configuration files after
        // modules are loaded. These effectively overide configuration
        // provided by modules themselves. Paths may use GLOB_BRACE notation.
        'config_glob_paths' => array(
        ),

        // Whether or not to enable a configuration cache.
        // If enabled, the merged configuration will be cached and used in
        // subsequent requests.
        'config_cache_enabled' => $booleanValue,

        // The key used to create the configuration cache file name.
        'config_cache_key' => $stringKey,

        // The path in which to cache merged configuration.
        'cache_dir' => $stringPath,
    ),

    // Initial configuration with which to seed the ServiceManager.
    // Should be compatible with Zend\ServiceManager\Config.
    'service_manager' => array(
    ),
);
```

Default Configuration Options

The following options are available when using the default services configured by the `ServiceManagerConfig` and `ViewManager`.

```
<?php
return array(
    // The following are used to configure controller or controller plugin loading
    'controller' => array(
        // Map of controller "name" to class
        // This should be used if you do not need to inject any dependencies
        // in your controller
        'classes' => array(
        ),

        // Map of controller "name" to factory for creating controller instance
        // You may provide either the class name of a factory, or a PHP callback.
        'factories' => array(
        ),

        // Map of controller plugin names to their classes
        'map' => array(
        ),
    ),

    // The following is used to configure a Zend\Di\Di instance.
```

```
    // The array should be in a format that Zend\Di\Config can understand.
    'di' => array(
    ),

    // Configuration for the Router service
    // Can contain any router configuration, but typically will always define
    // the routes for the application. See the router documentation for details
    // on route configuration.
    'router' => array(
        'routes' => array(
        ),
    ),

    // ViewManager configuration
    'view_manager' => array(
        // Defined helpers.
        // Typically helper name/helper class pairs. Can contain values without keys
        // that refer to either Traversable classes or Zend\Loader\PluginClassLoader
        // instances as well.
        'helper_map' => array(
            'foo' => 'My\Helper\Foo',      // name/class pair
            'Zend\Form\View\HelperLoader', // additional helper loader to seed
        ),

        // Base URL path to the application
        'base_path' => $stringBasePath,

        // Doctype with which to seed the Doctype helper
        'doctype' => $doctypeHelperConstantString, // e.g. HTML5, XHTML1

        // TemplateMapResolver configuration
        // template/path pairs
        'template_map' => array(
        ),

        // TemplatePathStack configuration
        // module/view script path pairs
        'template_path_stack' => array(
        ),

        // Layout template name
        'layout' => $layoutTemplateName, // e.g., 'layout/layout'

        // ExceptionStrategy configuration
        'display_exceptions' => $bool, // display exceptions in template
        'exception_template' => $stringTemplateName, // e.g. 'error'

        // RouteNotFoundStrategy configuration
        'display_not_found_reason' => $bool, // display 404 reason in template
        'not_found_template' => $stringTemplateName, // e.g. '404'

        // Additional strategies to attach
        // These should be class names or service names of View strategy classes
        // that act as ListenerAggregates. They will be attached at priority 100,
        // in the order registered.
        'strategies' => array(
            'ViewJsonStrategy', // register JSON renderer strategy
            'ViewFeedStrategy', // register Feed renderer strategy
```

```
80              ),
81          ),
82  );
```

CHAPTER 117

Routing

Routing is the act of matching a request to a given controller.

Typically, routing will examine the request URI, and attempt to match the URI path segment against provided constraints. If the constraints match, a set of "matches" are returned, one of which should be the controller name to execute. Routing can utilize other portions of the request URI or environment as well – for example, the host or scheme, query parametes, headers, request method, and more.

Routing has been written from the ground up for Zend Framework 2.0. Execution is quite similar, but the internal workings are more consistent, performant, and often simpler.

The base unit of routing is a `Route`:

```
namespace Zend\Mvc\Router;

use zend\Stdlib\RequestInterface as Request;

interface Route
{
    public static function factory(array $options = array());
    public function match(Request $request);
    public function assemble(array $params = array(), array $options = array());
}
```

A `Route` accepts a `Request`, and determines if it matches. If so, it returns a `RouteMatch` object:

```
namespace Zend\Mvc\Router;

class RouteMatch
{
    public function __construct(array $params);
    public function setParam($name, $value);
    public function merge(RouteMatch $match);
    public function getParam($name, $default = null);
    public function getRoute();
}
```

Typically, when a `Route` matches, it will define one or more parameters. These are passed into the `RouteMatch`, and objects may query the `RouteMatch` for their values.

```
$id = $routeMatch->getParam('id', false);
if (!$id) {
    throw new Exception('Required identifier is missing!');
}
$entity = $resource->get($id);
```

Usually you will have multiple routes you wish to test against. In order to facilitate this, you will use a route aggregate, usually implementing `RouteStack`:

```
namespace Zend\Mvc\Router;

interface RouteStack extends Route
{
    public function addRoute($name, $route, $priority = null);
    public function addRoutes(array $routes);
    public function removeRoute($name);
}
```

Typically, routes should be queried in a LIFO order, and hence the reason behind the name `RouteStack`. Zend Framework provides two implementations of this interface, `SimpleRouteStack` and `TreeRouteStack`. In each, you register routes either one at a time using `addRoute()`, or in bulk using `addRoutes()`.

```
// One at a time:
$route = Literal::factory(array(
    'route' => '/foo',
    'defaults' => array(
        'controller' => 'foo-index',
        'action'     => 'index',
    ),
));
$router->addRoute('foo', $route);

$router->addRoutes(array(
    // using already instantiated routes:
    'foo' => $route,

    // providing configuration to allow lazy-loading routes:
    'bar' => array(
        'type' => 'literal',
        'options' => array(
            'route' => '/bar',
            'defaults' => array(
                'controller' => 'bar-index',
                'action'     => 'index',
            ),
        ),
    ),
));
```

Router Types

Two routers are provided, the `SimpleRouteStack` and `TreeRouteStack`. Each works with the above interface, but utilize slightly different options and execution paths.

SimpleRouteStack

This router simply takes individual routes that provide their full matching logic in one go, and loops through them in LIFO order until a match is found. As such, routes that will match most often should be registered last, and least common routes first. Additionally, you will need to ensure that routes that potentially overlap are registered such that the most specific match will match first (i.e., register later). Alternatively, you can set priorities by giving the priority as third parameter to the `addRoute()` method, specifying the priority in the route specifications or setting the priority property within a route instance before adding it to the route stack.

TreeRouteStack

`Zend\Mvc\Router\Http\TreeRouteStack` provides the ability to register trees of routes, and will use a B-tree algorithm to match routes. As such, you register a single route with many children.

A `TreeRouteStack` will consist of the following configuration:

- A base "route", which describes the base match needed, the root of the tree.
- An optional "route_broker", which is a configured `Zend\Mvc\Router\RouteBroker` that can lazy-load routes.
- The option "may_terminate", which hints to the router that no other segments will follow it.
- An optional "child_routes" array, which contains additional routes that stem from the base "route" (i.e., build from it). Each child route can itself be a `TreeRouteStack` if desired; in fact, the `Part` route works exactly this way.

When a route matches against a `TreeRouteStack`, the matched parameters from each segment of the tree will be returned.

A `TreeRouteStack` can be your sole route for your application, or describe particular path segments of the application.

An example of a `TreeRouteStack` is provided in the documentation of the `Part` route.

Route Types

Zend Framework 2.0 ships with the following route types.

Zend\Mvc\Router\Http\Hostname

The `Hostname` route attempts to match the hostname registered in the request against specific criteria. Typically, this will be in one of the following forms:

- "subdomain.domain.tld"
- ":subdomain.domain.tld"

In the above, the second route would return a "subdomain" key as part of the route match.

For any given hostname segment, you may also provide a constraint. As an example, if the "subdomain" segment needed to match only if it started with "fw" and contained exactly 2 digits following, the following route would be needed:

```
$route = Hostname::factory(array(
    'route' => ':subdomain.domain.tld',
    'constraints' => array(
        'subdomain' => 'fw\d{2}'
    ),
));
```

In the above example, only a "subdomain" key will be returned in the `RouteMatch`. If you wanted to also provide other information based on matching, or a default value to return for the subdomain, you need to also provide defaults.

```
$route = Hostname::factory(array(
    'route' => ':subdomain.domain.tld',
    'constraints' => array(
        'subdomain' => 'fw\d{2}'
    ),
    'defaults' => array(
        'type' => 'json',
    ),
));
```

When matched, the above will return two keys in the `RouteMatch`, "subdomain" and "type".

Zend\Mvc\Router\Http\Literal

The `Literal` route is for doing exact matching of the URI path. Configuration therefore is solely the path you want to match, and the "defaults", or parameters you want returned on a match.

```
$route = Literal::factory(array(
    'route' => '/foo',
    'defaults' => array(
        'controller' => 'foo-index',
    ),
));
```

The above route would match a path "/foo", and return the key "controller" in the `RouteMatch`, with the value "foo-index".

Zend\Mvc\Router\Http\Method

The `Method` route is used to match the http method or 'verb' specified in the request (See RFC 2616 Sec. 5.1.1). It can optionally be configured to match against multiple methods by providing a comma-separated list of method tokens.

```
$route = Method::factory(array(
    'verb' => 'post,put',
    'defaults' => array(
        'action' => 'form-submit'
    ),
));
```

The above route would match an http "POST" or "PUT" request and return a `RouteMatch` object containing a key "action" with a value of "form-submit".

Zend\Mvc\Router\Http\Part

A `Part` route allows crafting a tree of possible routes based on segments of the URI path. It actually extends the `TreeRouteStack`.

`Part` routes are difficult to describe, so we'll simply provide a sample one here.

```
$route = Part::factory(array(
    'route' => array(
        'type'    => 'literal',
        'options' => array(
            'route'    => '/',
            'defaults' => array(
                'controller' => 'ItsHomePage',
            ),
        )
    ),
    'may_terminate' => true,
    'route_broker'  => $routeBroker,
    'child_routes'  => array(
        'blog' => array(
            'type'    => 'literal',
            'options' => array(
                'route'    => 'blog',
                'defaults' => array(
                    'controller' => 'ItsBlog',
                ),
            ),
            'may_terminate' => true,
            'child_routes'  => array(
                'rss' => array(
                    'type'    => 'literal',
                    'options' => array(
                        'route'    => '/rss',
                        'defaults' => array(
                            'controller' => 'ItsRssBlog',
                        ),
                    ),
                    'child_routes'  => array(
                        'sub' => array(
                            'type'    => 'literal',
                            'options' => array(
                                'route'    => '/sub',
                                'defaults' => array(
                                    'action' => 'ItsSubRss',
                                ),
                            )
                        ),
                    ),
                ),
            ),
        ),
        'forum' => array(
            'type'    => 'literal',
            'options' => array(
                'route'    => 'forum',
                'defaults' => array(
                    'controller' => 'ItsForum',
                ),
```

```
            ),
        ),
    ),
));
```

The above would match the following:

- "/" would load the "ItsHomePage" controller
- "/blog" would load the "ItsBlog" controller
- "/blog/rss" would load the "ItsRssBlog" controller
- "/blog/rss/sub" would load the "ItsSubRss" controller
- "/forum" would load the "ItsForum" controller

You may use any route type as a child route of a `Part` route.

Zend\Mvc\Router\Http\Regex

A `Regex` route utilizes a regular expression to match against the URI path. Any valid regular expession is allowed; our recommendation is to use named captures for any values you want to return in the `RouteMatch`.

Since regular expression routes are often complex, you must specify a "spec" or specification to use when assembling URLs from regex routes. The spec is simply a string; replacements are identified using "%keyname%" within the string, with the keys coming from either the captured values or named parameters passed to the `assemble()` method.

Just like other routes, the `Regex` route can accept "defaults", parameters to include in the `RouteMatch` when succesfully matched.

```
$route = Regex::factory(array(
    'regex' => '/blog/(?<id>[a-zA-Z0-9_-]+)(\.(?<format>(json|html|xml|rss)))?',
    'defaults' => array(
        'controller' => 'blog-entry',
        'format'     => 'html',
    ),
    'spec' => '/blog/%id%.%format%',
));
```

The above would match "/blog/001-some-blog_slug-here.html", and return three items in the `RouteMatch`, an "id", the "controller", and the "format". When assembling a URL from this route, the "id" and "format" values would be used to fill the specification.

Zend\Mvc\Router\Http\Scheme

The `Scheme` route matches the URI scheme only, and must be an exact match. As such, this route, like the `Literal` route, simply takes what you want to match and the "defaults", parameters to return on a match.

```
$route = Scheme::factory(array(
    'scheme' => 'https',
    'defaults' => array(
        'https' => true,
    ),
));
```

The above route would match the "https" scheme, and return the key "https" in the `RouteMatch` with a boolean `true` value.

Zend\Mvc\Router\Http\Segment

A `Segment` route allows matching any segment of a URI path. Segments are denoted using a colon, followed by alphanumeric characters; if a segment is optional, it should be surrounded by brackets. As an example, "/:foo[/:bar]" would match a "/" followed by text and assign it to the key "foo"; if any additional "/" characters are found, any text following the last one will be assigned to the key "bar".

The separation between literal and named segments can be anything. For example, the above could be done as "/:foo{-}[-:bar] as well. The {-} after the :foo parameter indicates a set of one or more delimiters, after which matching of the parameter itself ends.

Each segment may have constraints associated with it. Each constraint should simply be a regular expression expressing the conditions under which that segment should match.

Also, as you can in other routes, you may provide defaults to use; these are particularly useful when using optional segments.

As a complex example:

```
$route = Segment::factory(array(
    'route' => '/:controller[/:action]',
    'constraints' => array(
        'controller' => '[a-zA-Z][a-zA-Z0-9_-]+',
        'action'     => '[a-zA-Z][a-zA-Z0-9_-]+',
    ),
    'defaults' => array(
        'controller' => 'application-index',
        'action'     => 'index',
    ),
));
```

Zend\Mvc\Router\Http\Query

The `Query` route part allows you to specify and capture query string parameters for a given route.

The intention of the `Query` part is that you do not instantiate it in its own right but to use it as a child of another route part.

An example of its usage would be

```
$route = Part::factory(array(
    'home' => array(
        'page'    => 'segment',
        'options' => array(
            'route'    => '/page[/:name]',
            'constraints' => array(
                'controller' => '[a-zA-Z][a-zA-Z0-9_-]*',
                'action'     => '[a-zA-Z][a-zA-Z0-9_-]*',
            ),
            'defaults' => array(
                'controller' => 'page',
                'action'     => 'index',
            ),
        )
    ),
    'may_terminate' => true,
    'route_broker'  => $routeBroker,
    'child_routes'  => array(
```

```
        'query' => array(
            'type' => 'Query',
        ),
    ),
));
```

As you can see, it's pretty straight forward to specify the query part. This then allows you to create query strings using the url view helper.

```
$this->url(
    'page/query',
    array(
        'name'=>'my-test-page',
        'format' => 'rss',
        'limit' => 10,
    )
);
```

As you can see above, you must add "/query" to your route name in order to append a query string. If you do not specify "/query" in the route name then no query string will be appended.

Our example "page" route has only one defined parameter of "name" ("/page[/:name]"), meaning that the remaining parameters of "format" and "limit" will then be appended as a query string.

The output from our example should then be "/page/mys-test-page?format=rss&limit=10"

CHAPTER 118

The MvcEvent

The ZF2 MVC layer incorporates and utilizes a custom `Zend\EventManager\EventDescription` type, `Zend\Mvc\MvcEvent`. This event is created during `Zend\Mvc\Application::run()`, and is passed directly to all events that method triggers. Additionally, if you mark your controllers with the `Zend\Mvc\InjectApplicationEvent` interface, it will be injected into those controllers.

The `MvcEvent` adds accessors and mutators for the following:

- `Application` object
- `Request` object
- `Response` object
- `Router` object
- `RouteMatch` object
- "Result", usually the result of dispatching a controller
- `ViewModel` object, typically representing the layout view model

The methods it defines are:

- `setApplication($application)`
- `getApplication()`
- `setRequest($request)`
- `getRequest()`
- `setResponse($response)`
- `getResponse()`
- `setRouter($router)`
- `getRouter()`
- `setRouteMatch($routeMatch)`
- `getRouteMatch()`

- `setResult($result)`
- `getResult()`
- `setViewModel($viewModel)`
- `getViewModel()`

The `Application`, `Request`, `Response`, `Router`, and `ViewModel` are all injected during the `bootstrap` event. Following the `route` event, it will be injected also with the `RouteMatch` object encapsulating the results of routing.

Since this object is passed around throughout the MVC, it is a common location for retrieving the results of routing, the router, and the request and response objects. Additionally, we encourage setting the results of execution in the event, to allow event listeners to introspect them and utilize them within their execution. As an example, the results could be passed into a view renderer.

CHAPTER 119

Available Controllers

Controllers in the MVC layer simply need to be objects implementing `Zend\Stdlib\DispatchableInterface`. That interface describes a single method:

```
use Zend\Stdlib\DispatchableInterface;
use Zend\Stdlib\RequestInterface as Request;
use Zend\Stdlib\ResponseInterface as Response;

class Foo implements DispatchableInterface
{
    public function dispatch(Request $request, Response $response = null)
    {
        // ... do something, and preferably return a Response ...
    }
}
```

While this pattern is simple enough, chances are you don't want to implement custom dispatch logic for every controller (particularly as it's not unusual or uncommon for a single controller to handle several related types of requests).

The MVC also defines several interfaces that, when implemented, can provide controllers with additional capabilities.

Common Interfaces Used With Controllers

InjectApplicationEvent

The `Zend\Mvc\InjectApplicationEventInterface` hints to the `Application` instance that it should inject its `MvcEvent` into the controller itself. Why would this be useful?

Recall that the `MvcEvent` composes a number of objects: the `Request` and `Response`, naturally, but also the router, the route matches (a `RouteMatch` instance), and potentially the "result" of dispatching.

A controller that has the `MvcEvent` injected, then, can retrieve or inject these. As an example:

```
$matches = $this->getEvent()->getRouteMatch();
$id      = $matches->getParam('id', false);
if (!$id) {
    $this->getResponse();
    $response->setStatusCode(500);
    $this->getEvent()->setResult('Invalid identifier; cannot complete request');
    return;
}
```

The `InjectApplicationEventInterface` defines simply two methods:

```
use Zend\EventManager\EventDescription as Event;

public function setEvent(Event $event);
public function getEvent($event);
```

ServiceManagerAware

In most cases, you should define your controllers such that dependencies are injected by the application's `ServiceManager`, via either constructor arguments or setter methods.

However, occasionally you may have objects you wish to use in your controller that are only valid for certain code paths. Examples include forms, paginators, navigation, etc. In these cases, you may decide that it doesn't make sense to inject those objects every time the controller is used.

The `ServiceManagerAwareInterface` interface hints to the `ServiceManager` that it should inject itself into the controller. It defines simply one method:

```
use Zend\ServiceManager\ServiceManager;
use Zend\ServiceManager\ServiceManagerAwareInterface;

public function setServiceManager(ServiceManager $serviceManager);
```

EventManagerAware

Typically, it's nice to be able to tie into a controller's workflow without needing to extend it or hardcode behavior into it. The solution for this at the framework level is to use the `EventManager`.

You can hint to the `ServiceManager` that you want an `EventManager` injected by implementing the interfaces `EventManagerAwareInterface` and `EventsCapableInterface`; the former tells the `ServiceManager` to inject an `EventManager`, the latter to other objects that this class has an accessible `EventManager` instance.

Combined, you define two methods. The first, a setter, should also set any `EventManager` identifiers you want to listen on, and the second, a getter, should simply return the composed `EventManager` instance

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

public function setEventManager(EventManagerInterface $events);
public function getEventManager();
```

Pluggable

Code re-use is a common goal for developers. Another common goal is convenience. However, this is often difficult to achieve cleanly in abstract, general systems.

Within your controllers, you'll often find yourself repeating tasks from one controller to another. Some common examples:

- Generating URLs
- Redirecting
- Setting and retrieving flash messages (self-expiring session messages)
- Invoking and dispatching additional controllers

To facilitate these actions while also making them available to alternate controller implementations, we've created a `PluginBroker` implementation for the controller layer, `Zend\Mvc\Controller\PluginBroker`, building on the `Zend\Loader\PluginBroker` functionality. To utilize it, you simply need to implement the `Zend\Loader\Pluggable` interface, and set up your code to use the controller-specific implementation by default:

```
use Zend\Loader\Broker;
use Zend\Mvc\Controller\PluginBroker;

public function setBroker(Broker $broker)
{
    $this->broker = $broker;
    return $this;
}

public function getBroker()
{
    if (!$this->broker instanceof Broker) {
        $this->setBroker(new PluginBroker);
    }
    return $this->broker;
}

public function plugin($plugin, array $options = null)
{
    return $this->getBroker()->load($plugin, $options);
}
```

The AbstractActionController

Implementing each of the above interfaces is a lesson in redundancy; you won't often want to do it. As such, we've developed two abstract, base controllers you can extend to get started.

The first is `Zend\Mvc\Controller\AbstractActionController`. This controller implements each of the above interfaces, and uses the following assumptions:

- An "action" parameter is expected in the `RouteMatch` object composed in the attached `MvcEvent`. If none is found, a `notFoundAction()` is invoked.
- The "action" parameter is converted to a camelCased format and appended with the word "Action" to create a method name. As examples: "foo" maps to "fooAction", "foo-bar" or "foo.bar" or "foo_bar" to "fooBarAction".

The controller then checks to see if that method exists. If not, the `notFoundAction()` method is invoked; otherwise, the discovered method.

- The results of executing the given action method are injected into the `MvcEvent`'s "result" property (via `setResult()`, and accesible via `getResult()`).

Essentially, a route mapping to an `AbstractActionController` needs to return both "controller" and "action" keys in its matches.

Creation of action controllers is then reasonably trivial:

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class BarController extends AbstractActionController
{
    public function bazAction()
    {
        return array('title' => __METHOD__);
    }

    public function batAction()
    {
        return array('title' => __METHOD__);
    }
}
```

Interfaces and Collaborators

`AbstractActionController` implements each of the following interfaces:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Loader\Pluggable`
- `Zend\Mvc\InjectApplicationEventInterface`
- `Zend\ServiceManager\ServiceManagerAwareInterface`
- `Zend\EventManager\EventManagerAwareInterface`
- `Zend\EventManager\EventsCapableInterface`

The composed `EventManager` will be configured to listen on the following contexts:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Mvc\Controller\AbstractActionController`

Additionally, if you extend the class, it will listen on the extending class's name.

The AbstractRestfulController

The second abstract controller ZF2 provides is `Zend\Mvc\Controller\AbstractRestfulController`. This controller provides a naive RESTful implementation that simply maps HTTP request methods to controller methods, using the following matrix:

- **GET** maps to either `get()` or `getList()`, depending on whether or not an "id" parameter is found in the route matches. If one is, it is passed as an argument to `get()`; if not, `getList()` is invoked. In the former case, you should provide a representation of the given entity with that identification; in the latter, you should provide a list of entities.
- **POST** maps to `create()`. That method expects a `$data` argument, usually the `$_POST` superglobal array. The data should be used to create a new entitiy, and the response should typically be an HTTP 201 response with the Location header indicating the URI of the newly created entity and the response body providing the representation.
- **PUT** maps to `update()`, and requires that an "id" parameter exists in the route matches; that value is passed as an argument to the method. It should attempt to update the given entity, and, if successful, return either a 200 or 202 response status, as well as the representation of the entity.
- **DELETE** maps to `delete()`, and requires that an "id" parameter exists in the route matches; that value is passed as an argument to the method. It should attempt to delete the given entity, and, if successful, return either a 200 or 204 response status.

Additionally, you can map "action" methods to the `AbstractRestfulController`, just as you would in the `AbstractActionController`; these methods will be suffixed with "Action", differentiating them from the RESTful methods listed above. This allows you to perform such actions as providing forms used to submit to the various RESTful methods, or to add RPC methods to your RESTful API.

Interfaces and Collaborators

`AbstractRestfulController` implements each of the following interfaces:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Loader\Pluggable`
- `Zend\Mvc\InjectApplicationEventInterface`
- `Zend\ServiceManager\ServiceManagerAwareInterface`
- `Zend\EventManager\EventManagerAwareInterface`
- `Zend\EventManager\EventsCapableInterface`

The composed `EventManager` will be configured to listen on the following contexts:

- `Zend\Stdlib\DispatchableInterface`
- `Zend\Mvc\Controller\AbstractActionController`

Additionally, if you extend the class, it will listen on the extending class's name.

CHAPTER 120

Controller Plugins

When using the `AbstractActionController` or `AbstractRestfulController`, or if you compose the `Zend\Mvc\Controller\PluginBroker` in your custom controllers, you have access to a number of pre-built plugins. Additionally, you can register your own custom plugins with the broker, just as you would with `Zend\Loader\PluginBroker`.

The built-in plugins are:

- `Zend\Mvc\Controller\Plugin\FlashMessenger`
- `Zend\Mvc\Controller\Plugin\Forward`
- `Zend\Mvc\Controller\Plugin\PostRedirectGet`
- `Zend\Mvc\Controller\Plugin\Redirect`
- `Zend\Mvc\Controller\Plugin\Url`

If your controller implements the `Zend\Loader\Pluggable` interface, you can access these using their shortname via the `plugin()` method:

```
$plugin = $this->plugin('url');
```

For an extra layer of convenience, both `AbstractActionController` and `AbstractRestfulController` have `__call()` implementations that allow you to retrieve plugins via method calls:

```
$plugin = $this->url();
```

The FlashMessenger

The `FlashMessenger` is a plugin designed to create and retrieve self-expiring, session-based messages. It exposes a number of methods:

- `setSessionManager()` allows you to specify an alternate session manager, if desired.
- `getSessionManager()` allows you to retrieve the session manager registered.

- `getContainer()` returns the `Zend\Session\Container` instance in which the flash messages are stored.
- `setNamespace()` allows you to specify a specific namespace in the container in which to store or from which to retrieve flash messages.
- `getNamespace()` retrieves the name of the flash message namespace.
- `addMessage()` allows you to add a message to the current namespace of the session container.
- `hasMessages()` lets you determine if there are any flash messages from the current namespace in the session container.
- `getMessages()` retrieves the flash messages from the current namespace of the session container.
- `clearMessages()` clears all flash messages in current namespace of the session container.
- `hasCurrentMessages()` indicates whether any messages were added during the current request.
- `getCurrentMessages()` retrieves any messages added during the current request.
- `clearCurrentMessages()` removes any messages added during the current request.

Additionally, the `FlashMessenger` implements both `IteratorAggregate` and `Countable`, allowing you to iterate over and count the flash messages in the current namespace within the session container.

Examples

```
public function processAction()
{
    // ... do some work ...
    $this->flashMessenger()->addMessage('You are now logged in.');
    return $this->redirect()->toRoute('user-success');
}

public function successAction()
{
    $return = array('success' => true);
    $flashMessenger = $this->flashMessenger();
    if ($flashMessenger->hasMessages()) {
        $return['messages'] = $flashMessenger->getMessages();
    }
    return $return;
}
```

The Forward Plugin

Occasionally, you may want to dispatch additional controllers from within the matched controller – for instance, you might use this approach to build up "widgetized" content. The `Forward` plugin helps enable this.

For the `Forward` plugin to work, the controller calling it must be `ServiceManagerAware`; otherwise, the plugin will be unable to retrieve a configured and injected instance of the requested controller.

The plugin exposes a single method, `dispatch()`, which takes two arguments:

- `$name`, the name of the controller to invoke. This may be either the fully qualified class name, or an alias defined and recognized by the `ServiceManager` instance attached to the invoking controller.

- `$params` is an optional array of parameters with which to see a `RouteMatch` object for purposes of this specific request.

`Forward` returns the results of dispatching the requested controller; it is up to the developer to determine what, if anything, to do with those results. One recommendation is to aggregate them in any return value from the invoking controller.

As an example:

```
$foo = $this->forward()->dispatch('foo', array('action' => 'process'));
return array(
    'somekey' => $somevalue,
    'foo'     => $foo,
);
```

The Post/Redirect/Get Plugin

When a user sends a POST request (e.g. after submitting a form), their browser will try to protect them from sending the POST again, breaking the back button, causing browser warnings and pop-ups, and sometimes reposting the form. Instead, when receiving a POST, we should store the data in a session container and redirect the user to a GET request.

This plugin can be invoked with two arguments:

- `$redirect`, a string containing the redirect location which can either be a named route or a URL, based on the contents of the second parameter.
- `$redirectToUrl`, a boolean that when set to TRUE, causes the first parameter to be treated as a URL instead of a route name (this is required when redirecting to a URL instead of a route). This argument defaults to false.

```
// Pass in the route/url you want to redirect to after the POST
$prg = $this->prg('/user/register', true);

if ($prg instanceof \Zend\Http\PhpEnvironment\Response) {
    // returned a response to redirect us
    return $prg;
} elseif ($prg === false) {
    // this wasn't a POST request, but there were no params in the flash messenger
    // probably this is the first time the form was loaded
    return array('form' => $myForm);
}

// $prg is an array containing the POST params from the previous request
$form->setData($prg);

// ... your form processing code here
```

The Redirect Plugin

Redirections are quite common operations within applications. If done manually, you will need to do the following steps:

- Assemble a url using the router
- Create and inject a "Location" header into the `Response` object, pointing to the assembled URL
- Set the status code of the `Response` object to one of the 3xx HTTP statuses.

The `Redirect` plugin does this work for you. It offers two methods:

- `toRoute($route, array $params = array(), array $options = array())`: Redirects to a named route, using the provided `$params` and `$options` to assembled the URL.
- `toUrl($url)`: Simply redirects to the given URL.

In each case, the `Response` object is returned. If you return this immediately, you can effectively short-circuit execution of the request.

One note: this plugin requires that the controller invoking it implements `InjectApplicationEvent`, and thus has an `MvcEvent` composed, as it retrieves the router from the event object.

As an example:

```
return $this->redirect()->toRoute('login-success');
```

The Url Plugin

Often you may want to generate URLs from route definitions within your controllers – in order to seed the view, generate headers, etc. While the `MvcEvent` object composes the router, doing so manually would require this workflow:

```
$router = $this->getEvent()->getRouter();
$url    = $router->assemble($params, array('name' => 'route-name'));
```

The `Url` helper makes this slightly more convenient:

```
$url = $this->url()->fromRoute('route-name', $params);
```

The `fromRoute()` method is the only public method defined, and has the following signature:

```
public function fromRoute($route, array $params = array(), array $options = array())
```

One note: this plugin requires that the controller invoking it implements `InjectApplicationEvent`, and thus has an `MvcEvent` composed, as it retrieves the router from the event object.

CHAPTER 121

Examples

Controllers

Accessing the Request and Response

When using `AbstractActionController` or `AbstractRestfulController`, the request and response object are composed directly into the controller as soon as `dispatch()` is called. You may access them in the following ways:

```
// Using explicit accessor methods
$request  = $this->getRequest();
$response = $this->getResponse();

// Using direct property access
$request  = $this->request;
$response = $this->response;
```

Additionally, if your controller implements `InjectApplicationEventInterface` (as both `AbstractActionController` and `AbstractRestfulController` do), you can access these objects from the attached `MvcEvent`:

```
$event    = $this->getEvent();
$request  = $event->getRequest();
$response = $event->getResponse();
```

The above can be useful when composing event listeners into your controller.

Accessing routing parameters

The parameters returned when routing completes are wrapped in a `Zend\Mvc\Router\RouteMatch` object. This object is detailed in the section on routing.

Within your controller, if you implement `InjectApplicationEventInterface` (as both `AbstractActionController` and `AbstractRestfulController` do), you can access this object from the attached `MvcEvent`:

```
$event   = $this->getEvent();
$matches = $event->getRouteMatch();
```

Once you have the `RouteMatch` object, you can pull parameters from it.

Returning early

You can effectively short-circuit execution of the application at any point by returning a `Response` from your controller or any event. When such a value is discovered, it halts further execution of the event manager, bubbling up to the `Application` instance, where it is immediately returned.

As an example, the `Redirect` plugin returns a `Response`, which can be returned immediately so as to complete the request as quickly as possible. Other use cases might be for returning JSON or XML results from web service endpoints, returning "401 Forbidden" results, etc.

Bootstrapping

Registering module-specific listeners

Often you may want module-specific listeners. As an example, this would be a simple and effective way to introduce authorization, logging, or caching into your application.

Each `Module` class can have an optional `onBootstrap()` method. Typically, you'll do module-specific configuration here, or setup event listeners for you module here. The `onBootstrap()` method is called for **every** module on **every** page request and should **only** be used for performing **lightweight** tasks such as registering event listeners.

The base `Application` class shipped with the framework has an `EventManager` associated with it, and once the modules are initialized, it triggers a "bootstrap" event, with a `getApplication()` method on the event.

So, one way to accomplish module-specific listeners is to listen to that event, and register listeners at that time. As an example:

```
namespace SomeCustomModule;

class Module
{
    public function onBootstrap($e)
    {
        $application = $e->getApplication();
        $config      = $application->getConfiguration();
        $view        = $application->getServiceManager()->get('View');
        $view->headTitle($config['view']['base_title']);

        $listeners   = new Listeners\ViewListener();
        $listeners->setView($view);
        $application->getEventManager()->attachAggregate($listeners);
    }
}
```

The above demonstrates several things. First, it demonstrates a listener on the application's "bootstrap" event (the `onBootstrap()` method). Second, it demonstrates that listener, and how it can be used to register listeners with

the application. It grabs the `Application` instance; from the `Application`, it is able to grab the attached service manager and configuration. These are then used to retrieve the view, configure some helpers, and then register a listener aggregate with the application event manager.

CHAPTER 122

Introduction

The `Zend\Permissions\Acl` component provides a lightweight and flexible access control list (*ACL*) implementation for privileges management. In general, an application may utilize such *ACL*'s to control access to certain protected objects by other requesting objects.

For the purposes of this documentation:

- a **resource** is an object to which access is controlled.
- a **role** is an object that may request access to a Resource.

Put simply, **roles request access to resources**. For example, if a parking attendant requests access to a car, then the parking attendant is the requesting role, and the car is the resource, since access to the car may not be granted to everyone.

Through the specification and use of an *ACL*, an application may control how roles are granted access to resources.

Resources

Creating a resource using `Zend\Permissions\Acl\Acl` is very simple. A resource interface `Zend\Permissions\Acl\Resource\ResourceInterface` is provided to facilitate creating resources in an application. A class need only implement this interface, which consists of a single method, `getResourceId()`, for `Zend\Permissions\Acl\Acl` to recognize the object as a resource. Additionally, `Zend\Permissions\Acl\Resource\GenericResource` is provided as a basic resource implementation for developers to extend as needed.

`Zend\Permissions\Acl\Acl` provides a tree structure to which multiple resources can be added. Since resources are stored in such a tree structure, they can be organized from the general (toward the tree root) to the specific (toward the tree leaves). Queries on a specific resource will automatically search the resource's hierarchy for rules assigned to ancestor resources, allowing for simple inheritance of rules. For example, if a default rule is to be applied to each building in a city, one would simply assign the rule to the city, instead of assigning the same rule to each building. Some buildings may require exceptions to such a rule, however, and this can be achieved in `Zend\Permissions\Acl\Acl` by assigning such exception rules to each building that requires such an exception. A resource may inherit from only one parent resource, though this parent resource can have its own parent resource, etc.

`Zend\Permissions\Acl\Acl` also supports privileges on resources (e.g., "create", "read", "update", "delete"), so the developer can assign rules that affect all privileges or specific privileges on one or more resources.

Roles

As with resources, creating a role is also very simple. All roles must implement `Zend\Permissions\Acl\Role\RoleInterface`. This interface consists of a single method, `getRoleId()`. Additionally, `Zend\Permissions\Acl\Role\GenericRole` is provided by the `Zend\Permissions\Acl` component as a basic role implementation for developers to extend as needed.

In `Zend\Permissions\Acl\Acl`, a role may inherit from one or more roles. This is to support inheritance of rules among roles. For example, a user role, such as "sally", may belong to one or more parent roles, such as "editor" and "administrator". The developer can assign rules to "editor" and "administrator" separately, and "sally" would inherit such rules from both, without having to assign rules directly to "sally".

Though the ability to inherit from multiple roles is very useful, multiple inheritance also introduces some degree of complexity. The following example illustrates the ambiguity condition and how `Zend\Permissions\Acl\Acl` solves it.

Multiple Inheritance among Roles

The following code defines three base roles - "guest", "member", and "admin" - from which other roles may inherit. Then, a role identified by "someUser" is established and inherits from the three other roles. The order in which these roles appear in the `$parents` array is important. When necessary, `Zend\Permissions\Acl\Acl` searches for access rules defined not only for the queried role (herein, "someUser"), but also upon the roles from which the queried role inherits (herein, "guest", "member", and "admin"):

```
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;
use Zend\Permissions\Acl\Resource\GenericResource as Resource;

$acl = new Acl();

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

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

$acl->addResource(new Resource('someResource'));

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

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

Since there is no rule specifically defined for the "someUser" role and "someResource", `Zend\Permissions\Acl\Acl` must search for rules that may be defined for roles that "someUser" inherits. First, the "admin" role is visited, and there is no access rule defined for it. Next, the "member" role is visited, and `Zend\Permissions\Acl\Acl` finds that there is a rule specifying that "member" is allowed access to "someResource".

If `Zend\Permissions\Acl\Acl` were to continue examining the rules defined for other parent roles, however, it would find that "guest" is denied access to "someResource". This fact introduces an ambiguity because now

"someUser" is both denied and allowed access to "someResource", by reason of having inherited conflicting rules from different parent roles.

`Zend\Permissions\Acl\Acl` resolves this ambiguity by completing a query when it finds the first rule that is directly applicable to the query. In this case, since the "member" role is examined before the "guest" role, the example code would print "allowed".

Note: When specifying multiple parents for a role, keep in mind that the last parent listed is the first one searched for rules applicable to an authorization query.

Creating the Access Control List

An Access Control List (*ACL*) can represent any set of physical or virtual objects that you wish. For the purposes of demonstration, however, we will create a basic Content Management System (*CMS*) *ACL* that maintains several tiers of groups over a wide variety of areas. To create a new *ACL* object, we instantiate the *ACL* with no parameters:

```
use Zend\Permissions\Acl\Acl;
$acl = new Acl();
```

Note: Until a developer specifies an "allow" rule, `Zend\Permissions\Acl\Acl` denies access to every privilege upon every resource by every role.

Registering Roles

CMS's will nearly always require a hierarchy of permissions to determine the authoring capabilities of its users. There may be a 'Guest' group to allow limited access for demonstrations, a 'Staff' group for the majority of *CMS* users who perform most of the day-to-day operations, an 'Editor' group for those responsible for publishing, reviewing, archiving and deleting content, and finally an 'Administrator' group whose tasks may include all of those of the other groups as well as maintenance of sensitive information, user management, back-end configuration data, backup and export. This set of permissions can be represented in a role registry, allowing each group to inherit privileges from 'parent' groups, as well as providing distinct privileges for their unique group only. The permissions may be expressed as follows:

Table 122.1: Access Controls for an Example CMS

Name	Unique Permissions	Inherit Permissions From
Guest	View	N/A
Staff	Edit, Submit, Revise	Guest
Editor	Publish, Archive, Delete	Staff
Administrator	(Granted all access)	N/A

For this example, `Zend\Permissions\Acl\Role\GenericRole` is used, but any object that implements `Zend\Permissions\Acl\Role\RoleInterface` is acceptable. These groups can be added to the role registry as follows:

```
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;

$acl = new Acl();

```

```php
// Add groups to the Role registry using Zend\Permissions\Acl\Role\GenericRole
// Guest does not inherit access controls
$roleGuest = new Role('guest');
$acl->addRole($roleGuest);

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

/*
Alternatively, the above could be written:
$acl->addRole(new Role('staff'), 'guest');
*/

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

// Administrator does not inherit access controls
$acl->addRole(new Role('administrator'));
```

Defining Access Controls

Now that the *ACL* contains the relevant roles, rules can be established that define how resources may be accessed by roles. You may have noticed that we have not defined any particular resources for this example, which is simplified to illustrate that the rules apply to all resources. `Zend\Permissions\Acl\Acl` provides an implementation whereby rules need only be assigned from general to specific, minimizing the number of rules needed, because resources and roles inherit rules that are defined upon their ancestors.

Note: In general, `Zend\Permissions\Acl\Acl` obeys a given rule if and only if a more specific rule does not apply.

Consequently, we can define a reasonably complex set of rules with a minimum amount of code. To apply the base permissions as defined above:

```php
use Zend\Permissions\Acl\Acl;
use Zend\Permissions\Acl\Role\GenericRole as Role;

$acl = new Acl();

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

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

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

// Staff inherits view privilege from guest, but also needs additional
```

```php
// 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 an ACL

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

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

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

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

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

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

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

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

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

CHAPTER 123

Refining Access Controls

Precise Access Controls

The basic *ACL* as defined in the *previous section* shows how various privileges may be allowed upon the entire *ACL* (all resources). In practice, however, access controls tend to have exceptions and varying degrees of complexity. `Zend\Permissions\Acl\Acl` allows 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':

```
1 // The new marketing group inherits permissions from staff
2 use Zend\Permissions\Acl\Acl;
3 use Zend\Permissions\Acl\Role\GenericRole as Role;
4 use Zend\Permissions\Acl\Resource\GenericResource as Resource;
5
6 $acl = new Acl();
7
8 $acl->addRole(new Role('marketing'), 'staff');
```

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

```
1 // Create Resources for the rules
2
3 // newsletter
4 $acl->addResource(new Resource('newsletter'));
```

```
5
6  // news
7  $acl->addResource(new Resource('news'));
8
9  // latest news
10 $acl->addResource(new Resource('latest'), 'news');
11
12 // announcement news
13 $acl->addResource(new Resource('announcement'), 'news');
```

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

```
1  // Marketing must be able to publish and archive newsletters and the
2  // latest news
3  $acl->allow('marketing',
4              array('newsletter', 'latest'),
5              array('publish', 'archive'));
6
7  // Staff (and marketing, by inheritance), are denied permission to
8  // revise the latest news
9  $acl->deny('staff', 'latest', 'revise');
10
11 // Everyone (including administrators) are denied permission to
12 // archive news announcements
13 $acl->deny(null, 'announcement', 'archive');
```

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

```
1  echo $acl->isAllowed('staff', 'newsletter', 'publish') ?
2       "allowed" : "denied";
3  // denied
4
5  echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
6       "allowed" : "denied";
7  // allowed
8
9  echo $acl->isAllowed('staff', 'latest', 'publish') ?
10      "allowed" : "denied";
11 // denied
12
13 echo $acl->isAllowed('marketing', 'latest', 'publish') ?
14      "allowed" : "denied";
15 // allowed
16
17 echo $acl->isAllowed('marketing', 'latest', 'archive') ?
18      "allowed" : "denied";
19 // allowed
20
21 echo $acl->isAllowed('marketing', 'latest', 'revise') ?
22      "allowed" : "denied";
23 // denied
24
25 echo $acl->isAllowed('editor', 'announcement', 'archive') ?
26      "allowed" : "denied";
27 // denied
28
29 echo $acl->isAllowed('administrator', 'announcement', 'archive') ?
30      "allowed" : "denied";
```

```
// 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
```

CHAPTER 124

Advanced Usage

Storing ACL Data for Persistence

The `Zend\Permissions\Acl` component was designed in such a way that it does not require any particular backend technology such as a database or cache server for storage of the *ACL* data. Its complete *PHP* implementation enables customized administration tools to be built upon `Zend\Permissions\Acl\Acl` with relative ease and flexibility. Many situations require some form of interactive maintenance of the *ACL*, and `Zend\Permissions\Acl\Acl` provides methods for setting up, and querying against, the access controls of an application.

Storage of *ACL* data is therefore left as a task for the developer, since use cases are expected to vary widely for various situations. Because `Zend\Permissions\Acl\Acl` is serializable, *ACL* objects may be serialized with *PHP*'s serialize() function, and the results may be stored anywhere the developer should desire, such as a file, database, or caching mechanism.

Writing Conditional ACL Rules with Assertions

Sometimes a rule for allowing or denying a role access to a resource should not be absolute but dependent upon various criteria. For example, suppose that certain access should be allowed, but only between the hours of 8:00am and 5:00pm. Another example would be denying access because a request comes from an IP address that has been flagged as a source of abuse. `Zend\Permissions\Acl\Acl` has built-in support for implementing rules based on whatever conditions the developer needs.

`Zend\Permissions\Acl\Acl` provides support for conditional rules with `Zend\Permissions\Acl\Assertion\AssertionInterface`. In order to use the rule assertion interface, a developer writes a class that implements the `assert()` method of the interface:

```
class CleanIPAssertion implements Zend\Permissions\Acl\Assertion\AssertionInterface
{
    public function assert(Zend\Permissions\Acl $acl,
                           Zend\Permissions\Acl\Role\RoleInterface $role = null,
                           Zend\Permissions\Acl\Resource\ResourceInterface $resource␣
↪= null,
```

```
                                $privilege = null)
    {
        return $this->_isCleanIP($_SERVER['REMOTE_ADDR']);
    }

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

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

```
use Zend\Permissions\Acl\Acl;

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

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

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

CHAPTER 125

Zend\ServiceManager

The Service Locator design pattern is implemented by the `ServiceManager`. The Service Locator is a service/object locator, tasked with retrieving other objects. You may interact with the `ServiceManger` via the following methods

- `has($name)`, tests whether the `ServiceManager` has a named service;
- `get($name)`, retrieves a service by the given name.

In addition to above methods, the `ServiceManger` can be instantiated via the following features:

- **Service registration**. You can register an object under a given name (*$services->setService('foo', $object)*).
- **Lazy-loaded service objects**. You can tell the manager what class to instantiate on first request (*$services->setInvokableClass('foo', 'FullyQualifiedClassname')*).
- **Service factories**. Instead of an actual object instance or a class name, you can tell the manager to invoke the provided factory in order to get the object instance. Factories may be either any PHP callable, an object implementing `Zend\ServiceManager\FactoryInterface`, or the name of a class implementing that interface.
- **Service aliasing**. You can tell the manager that when a particular name is requested, use the provided name instead. You can alias to a known service, a lazy-loaded service, a factory, or even other aliases.
- **Abstract factories**. An abstract factory can be considered a "fallback" – if the service does not exist in the manager, it will then pass it to any abstract factories attached to it until one of them is able to return an object.
- **Initializers**. You may want certain injection points always populated – as an example, any object you load via the service manager that implements `Zend\EventManager\EventManagerAware` should likely receive an `EventManager` instance. **Initializers** are PHP callbacks or classes implementing `Zend\ServiceManager\InitializerInterface`; they receive the new instance, and can then manipulate it.

In addition to the above, the `ServiceManager` also provides optional ties to `Zend\Di`, allowing `Di` to act as an initializer or an abstract factory for the manager.

CHAPTER 126

Zend\ServiceManager Quick Start

By default, Zend Framework utilizes `Zend\ServiceManager` within the MVC layer. As such, in most cases you'll be providing services, invokable classes, aliases, and factories either via configuration or via your module classes.

By default, the module manager listener `Zend\ModuleManager\Listener\ServiceListener` will do the following:

- For modules implementing the `Zend\ModuleManager\Feature\ServiceProvider` interface, or the `getServiceConfig()` method, it will call that method and merge the configuration.
- After all modules have been processed, it will grab the configuration from the registered `Zend\ModuleManager\Feature\ConfigListener`, and merge any configuration under the `service_manager` key.
- Finally, it will use the merged configuration to configure the `ServiceManager`.

In most cases, you won't interact with the `ServiceManager`, other than to provide services to it; your application will typically rely on good configuration in the `ServiceManager` to ensure that classes are configured correctly with their dependencies. When creating factories, however, you may want to interact with the `ServiceManager` to retrieve other services to inject as dependencies. Additionally, there are some cases where you may want to receive the `ServiceManager` to lazy-retrieve dependencies; as such, you'll want to implement `ServiceManagerAwareInterface`, and learn the API of the `ServiceManager`.

Using Configuration

Configuration requires a `service_manager` key at the top level of your configuration, with one or more of the following sub-keys:

- **abstract_factories**, which should be an array of abstract factory class names.
- **aliases**, which should be an associative array of alias name/target name pairs (where the target name may also be an alias).
- **factories**, an array of service name/factory class name pairs. The factories should be either classes implementing `Zend\ServiceManager\FactoryInterface` or invokable classes. If you are using PHP configuration files, you may provide any PHP callable as the factory.

- **invokables**, an array of service name/class name pairs. The class name should be class that may be directly instantiated without any constructor arguments.
- **services**, an array of service name/object pairs. Clearly, this will only work with PHP configuration.
- **shared**, an array of service name/boolean pairs, indicating whether or not a service should be shared. By default, the `ServiceManager` assumes all services are shared, but you may specify a boolean false value here to indicate a new instance should be returned.

Modules as Service Providers

Modules may act as service configuration providers. To do so, the Module class must either implement `Zend\ModuleManager\Feature\ServiceProviderInterface` or simply the method `getServiceConfig()` (which is also defined in the interface). This method must return one of the following:

- An array (or `Traversable` object) of configuration compatible with `Zend\ServiceManager\Config`. (Basically, it should have the keys for configuration as discussed in *the previous section*.
- A string providing the name of a class implementing `Zend\ServiceManager\ConfigInterface`.
- An instance of either `Zend\ServiceManager\Config`, or an object implementing `Zend\ServiceManager\ConfigInterface`.

As noted previously, this configuration will be merged with the configuration returned from other modules as well as configuration files, prior to being passed to the `ServiceManager`; this allows overriding configuration from modules easily.

Examples

Sample configuration

The following is valid configuration for any configuration being merged in your application, and demonstrates each of the possible configuration keys. Configuration is merged in the following order:

- Configuration returned from Module classes via the `getServiceConfig()` method, in the order in which modules are processed.
- Module configuration under the `service_manager` key, in the order in which modules are processed.
- Application configuration under the `config/autoload/` directory, in the order in which they are processed.

As such, you have a variety of ways to override service manager configuration settings.

```
<?php
// a module configuration, "module/SomeModule/config/module.config.php"
return array(
    'service_manager' => array(
        'abstract_factories' => array(
            // Valid values include names of classes implementing
            // AbstractFactoryInterface, instances of classes implementing
            // AbstractFactoryInterface, or any PHP callbacks
            'SomeModule\Service\FallbackFactory',
        ),
        'aliases' => array(
            // Aliasing a FQCN to a service name
            'SomeModule\Model\User' => 'User',
```

```
            // Aliasing a name to a known service name
            'AdminUser' => 'User',
            // Aliasing to an alias
            'SuperUser' => 'AdminUser',
        ),
        'factories' => array(
            // Keys are the service names.
            // Valid values include names of classes implementing
            // FactoryInterface, instances of classes implementing
            // FactoryInterface, or any PHP callbacks
            'User'     => 'SomeModule\Service\UserFactory',
            'UserForm' => function ($serviceManager) {
                $form = new SomeModule\Form\User();

                // Retrieve a dependency from the service manager and inject it!
                $form->setInputFilter($serviceManager->get('UserInputFilter'));
                return $form;
            },
        ),
        'invokables' => array(
            // Keys are the service names
            // Values are valid class names to instantiate.
            'UserInputFiler' => 'SomeModule\InputFilter\User',
        ),
        'services' => array(
            // Keys are the service names
            // Values are objects
            'Auth' => new SomeModule\Authentication\AuthenticationService(),
        ),
        'shared' => array(
            // Usually, you'll only indicate services that should _NOT_ be
            // shared -- i.e., ones where you want a different instance
            // every time.
            'UserForm' => false,
        ),
    ),
);
```

Note: Configuration and PHP

Typically, you should not have your configuration files create new instances of objects or even closures for factories; at the time of configuration, not all autoloading may be in place, and if another configuration overwrites this one later, you're now spending CPU and memory performing work that is ultimately lost.

For instances that require factories, write a factory. If you'd like to inject specific, configured instances, use the Module class to do so, or a listener.

Module returning an array

The following demonstrates returning an array of configuration from a module class. It is substantively the same as the array configuration from the previous example.

```
namespace SomeModule;

class Module
```

```
{
    public function getServiceConfig()
    {
        return array(
            'abstract_factories' => array(
                // Valid values include names of classes implementing
                // AbstractFactoryInterface, instances of classes implementing
                // AbstractFactoryInterface, or any PHP callbacks
                'SomeModule\Service\FallbackFactory',
            ),
            'aliases' => array(
                // Aliasing a FQCN to a service name
                'SomeModule\Model\User' => 'User',
                // Aliasing a name to a known service name
                'AdminUser' => 'User',
                // Aliasing to an alias
                'SuperUser' => 'AdminUser',
            ),
            'factories' => array(
                // Keys are the service names.
                // Valid values include names of classes implementing
                // FactoryInterface, instances of classes implementing
                // FactoryInterface, or any PHP callbacks
                'User'     => 'SomeModule\Service\UserFactory',
                'UserForm' => function ($serviceManager) {
                    // Note: we're already in the "SomeModule" namespace
                    $form = new Form\User();

                    // Retrieve a dependency from the service manager and inject it!
                    $form->setInputFilter($serviceManager->get('UserInputFilter'),
                    return $form;
                },
            ),
            'invokables' => array(
                // Keys are the service names
                // Values are valid class names to instantiate.
                'UserInputFiler' => 'SomeModule\InputFilter\User',
            ),
            'services' => array(
                // Keys are the service names
                // Values are objects
                // Note: we're already in the "SomeModule" namespace
                'Auth' => new Authentication\AuthenticationService(),
            ),
            'shared' => array(
                // Usually, you'll only indicate services that should _NOT_ be
                // shared -- i.e., ones where you want a different instance
                // every time.
                'UserForm' => false,
            ),
        );
    }
}
```

Returning a Configuration instance

First, let's create a class that holds configuration.

```
namespace SomeModule\Service;

use SomeModule\Authentication;
use SomeModule\Form;
use Zend\ServiceManager\Config;
use Zend\ServiceManager\ServiceManager;

class ServiceConfiguration extends Configuration
{
    /**
     * This is hard-coded for brevity.
     */
    public function configureServiceManager(ServiceManager $serviceManager)
    {
        $serviceManager->setFactory('User', 'SomeModule\Service\UserFactory');
        $serviceManager->setFactory('UserForm', function ($serviceManager) {
            $form = new Form\User();

            // Retrieve a dependency from the service manager and inject it!
            $form->setInputFilter($serviceManager->get('UserInputFilter'),
            return $form;
        });
        $serviceManager->setInvokableClass('UserInputFilter',
↪'SomeModule\InputFilter\User');
        $serviceManager->setService('Auth', new␣
↪Authentication\AuthenticationService());
        $serviceManager->setAlias('SomeModule\Model\User', 'User');
        $serviceManager->setAlias('AdminUser', 'User');
        $serviceManager->setAlias('SuperUser', 'AdminUser');
        $serviceManager->setShared('UserForm', false);
    }
}
```

Now, we'll consume it from our Module.

```
namespace SomeModule;

// We could implement Zend\ModuleManager\Feature\ServiceProviderInterface.
// However, the module manager will still find the method without doing so.
class Module
{
    public function getServiceConfig()
    {
        return new Service\ServiceConfiguration();
        // OR:
        // return 'SomeModule\Service\ServiceConfiguration';
    }
}
```

Creating a ServiceManager-aware class

By default, the Zend Framework MVC registers an initializer that will inject the `ServiceManager` instance into any class implementing `Zend\ServiceManager\ServiceManagerAwareInterface`. The default controller implementations implement this interface, as do a small number of other objects. A simple implementation looks like the following.

```
namespace SomeModule\Controller\BareController;

use Zend\ServiceManager\ServiceManager;
use Zend\ServiceManager\ServiceManagerAwareInterface;
use Zend\Stdlib\DispatchableInterface as Dispatchable;
use Zend\Stdlib\RequestInterface as Request;
use Zend\Stdlib\ResponseInterface as Response;

class BareController implements
    Dispatchable,
    ServiceManagerAwareInterface
{
    protected $services;

    public function setServiceManager(ServiceManager $serviceManager)
    {
        $this->services = $serviceManager;
    }

    public function dispatch(Request $request, Response $response = null)
    {
        // ...

        // Retrieve something from the service manager
        $router = $this->services->get('Router');

        // ...
    }
}
```

CHAPTER 127

Zend\Stdlib\Hydrator

Hydration is the act of populating an object from a set of data.

The `Hydrator` is a simple component to provide mechanisms both for hydrating objects, as well as extracting data sets from them.

The component consists of an interface, and several implementations for common use cases.

HydratorInterface

```
namespace Zend\Stdlib\Hydrator;

interface HydratorInterface
{
    /**
     * Extract values from an object
     *
     * @param  object $object
     * @return array
     */
    public function extract($object);

    /**
     * Hydrate $object with the provided $data.
     *
     * @param  array $data
     * @param  object $object
     * @return void
     */
    public function hydrate(array $data, $object);
}
```

Usage

Usage is quite simple: simply instantiate the hydrator, and then pass information to it.

```
use Zend\Stdlib\Hydrator;
$hydrator = new Hydrator\ArraySerializable();

$object = new ArrayObject(array());

$hydrator->hydrate($someData, $object);

// or, if the object has data we want as an array:
$data = $hydrator->extract($object);
```

Available Implementations

- **ZendStdlibHydratorArraySerializable**

  Follows the definition of `ArrayObject`. Objects must implement either the the `exchangeArray()` or `populate()` methods to support hydration, and the `getArrayCopy()` method to support extraction.

- **ZendStdlibHydratorClassMethods**

  Any data key matching a setter method will be called in order to hydrate; any method matching a getter method will be called for extraction.

- **ZendStdlibHydratorObjectProperty**

  Any data key matching a publically accessible property will be hydrated; any public properties will be used for extration.

CHAPTER 128

Zend\Uri

Overview

`Zend\Uri` is a component that aids in manipulating and validating Uniform Resource Identifiers (URIs)[1]. `Zend\Uri` exists primarily to service other components, such as `Zend\Http\`, but is also useful as a standalone utility.

*URI*s always begin with a scheme, followed by a colon. The construction of the many different schemes varies significantly. The `Zend\Uri` component provides the `Zend\Uri\UriFactory` that returns a class implementing the `Zend\Uri\UriInterface` which specializes in the scheme if such a class is registered with the Factory.

Creating a New URI

`Zend\Uri\UriFactory` will build a new *URI* from scratch if only a scheme is passed to `Zend\Uri\UriFactory::factory()`.

Creating a New URI with ZendUriUriFactory::factory()

```
1 // To create a new URI from scratch, pass only the scheme
2 // followed by a colon.
3 $uri = Zend\Uri\UriFactory::factory('http:');
4
5 // $uri instanceof Zend\Uri\UriInterface
```

To create a new *URI* from scratch, pass only the scheme followed by a colon to `Zend\Uri\UriFactory::factory()`[2]. If an unsupported scheme is passed and no scheme-specific class is specified, a `Zend\Uri\Exception\InvalidArgumentException` will be thrown.

[1] See http://www.ietf.org/rfc/rfc3986.txt for more information on URIs

[2] At the time of writing, `Zend\Uri` provides built-in support for the following schemes: *HTTP*, *HTTPS*, *MAILTO* and *FILE*

If the scheme or *URI* passed is supported, `Zend\Uri\UriFactory::factory()` will return a class implementing `Zend\Uri\UriInterface` that specializes in the scheme to be created.

Creating a New Custom-Class URI

You can specify a custom class to be used when using the `Zend\Uri\UriFactory` by registering your class with the Factory using `\Zend\Uri\UriFactory::registerScheme()` which takes the scheme as first parameter. This enables you to create your own *URI*-class and instantiate new URI objects based on your own custom classes.

The 2nd parameter passed to `Zend\Uri\UriFactory::registerScheme()` must be a string with the name of a class implementing `Zend\Uri\UriInterface`. The class must either be alredy loaded, or be loadable by the autoloader.

Creating a URI using a custom class

```
// Create a new 'ftp' URI based on a custom class
use Zend\Uri\UriFactory

UriFactory::registerScheme('ftp', 'MyNamespace\MyClass');

$ftpUri = UriFactory::factory(
    'ftp://user@ftp.example.com/path/file'
);

// $ftpUri is an instance of MyLibrary\MyClass, which implements
// Zend\Uri\UriInterface
```

Manipulating an Existing URI

To manipulate an existing *URI*, pass the entire *URI* as string to `Zend\Uri\UriFactory::factory()`.

Manipulating an Existing URI with Zend\Uri\UriFactory::factory()

```
// To manipulate an existing URI, pass it in.
$uri = Zend\Uri\UriFactory::factory('http://www.zend.com');

// $uri instanceof Zend\Uri\UriInterface
```

The *URI* will be parsed and validated. If it is found to be invalid, a `Zend\Uri\Exception\InvalidArgumentException` will be thrown immediately. Otherwise, `Zend\Uri\UriFactory::factory()` will return a class implementing `Zend\Uri\UriInterface` that specializes in the scheme to be manipulated.

Common Instance Methods

The *ZendUriUriInterface'* defines several instance methods that are useful for working with any kind of *URI*.

Getting the Scheme of the URI

The scheme of the *URI* is the part of the *URI* that precedes the colon. For example, the scheme of `http://johndoe@example.com/my/path?query#token` is 'http'.

Getting the Scheme from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');

$scheme = $uri->getScheme();  // "mailto"
```

The `getScheme()` instance method returns only the scheme part of the *URI* object.

Getting the Userinfo of the URI

The userinfo of the *URI* is the optional part of the *URI* that follows the colon and comes before the host-part. For example, the userinfo of `http://johndoe@example.com/my/path?query#token` is 'johndoe'.

Getting the Username from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');

$scheme = $uri->getUserinfo();  // "john.doe"
```

The `getUserinfo()` method returns only the userinfo part of the *URI* object.

Getting the host of the URI

The host of the *URI* is the optional part of the *URI* that follows the user-part and comes before the path-part. For example, the host of `http://johndoe@example.com/my/path?query#token` is 'example.com'.

Getting the host from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('mailto:john.doe@example.com');

$scheme = $uri->getHost();  // "example.com"
```

The `getHost()` method returns only the host part of the *URI* object.

Getting the port of the URI

The port of the *URI* is the optional part of the *URI* that follows the host-part and comes before the path-part. For example, the host of `http://johndoe@example.com:80/my/path?query#token` is '80'. The URI-class can define default-ports that can be returned when no port is given in the `URI`.

Getting the port from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:8080');

$scheme = $uri->getPort();  // "8080"
```

Getting a default port from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com');

$scheme = $uri->getPort();  // "80"
```

The `getHost()` method returns only the port part of the *URI* object.

Getting the path of the URI

The path of the *URI* is the mandatory part of the *URI* that follows the port and comes before the query-part. For example, the path of `http://johndoe@example.com:80/my/path?query#token` is '/my/path'.

Getting the path from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getPath();  // "/my/path"
```

The `getPath()` method returns only the path of the *URI* object.

Getting the query-part of the URI

The query-part of the *URI* is the optional part of the *URI* that follows the path and comes before the fragment. For example, the query of `http://johndoe@example.com:80/my/path?query#token` is 'query'.

Getting the query from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getQuery();  // "a=b&c=d"
```

The `getQuery()` method returns only the query-part of the *URI* object.

Getting the query as array from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getQueryAsArray();
// array(
//  'a' => 'b',
```

```
//   'c' => 'd',
// )
```

The query-part often contains key=value pairs and therefore can be split into an associative array. This array can be retrieved using `getQueryAsArray()`

Getting the fragment-part of the URI

The fragment-part of the *URI* is the optional part of the *URI* that follows the query. For example, the fragment of `http://johndoe@example.com:80/my/path?query#token` is 'token'.

Getting the fragment from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://example.com:80/my/path?a=b&c=d#token');

$scheme = $uri->getFragment();  // "token"
```

The `getFragment()` method returns only the fragment-part of the *URI* object.

Getting the Entire URI

Getting the Entire URI from a Zend\Uri\UriInterface Object

```
$uri = Zend\Uri\UriFactory::factory('http://www.zend.com');

echo $uri->toString();  // "http://www.zend.com"

// Alternate method:
echo (string) $uri;     // "http://www.zend.com"
```

The `toString()` method returns the string representation of the entire *URI*.

The `Zend\Uri\UriInterface` defines also a magic `__toString()` method that returns the string representation of the *URI* when the Object is cast to a string.

Validating the URI

When using `Zend\Uri\UriFactory::factory()` the given *URI* will always be validated and a `Zend\Uri\Exception\InvalidArgumentException` will be thrown when the *URI* is invalid. However, after the `Zend\Uri\UriInterface` is instantiated for a new *URI* or an existing valid one, it is possible that the *URI* can later become invalid after it is manipulated.

Validating a Zend_Uri_* Object

```
$uri = Zend\Uri\UriFactory::factory('http://www.zend.com');

$isValid = $uri->isValid();  // TRUE
```

The `isValid()` instance method provides a means to check that the *URI* object is still valid.

CHAPTER 129

Introduction

The `Zend\Validator` component provides a set of commonly needed validators. It also provides a simple validator chaining mechanism by which multiple validators may be applied to a single datum in a user-defined order.

What is a validator?

A validator examines its input with respect to some requirements and produces a boolean result - whether the input successfully validates against the requirements. If the input does not meet the requirements, a validator may additionally provide information about which requirement(s) the input does not meet.

For example, a web application might require that a username be between six and twelve characters in length and may only contain alphanumeric characters. A validator can be used for ensuring that 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\Validator\ValidatorInterface`, which defines two methods, `isValid()` and `getMessages()`. The `isValid()` method performs validation upon the provided value, returning `TRUE` if and only if the value passes against the validation criteria.

If `isValid()` returns `FALSE`, the `getMessages()` returns an array of messages explaining the reason(s) for validation failure. The array keys are short strings that identify the reasons for validation failure, and the array values are the corresponding human-readable string messages. The keys and values are class-dependent; each validation class defines its own set of validation failure messages and the unique keys that identify them. Each class also has a const definition that matches each identifier for a validation failure cause.

Note: The `getMessages()` methods return validation failure information only for the most recent `isValid()` call. Each call to `isValid()` clears any messages and errors caused by a previous `isValid()` call, because it's

likely that each call to `isValid()` is made for a different input value.

The following example illustrates validation of an e-mail address:

```
$validator = new Zend\Validator\EmailAddress();

if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $messageId => $message) {
        echo "Validation failure '$messageId': $message\n";
    }
}
```

Customizing messages

Validator classes provide a `setMessage()` method with which you can specify the format of a message returned by `getMessages()` in case of validation failure. The first argument of this method is a string containing the error message. You can include tokens in this string which will be substituted with data relevant to the validator. The token **%value%** is supported by all validators; this is substituted with the value you passed to `isValid()`. Other tokens may be supported on a case-by-case basis in each validation class. For example, **%max%** is a token supported by `Zend\Validator\LessThan`. The `getMessageVariables()` method returns an array of variable tokens supported by the validator.

The second optional argument is a string that identifies the validation failure message template to be set, which is useful when a validation class defines more than one cause for failure. If you omit the second argument, `setMessage()` assumes the message you specify should be used for the first message template declared in the validation class. Many validation classes only have one error message template defined, so there is no need to specify which message template you are changing.

```
$validator = new Zend\Validator\StringLength(8);

$validator->setMessage(
    'The string \'%value%\' is too short; it must be at least %min% ' .
    'characters',
    Zend\Validator\StringLength::TOO_SHORT);

if (!$validator->isValid('word')) {
    $messages = $validator->getMessages();
    echo current($messages);

    // "The string 'word' is too short; it must be at least 8 characters"
}
```

You can set multiple messages using the `setMessages()` method. Its argument is an array containing key/message pairs.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));

$validator->setMessages( array(
    Zend\Validator\StringLength::TOO_SHORT =>
        'The string \'%value%\' is too short',
    Zend\Validator\StringLength::TOO_LONG  =>
```

```
        'The string \'%value%\' is too long'
));
```

If your application requires even greater flexibility with which it reports validation failures, you can access properties by the same name as the message tokens supported by a given validation class. The `value` property is always available in a validator; it is the value you specified as the argument of `isValid()`. Other properties may be supported on a case-by-case basis in each validation class.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));

if (!validator->isValid('word')) {
    echo 'Word failed: '
         . $validator->value
         . '; its length is not between '
         . $validator->min
         . ' and '
         . $validator->max
         . "\n";
}
```

Translating messages

Validator classes provide a `setTranslator()` method with which you can specify a instance of `Zend\I18n\Translator\Translator` which will translate the messages in case of a validation failure. The `getTranslator()` method returns the set translator instance.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
$translate = new Zend\I18n\Translator\Translator();
// configure the translator...

$validator->setTranslator($translate);
```

With the static `setDefaultTranslator()` method you can set a instance of `Zend\I18n\Translator\Translator` which will be used for all validation classes, and can be retrieved with `getDefaultTranslator()`. This prevents you from setting a translator manually for all validator classes, and simplifies your code.

```
$translate = new Zend\I18n\Translator\Translator();
// configure the translator...

Zend\Validator\AbstractValidator::setDefaultTranslator($translate);
```

Note: When you have set an application wide locale within your registry, then this locale will be used as default translator.

Sometimes it is necessary to disable the translator within a validator. To archive this you can use the `setDisableTranslator()` method, which accepts a boolean parameter, and `isTranslatorDisabled()` to get the set value.

```
$validator = new Zend\Validator\StringLength(array('min' => 8, 'max' => 12));
if (!$validator->isTranslatorDisabled()) {
    $validator->setDisableTranslator();
}
```

It is also possible to use a translator instead of setting own messages with `setMessage()`. But doing so, you should keep in mind, that the translator works also on messages you set your own.

CHAPTER 130

Standard Validation Classes

Zend Framework comes with a standard set of validation classes, which are ready for you to use.

CHAPTER 131

Alnum

`Zend\Validator\Alnum` allows you to validate if a given value contains only alphabetical characters and digits. There is no length limitation for the input you want to validate.

Supported options for Zend\Validator\Alnum

The following options are supported for `Zend\Validator\Alnum`:

- **allowWhiteSpace**: If whitespace characters are allowed. This option defaults to `FALSE`

Basic usage

A basic example is the following one:

```
$validator = new Zend\Validator\Alnum();
if ($validator->isValid('Abcd12')) {
    // value contains only allowed chars
} else {
    // false
}
```

Using whitespaces

Per default whitespaces are not accepted because they are not part of the alphabet. Still, there is a way to accept them as input. This allows to validate complete sentences or phrases.

To allow the usage of whitespaces you need to give the `allowWhiteSpace` option. This can be done while creating an instance of the validator, or afterwards by using `setAllowWhiteSpace()`. To get the actual state you can use `getAllowWhiteSpace()`.

```php
$validator = new Zend\Validator\Alnum(array('allowWhiteSpace' => true));
if ($validator->isValid('Abcd and 12')) {
    // value contains only allowed chars
} else {
    // false
}
```

Using different languages

When using `Zend\Validator\Alnum` then the language which the user sets within his browser will be used to set the allowed characters. This means when your user sets **de** for german then he can also enter characters like **ä**, **ö** and **ü** additionally to the characters from the english alphabet.

Which characters are allowed depends completely on the used language as every language defines it's own set of characters.

There are actually 3 languages which are not accepted in their own script. These languages are **korean**, **japanese** and **chinese** because this languages are using an alphabet where a single character is build by using multiple characters.

In the case you are using these languages, the input will only be validated by using the english alphabet.

CHAPTER 132

Alpha

`Zend\Validator\Alpha` allows you to validate if a given value contains only alphabetical characters. There is no length limitation for the input you want to validate. This validator is related to the `Zend\Validator\Alnum` validator with the exception that it does not accept digits.

Supported options for Zend\Validator\Alpha

The following options are supported for `Zend\Validator\Alpha`:

- **allowWhiteSpace**: If whitespace characters are allowed. This option defaults to `FALSE`

Basic usage

A basic example is the following one:

```
$validator = new Zend\Validator\Alpha();
if ($validator->isValid('Abcd')) {
    // value contains only allowed chars
} else {
    // false
}
```

Using whitespaces

Per default whitespaces are not accepted because they are not part of the alphabet. Still, there is a way to accept them as input. This allows to validate complete sentences or phrases.

To allow the usage of whitespaces you need to give the `allowWhiteSpace` option. This can be done while creating an instance of the validator, or afterwards by using `setAllowWhiteSpace()`. To get the actual state you can use `getAllowWhiteSpace()`.

```php
$validator = new Zend\Validator\Alpha(array('allowWhiteSpace' => true));
if ($validator->isValid('Abcd and efg')) {
    // value contains only allowed chars
} else {
    // false
}
```

Using different languages

When using `Zend\Validator\Alpha` then the language which the user sets within his browser will be used to set the allowed characters. This means when your user sets **de** for german then he can also enter characters like **ä**, **ö** and **ü** additionally to the characters from the english alphabet.

Which characters are allowed depends completely on the used language as every language defines it's own set of characters.

There are actually 3 languages which are not accepted in their own script. These languages are **korean**, **japanese** and **chinese** because this languages are using an alphabet where a single character is build by using multiple characters.

In the case you are using these languages, the input will only be validated by using the english alphabet.

CHAPTER 133

Barcode

`Zend\Validator\Barcode` allows you to check if a given value can be represented as barcode.

`Zend\Validator\Barcode` supports multiple barcode standards and can be extended with proprietary barcode implementations very easily. The following barcode standards are supported:

- **CODABAR**: Also known as Code-a-bar.

  This barcode has no length limitation. It supports only digits, and 6 special chars. Codabar is a self-checking barcode. This standard is very old. Common use cases are within airbills or photo labs where multi-part forms are used with dot-matrix printers.

- **CODE128**: CODE128 is a high density barcode.

  This barcode has no length limitation. It supports the first 128 ascii characters. When used with printing characters it has an checksum which is calculated modulo 103. This standard is used worldwide as it supports upper and lowercase characters.

- **CODE25**: Often called "two of five" or "Code25 Industrial".

  This barcode has no length limitation. It supports only digits, and the last digit can be an optional checksum which is calculated with modulo 10. This standard is very old and nowadays not often used. Common use cases are within the industry.

- **CODE25INTERLEAVED**: Often called "Code 2 of 5 Interleaved".

  This standard is a variant of CODE25. It has no length limitation, but it must contain an even amount of characters. It supports only digits, and the last digit can be an optional checksum which is calculated with modulo 10. It is used worldwide and common on the market.

- **CODE39**: CODE39 is one of the oldest available codes.

  This barcode has a variable length. It supports digits, upper cased alphabetical characters and 7 special characters like whitespace, point and dollar sign. It can have an optional checksum which is calculated with modulo 43. This standard is used worldwide and common within the industry.

- **CODE39EXT**: CODE39EXT is an extension of CODE39.

  This barcode has the same properties as CODE39. Additionally it allows the usage of all 128 ASCII characters. This standard is used worldwide and common within the industry.

- **CODE93**: CODE93 is the successor of CODE39.

  This barcode has a variable length. It supports digits, alphabetical characters and 7 special characters. It has an optional checksum which is calculated with modulo 47 and contains 2 characters. This standard produces a denser code than CODE39 and is more secure.

- **CODE93EXT**: CODE93EXT is an extension of CODE93.

  This barcode has the same properties as CODE93. Additionally it allows the usage of all 128 ASCII characters. This standard is used worldwide and common within the industry.

- **EAN2**: EAN is the shortcut for "European Article Number".

  These barcode must have 2 characters. It supports only digits and does not have a checksum. This standard is mainly used as addition to EAN13 (ISBN) when printed on books.

- **EAN5**: EAN is the shortcut for "European Article Number".

  These barcode must have 5 characters. It supports only digits and does not have a checksum. This standard is mainly used as addition to EAN13 (ISBN) when printed on books.

- **EAN8**: EAN is the shortcut for "European Article Number".

  These barcode can have 7 or 8 characters. It supports only digits. When it has a length of 8 characters it includes a checksum. This standard is used worldwide but has a very limited range. It can be found on small articles where a longer barcode could not be printed.

- **EAN12**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 12 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used within the USA and common on the market. It has been superseded by EAN13.

- **EAN13**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 13 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used worldwide and common on the market.

- **EAN14**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 14 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is used worldwide and common on the market. It is the successor for EAN13.

- **EAN18**: EAN is the shortcut for "European Article Number".

  This barcode must have a length of 18 characters. It support only digits. The last digit is always a checksum digit which is calculated with modulo 10. This code is often used for the identification of shipping containers.

- **GTIN12**: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN12 and is its successor. It's commonly used within the USA.

- **GTIN13**: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN13 and is its successor. It is used worldwide by industry.

- **GTIN14**: GTIN is the shortcut for "Global Trade Item Number".

  This barcode uses the same standard as EAN14 and is its successor. It is used worldwide and common on the market.

- **IDENTCODE**: Identcode is used by Deutsche Post and DHL. It's an specialized implementation of Code25.

  This barcode must have a length of 12 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is mainly used by the companies DP and DHL.

- **INTELLIGENTMAIL**: Intelligent Mail is a postal barcode.

  This barcode can have a length of 20, 25, 29 or 31 characters. It supports only digits, and contains no checksum. This standard is the successor of *PLANET* and *POSTNET*. It is mainly used by the United States Postal Services.

- **ISSN**: *ISSN* is the abbreviation for International Standard Serial Number.

  This barcode can have a length of 8 or 13 characters. It supports only digits, and the last digit must be a checksum digit which is calculated with modulo 11. It is used worldwide for printed publications.

- **ITF14**: ITF14 is the GS1 implementation of an Interleaved Two of Five bar code.

  This barcode is a special variant of Interleaved 2 of 5. It must have a length of 14 characters and is based on GTIN14. It supports only digits, and the last digit must be a checksum digit which is calculated with modulo 10. It is used worldwide and common within the market.

- **LEITCODE**: Leitcode is used by Deutsche Post and DHL. It's an specialized implementation of Code25.

  This barcode must have a length of 14 characters. It supports only digits, and the last digit is always a checksum which is calculated with modulo 10. This standard is mainly used by the companies DP and DHL.

- **PLANET**: Planet is the abbreviation for Postal Alpha Numeric Encoding Technique.

  This barcode can have a length of 12 or 14 characters. It supports only digits, and the last digit is always a checksum. This standard is mainly used by the United States Postal Services.

- **POSTNET**: Postnet is used by the US Postal Service.

  This barcode can have a length of 6, 7, 10 or 12 characters. It supports only digits, and the last digit is always a checksum. This standard is mainly used by the United States Postal Services.

- **ROYALMAIL**: Royalmail is used by Royal Mail.

  This barcode has no defined length. It supports digits, uppercase letters, and the last digit is always a checksum. This standard is mainly used by Royal Mail for their Cleanmail Service. It is also called *RM4SCC*.

- **SSCC**: SSCC is the shortcut for "Serial Shipping Container Code".

  This barcode is a variant of EAN barcode. It must have a length of 18 characters and supports only digits. The last digit must be a checksum digit which is calculated with modulo 10. It is commonly used by the transport industry.

- **UPCA**: UPC is the shortcut for "Universal Product Code".

  This barcode preceded EAN13. It must have a length of 12 characters and supports only digits. The last digit must be a checksum digit which is calculated with modulo 10. It is commonly used within the USA.

- **UPCE**: UPCE is the short variant from UPCA.

  This barcode is a smaller variant of UPCA. It can have a length of 6, 7 or 8 characters and supports only digits. When the barcode is 8 chars long it includes a checksum which is calculated with modulo 10. It is commonly used with small products where a UPCA barcode would not fit.

Supported options for Zend\Validator\Barcode

The following options are supported for `Zend\Validator\Barcode`:

- **adapter**: Sets the barcode adapter which will be used. Supported are all above noted adapters. When using a self defined adapter, then you have to set the complete class name.
- **checksum**: `TRUE` when the barcode should contain a checksum. The default value depends on the used adapter. Note that some adapters don't allow to set this option.

- **options**: Defines optional options for a self written adapters.

Basic usage

To validate if a given string is a barcode you just need to know its type. See the following example for an EAN13 barcode:

```
$valid = new Zend\Validator\Barcode('EAN13');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Optional checksum

Some barcodes can be provided with an optional checksum. These barcodes would be valid even without checksum. Still, when you provide a checksum, then you should also validate it. By default, these barcode types perform no checksum validation. By using the `checksum` option you can define if the checksum will be validated or ignored.

```
$valid = new Zend\Validator\Barcode(array(
    'adapter'  => 'EAN13',
    'checksum' => false,
));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Note: Reduced security by disabling checksum validation

By switching off checksum validation you will also reduce the security of the used barcodes. Additionally you should note that you can also turn off the checksum validation for those barcode types which must contain a checksum value. Barcodes which would not be valid could then be returned as valid even if they are not.

Writing custom adapters

You may write custom barcode validators for usage with `Zend\Validator\Barcode`; this is often necessary when dealing with proprietary barcode types. To write your own barcode validator, you need the following information.

- **Length**: The length your barcode must have. It can have one of the following values:
  - **Integer**: A value greater 0, which means that the barcode must have this length.
  - **-1**: There is no limitation for the length of this barcode.
  - **"even"**: The length of this barcode must have a even amount of digits.
  - **"odd"**: The length of this barcode must have a odd amount of digits.
  - **array**: An array of integer values. The length of this barcode must have one of the set array values.

- **Characters**: A string which contains all allowed characters for this barcode. Also the integer value 128 is allowed, which means the first 128 characters of the ASCII table.
- **Checksum**: A string which will be used as callback for a method which does the checksum validation.

Your custom barcode validator must extend `Zend\Validator\Barcode\AbstractAdapter` or implement `Zend\Validator\Barcode\AdapterInterface`.

As an example, let's create a validator that expects an even number of characters that include all digits and the letters 'ABCDE', and which requires a checksum.

```
class My\Barcode\MyBar extends Zend\Validator\Barcode\AbstractAdapter
{
    protected $length     = 'even';
    protected $characters = '0123456789ABCDE';
    protected $checksum   = 'mod66';

    protected function mod66($barcode)
    {
        // do some validations and return a boolean
    }
}

$valid = new Zend\Validator\Barcode('My\Barcode\MyBar');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

CHAPTER 134

Between

`Zend\Validator\Between` allows you to validate if a given value is between two other values.

Note: ZendValidatorBetween supports only number validation

It should be noted that `Zend\Validator\Between` supports only the validation of numbers. Strings or dates can not be validated with this validator.

Supported options for Zend\Validator\Between

The following options are supported for `Zend\Validator\Between`:

- **inclusive**: Defines if the validation is inclusive the minimum and maximum border values or exclusive. It defaults to `TRUE`.
- **max**: Sets the maximum border for the validation.
- **min**: Sets the minimum border for the validation.

Default behaviour for Zend\Validator\Between

Per default this validator checks if a value is between `min` and `max` where both border values are allowed as value.

```
$valid  = new Zend\Validator\Between(array('min' => 0, 'max' => 10));
$value  = 10;
$result = $valid->isValid($value);
// returns true
```

In the above example the result is `TRUE` due to the reason that per default the search is inclusively the border values. This means in our case that any value from '0' to '10' is allowed. And values like '-1' and '11' will return `FALSE`.

Validation exclusive the border values

Sometimes it is useful to validate a value by excluding the border values. See the following example:

```
$valid  = new Zend\Validator\Between(
    array(
        'min' => 0,
        'max' => 10,
        'inclusive' => false
    )
);
$value  = 10;
$result = $valid->isValid($value);
// returns false
```

The example is almost equal to our first example but we excluded the border value. Now the values '0' and '10' are no longer allowed and will return `FALSE`.

CHAPTER 135

Callback

`Zend\Validator\Callback` allows you to provide a callback with which to validate a given value.

Supported options for Zend\Validator\Callback

The following options are supported for `Zend\Validator\Callback`:

- **callback**: Sets the callback which will be called for the validation.
- **options**: Sets the additional options which will be given to the callback.

Basic usage

The simplest usecase is to have a single function and use it as a callback. Let's expect we have the following function.

```
function myMethod($value)
{
    // some validation
    return true;
}
```

To use it within `Zend\Validator\Callback` you just have to call it this way:

```
$valid = new Zend\Validator\Callback('myMethod');
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Usage with closures

PHP 5.3 introduces closures, which are basically self-contained or **anonymous** functions. *PHP* considers closures another form of callback, and, as such, may be used with `Zend\Validator\Callback`. As an example:

```
$valid = new Zend\Validator\Callback(function($value){
    // some validation
    return true;
});

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Usage with class-based callbacks

Of course it's also possible to use a class method as callback. Let's expect we have the following class method:

```
class MyClass
{
    public function myMethod($value)
    {
        // some validation
        return true;
    }
}
```

The definition of the callback is in this case almost the same. You have just to create an instance of the class before the method and create an array describing the callback:

```
$object = new MyClass;
$valid = new Zend\Validator\Callback(array($object, 'myMethod'));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

You may also define a static method as a callback. Consider the following class definition and validator usage:

```
class MyClass
{
    public static function test($value)
    {
        // some validation
        return true;
    }
}

$valid = new Zend\Validator\Callback(array('MyClass', 'test'));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
```

```
    // input is invalid
}
```

Finally, if you are using *PHP* 5.3, you may define the magic method `__invoke()` in your class. If you do so, simply providing an instance of the class as the callback will also work:

```
class MyClass
{
    public function __invoke($value)
    {
        // some validation
        return true;
    }
}

$object = new MyClass();
$valid = new Zend\Validator\Callback($object);
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Adding options

`Zend\Validator\Callback` also allows the usage of options which are provided as additional arguments to the callback.

Consider the following class and method definition:

```
class MyClass
{
    function myMethod($value, $option)
    {
        // some validation
        return true;
    }
}
```

There are two ways to inform the validator of additional options: pass them in the constructor, or pass them to the `setOptions()` method.

To pass them to the constructor, you would need to pass an array containing two keys, "callback" and "options":

```
$valid = new Zend\Validator\Callback(array(
    'callback' => array('MyClass', 'myMethod'),
    'options'  => $option,
));

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Otherwise, you may pass them to the validator after instantiation:

```php
$valid = new Zend\Validator\Callback(array('MyClass', 'myMethod'));
$valid->setOptions($option);

if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

When there are additional values given to `isValid()` then these values will be added immediately after `$value`.

```php
$valid = new Zend\Validator\Callback(array('MyClass', 'myMethod'));
$valid->setOptions($option);

if ($valid->isValid($input, $additional)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

When making the call to the callback, the value to be validated will always be passed as the first argument to the callback followed by all other values given to `isValid()`; all other options will follow it. The amount and type of options which can be used is not limited.

CHAPTER 136

CreditCard

`Zend\Validator\CreditCard` allows you to validate if a given value could be a credit card number.

A credit card contains several items of metadata, including a hologram, account number, logo, expiration date, security code and the card holder name. The algorithms for verifying the combination of metadata are only known to the issuing company, and should be verified with them for purposes of payment. However, it's often useful to know whether or not a given number actually falls within the ranges of possible numbers **prior** to performing such verification, and, as such, `Zend\Validator\CreditCard` simply verifies that the credit card number provided is well-formed.

For those cases where you have a service that can perform comprehensive verification, `Zend\Validator\CreditCard` also provides the ability to attach a service callback to trigger once the credit card number has been deemed valid; this callback will then be triggered, and its return value will determine overall validity.

The following issuing institutes are accepted:

- **American Express**

  China UnionPay

  Diners Club Card Blanche

  Diners Club International

  Diners Club US & Canada

  Discover Card

  JCB

  Laser

  Maestro

  MasterCard

  Solo

  Visa

  Visa Electron

Note: Invalid institutes

The institutes **Bankcard** and **Diners Club enRoute** do not exist anymore. Therefore they are treated as invalid.

Switch has been rebranded to **Visa** and is therefore also treated as invalid.

Supported options for Zend\Validator\CreditCard

The following options are supported for `Zend\Validator\CreditCard`:

- **service**: A callback to an online service which will additionally be used for the validation.
- **type**: The type of credit card which will be validated. See the below list of institutes for details.

Basic usage

There are several credit card institutes which can be validated by `Zend\Validator\CreditCard`. Per default, all known institutes will be accepted. See the following example:

```
$valid = new Zend\Validator\CreditCard();
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

The above example would validate against all known credit card institutes.

Accepting defined credit cards

Sometimes it is necessary to accept only defined credit card institutes instead of all; e.g., when you have a webshop which accepts only Visa and American Express cards. `Zend\Validator\CreditCard` allows you to do exactly this by limiting it to exactly these institutes.

To use a limitation you can either provide specific institutes at initiation, or afterwards by using `setType()`. Each can take several arguments.

You can provide a single institute:

```
$valid = new Zend\Validator\CreditCard(
    Zend\Validator\CreditCard::AMERICAN_EXPRESS
);
```

When you want to allow multiple institutes, then you can provide them as array:

```
$valid = new Zend\Validator\CreditCard(array(
    Zend\Validator\CreditCard::AMERICAN_EXPRESS,
    Zend\Validator\CreditCard::VISA
));
```

And as with all validators, you can also pass an associative array of options or an instance of `Traversable`. In this case you have to provide the institutes with the `type` array key as simulated here:

```
$valid = new Zend\Validator\CreditCard(array(
    'type' => array(Zend\Validator\CreditCard::AMERICAN_EXPRESS)
));
```

Table 136.1: Constants for credit card institutes

Institute	Constant
American Express	AMERICAN_EXPRESS
China UnionPay	UNIONPAY
Diners Club Card Blanche	DINERS_CLUB
Diners Club International	DINERS_CLUB
Diners Club US & Canada	DINERS_CLUB_US
Discover Card	DISCOVER
JCB	JCB
Laser	LASER
Maestro	MAESTRO
MasterCard	MASTERCARD
Solo	SOLO
Visa	VISA
Visa Electron	VISA

You can also set or add institutes afterward instantiation by using the methods `setType()`, `addType()` and `getType()`.

```
$valid = new Zend\Validator\CreditCard();
$valid->setType(array(
    Zend\Validator\CreditCard::AMERICAN_EXPRESS,
    Zend\Validator\CreditCard::VISA
));
```

Note: Default institute

When no institute is given at initiation then `ALL` will be used, which sets all institutes at once.

In this case the usage of `addType()` is useless because all institutes are already added.

Validation by using foreign APIs

As said before `Zend\Validator\CreditCard` will only validate the credit card number. Fortunately, some institutes provide online *API*s which can validate a credit card number by using algorithms which are not available to the public. Most of these services are paid services. Therefore, this check is deactivated per default.

When you have access to such an *API*, then you can use it as an add on for `Zend\Validator\CreditCard` and increase the security of the validation.

To do so, you simply need to give a callback which will be called when the generic validation has passed. This prevents the *API* from being called for invalid numbers, which increases the performance of the application.

`setService()` sets a new service, and `getService()` returns the set service. As a configuration option, you can give the array key '`service`' at initiation. For details about possible options take a look into *Callback*.

```php
// Your service class
class CcService
{
    public function checkOnline($cardnumber, $types)
    {
        // some online validation
    }
}

// The validation
$service = new CcService();
$valid   = new Zend\Validator\CreditCard(Zend\Validator\CreditCard::VISA);
$valid->setService(array($service, 'checkOnline'));
```

As you can see the callback method will be called with the credit card number as the first parameter, and the accepted types as the second parameter.

Ccnum

Note: The `Ccnum` validator has been deprecated in favor of the `CreditCard` validator. For security reasons you should use CreditCard instead of Ccnum.

CHAPTER 137

Date

`Zend\Validator\Date` allows you to validate if a given value contains a date. This validator validates also localized input.

Supported options for Zend\Validator\Date

The following options are supported for `Zend\Validator\Date`:

- **format**: Sets the format which is used to write the date.
- **locale**: Sets the locale which will be used to validate date values.

Default date validation

The easiest way to validate a date is by using the default date format. It is used when no locale and no format has been given.

```
1 $validator = new Zend\Validator\Date();
2
3 $validator->isValid('2000-10-10');   // returns true
4 $validator->isValid('10.10.2000'); // returns false
```

The default date format for `Zend\Validator\Date` is 'yyyy-MM-dd'.

Localized date validation

`Zend\Validator\Date` validates also dates which are given in a localized format. By using the `locale` option you can define the locale which the date format should use for validation.

```
$validator = new Zend\Validator\Date(array('locale' => 'de'));

$validator->isValid('10.Feb.2010'); // returns true
$validator->isValid('10.May.2010'); // returns false
```

The `locale` option sets the default date format. In the above example this is 'dd.MM.yyyy' which is defined as default date format for 'de'.

Self defined date validation

`Zend\Validator\Date` supports also self defined date formats. When you want to validate such a date you can use the `format` option.

```
$validator = new Zend\Validator\Date(array('format' => 'yyyy'));

$validator->isValid('2010'); // returns true
$validator->isValid('May');  // returns false
```

Of course you can combine `format` and `locale`. In this case you can also use localized month or day names.

```
$validator = new Zend\Validator\Date(array('format' => 'yyyy MMMM', 'locale' => 'de'));

$validator->isValid('2010 Dezember'); // returns true
$validator->isValid('2010 June');     // returns false
```

CHAPTER 138

Db\RecordExists and Db\NoRecordExists

`Zend\Validator\Db\RecordExists` and `Zend\Validator\Db\NoRecordExists` provide a means to test whether a record exists in a given table of a database, with a given value.

Supported options for Zend\Validator\Db_*

The following options are supported for `Zend\Validator\Db\NoRecordExists` and `Zend\Validator\Db\RecordExists`:

- **adapter**: The database adapter which will be used for the search.
- **exclude**: Sets records which will be excluded from the search.
- **field**: The database field within this table which will be searched for the record.
- **schema**: Sets the schema which will be used for the search.
- **table**: The table which will be searched for the record.

Basic usage

An example of basic usage of the validators:

```
//Check that the email address exists in the database
$validator = new Zend\Validator\Db\RecordExists(
    array(
        'table' => 'users',
        'field' => 'emailaddress'
    )
);

if ($validator->isValid($emailaddress)) {
    // email address appears to be valid
```

```
} else {
    // email address is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

The above will test that a given email address is in the database table. If no record is found containing the value of `$emailaddress` in the specified column, then an error message is displayed.

```
//Check that the username is not present in the database
$validator = new Zend\Validator\Db\NoRecordExists(
    array(
        'table' => 'users',
        'field' => 'username'
    )
);
if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above will test that a given username is not in the database table. If a record is found containing the value of `$username` in the specified column, then an error message is displayed.

Excluding records

`Zend\Validator\Db\RecordExists` and `Zend\Validator\Db\NoRecordExists` also provide a means to test the database, excluding a part of the table, either by providing a where clause as a string, or an array with the keys "field" and "value".

When providing an array for the exclude clause, the **!=** operator is used, so you can check the rest of a table for a value before altering a record (for example on a user profile form)

```
//Check no other users have the username
$user_id   = $user->getId();
$validator = new Zend\Validator\Db\NoRecordExists(
    array(
        'table' => 'users',
        'field' => 'username',
        'exclude' => array(
            'field' => 'id',
            'value' => $user_id
        )
    )
);

if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
```

```
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above example will check the table to ensure no records other than the one where `id = $user_id` contains the value $username.

You can also provide a string to the exclude clause so you can use an operator other than **!=**. This can be useful for testing against composite keys.

```
$email     = 'user@example.com';
$clause    = $db->quoteInto('email = ?', $email);
$validator = new Zend\Validator\Db\RecordExists(
    array(
        'table'   => 'users',
        'field'   => 'username',
        'exclude' => $clause
    )
);

if ($validator->isValid($username)) {
    // username appears to be valid
} else {
    // username is invalid; print the reason
    $messages = $validator->getMessages();
    foreach ($messages as $message) {
        echo "$message\n";
    }
}
```

The above example will check the 'users' table to ensure that only a record with both the username `$username` and with the email `$email` is valid.

Database Adapters

You can also specify an adapter. This will allow you to work with applications using multiple database adapters, or where you have not set a default adapter. As in the example below:

```
$validator = new Zend\Validator\Db\RecordExists(
    array(
        'table' => 'users',
        'field' => 'id',
        'adapter' => $dbAdapter
    )
);
```

Database Schemas

You can specify a schema within your database for adapters such as PostgreSQL and DB/2 by simply supplying an array with `table` and `schema` keys. As in the example below:

```php
$validator = new Zend\Validator\Db\RecordExists(
    array(
        'table'  => 'users',
        'schema' => 'my',
        'field'  => 'id'
    )
);
```

CHAPTER 139

Digits

`Zend\Validator\Digits` validates if a given value contains only digits.

Supported options for Zend\Validator\Digits

There are no additional options for `Zend\Validator\Digits`:

Validating digits

To validate if a given value contains only digits and no other characters, simply call the validator like shown in this example:

```
1 $validator = new Zend\Validator\Digits();
2
3 $validator->isValid("1234567890"); // returns true
4 $validator->isValid(1234);         // returns true
5 $validator->isValid('1a234');      // returns false
```

Note: Validating numbers

When you want to validate numbers or numeric values, be aware that this validator only validates digits. This means that any other sign like a thousand separator or a comma will not pass this validator. In this case you should use `Zend\Validator\Int` or `Zend\Validator\Float`.

CHAPTER 140

EmailAddress

`Zend\Validator\EmailAddress` allows you to validate an email address. The validator first splits the email address on local-part @ hostname and attempts to match these against known specifications for email addresses and hostnames.

Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\EmailAddress();
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This will match the email address `$email` and on failure populate `getMessages()` with useful error messages.

Options for validating Email Addresses

`Zend\Validator\EmailAddress` supports several options which can either be set at initiation, by giving an array with the related options, or afterwards, by using `setOptions()`. The following options are supported:

- **allow**: Defines which type of domain names are accepted. This option is used in conjunction with the hostname option to set the hostname validator. For more information about possible values of this option, look at *Hostname* and possible `ALLOW*` constants. This option defaults to `ALLOW_DNS`.

- **deep**: Defines if the servers MX records should be verified by a deep check. When this option is set to `TRUE` then additionally to MX records also the A, A6 and `AAAA` records are used to verify if the server accepts emails. This option defaults to `FALSE`.
- **domain**: Defines if the domain part should be checked. When this option is set to `FALSE`, then only the local part of the email address will be checked. In this case the hostname validator will not be called. This option defaults to `TRUE`.
- **hostname**: Sets the hostname validator with which the domain part of the email address will be validated.
- **mx**: Defines if the MX records from the server should be detected. If this option is defined to `TRUE` then the MX records are used to verify if the server accepts emails. This option defaults to `FALSE`.

```
$validator = new Zend\Validator\EmailAddress();
$validator->setOptions(array('domain' => false));
```

Complex local parts

`Zend\Validator\EmailAddress` will match any valid email address according to RFC2822. For example, valid emails include **bob@domain.com**, **bob+jones@domain.us**, **"bob@jones"@domain.com** and **"bob jones"@domain.com**

Some obsolete email formats will not currently validate (e.g. carriage returns or a "\" character in an email address).

Validating only the local part

If you need `Zend\Validator\EmailAddress` to check only the local part of an email address, and want to disable validation of the hostname, you can set the `domain` option to `FALSE`. This forces `Zend\Validator\EmailAddress` not to validate the hostname part of the email address.

```
$validator = new Zend\Validator\EmailAddress();
$validator->setOptions(array('domain' => FALSE));
```

Validating different types of hostnames

The hostname part of an email address is validated against *Zend\Validator\Hostname*. By default only DNS hostnames of the form `domain.com` are accepted, though if you wish you can accept IP addresses and Local hostnames too.

To do this you need to instantiate `Zend\Validator\EmailAddress` passing a parameter to indicate the type of hostnames you want to accept. More details are included in `Zend\Validator\Hostname`, though an example of how to accept both DNS and Local hostnames appears below:

```
$validator = new Zend\Validator\EmailAddress(
                    Zend\Validator\Hostname::ALLOW_DNS |
                    Zend\Validator\Hostname::ALLOW_LOCAL);
if ($validator->isValid($email)) {
    // email appears to be valid
} else {
    // email is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
```

```
    }
}
```

Checking if the hostname actually accepts email

Just because an email address is in the correct format, it doesn't necessarily mean that email address actually exists. To help solve this problem, you can use MX validation to check whether an MX (email) entry exists in the DNS record for the email's hostname. This tells you that the hostname accepts email, but doesn't tell you the exact email address itself is valid.

MX checking is not enabled by default. To enable MX checking you can pass a second parameter to the `Zend\Validator\EmailAddress` constructor.

```
$validator = new Zend\Validator\EmailAddress(
    array(
        'allow' => Zend\Validator\Hostname::ALLOW_DNS,
        'mx'    => true
    )
);
```

Note: MX Check under Windows

Within Windows environments MX checking is only available when *PHP* 5.3 or above is used. Below *PHP* 5.3 MX checking will not be used even if it's activated within the options.

Alternatively you can either pass `TRUE` or `FALSE` to `setValidateMx()` to enable or disable MX validation.

By enabling this setting network functions will be used to check for the presence of an MX record on the hostname of the email address you wish to validate. Please be aware this will likely slow your script down.

Sometimes validation for MX records returns `FALSE`, even if emails are accepted. The reason behind this behaviour is, that servers can accept emails even if they do not provide a MX record. In this case they can provide A, A6 or `AAAA` records. To allow `Zend\Validator\EmailAddress` to check also for these other records, you need to set deep MX validation. This can be done at initiation by setting the `deep` option or by using `setOptions()`.

```
$validator = new Zend\Validator\EmailAddress(
    array(
        'allow' => Zend\Validator\Hostname::ALLOW_DNS,
        'mx'    => true,
        'deep'  => true
    )
);
```

Sometimes it can be useful to get the server's MX information which have been used to do further processing. Simply use `getMXRecord()` after validation. This method returns the received MX record including weight and sorted by it.

Warning: Performance warning

You should be aware that enabling MX check will slow down you script because of the used network functions. Enabling deep check will slow down your script even more as it searches the given server for 3 additional types.

Note: Disallowed IP addresses

You should note that MX validation is only accepted for external servers. When deep MX validation is enabled, then local IP addresses like `192.168.*` or `169.254.*` are not accepted.

Validating International Domains Names

`Zend\Validator\EmailAddress` will also match international characters that exist in some domains. This is known as International Domain Name (IDN) support. This is enabled by default, though you can disable this by changing the setting via the internal `Zend\Validator\Hostname` object that exists within `Zend\Validator\EmailAddress`.

```
1 $validator->getHostnameValidator()->setValidateIdn(false);
```

More information on the usage of `setValidateIdn()` appears in the `Zend\Validator\Hostname` documentation.

Please note IDNs are only validated if you allow DNS hostnames to be validated.

Validating Top Level Domains

By default a hostname will be checked against a list of known TLDs. This is enabled by default, though you can disable this by changing the setting via the internal `Zend\Validator\Hostname` object that exists within `Zend\Validator\EmailAddress`.

```
1 $validator->getHostnameValidator()->setValidateTld(false);
```

More information on the usage of `setValidateTld()` appears in the `Zend\Validator\Hostname` documentation.

Please note TLDs are only validated if you allow DNS hostnames to be validated.

Setting messages

`Zend\Validator\EmailAddress` makes also use of `Zend\Validator\Hostname` to check the hostname part of a given email address. As with Zend Framework 1.10 you can simply set messages for `Zend\Validator\Hostname` from within `Zend\Validator\EmailAddress`.

```
1 $validator = new Zend\Validator\EmailAddress();
2 $validator->setMessages(
3     array(
4         Zend\Validator\Hostname::UNKNOWN_TLD => 'I don't know the TLD you gave'
5     )
6 );
```

Before Zend Framework 1.10 you had to attach the messages to your own `Zend\Validator\Hostname`, and then set this validator within `Zend\Validator\EmailAddress` to get your own messages returned.

CHAPTER 141

Float

`Zend\Validator\Float` allows you to validate if a given value contains a floating-point value. This validator validates also localized input.

Supported options for Zend\Validator\Float

The following options are supported for `Zend\Validator\Float`:

- **locale**: Sets the locale which will be used to validate localized float values.

Simple float validation

The simplest way to validate a float is by using the system settings. When no option is used, the environment locale is used for validation:

```
1 $validator = new Zend\Validator\Float();
2
3 $validator->isValid(1234.5);   // returns true
4 $validator->isValid('10a01'); // returns false
5 $validator->isValid('1,234.5'); // returns true
```

In the above example we expected that our environment is set to "en" as locale.

Localized float validation

Often it's useful to be able to validate also localized values. Float values are often written different in other countries. For example using english you will write "1.5". In german you may write "1,5" and in other languages you may use grouping.

`Zend\Validator\Float` is able to validate such notations. But it is limited to the locale you set. See the following code:

```
$validator = new Zend\Validator\Float(array('locale' => 'de'));

$validator->isValid(1234.5); // returns true
$validator->isValid("1 234,5"); // returns false
$validator->isValid("1.234"); // returns true
```

As you can see, by using a locale, your input is validated localized. Using a different notation you get a `FALSE` when the locale forces a different notation.

The locale can also be set afterwards by using `setLocale()` and retrieved by using `getLocale()`.

CHAPTER 142

GreaterThan

`Zend\Validator\GreaterThan` allows you to validate if a given value is greater than a minimum border value.

Note: ZendValidatorGreaterThan supports only number validation

It should be noted that `Zend\Validator\GreaterThan` supports only the validation of numbers. Strings or dates can not be validated with this validator.

Supported options for Zend\Validator\GreaterThan

The following options are supported for `Zend\Validator\GreaterThan`:

- **inclusive**: Defines if the validation is inclusive the minimum border value or exclusive. It defaults to `FALSE`.
- **min**: Sets the minimum allowed value.

Basic usage

To validate if a given value is greater than a defined border simply use the following example.

```
1 $valid  = new Zend\Validator\GreaterThan(array('min' => 10));
2 $value  = 8;
3 $return = $valid->isValid($value);
4 // returns false
```

The above example returns `TRUE` for all values which are greater than 10.

Validation inclusive the border value

Sometimes it is useful to validate a value by including the border value. See the following example:

```
$valid  = new Zend\Validator\GreaterThan(
    array(
        'min' => 10,
        'inclusive' => true
    )
);
$value  = 10;
$result = $valid->isValid($value);
// returns true
```

The example is almost equal to our first example but we included the border value. Now the value '10' is allowed and will return `TRUE`.

CHAPTER 143

Hex

`Zend\Validator\Hex` allows you to validate if a given value contains only hexadecimal characters. These are all characters from **0 to 9** and **A to F** case insensitive. There is no length limitation for the input you want to validate.

```
$validator = new Zend\Validator\Hex();
if ($validator->isValid('123ABC')) {
    // value contains only hex chars
} else {
    // false
}
```

Note: Invalid characters

All other characters will return false, including whitespace and decimal point. Also unicode zeros and numbers from other scripts than latin will not be treated as valid.

Supported options for Zend\Validator\Hex

There are no additional options for `Zend\Validator\Hex`:

CHAPTER 144

Hostname

`Zend\Validator\Hostname` allows you to validate a hostname against a set of known specifications. It is possible to check for three different types of hostnames: a *DNS* Hostname (i.e. `domain.com`), IP address (i.e. 1.2.3.4), and Local hostnames (i.e. localhost). By default only *DNS* hostnames are matched.

Supported options for Zend\Validator\Hostname

The following options are supported for `Zend\Validator\Hostname`:

- **allow**: Defines the sort of hostname which is allowed to be used. See *Hostname types* for details.
- **idn**: Defines if *IDN* domains are allowed or not. This option defaults to `TRUE`.
- **ip**: Allows to define a own IP validator. This option defaults to a new instance of `Zend\Validator\Ip`.
- **tld**: Defines if *TLD*s are validated. This option defaults to `TRUE`.

Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\Hostname();
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

This will match the hostname `$hostname` and on failure populate `getMessages()` with useful error messages.

Validating different types of hostnames

You may find you also want to match IP addresses, Local hostnames, or a combination of all allowed types. This can be done by passing a parameter to `Zend\Validator\Hostname` when you instantiate it. The parameter should be an integer which determines what types of hostnames are allowed. You are encouraged to use the `Zend\Validator\Hostname` constants to do this.

The `Zend\Validator\Hostname` constants are: `ALLOW_DNS` to allow only *DNS* hostnames, `ALLOW_IP` to allow IP addresses, `ALLOW_LOCAL` to allow local network names, `ALLOW_URI` to allow RFC3986-compliant addresses, and `ALLOW_ALL` to allow all four above types.

Note: Additional Information on ALLOW_URI

`ALLOW_URI` allows to check hostnames according to RFC3986. These are registered names which are used by *WINS*, NetInfo and also local hostnames like those defined within your `.hosts` file.

To just check for IP addresses you can use the example below:

```
$validator = new Zend\Validator\Hostname(Zend\Validator\Hostname::ALLOW_IP);
if ($validator->isValid($hostname)) {
    // hostname appears to be valid
} else {
    // hostname is invalid; print the reasons
    foreach ($validator->getMessages() as $message) {
        echo "$message\n";
    }
}
```

As well as using `ALLOW_ALL` to accept all common hostnames types you can combine these types to allow for combinations. For example, to accept *DNS* and Local hostnames instantiate your `Zend\Validator\Hostname` object as so:

```
$validator = new Zend\Validator\Hostname(Zend\Validator\Hostname::ALLOW_DNS |
                                         Zend\Validator\Hostname::ALLOW_IP);
```

Validating International Domains Names

Some Country Code Top Level Domains (ccTLDs), such as 'de' (Germany), support international characters in domain names. These are known as International Domain Names (*IDN*). These domains can be matched by `Zend\Validator\Hostname` via extended characters that are used in the validation process.

Note: IDN domains

Until now more than 50 ccTLDs support *IDN* domains.

To match an *IDN* domain it's as simple as just using the standard Hostname validator since *IDN* matching is enabled by default. If you wish to disable *IDN* validation this can be done by either passing a parameter to the `Zend\Validator\Hostname` constructor or via the `setValidateIdn()` method.

You can disable *IDN* validation by passing a second parameter to the `Zend\Validator\Hostname` constructor in the following way.

```php
$validator =
    new Zend\Validator\Hostname(
        array(
            'allow' => Zend\Validator\Hostname::ALLOW_DNS,
            'idn'   => false
        )
    );
```

Alternatively you can either pass `TRUE` or `FALSE` to `setValidateIdn()` to enable or disable *IDN* validation. If you are trying to match an *IDN* hostname which isn't currently supported it is likely it will fail validation if it has any international characters in it. Where a ccTLD file doesn't exist in `Zend/Validator/Hostname` specifying the additional characters a normal hostname validation is performed.

Note: IDN validation

Please note that *IDN*s are only validated if you allow *DNS* hostnames to be validated.

Validating Top Level Domains

By default a hostname will be checked against a list of known *TLD*s. If this functionality is not required it can be disabled in much the same way as disabling *IDN* support. You can disable *TLD* validation by passing a third parameter to the `Zend\Validator\Hostname` constructor. In the example below we are supporting *IDN* validation via the second parameter.

```php
$validator =
    new Zend\Validator\Hostname(
        array(
            'allow' => Zend\Validator\Hostname::ALLOW_DNS,
            'idn'   => true,
            'tld'   => false
        )
    );
```

Alternatively you can either pass `TRUE` or `FALSE` to `setValidateTld()` to enable or disable *TLD* validation.

Note: TLD validation

Please note *TLD*s are only validated if you allow *DNS* hostnames to be validated.

CHAPTER 145

Iban

`Zend\Validator\Iban` validates if a given value could be a *IBAN* number. *IBAN* is the abbreviation for "International Bank Account Number".

Supported options for Zend\Validator\Iban

The following options are supported for `Zend\Validator\Iban`:

- **locale**: Sets the locale which is used to get the *IBAN* format for validation.

IBAN validation

IBAN numbers are always related to a country. This means that different countries use different formats for their *IBAN* numbers. This is the reason why *IBAN* numbers always need a locale. By knowing this we already know how to use `Zend\Validator\Iban`.

Application wide locale

We could use the application wide locale. This means that when no option is given at initiation, `Zend\Validator\Iban` searches for the application wide locale. See the following code snippet:

```
// within bootstrap
Locale::setDefault('de_AT');

// within the module
$validator = new Zend\Validator\Iban();

if ($validator->isValid('AT611904300234573201')) {
    // IBAN appears to be valid
} else {
```

```
    // IBAN is not valid
}
```

Note: Application wide locale

Of course this works only when an application wide locale was set within the registry previously. Otherwise `Locale` will try to use the locale which the client sends or, when non has been send, it uses the environment locale. Be aware that this can lead to unwanted behaviour within the validation.

Ungreedy IBAN validation

Sometime it is useful, just to validate if the given value **is** a *IBAN* number or not. This means that you don't want to validate it against a defined country. This can be done by using a `FALSE` as locale.

```
$validator = new Zend\Validator\Iban(array('locale' => false));
// Note: you can also set a FALSE as single parameter

if ($validator->isValid('AT611904300234573201')) {
    // IBAN appears to be valid
} else {
    // IBAN is not valid
}
```

So **any** *IBAN* number will be valid. Note that this should not be done when you accept only accounts from a single country.

Region aware IBAN validation

To validate against a defined country, you just need to give the wished locale. You can do this by the option `locale` and also afterwards by using `setLocale()`.

```
$validator = new Zend\Validator\Iban(array('locale' => 'de_AT'));

if ($validator->isValid('AT611904300234573201')) {
    // IBAN appears to be valid
} else {
    // IBAN is not valid
}
```

Note: Use full qualified locales

You must give a full qualified locale, otherwise the country could not be detected correct because languages are spoken in multiple countries.

CHAPTER 146

Identical

`Zend\Validator\Identical` allows you to validate if a given value is identical with an set haystack.

Supported options for Zend\Validator\Identical

The following options are supported for `Zend\Validator\Identical`:

- **strict**: Defines if the validation should be done strict. The default value is `TRUE`.
- **token**: Sets the token with which the input will be validated against.

Basic usage

To validate if two values are identical you need to set the origin value as haystack. See the following example which validates two strings.

```
$valid = new Zend\Validator\Identical('origin');
if ($valid->isValid($value) {
    return true;
}
```

The validation will only then return `TRUE` when both values are 100% identical. In our example, when `$value` is 'origin'.

You can set the wished token also afterwards by using the method `setToken()` and `getToken()` to get the actual set token.

Identical objects

Of course Zend\Validator\Identical can not only validate strings, but also any other variable type like Boolean, Integer, Float, Array or even Objects. As already noted Haystack and Value must be identical.

```
$valid = new Zend\Validator\Identical(123);
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

Note: Type comparison

You should be aware that also the type of a variable is used for validation. This means that the string **'3'** is not identical with the integer **3**. When you want such a non strict validation you must set the strict option.

Form elements

Zend\Validator\Identical supports also the comparison of form elements. This can be done by using the element's name as token. See the following example:

```
$form->addElement('password', 'elementOne');
$form->addElement('password', 'elementTwo', array(
    'validators' => array(
        array('identical', false, array('token' => 'elementOne'))
    )
));
```

By using the elements name from the first element as token for the second element, the validator validates if the second element is equal with the first element. In the case your user does not enter two identical values, you will get an validation error.

Strict validation

As mentioned before Zend\Validator\Identical validates tokens strict. You can change this behaviour by using the strict option. The default value for this property is TRUE.

```
$valid = new Zend\Validator\Identical(array('token' => 123, 'strict' => FALSE));
$input = '123';
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

The difference to the previous example is that the validation returns in this case TRUE, even if you compare a integer with string value as long as the content is identical but not the type.

For convenience you can also use setStrict() and getStrict().

Configuration

As all other validators also `Zend\Validator\Identical` supports the usage of configuration settings as input parameter. This means that you can configure this validator with an `Traversable` instance.

But this adds one case which you have to be aware. When you are using an array as haystack then you should wrap it within an '`token`' key when it could contain only one element.

```
$valid = new Zend\Validator\Identical(array('token' => 123));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

The above example validates the integer 123. The reason for this special case is, that you can configure the token which has to be used by giving the '`token`' key.

So, when your haystack contains one element and this element is named '`token`' then you have to wrap it like shown in the example below.

```
$valid = new Zend\Validator\Identical(array('token' => array('token' => 123)));
if ($valid->isValid($input)) {
    // input appears to be valid
} else {
    // input is invalid
}
```

CHAPTER 147

InArray

`Zend\Validator\InArray` allows you to validate if a given value is contained within an array. It is also able to validate multidimensional arrays.

Supported options for Zend\Validator\InArray

The following options are supported for `Zend\Validator\InArray`:

- **haystack**: Sets the haystack for the validation.
- **recursive**: Defines if the validation should be done recursive. This option defaults to `FALSE`.
- **strict**: Three modes of comparison are offered owing to an often overlooked, and potentially dangerous security issue when validating string input from user input.
  - *InArray::COMPARE_STRICT*

    This is a normal in_array strict comparison that checks value and type.
  - *InArray::COMPARE_NOT_STRICT*

    This is a normal in_array non-strict comparison that checks value only.

> **Warning:** This mode may give false positives when strings are compared against ints or floats owing to in_array's behaviour of converting strings to int in such cases. Therefore, *"foo"* would become *0*, *"43foo"* would become *43*, while *"foo43"* would also become *0*.

- *InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_VULNERABILITY*

  To remedy the above warning, this mode offers a middle-ground which allows string representations of numbers to be successfully matched against either their string or int counterpart and vice versa. For example: *"0"* will successfully match against *0*, but *"foo"* would not match against *0* as would be true in the `*COMPARE_NOT_STRICT*` mode. This is the safest option to use when validating web input, and is the default.

Defines if the validation should be done strict. This option defaults to `FALSE`.

Simple array validation

The simplest way, is just to give the array which should be searched against at initiation:

```
$validator = new Zend\Validator\InArray(array('value1', 'value2',...'valueN'));
if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

This will behave exactly like *PHP*'s `in_array()` method.

Note: Per default this validation is not strict nor can it validate multidimensional arrays.

Alternatively, you can define the array to validate against after object construction by using the `setHaystack()` method. `getHaystack()` returns the actual set haystack array.

```
$validator = new Zend\Validator\InArray();
$validator->setHaystack(array('value1', 'value2',...'valueN'));

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

Array validation modes

As previously mentioned, there are possible security issues when using the default non-strict comparison mode, so rather than restricting the developer, we've chosen to offer both strict and non-strict comparisons and adding a safer middle-ground.

It's possible to set the strict mode at initialisation and afterwards with the `setStrict` method. `InArray::COMPARE_STRICT` equates to `true` and `InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_VULNERABILITY` equates to false.

```
// defaults to InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_VULNERABILITY
$validator = new Zend\Validator\InArray(
    array(
         'haystack' => array('value1', 'value2',...'valueN'),
    )
);

// set strict mode
$validator = new Zend\Validator\InArray(
    array(
         'haystack' => array('value1', 'value2',...'valueN'),
         'strict'   => InArray::COMPARE_STRICT  // equates to ``true``
    )
```

```
);

// set non-strict mode
$validator = new Zend\Validator\InArray(
    array(
         'haystack' => array('value1', 'value2',...'valueN'),
         'strict'   => InArray:COMPARE_NOT_STRICT  // equates to ``false``
    )
);

// or

$validator->setStrict(InArray::COMPARE_STRICT);
$validator->setStrict(InArray::COMPARE_NOT_STRICT);
$validator->setStrict(InArray::COMPARE_NOT_STRICT_AND_PREVENT_STR_TO_INT_
↪VULNERABILITY);
```

Note: Note that the **strict** setting is per default `FALSE`.

Recursive array validation

In addition to *PHP*'s `in_array()` method this validator can also be used to validate multidimensional arrays.

To validate multidimensional arrays you have to set the **recursive** option.

```
$validator = new Zend\Validator\InArray(
    array(
        'haystack' => array(
            'firstDimension' => array('value1', 'value2',...'valueN'),
            'secondDimension' => array('foo1', 'foo2',...'fooN')),
        'recursive' => true
    )
);

if ($validator->isValid('value')) {
    // value found
} else {
    // no value found
}
```

Your array will then be validated recursively to see if the given value is contained. Additionally you could use `setRecursive()` to set this option afterwards and `getRecursive()` to retrieve it.

```
$validator = new Zend\Validator\InArray(
    array(
        'firstDimension' => array('value1', 'value2',...'valueN'),
        'secondDimension' => array('foo1', 'foo2',...'fooN')
    )
);

$validator->setRecursive(true);

if ($validator->isValid('value')) {
    // value found
```

```
} else {
    // no value found
}
```

Note: Default setting for recursion

Per default the recursive validation is turned off.

Note: Option keys within the haystack

When you are using the keys 'haystack', 'strict' or 'recursive' within your haystack, then you must wrap the haystack key.

CHAPTER 148

Int

`Zend\Validator\Int` validates if a given value is an integer. Also localized integer values are recognised and can be validated.

Supported options for Zend\Validator\Int

The following options are supported for `Zend\Validator\Int`:

- **locale**: Sets the locale which will be used to validate localized integers.

Simple integer validation

The simplest way to validate an integer is by using the system settings. When no option is used, the environment locale is used for validation:

```
$validator = new Zend\Validator\Int();

$validator->isValid(1234);   // returns true
$validator->isValid(1234.5); // returns false
$validator->isValid('1,234'); // returns true
```

In the above example we expected that our environment is set to "en" as locale. As you can see in the third example also grouping is recognised.

Localized integer validation

Often it's useful to be able to validate also localized values. Integer values are often written different in other countries. For example using english you can write "1234" or "1,234". Both are integer values but the grouping is optional. In german for example you may write "1.234" and in french "1 234".

`Zend\Validator\Int` is able to validate such notations. But it is limited to the locale you set. This means that it not simply strips off the separator, it validates if the correct separator is used. See the following code:

```
$validator = new Zend\Validator\Int(array('locale' => 'de'));

$validator->isValid(1234); // returns true
$validator->isValid("1,234"); // returns false
$validator->isValid("1.234"); // returns true
```

As you can see, by using a locale, your input is validated localized. Using the english notation you get a `FALSE` when the locale forces a different notation.

The locale can also be set afterwards by using `setLocale()` and retrieved by using `getLocale()`.

CHAPTER 149

Ip

`Zend\Validator\Ip` allows you to validate if a given value is an IP address. It supports the IPv4, IPv6 and IPvFeature definitions.

Supported options for Zend\Validator\Ip

The following options are supported for `Zend\Validator\Ip`:

- **allowipv4**: Defines if the validator allows IPv4 addresses. This option defaults to `TRUE`.
- **allowipv6**: Defines if the validator allows IPv6 addresses. This option defaults to `TRUE`.
- **allowipvfuture**: Defines if the validator allows IPvFuture addresses. This option defaults to `false`.
- **allowliteral**: Defines if the validator allows IPv6 or IPvFuture with URI literal style (the IP surrounded by brackets). This option defaults to `true`.

Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\Ip();
if ($validator->isValid($ip)) {
    // ip appears to be valid
} else {
    // ip is invalid; print the reasons
}
```

Note: Invalid IP addresses

Keep in mind that `Zend\Validator\Ip` only validates IP addresses. Addresses like '`mydomain.com`' or '`192.168.50.1/index.html`' are no valid IP addresses. They are either hostnames or valid *URL*s but not IP addresses.

Note: IPv6/IPvFuture validation

`Zend\Validator\Ip` validates IPv6/IPvFuture addresses with regex. The reason is that the filters and methods from *PHP* itself don't follow the *RFC*. Many other available classes also don't follow it.

Validate IPv4 or IPV6 alone

Sometimes it's useful to validate only one of the supported formats. For example when your network only supports IPv4. In this case it would be useless to allow IPv6 within this validator.

To limit `Zend\Validator\Ip` to one protocol you can set the options `allowipv4` or `allowipv6` to `FALSE`. You can do this either by giving the option to the constructor or by using `setOptions()` afterwards.

```
$validator = new Zend\Validator\Ip(array('allowipv6' => false));
if ($validator->isValid($ip)) {
    // ip appears to be valid ipv4 address
} else {
    // ip is no ipv4 address
}
```

Note: Default behaviour

The default behaviour which `Zend\Validator\Ip` follows is to allow both standards.

CHAPTER 150

Isbn

`Zend\Validator\Isbn` allows you to validate an *ISBN-10* or *ISBN-13* value.

Supported options for Zend\Validator\Isbn

The following options are supported for `Zend\Validator\Isbn`:

- **separator**: Defines the allowed separator for the *ISBN* number. It defaults to an empty string.
- **type**: Defines the allowed type of *ISBN* numbers. It defaults to `Zend\Validator\Isbn::AUTO`. For details take a look at *this section*.

Basic usage

A basic example of usage is below:

```
$validator = new Zend\Validator\Isbn();
if ($validator->isValid($isbn)) {
    // isbn is valid
} else {
    // isbn is not valid
}
```

This will validate any *ISBN-10* and *ISBN-13* without separator.

Setting an explicit ISBN validation type

An example of an *ISBN* type restriction is below:

```php
$validator = new Zend\Validator\Isbn();
$validator->setType(Zend\Validator\Isbn::ISBN13);
// OR
$validator = new Zend\Validator\Isbn(array(
    'type' => Zend\Validator\Isbn::ISBN13,
));

if ($validator->isValid($isbn)) {
    // this is a valid ISBN-13 value
} else {
    // this is an invalid ISBN-13 value
}
```

The above will validate only *ISBN-13* values.

Valid types include:

- `Zend\Validator\Isbn::AUTO` (default)
- `Zend\Validator\Isbn::ISBN10`
- `Zend\Validator\Isbn::ISBN13`

Specifying a separator restriction

An example of separator restriction is below:

```php
$validator = new Zend\Validator\Isbn();
$validator->setSeparator('-');
// OR
$validator = new Zend\Validator\Isbn(array(
    'separator' => '-',
));

if ($validator->isValid($isbn)) {
    // this is a valid ISBN with separator
} else {
    // this is an invalid ISBN with separator
}
```

Note: Values without separator

This will return `FALSE` if `$isbn` doesn't contain a separator **or** if it's an invalid *ISBN* value.

Valid separators include:

- "" (empty) (default)
- "-" (hyphen)
- " " (space)

CHAPTER 151

LessThan

`Zend\Validator\LessThan` allows you to validate if a given value is less than a maximum border value.

Note: ZendValidatorLessThan supports only number validation

It should be noted that `Zend\Validator\LessThan` supports only the validation of numbers. Strings or dates can not be validated with this validator.

Supported options for Zend\Validator\LessThan

The following options are supported for `Zend\Validator\LessThan`:

- **inclusive**: Defines if the validation is inclusive the maximum border value or exclusive. It defaults to `FALSE`.
- **max**: Sets the maximum allowed value.

Basic usage

To validate if a given value is less than a defined border simply use the following example.

```
$valid  = new Zend\Validator\LessThan(array('max' => 10));
$value  = 12;
$return = $valid->isValid($value);
// returns false
```

The above example returns `TRUE` for all values which are lower than 10.

Validation inclusive the border value

Sometimes it is useful to validate a value by including the border value. See the following example:

```
$valid  = new Zend\Validator\LessThan(
    array(
        'max' => 10,
        'inclusive' => true
    )
);
$value  = 10;
$result = $valid->isValid($value);
// returns true
```

The example is almost equal to our first example but we included the border value. Now the value '10' is allowed and will return `TRUE`.

CHAPTER 152

NotEmpty

This validator allows you to validate if a given value is not empty. This is often useful when working with form elements or other user input, where you can use it to ensure required elements have values associated with them.

Supported options for Zend\Validator\NotEmpty

The following options are supported for `Zend\Validator\NotEmpty`:

- **type**: Sets the type of validation which will be processed. For details take a look into *this section*.

Default behaviour for Zend\Validator\NotEmpty

By default, this validator works differently than you would expect when you've worked with *PHP*'s `empty()` function. In particular, this validator will evaluate both the integer **0** and string '**0**' as empty.

```
$valid = new Zend\Validator\NotEmpty();
$value  = '';
$result = $valid->isValid($value);
// returns false
```

Note: Default behaviour differs from PHP

Without providing configuration, `Zend\Validator\NotEmpty`'s behaviour differs from *PHP*.

Changing behaviour for Zend\Validator\NotEmpty

Some projects have differing opinions of what is considered an "empty" value: a string with only whitespace might be considered empty, or **0** may be considered non-empty (particularly for boolean sequences). To accommodate differing

needs, Zend\Validator\NotEmpty allows you to configure which types should be validated as empty and which not.

The following types can be handled:

- **boolean**: Returns FALSE when the boolean value is FALSE.
- **integer**: Returns FALSE when an integer **0** value is given. Per default this validation is not activated and returns TRUE on any integer values.
- **float**: Returns FALSE when an float **0.0** value is given. Per default this validation is not activated and returns TRUE on any float values.
- **string**: Returns FALSE when an empty string **''** is given.
- **zero**: Returns FALSE when the single character zero (**'0'**) is given.
- **empty_array**: Returns FALSE when an empty **array** is given.
- **null**: Returns FALSE when an NULL value is given.
- **php**: Returns FALSE on the same reasons where *PHP* method empty() would return TRUE.
- **space**: Returns FALSE when an string is given which contains only whitespaces.
- **object**: Returns TRUE. FALSE will be returned when object is not allowed but an object is given.
- **object_string**: Returns FALSE when an object is given and it's __toString() method returns an empty string.
- **object_count**: Returns FALSE when an object is given, it has an Countable interface and it's count is 0.
- **all**: Returns FALSE on all above types.

All other given values will return TRUE per default.

There are several ways to select which of the above types are validated. You can give one or multiple types and add them, you can give an array, you can use constants, or you can give a textual string. See the following examples:

```
// Returns false on 0
$validator = new Zend\Validator\NotEmpty(Zend\Validator\NotEmpty::INTEGER);

// Returns false on 0 or '0'
$validator = new Zend\Validator\NotEmpty(
    Zend\Validator\NotEmpty::INTEGER + Zend\Validator\NotEmpty::ZERO
);

// Returns false on 0 or '0'
$validator = new Zend\Validator\NotEmpty(array(
    Zend\Validator\NotEmpty::INTEGER,
    Zend\Validator\NotEmpty::ZERO
));

// Returns false on 0 or '0'
$validator = new Zend\Validator\NotEmpty(array(
    'integer',
    'zero',
));
```

You can also provide an instance of Traversable to set the desired types. To set types after instantiation, use the setType() method.

CHAPTER 153

PostCode

`Zend\Validator\PostCode` allows you to determine if a given value is a valid postal code. Postal codes are specific to cities, and in some locales termed *ZIP* codes.

`Zend\Validator\PostCode` knows more than 160 different postal code formats. To select the correct format there are 2 ways. You can either use a fully qualified locale or you can set your own format manually.

Using a locale is more convenient as Zend Framework already knows the appropriate postal code format for each locale; however, you need to use the fully qualified locale (one containing a region specifier) to do so. For instance, the locale "de" is a locale but could not be used with `Zend\Validator\PostCode` as it does not include the region; "de_AT", however, would be a valid locale, as it specifies the region code ("AT", for Austria).

```
$validator = new Zend\Validator\PostCode('de_AT');
```

When you don't set a locale yourself, then `Zend\Validator\PostCode` will use the application wide set locale, or, when there is none, the locale returned by `Locale`.

```
// application wide locale within your bootstrap
Locale::setDefault('de_AT');

$validator = new Zend\Validator\PostCode();
```

You can also change the locale afterwards by calling `setLocale()`. And of course you can get the actual used locale by calling `getLocale()`.

```
$validator = new Zend\Validator\PostCode('de_AT');
$validator->setLocale('en_GB');
```

Postal code formats are simply regular expression strings. When the international postal code format, which is used by setting the locale, does not fit your needs, then you can also manually set a format by calling `setFormat()`.

```
$validator = new Zend\Validator\PostCode('de_AT');
$validator->setFormat('AT-\d{5}');
```

Note: Conventions for self defined formats

When using self defined formats you should omit the starting (`'/^'`) and ending tags (`'$/'`). They are attached automatically.

You should also be aware that postcode values are always be validated in a strict way. This means that they have to be written standalone without additional characters when they are not covered by the format.

Constructor options

At it's most basic, you may pass a string representing a fully qualified locale to the constructor of `Zend\Validator\PostCode`.

```
$validator = new Zend\Validator\PostCode('de_AT');
$validator = new Zend\Validator\PostCode($locale);
```

Additionally, you may pass either an array or a `Traversable` instance to the constructor. When you do so, you must include either the key "locale" or "format"; these will be used to set the appropriate values in the validator object.

```
$validator = new Zend\Validator\PostCode(array(
    'locale' => 'de_AT',
    'format' => 'AT_\d+'
));
```

Supported options for Zend\Validator\PostCode

The following options are supported for `Zend\Validator\PostCode`:

- **format**: Sets a postcode format which will be used for validation of the input.
- **locale**: Sets a locale from which the postcode will be taken from.

CHAPTER 154

Regex

This validator allows you to validate if a given string conforms a defined regular expression.

Supported options for Zend\Validator\Regex

The following options are supported for `Zend\Validator\Regex`:

- **pattern**: Sets the regular expression pattern for this validator.

Validation with Zend\Validator\Regex

Validation with regular expressions allows to have complicated validations being done without writing a own validator. The usage of regular expression is quite common and simple. Let's look at some examples:

```
$validator = new Zend\Validator\Regex(array('pattern' => '/^Test/');

$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns true
$validator->isValid("Pest"); // returns false
```

As you can see, the pattern has to be given using the same syntax as for `preg_match()`. For details about regular expressions take a look into PHP's manual about PCRE pattern syntax.

Pattern handling

It is also possible to set a different pattern afterwards by using `setPattern()` and to get the actual set pattern with `getPattern()`.

```php
$validator = new Zend\Validator\Regex(array('pattern' => '/^Test/');
$validator->setPattern('ing$/');

$validator->isValid("Test"); // returns false
$validator->isValid("Testing"); // returns true
$validator->isValid("Pest"); // returns false
```

CHAPTER 155

Sitemap Validators

The following validators conform to the Sitemap XML protocol.

Sitemap\Changefreq

Validates whether a string is valid for using as a 'changefreq' element in a Sitemap *XML* document. Valid values are: 'always', 'hourly', 'daily', 'weekly', 'monthly', 'yearly', or 'never'.

Returns `TRUE` if and only if the value is a string and is equal to one of the frequencies specified above.

Sitemap\Lastmod

Validates whether a string is valid for using as a 'lastmod' element in a Sitemap *XML* document. The lastmod element should contain a *W3C* date string, optionally discarding information about time.

Returns `TRUE` if and only if the given value is a string and is valid according to the protocol.

Sitemap Lastmod Validator

```
$validator = new Zend\Validator\Sitemap\Lastmod();

$validator->isValid('1999-11-11T22:23:52-02:00'); // true
$validator->isValid('2008-05-12T00:42:52+02:00'); // true
$validator->isValid('1999-11-11'); // true
$validator->isValid('2008-05-12'); // true

$validator->isValid('1999-11-11t22:23:52-02:00'); // false
$validator->isValid('2008-05-12T00:42:60+02:00'); // false
$validator->isValid('1999-13-11'); // false
```

```
$validator->isValid('2008-05-32'); // false
$validator->isValid('yesterday'); // false
```

Sitemap\Loc

Validates whether a string is valid for using as a 'loc' element in a Sitemap *XML* document. This uses `Zend\Uri\Uri::isValid()` internally. Read more at URI Validation.

Sitemap\Priority

Validates whether a value is valid for using as a 'priority' element in a Sitemap *XML* document. The value should be a decimal between 0.0 and 1.0. This validator accepts both numeric values and string values.

Sitemap Priority Validator

```
$validator = new Zend\Validator\Sitemap\Priority();

$validator->isValid('0.1'); // true
$validator->isValid('0.789'); // true
$validator->isValid(0.8); // true
$validator->isValid(1.0); // true

$validator->isValid('1.1'); // false
$validator->isValid('-0.4'); // false
$validator->isValid(1.00001); // false
$validator->isValid(0xFF); // false
$validator->isValid('foo'); // false
```

Supported options for Zend\Validator\Sitemap_*

There are no supported options for any of the Sitemap validators.

CHAPTER 156

Step

`Zend\Validator\Step` allows you to validate if a given value is a valid step value. This validator requires the value to be a numeric value (either string, int or float).

Supported options for Zend\Validator\Step

The following options are supported for `Zend\Validator\Step`:

- **baseValue**: This is the base value from which the step should be computed. This option defaults to `0`
- **step**: This is the step value. This option defaults to `1`

Basic usage

A basic example is the following one:

```
$validator = new Zend\Validator\Step();
if ($validator->isValid(1)) {
// value is a valid step value
} else {
// false
}
```

Using floating-point values

This validator also supports floating-point base value and step value. Here is a basic example of this feature:

```
$validator = new Zend\Validator\Step(array(
'baseValue' => 1.1,
'step' => 2.2
```

```
));

echo $validator->isValid(1.1); // prints true
echo $validator->isValid(3.3); // prints true
echo $validator->isValid(3.35); // prints false
echo $validator->isValid(2.2); // prints false
```

CHAPTER 157

StringLength

This validator allows you to validate if a given string is between a defined length.

Note: ZendValidatorStringLength supports only string validation

It should be noted that `Zend\Validator\StringLength` supports only the validation of strings. Integers, floats, dates or objects can not be validated with this validator.

Supported options for Zend\Validator\StringLength

The following options are supported for `Zend\Validator\StringLength`:

- **encoding**: Sets the `ICONV` encoding which has to be used for this string.
- **min**: Sets the minimum allowed length for a string.
- **max**: Sets the maximum allowed length for a string.

Default behaviour for Zend\Validator\StringLength

Per default this validator checks if a value is between `min` and `max`. But for `min` the default value is **0** and for `max` it is **NULL** which means unlimited.

So per default, without giving any options, this validator only checks if the input is a string.

Limiting the maximum allowed length of a string

To limit the maximum allowed length of a string you need to set the `max` property. It accepts an integer value as input.

```
$validator = new Zend\Validator\StringLength(array('max' => 6));

$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns false
```

You can set the maximum allowed length also afterwards by using the `setMax()` method. And `getMax()` to retrieve the actual maximum border.

```
$validator = new Zend\Validator\StringLength();
$validator->setMax(6);

$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns false
```

Limiting the minimal required length of a string

To limit the minimal required length of a string you need to set the `min` property. It accepts also an integer value as input.

```
$validator = new Zend\Validator\StringLength(array('min' => 5));

$validator->isValid("Test"); // returns false
$validator->isValid("Testing"); // returns true
```

You can set the minimal requested length also afterwards by using the `setMin()` method. And `getMin()` to retrieve the actual minimum border.

```
$validator = new Zend\Validator\StringLength();
$validator->setMin(5);

$validator->isValid("Test"); // returns false
$validator->isValid("Testing"); // returns true
```

Limiting a string on both sides

Sometimes it is required to get a string which has a maximal defined length but which is also minimal chars long. For example when you have a textbox where a user can enter his name, then you may want to limit the name to maximum 30 chars but want to get sure that he entered his name. So you limit the minimum required length to 3 chars. See the following example:

```
$validator = new Zend\Validator\StringLength(array('min' => 3, 'max' => 30));

$validator->isValid("."); // returns false
$validator->isValid("Test"); // returns true
$validator->isValid("Testing"); // returns true
```

Note: Setting a lower maximum border than the minimum border

When you try to set a lower maximum value as the actual minimum value, or a higher minimum value as the actual maximum value, then an exception will be raised.

Encoding of values

Strings are always using a encoding. Even when you don't set the encoding explicit, *PHP* uses one. When your application is using a different encoding than *PHP* itself then you should set an encoding yourself.

You can set your own encoding at initiation with the `encoding` option, or by using the `setEncoding()` method. We assume that your installation uses *ISO* and your application it set to *ISO*. In this case you will see the below behaviour.

```
$validator = new Zend\Validator\StringLength(
    array('min' => 6)
);
$validator->isValid("Ärger"); // returns false

$validator->setEncoding("UTF-8");
$validator->isValid("Ärger"); // returns true

$validator2 = new Zend\Validator\StringLength(
    array('min' => 6, 'encoding' => 'UTF-8')
);
$validator2->isValid("Ärger"); // returns true
```

So when your installation and your application are using different encodings, then you should always set an encoding yourself.

CHAPTER 158

Validator Chains

Often multiple validations should be applied to some value in a particular order. The following code demonstrates a way to solve the example from the *introduction*, where a username must be between 6 and 12 alphanumeric characters:

```
// Create a validator chain and add validators to it
$validatorChain = new Zend\Validator\ValidatorChain();
$validatorChain->addValidator(
                    new Zend\Validator\StringLength(array('min' => 6,
                                                          'max' => 12)))
               ->addValidator(new Zend\Validator\Alnum());

// Validate the username
if ($validatorChain->isValid($username)) {
    // username passed validation
} else {
    // username failed validation; print reasons
    foreach ($validatorChain->getMessages() as $message) {
        echo "$message\n";
    }
}
```

Validators are run in the order they were added to `Zend\Validator\ValidatorChain`. In the above example, the username is first checked to ensure that its length is between 6 and 12 characters, and then it is checked to ensure that it contains only alphanumeric characters. The second validation, for alphanumeric characters, is performed regardless of whether the first validation, for length between 6 and 12 characters, succeeds. This means that if both validations fail, `getMessages()` will return failure messages from both validators.

In some cases it makes sense to have a validator break the chain if its validation process fails. `Zend\Validator\ValidatorChain` supports such use cases with the second parameter to the `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\Validator\StringLength(array('min' => 6,
                                                          'max' => 12)),
                    true)
                ->addValidator(new Zend\Validator\Alnum());
```

Any object that implements `Zend\Validator\ValidatorInterface` may be used in a validator chain.

CHAPTER 159

Writing Validators

`Zend\Validator\AbstractValidator` supplies a set of commonly needed validators, but inevitably, developers will wish to write custom validators for their particular needs. The task of writing a custom validator is described in this section.

`Zend\Validator\ValidatorInterface` defines two methods, `isValid()` and `getMessages()`, that may be implemented by user classes in order to create custom validation objects. An object that implements `Zend\Validator\AbstractValidator` interface may be added to a validator chain with `Zend\Validator\ValidatorChain::addValidator()`. Such objects may also be used with Zend\Filter\Input.

As you may already have inferred from the above description of `Zend\Validator\ValidatorInterface`, validation classes provided with Zend Framework return a boolean value for whether or not a value validates successfully. They also provide information about **why** a value failed validation. The availability of the reasons for validation failures may be valuable to an application for various purposes, such as providing statistics for usability analysis.

Basic validation failure message functionality is implemented in `Zend\Validator\AbstractValidator`. To include this functionality when creating a validation class, simply extend `Zend\Validator\AbstractValidator`. In the extending class you would implement the `isValid()` method logic and define the message variables and message templates that correspond to the types of validation failures that can occur. If a value fails your validation tests, then `isValid()` should return `FALSE`. If the value passes your validation tests, then `isValid()` should return `TRUE`.

In general, the `isValid()` method should not throw any exceptions, except where it is impossible to determine whether or not the input value is valid. A few examples of reasonable cases for throwing an exception might be if a file cannot be opened, an *LDAP* server could not be contacted, or a database connection is unavailable, where such a thing may be required for validation success or failure to be determined.

Creating a Simple Validation Class

The following example demonstrates how a very simple custom validator might be written. In this case the validation rules are simply that the input value must be a floating point value.

```php
class MyValid\Float extends Zend\Validator\AbstractValidator
{
    const FLOAT = 'float';

    protected $messageTemplates = array(
        self::FLOAT => "'%value%' is not a floating point value"
    );

    public function isValid($value)
    {
        $this->setValue($value);

        if (!is_float($value)) {
            $this->error(self::FLOAT);
            return false;
        }

        return true;
    }
}
```

The class defines a template for its single validation failure message, which includes the built-in magic parameter, **%value%**. The call to `setValue()` prepares the object to insert the tested value into the failure message automatically, should the value fail validation. The call to `error()` tracks a reason for validation failure. Since this class only defines one failure message, it is not necessary to provide `error()` with the name of the failure message template.

Writing a Validation Class having Dependent Conditions

The following example demonstrates a more complex set of validation rules, where it is required that the input value be numeric and within the range of minimum and maximum boundary values. An input value would fail validation for exactly one of the following reasons:

- The input value is not numeric.
- The input value is less than the minimum allowed value.
- The input value is more than the maximum allowed value.

These validation failure reasons are then translated to definitions in the class:

```php
class MyValid\NumericBetween extends Zend\Validator\AbstractValidator
{
    const MSG_NUMERIC = 'msgNumeric';
    const MSG_MINIMUM = 'msgMinimum';
    const MSG_MAXIMUM = 'msgMaximum';

    public $minimum = 0;
    public $maximum = 100;

    protected $messageVariables = array(
        'min' => 'minimum',
        'max' => 'maximum'
    );

    protected $messageTemplates = array(
        self::MSG_NUMERIC => "'%value%' is not numeric",
        self::MSG_MINIMUM => "'%value%' must be at least '%min%'",
```

```
        self::MSG_MAXIMUM => "'%value%' must be no more than '%max%'"
    );

    public function isValid($value)
    {
        $this->setValue($value);

        if (!is_numeric($value)) {
            $this->error(self::MSG_NUMERIC);
            return false;
        }

        if ($value < $this->minimum) {
            $this->error(self::MSG_MINIMUM);
            return false;
        }

        if ($value > $this->maximum) {
            $this->error(self::MSG_MAXIMUM);
            return false;
        }

        return true;
    }
}
```

The public properties `$minimum` and `$maximum` have been established to provide the minimum and maximum boundaries, respectively, for a value to successfully validate. The class also defines two message variables that correspond to the public properties and allow `min` and `max` to be used in message templates as magic parameters, just as with `value`.

Note that if any one of the validation checks in `isValid()` fails, an appropriate failure message is prepared, and the method immediately returns `FALSE`. These validation rules are therefore sequentially dependent. That is, if one test should fail, there is no need to test any subsequent validation rules. This need not be the case, however. The following example illustrates how to write a class having independent validation rules, where the validation object may return multiple reasons why a particular validation attempt failed.

Validation with Independent Conditions, Multiple Reasons for Failure

Consider writing a validation class for password strength enforcement - when a user is required to choose a password that meets certain criteria for helping secure user accounts. Let us assume that the password security criteria enforce that the password:

- is at least 8 characters in length,
- contains at least one uppercase letter,
- contains at least one lowercase letter,
- and contains at least one digit character.

The following class implements these validation criteria:

```
class MyValid\PasswordStrength extends Zend\Validator\AbstractValidator
{
    const LENGTH = 'length';
    const UPPER  = 'upper';
    const LOWER  = 'lower';
```

```
    const DIGIT  = 'digit';

    protected $messageTemplates = array(
        self::LENGTH => "'%value%' must be at least 8 characters in length",
        self::UPPER  => "'%value%' must contain at least one uppercase letter",
        self::LOWER  => "'%value%' must contain at least one lowercase letter",
        self::DIGIT  => "'%value%' must contain at least one digit character"
    );

    public function isValid($value)
    {
        $this->setValue($value);

        $isValid = true;

        if (strlen($value) < 8) {
            $this->error(self::LENGTH);
            $isValid = false;
        }

        if (!preg_match('/[A-Z]/', $value)) {
            $this->error(self::UPPER);
            $isValid = false;
        }

        if (!preg_match('/[a-z]/', $value)) {
            $this->error(self::LOWER);
            $isValid = false;
        }

        if (!preg_match('/\d/', $value)) {
            $this->error(self::DIGIT);
            $isValid = false;
        }

        return $isValid;
    }
}
```

Note that the four criteria tests in `isValid()` do not immediately return `FALSE`. This allows the validation class to provide **all** of the reasons that the input password failed to meet the validation requirements. if, for example, a user were to input the string "#$%" as a password, `isValid()` would cause all four validation failure messages to be returned by a subsequent call to `getMessages()`.

CHAPTER 160

Validation Messages

Each validator which is based on `Zend\Validator\ValidatorInterface` provides one or multiple messages in the case of a failed validation. You can use this information to set your own messages, or to translate existing messages which a validator could return to something different.

These validation messages are constants which can be found at top of each validator class. Let's look into `Zend\Validator\GreaterThan` for an descriptive example:

```
protected $messageTemplates = array(
    self::NOT_GREATER => "'%value%' is not greater than '%min%'",
);
```

As you can see the constant `self::NOT_GREATER` refers to the failure and is used as key, and the message itself is used as value of the message array.

You can retrieve all message templates from a validator by using the `getMessageTemplates()` method. It returns you the above array which contains all messages a validator could return in the case of a failed validation.

```
$validator = new Zend\Validator\GreaterThan();
$messages  = $validator->getMessageTemplates();
```

Using the `setMessage()` method you can set another message to be returned in case of the specified failure.

```
$validator = new Zend\Validator\GreaterThan();
$validator->setMessage(
    'Please enter a lower value',
    Zend\Validator\GreaterThan::NOT_GREATER
);
```

The second parameter defines the failure which will be overridden. When you omit this parameter, then the given message will be set for all possible failures of this validator.

Using pre-translated validation messages

Zend Framework is shipped with more than 45 different validators with more than 200 failure messages. It can be a tedious task to translate all of these messages. But for your convenience Zend Framework comes with already pre-translated validation messages. You can find them within the path `/resources/languages` in your Zend Framework installation.

Note: Used path

The resource files are outside of the library path because all of your translations should also be outside of this path.

So to translate all validation messages to German for example, all you have to do is to attach a translator to `Zend\Validator\AbstractValidator` using these resource files.

```
$translator = new Zend\I18n\Translator\Translator();
$translator->addTranslationFile(
    'phpArray'
    'resources/languages/en.php',
    'default',
    'en_US
);
Zend\Validator\AbstractValidator::setDefaultTranslator($translator);
```

Note: Supported languages

This feature is very young, so the amount of supported languages may not be complete. New languages will be added with each release. Additionally feel free to use the existing resource files to make your own translations.

You could also use these resource files to rewrite existing translations. So you are not in need to create these files manually yourself.

Limit the size of a validation message

Sometimes it is necessary to limit the maximum size a validation message can have. For example when your view allows a maximum size of 100 chars to be rendered on one line. To simplify the usage, `Zend\Validator\AbstractValidator` is able to automatically limit the maximum returned size of a validation message.

To get the actual set size use `Zend\Validator\AbstractValidator::getMessageLength()`. If it is -1, then the returned message will not be truncated. This is default behaviour.

To limit the returned message size use `Zend\Validator\AbstractValidator::setMessageLength()`. Set it to any integer size you need. When the returned message exceeds the set size, then the message will be truncated and the string '**...**' will be added instead of the rest of the message.

```
Zend\Validator\AbstractValidator::setMessageLength(100);
```

Note: Where is this parameter used?

The set message length is used for all validators, even for self defined ones, as long as they extend `Zend\Validator\AbstractValidator`.

CHAPTER 161

Zend\View Quick Start

Overview

`Zend\View` provides the "View" layer of Zend Framework's MVC system. It is a multi-tiered system allowing a variety of mechanisms for extension, substitution, and more.

The components of the view layer are as follows:

- **Variables containers**, which hold variables and callbacks that you wish to represent in the view. Often-times, a Variables container will also provide mechanisms for context-specific escaping of variables and more.
- **View Models**, which hold Variables containers, specify the template to use, if any, and optionally provide rendering options (more on that below). View Models may be nested in order to represent complex structures.
- **Renderers**, which take View Models and provide a representation of them to return. Zend Framework ships three renderers by default: a "PHP" renderer which utilizes PHP templates in order to generate markup; a JSON renderer; and a Feed renderer, capable of generating RSS and Atom feeds.
- **Resolvers**, which resolve a template name to a resource a Renderer may consume. As an example, a resolver may take the name "blog/entry" and resolve it to a PHP view script.
- **The View**, which consists of strategies that map the current Request to a Renderer, and strategies for injecting the Response with the result of rendering.
- **Renderer and Response Strategies**. Renderer Strategies listen to the "renderer" event of the View, and decide which Renderer should be selected, based on the Request or other criteria. Response strategies are used to inject the Response object with the results of rendering – which may also include taking actions such as setting Content-Type headers.

Additionally, Zend Framework provides integration with the MVC via a number of event listeners in the `Zend\Mvc\View` namespace.

Usage

This manual section is designed to show you typical usage patterns of the view layer when using it within the Zend Framework MVC. The assumptions are that you are using *Dependency Injection*, and that you are using the default default MVC view strategies.

Configuration

The default configuration for the framework will typically work out-of-the-box. However, you will still need to select resolver strategies and configure them, as well as potentially indicate alternate template names for things like the site layout, 404 (not found) pages, and error pages. The code snippets below can be added to your configuration to accomplish this. We recommend adding it to a site-specific module, such as the "Application" module from the framework's "ZendSkeletonApplication", or to one of your autoloaded configurations within the `config/autoload/` directory.

```
return array(
    'di' => array(
        'instance' => array(
        // The above lines will likely already be present; it's the following
        // definitions that you will want to ensure are present within the DI
        // instance configuration.

            // Setup the View layer
            // This sets up an "AggregateResolver", which allows you to have
            // multiple template resolution strategies. We recommend using the
            // TemplateMapResolver as the primary solution, with the
            // TemplatePathStack as a backup.
            'Zend\View\Resolver\AggregateResolver' => array(
                'injections' => array(
                    'Zend\View\Resolver\TemplateMapResolver',
                    'Zend\View\Resolver\TemplatePathStack',
                ),
            ),

            // The TemplateMapResolver allows you to directly map template names
            // to specific templates. The following map would provide locations
            // for a "home" template, as well as for the "site/layout",
            // "site/error", and "site/404" templates, resolving them to view
            // scripts in this module.
            'Zend\View\Resolver\TemplateMapResolver' => array(
                'parameters' => array(
                    'map'  => array(
                        'home'        => __DIR__ . '/../view/home.phtml',
                        'site/layout' => __DIR__ . '/../view/site/layout.phtml',
                        'site/error'  => __DIR__ . '/../view/site/error.phtml',
                        'site/404'    => __DIR__ . '/../view/site/404.phtml',
                    ),
                ),
            ),

            // The TemplatePathStack takes an array of directories. Directories
            // are then searched in LIFO order (it's a stack) for the requested
            // view script. This is a nice solution for rapid application
            // development, but potentially introduces performance expense in
            // production due to the number of stat calls necessary.
            //
            // The following maps adds an entry pointing to the view directory
```

```
            // of the current module. Make sure your keys differ between modules
            // to ensure that they are not overwritten!
            'Zend\View\Resolver\TemplatePathStack' => array(
                'parameters' => array(
                    'paths'  => array(
                        'application' => __DIR__ . '/../view',
                    ),
                ),
            ),

            // We'll now define the PhpRenderer, and inject it with the
            // AggregateResolver we defined earlier. By default, the MVC layer
            // registers a rendering strategy that uses the PhpRenderer.
            'Zend\View\Renderer\PhpRenderer' => array(
                'parameters' => array(
                    'resolver' => 'Zend\View\Resolver\AggregateResolver',
                ),
            ),

            // By default, the MVC's default rendering strategy uses the
            // template name "layout" for the site layout. Let's tell it to use
            // "site/layout" (which we mapped via the TemplateMapResolver,
            // above).
            'Zend\Mvc\View\DefaultRenderingStrategy' => array(
                'parameters' => array(
                    'layoutTemplate' => 'site/layout',
                ),
            ),

            // By default, the MVC registers an "exception strategy", which is
            // triggered when a requested action raises an exception; it creates
            // a custom view model that wraps the exception, and selects a
            // template. This template is "error" by default; let's change it to
            // "site/error" (which we mapped via the TemplateMapResolver,
            // above).
            //
            // Additionally, we'll tell it that we want to display an exception
            // stack trace; you'll likely want to disable this by default.
            'Zend\Mvc\View\ExceptionStrategy' => array(
                'parameters' => array(
                    'displayExceptions' => true,
                    'exceptionTemplate' => 'site/error',
                ),
            ),

            // Another strategy the MVC registers by default is a "route not
            // found" strategy. Basically, this gets triggered if (a) no route
            // matches the current request, (b) the controller specified in the
            // route match cannot be found in the locator, (c) the controller
            // specified in the route match does not implement the
↪DispatchableInterface
            // interface, or (d) if a response from a controller sets the
            // response status to 404.
            //
            // The default template used in such situations is "error", just
            // like the exception strategy. Let's tell it to use the "site/404"
            // template, (which we mapped via the TemplateMapResolver, above).
            //
```

```
            // You can opt in to inject the reason for a 404 situation; see the
            // various Application::ERROR_* constants for a list of values.
            // Additionally, a number of 404 situations derive from exceptions
            // raised during routing or dispatching. You can opt-in to display
            // these.
            'Zend\Mvc\View\RouteNotFoundStrategy' => array(
                'parameters' => array(
                    'displayExceptions'     => true,
                    'displayNotFoundReason' => true,
                    'notFoundTemplate'      => 'site/404',
                ),
            ),
        ),
    ),
);
```

Controllers and View Models

`Zend\View\View` consumes `ViewModels`, passing them to the selected renderer. Where do you create these, though?

The most explicit way is to create them in your controllers and return them.

```
namespace Foo\Controller;

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

class BarController extends AbstractActionController
{
    public function doSomethingAction()
    {
        $view = new ViewModel(array(
            'message' => 'Hello world',
        ));
        $view->setTemplate('bar/do-something');
        return $view;
    }
}
```

This sets a "message" variable in the view model, and sets the template name "bar/do-something". The view model is then returned.

Considering that in most cases, you'll likely have a template name based on the controller and action, and simply be passing some variables, could this be made simpler? Definitely.

The MVC registers a couple of listeners for controllers to automate this. The first will look to see if you returned an associative array from your controller; if so, it will create a view model and inject this associative array as the view variables container; this view model then replaces the MVC event's result. It will also look to see if you returned nothing or null; if so, it will create a view model without any variables attached; this view model also replaces the MVC event's result.

The second listener checks to see if the MVC event result is a view model, and, if so, if it has a template associated with it. If not, it will inspect the controller matched during routing, and, if available, it's "action" parameter in order to create a template name. This will be "controller/action", with the controller and action normalized to lowercase, dash-separated words.

As an example, the controller `Bar\Controller\BazBatController`, with action "doSomethingCrazy", would be mapped to the template `baz-bat/do-something-crazy`.

In practice, that means our previous example could be re-written as follows:

```
namespace Foo\Controller;

use Zend\Mvc\Controller\AbstractActionController;

class BarController extends AbstractActionController
{
    public function doSomethingCrazyAction()
    {
        return array(
            'message' => 'Hello world',
        );
    }
}
```

The above method will likely work for a majority of use cases. When you need to specify a different template, explicitly create and return a view model, and specify the template manually.

The other use case you may have for explicit view models is if you wish to **nest** view models. Use cases include if you want to render templates to include within the main view you return.

As an example, you may want the view from the action to be one primary section that includes both an "article" and a couple of sidebars; one of the sidebars may include content from multiple views as well.

```
namespace Content\Controller;

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

class ArticleController extends AbstractActionController
{
    public function viewAction()
    {
        // get the article from the persistence layer, etc...

        $view = new ViewModel();

        $articleView = new ViewModel(array('article' => $article));
        $articleView->setTemplate('content/article');

        $primarySidebarView = new ViewModel();
        $primarySidebarView->setTemplate('content/main-sidebar');

        $secondarySidebarView = new ViewModel();
        $secondarySidebarView->setTemplate('content/secondary-sidebar');

        $sidebarBlockView = new ViewModel();
        $sidebarBlockView->setTemplate('content/block');

        $secondarySidebarView->addChild($sidebarBlockView, 'block');

        $view->addChild($articleView, 'article')
             ->addChild($primarySidebarView, 'sidebar_primary')
             ->addChild($secondarySidebarView, 'sidebar_secondary');

```

```
        return $view;
    }
}
```

The above will create and return a view model specifying the template "content/article". When the view is rendered, it will render three child views, the `$articleView`, `$primarySidebarView`, and `$secondarySidebarView`; these will be captured to the `$view`'s "article", "sidebar_primary", and "sidebar_secondary" variables, respectively, so that when it renders, you may include that content. Additionally, the `$secondarySidebarView` will include an additional view model, `$sidebarBlockView`, which will be captured to its "block" view variable.

To better visualize this, let's look at what the final content might look like, with comments detailing where each nested view model is injected.

Here are the templates:

```
<?php // "article/view" template ?>
<div class="sixteen columns content">
    <?php echo $this->article ?>

    <?php echo $this->sidebar_primary ?>

    <?php echo $this->sidebar_secondary ?>
</div>

<?php // "content/article" template ?>
    <!-- This is from the $articleView view model, and the "content/article"
         template -->
    <article class="twelve columns">
        <?php echo $this->escapeHtml('article') ?>
    </article>

<?php // "content/main-sidebar template ?>
    <!-- This is from the $primarySidebarView view model, and the
         "content/main-sidebar template -->
    <div class="two columns sidebar">
        sidebar content...
    </div>

<?php // "content/secondary-sidebar template ?>
    <!-- This is from the $secondarySidebarView view model, and the
         "content/secondary-sidebar template -->
    <div class="two columns sidebar">
        <?php echo $this->block ?>
    </div>

<?php // "content/block template ?>
        <!-- This is from the $sidebarBlockView view model, and the
             "content/block template -->
        <div class="block">
            block content...
        </div>
```

And here is the aggregate, generated content:

```
<!-- This is from the $view view model, and the "article/view" template -->
<div class="sixteen columns content">
    <!-- This is from the $articleView view model, and the "content/article"
```

```
         template -->
    <article class="twelve columns">
        Lorem ipsum ....
    </article>

    <!-- This is from the $primarySidebarView view model, and the
         "content/main-sidebar template -->
    <div class="two columns sidebar">
        sidebar content...
    </div>

    <!-- This is from the $secondarySidebarView view model, and the
         "content/secondary-sidebar template -->
    <div class="two columns sidebar">
        <!-- This is from the $sidebarBlockView view model, and the
             "content/block template -->
        <div class="block">
            block content...
        </div>
    </div>
</div>
```

As you can see, you can achieve very complex markup using nested views, while simultaneously keeping the details of rendering isolated from the request/reponse lifecycle of the controller.

Dealing with Layouts

Most sites enforce a cohesive look-and-feel, which we typically call the site "layout". The site layout includes the default stylesheets and JavaScript necessary, if any, as well as the basic markup structure into which all site content will be injected.

Within Zend Framework, layouts are handled via nesting of view models (see the *previous example* for examples of view model nesting). The MVC event composes a View Model which acts as the "root" for nested view models, as such, it should contain the skeleton, or layout, template for the site (configuration refers to this as the "layoutTemplate"). All other content is then rendered and captured to view variables ov this root view model.

The default rendering strategy sets the layout template as "layout". To change this, you can add some configuration for the Dependency Injector.

```
return array(
    'di' => array(
        'instance' => array(
        // The above lines will likely already be present; it's the following
        // definitions that you will want to ensure are present within the DI
        // instance configuration.

            // By default, the MVC's default rendering strategy uses the
            // template name "layout" for the site layout. Let's tell it to use
            // "site/layout" (which we mapped via the TemplateMapResolver,
            // above).
            'Zend\Mvc\View\DefaultRenderingStrategy' => array(
                'parameters' => array(
                    'baseTemplate' => 'site/layout',
                ),
            ),
        ),
```

```
    ),
);
```

A listener on the controllers, `Zend\Mvc\View\InjectViewModelListener`, will take a view model returned from a controller and inject it as a child of the root (layout) view model. By default, view models will capture to the "content" variable of the root view model. This means you can do the following in your layout view script.

```
<html>
    <head>
        <title><?php echo $this->headTitle() ?></title>
    </head>
    <body>
        <?php echo $this->content; ?>
    </body>
</html>
```

If you want to specify a different view variable to which to capture, explicitly create a view model in your controller, and set it's "capture to" value.

```
namespace Foo\Controller;

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

class BarController extends AbstractActionController
{
    public function doSomethingAction()
    {
        $view = new ViewModel(array(
            'message' => 'Hello world',
        ));

        // Capture to the layout view's "article" variable
        $view->setCaptureTo('article');

        return $view;
    }
}
```

There will be times you don't want to render a layout. For example, you might be answering an API call which expects JSON or an XML payload, or you might be answering an XHR request that expects a partial HTML payload. The simplest way to do this is to explicitly create and return a view model from your controller, and mark it as "terminal", which will hint to the MVC listener that normally injects the returned view model into the layout view model to instead replace the layout view model.

```
namespace Foo\Controller;

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

class BarController extends AbstractActionController
{
    public function doSomethingAction()
    {
        $view = new ViewModel(array(
            'message' => 'Hello world',
        ));
```

```
13
14          // Disable layouts; use this view model in the MVC event instead
15          $view->setTerminal(true);
16
17          return $view;
18      }
19  }
```

When discussing controllers and view models, we detailed a nested view model which contained an article and sidebars. Sometimes, you may want to provide additional view models to the layout, instead of nesting in the returned layout. This may be done by using the "layout" controller plugin, which returns the root view model; you can then call the same `addChild()` method on it as we did in that previous example.

```
1   namespace Content\Controller;
2
3   use Zend\Mvc\Controller\AbstractActionController;
4   use Zend\View\Model\ViewModel;
5
6   class ArticleController extends AbstractActionController
7   {
8       public function viewAction()
9       {
10          // get the article from the persistence layer, etc...
11
12          // Get the "layout" view model and inject a sidebar
13          $layout = $this->layout();
14          $sidebarView = new ViewModel();
15          $sidebarView->setTemplate('content/sidebar');
16          $layout->addChild($sidebarView, 'sidebar');
17
18          // Create and return a view model for the retrieved article
19          $view = new ViewModel(array('article' => $article));
20          $view->setTemplate('content/article');
21          return $view;
22      }
23  }
```

You could also use this technique to select a different layout, by simply calling the `setTemplate()` method of the layout view model.

```
1   namespace Content\Controller;
2
3   use Zend\Mvc\Controller\AbstractActionController;
4   use Zend\View\Model\ViewModel;
5
6   class ArticleController extends AbstractActionController
7   {
8       public function viewAction()
9       {
10          // get the article from the persistence layer, etc...
11
12          // Get the "layout" view model and set an alternate template
13          $layout = $this->layout();
14          $layout->setTemplate('article/layout');
15
16          // Create and return a view model for the retrieved article
17          $view = new ViewModel(array('article' => $article));
18          $view->setTemplate('content/article');
```

```
        return $view;
    }
}
```

Sometimes, you may want to access the layout from within your actual view scripts when using the `PhpRenderer`. Reasons might include wanting to change the layout template, or wanting to access or inject layout view variables. Similar to controllers, you can use the "layout" view plugin/helper. If you provide a string argument to it, you will change the template; if you provide no arguments the root layout view model is returned.

```
// Change the layout:
$this->layout('alternate/layout'); // OR
$this->layout()->setTemplate('alternate/layout');

// Access a layout variable.
// Since access to the base view model is relatively easy, it becomes a
// reasonable place to store things such as API keys, which other view scripts
// may need.
$layout       = $this->layout();
$disqusApiKey = false;
if (isset($layout->disqusApiKey)) {
    $disqusApiKey = $layout->disqusApiKey;
}

// Set a layout variable
$this->layout()->footer = $this->render('article/footer');
```

Commonly, you may want to alter the layout based on the module currently selected.

Another frequently requested feature is the ability to change a layout based on the current **module**. This requires (a) detecting if the controller matched in routing belongs to this module, and then (b) changing the template of the view model.

The place to do these actions is in a listener. It should listen either to the "route" event at low (negative) priority, or on the "dispatch" event, at any priority. Typically, you will register this during the bootstrap event.

```
namespace Content;

class Module
{
    public function onBootstrap($e)
    {
        // Register a dispatch event
        $app = $e->getParam('application');
        $app->getEventManager()->attach('dispatch', array($this, 'setLayout'), -100);
    }

    public function setLayout($e)
    {
        $matches    = $e->getRouteMatch();
        $controller = $matches->getParam('controller');
        if (0 !== strpos($controller, __NAMESPACE__, 0)) {
            // not a controller from this module
            return;
        }

        // Set the layout template
        $viewModel = $e->getViewModel();
        $viewModel->setTemplate('content/layout');
```

```
    }
}
```

Creating and Registering Alternate Rendering and Response Strategies

`Zend\View\View` does very little. Its workflow is essentially to martial a `ViewEvent`, and then trigger two events, "renderer" and "response". You can attach "strategies" to these events, using the methods `addRendererStrategy()` and `addResponseStrategy()`, respectively. A "renderer strategy" investigates the Request object (or any other criteria) in order to select a renderer (or fail to select one). A "response strategy" determines how to populate the Response based on the result of rendering.

Zend Framework ships with three rendering/response strategies that you can use within your application.

- `Zend\View\Strategy\PhpRendererStrategy`. This strategy is a "catch-all" in that it will always return the `Zend\View\Renderer\PhpRenderer`, and populate the Response body with the results of rendering.
- `Zend\View\Strategy\JsonStrategy`. This strategy inspects the Accept HTTP header, if present, and determines if the client has indicated it accepts an "application/json" response. If so, it will return the `Zend\View\Renderer\JsonRenderer`, and populate the Response body with the JSON value returned, as well as set a Content-Type header with a value of "application/json".
- `Zend\View\Strategy\FeedStrategy`. This strategy inspects the Accept HTTP header, if present, and determines if the client has indicated it accepts either an "application/rss+xml" or "application/atom+xml" response. If so, it will return the `Zend\View\Renderer\FeedRenderer`, setting the feed type to either "rss" or "atom", based on what was matched. Its Response strategy will populate the Response body with the generated feed, as well as set a Content-Type header with the appropriate value based on feed type.

By default, only the `PhpRendererStrategy` is registered, meaning you will need to register the other strategies yourself if you want to use them. Additionally, it means that you will likely want to register these at higher priority to ensure they match before the `PhpRendererStrategy`. As an example, let's register the `JsonStrategy`.

```
namespace Application;

class Module
{
    public function onBootstrap($e)
    {
        // Register a "render" event, at high priority (so it executes prior
        // to the view attempting to render)
        $app = $e->getParam('application');
        $app->getEventManager()->attach('render', array($this, 'registerJsonStrategy
↪'), 100);
    }

    public function registerJsonStrategy($e)
    {
        $app          = $e->getTarget();
        $locator      = $app->getServiceManager();
        $view         = $locator->get('Zend\View\View');
        $jsonStrategy = $locator->get('Zend\View\Strategy\JsonStrategy');

        // Attach strategy, which is a listener aggregate, at high priority
        $view->getEventManager()->attach($jsonStrategy, 100);
    }
}
```

The above will register the `JsonStrategy` with the "render" event, such that it executes prior to the `PhpRendererStrategy`, and thus ensure that a JSON payload is created when requested.

What if you want this to happen only in specific modules, or specific controllers? One way is similar to the last example in the *previous section on layouts*, where we detailed changing the layout for a specific module.

```
namespace Content;

class Module
{
    public function onBootstrap($e)
    {
        // Register a render event
        $app = $e->getParam('application');
        $app->getEventManager()->attach('render', array($this, 'registerJsonStrategy
↪'), 100);
    }

    public function registerJsonStrategy($e)
    {
        $matches    = $e->getRouteMatch();
        $controller = $matches->getParam('controller');
        if (0 !== strpos($controller, __NAMESPACE__, 0)) {
            // not a controller from this module
            return;
        }

        // Potentially, you could be even more selective at this point, and test
        // for specific controller classes, and even specific actions or request
        // methods.

        // Set the JSON strategy when controllers from this module are selected
        $app          = $e->getTarget();
        $locator      = $app->getServiceManager();
        $view         = $locator->get('Zend\View\View');
        $jsonStrategy = $locator->get('Zend\View\Strategy\JsonStrategy');

        // Attach strategy, which is a listener aggregate, at high priority
        $view->getEventManager()->attach($jsonStrategy, 100);
    }
}
```

While the above examples detail using the JSON strategy, the same could be done for the `FeedStrategy`.

What if you want to use a custom renderer? or if your app might allow a combination of JSON, Atom feeds, and HTML? At this point, you'll need to create your own custom strategies. Below is an example that more appropriately loops through the HTTP Accept header, and selects the appropriate renderer based on what is matched first.

```
namespace Content\View;

use Zend\EventManager\EventCollection;
use Zend\EventManager\ListenerAggregate;
use Zend\Feed\Writer\Feed;
use Zend\View\Renderer\FeedRenderer;
use Zend\View\Renderer\JsonRenderer;
use Zend\View\Renderer\PhpRenderer;

class AcceptStrategy implements ListenerAggregate
{
```

```
12      protected $feedRenderer;
13      protected $jsonRenderer;
14      protected $listeners = array();
15      protected $phpRenderer;
16
17      public function __construct(
18          PhpRenderer $phpRenderer,
19          JsonRenderer $jsonRenderer,
20          FeedRenderer $feedRenderer
21      ) {
22          $this->phpRenderer  = $phpRenderer;
23          $this->jsonRenderer = $jsonRenderer;
24          $this->feedRenderer = $feedRenderer;
25      }
26
27      public function attach(EventCollection $events, $priority = null)
28      {
29          if (null === $priority) {
30              $this->listeners[] = $events->attach('renderer', array($this,
    ↪'selectRenderer'));
31              $this->listeners[] = $events->attach('response', array($this,
    ↪'injectResponse'));
32          } else {
33              $this->listeners[] = $events->attach('renderer', array($this,
    ↪'selectRenderer'), $priority);
34              $this->listeners[] = $events->attach('response', array($this,
    ↪'injectResponse'), $priority);
35          }
36      }
37
38      public function detach(EventCollection $events)
39      {
40          foreach ($this->listeners as $index => $listener) {
41              if ($events->detach($listener)) {
42                  unset($this->listeners[$index]);
43              }
44          }
45      }
46
47      public function selectRenderer($e)
48      {
49          $request = $e->getRequest();
50          $headers = $request->getHeaders();
51
52          // No Accept header? return PhpRenderer
53          if (!$headers->has('accept')) {
54              return $this->phpRenderer;
55          }
56
57          $accept = $headers->get('accept');
58          foreach ($accept->getPrioritized() as $mediaType) {
59              if (0 === strpos($mediaType, 'application/json')) {
60                  return $this->jsonRenderer;
61              }
62              if (0 === strpos($mediaType, 'application/rss+xml')) {
63                  $this->feedRenderer->setFeedType('rss');
64                  return $this->feedRenderer;
65              }
```

```
            if (0 === strpos($mediaType, 'application/atom+xml')) {
                $this->feedRenderer->setFeedType('atom');
                return $this->feedRenderer;
            }
        }

        // Nothing matched; return PhpRenderer. Technically, we should probably
        // return an HTTP 415 Unsupported response.
        return $this->phpRenderer;
    }

    public function injectResponse($e)
    {
        $renderer = $e->getRenderer();
        $response = $e->getResponse();
        $result   = $e->getResult();

        if ($renderer === $this->jsonRenderer) {
            // JSON Renderer; set content-type header
            $headers = $response->getHeaders();
            $headers->addHeaderLine('content-type', 'application/json');
        } elseif ($renderer === $this->feedRenderer) {
            // Feed Renderer; set content-type header, and export the feed if
            // necessary
            $feedType  = $this->feedRenderer->getFeedType();
            $headers   = $response->getHeaders();
            $mediatype = 'application/'
                       . (('rss' == $feedType) ? 'rss' : 'atom')
                       . '+xml';
            $headers->addHeaderLine('content-type', $mediatype);

            // If the $result is a feed, export it
            if ($result instanceof Feed) {
                $result = $result->export($feedType);
            }
        } elseif ($renderer !== $this->phpRenderer) {
            // Not a renderer we support, therefor not our strategy. Return
            return;
        }

        // Inject the content
        $response->setContent($result);
    }
}
```

This strategy would be registered just as we demonstrated registering the `JsonStrategy` earlier. You would also need to define DI configuration to ensure the various renderers are injected when you retrieve the strategy from the application's locator instance.

CHAPTER 162

The PhpRenderer

`Zend\View\Renderer\PhpRenderer`"renders" view scripts written in PHP, capturing and returning the output. It composes Variable containers and/or View Models, a plugin broker for *helpers*, and optional filtering of the captured output.

The `PhpRenderer` is template system agnostic; you may use *PHP* as your template language, or create instances of other template systems and manipulate them within your view script. Anything you can do with PHP is available to you.

Usage

Basic usage consists of instantiating or otherwise obtaining an instance of the `PhpRenderer`, providing it with a resolver which will resolve templates to PHP view scripts, and then calling its `render()` method.

Instantiating a renderer is trivial:

```
1 use Zend\View\Renderer\PhpRenderer;
2
3 $renderer = new PhpRenderer();
```

Zend Framework ships with several types of "resolvers", which are used to resolve a template name to a resource a renderer can consume. The ones we will usually use with the `PhpRenderer` are:

- `Zend\View\Resolver\TemplateMapResolver`, which simply maps template names directly to view scripts.
- `Zend\View\Resolver\TemplatePathStack`, which creates a LIFO stack of script directories in which to search for a view script. By default, it appends the suffix ".phtml" to the requested template name, and then loops through the script directories; if it finds a file matching the requested template, it returns the full file path.
- `Zend\View\Resolver\AggregateResolver`, which allows attaching a FIFO queue of resolvers to consult.

We suggest using the `AggregateResolver`, as it allows you to create a multi-tiered strategy for resolving template names.

Programmatically, you would then do something like this:

```
use Zend\View\Renderer\PhpRenderer;
use Zend\View\Resolver;

$renderer = new PhpRenderer();

$resolver = new Resolver\AggregateResolver();

$map = new Resolver\TemplateMapResolver(array(
    'layout'      => __DIR__ . '/view/layout.phtml',
    'index/index' => __DIR__ . '/view/index/index.phtml',
));
$stack = new Resolver\TemplatePathStack(array(
    __DIR__ . '/view',
    $someOtherPath,
));

$resolver->attach($map)    // this will be consulted first
         ->attach($stack);
```

You can also specify a specific priority value when registering resolvers, with high, positive integers getting higher priority, and low, negative integers getting low priority, when resolving.

In an MVC application, you can configure this via DI quite easily:

```
return array(
    'di' => array(
        'instance' => array(
            'Zend\View\Resolver\AggregateResolver' => array(
                'injections' => array(
                    'Zend\View\Resolver\TemplateMapResolver',
                    'Zend\View\Resolver\TemplatePathStack',
                ),
            ),

            'Zend\View\Resolver\TemplateMapResolver' => array(
                'parameters' => array(
                    'map'  => array(
                        'layout'      => __DIR__ . '/view/layout.phtml',
                        'index/index' => __DIR__ . '/view/index/index.phtml',
                    ),
                ),
            ),
            'Zend\View\Resolver\TemplatePathStack' => array(
                'parameters' => array(
                    'paths'  => array(
                        'application' => __DIR__ . '/view',
                        'elsewhere'   => $someOtherPath,
                    ),
                ),
            ),
            'Zend\View\Renderer\PhpRenderer' => array(
                'parameters' => array(
                    'resolver' => 'Zend\View\Resolver\AggregateResolver',
                ),
            ),
        ),
    ),
```

```
);
```

Now that we have our `PhpRenderer` instance, and it can find templates, let's inject some variables. This can be done in 4 different ways.

- Pass an associative array (or `ArrayAccess` instance, or `Zend\View\Variables` instance) of items as the second argument to `render()`: *$renderer->render($templateName, array('foo' => 'bar'))*
- Assign a `Zend\View\Variables` instance, associative array, or `ArrayAccess` instance to the `setVars()` method.
- Assign variables as instance properties of the renderer: *$renderer->foo = 'bar'*. This essentially proxies to an instance of `Variables` composed internally in the renderer by default.
- Create a ViewModel instance, assign variables to that, and pass the ViewModel to the `render()` method:

```
use Zend\View\Model\ViewModel;
use Zend\View\Renderer\PhpRenderer;

$renderer = new PhpRenderer();

$model    = new ViewModel();
$model->setVariable('foo', 'bar');
// or
$model = new ViewModel(array('foo' => 'bar'));

$model->setTemplate($templateName);
$renderer->render($model);
```

Now, let's render something. As a simple example, let us say you have a list of book data.

```
// use a model to get the data for book authors and titles.
$data = array(
    array(
        'author' => 'Hernando de Soto',
        'title' => 'The Mystery of Capitalism'
    ),
    array(
        'author' => 'Henry Hazlitt',
        'title' => 'Economics in One Lesson'
    ),
    array(
        'author' => 'Milton Friedman',
        'title' => 'Free to Choose'
    )
);

// now assign the book data to a renderer instance
$renderer->books = $data;

// and render the template "booklist"
echo $renderer->render('booklist');
```

More often than not, you'll likely be using the MVC layer. As such, you should be thinking in terms of view models. Let's consider the following code from within an action method of a controller.

```
namespace Bookstore\Controller;

use Zend\Mvc\Controller\AbstractActionController;
```

```php

class BookController extends AbstractActionController
{
    public function listAction()
    {
        // do some work...

        // Assume $data is the list of books from the previous example
        $model = new ViewModel(array('books' => $data));

        // Optionally specify a template; if we don't, by default it will be
        // auto-determined based on the controller name and this action. In
        // this example, the template would resolve to "book/list", and thus
        // the file "book/list.phtml"; the following overrides that to set
        // the template to "booklist", and thus the file "booklist.phtml"
        // (note the lack of directory preceding the filename).
        $model->setTemplate('booklist');

        return $model
    }
}
```

This will then be rendered as if the following were executed:

```php
$renderer->render($model);
```

Now we need the associated view script. At this point, we'll assume that the template "booklist" resolves to the file `booklist.phtml`. This is a *PHP* script like any other, with one exception: it executes inside the scope of the `PhpRenderer` instance, which means that references to `$this` point to the `PhpRenderer` instance properties and methods. Thus, a very basic view script could look like this:

```php
<?php if ($this->books): ?>

    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escapeHtml($val['author']) ?></td>
            <td><?php echo $this->escapeHtml($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>

    </table>

<?php else: ?>

    <p>There are no books to display.</p>

<?php endif;?>
```

Note: Escape Output

The security mantra is "Filter input, escape output." If you are unsure of the source of a given variable – which is likely most of the time – you should escape it based on which HTML context it is being injected into. The primary contexts to be aware of are HTML Body, HTML Attribute, Javascript, CSS and URI. Each context has a dedicated helper available to apply the escaping strategy most appropriate to each context. You should be aware that escaping does vary significantly between contexts - there is no one single escaping strategy that can be globally applied.

In the example above, there are calls to an `escapeHtml()` method. The method is actually a *helper*, a plugin available via method overloading. Additional escape helpers provide the `escapeHtmlAttr()`, `escapeJs()`, `escapeCss()`, and `escapeUrl()` methods for each of the HTML contexts you are most likely to encounter.

By using the provided helpers and being aware of your variables' contexts, you will prevent your templates from running afoul of Cross-Site Scripting (XSS) vulnerabilities.

We've now toured the basic usage of the `PhpRenderer`. By now you should know how to instantiate the renderer, provide it with a resolver, assign variables and/or create view models, create view scripts, and render view scripts.

Options and Configuration

`Zend\View\Renderer\PhpRenderer` utilizes several collaborators in order to do its work. use the following methods to configure the renderer.

broker `setBroker(Zend\View\HelperBroker $broker)`

Set the broker instance used to load, register, and retrieve *helpers*.

resolver `setResolver(Zend\View\Resolver $resolver)`

Set the resolver instance.

filters `setFilterChain(Zend\Filter\FilterChain $filters)`

Set a filter chain to use as an output filter on rendered content.

vars `setVars(array|ArrayAccess|Zend\View\Variables $variables)`

Set the variables to use when rendering a view script/template.

canRenderTrees `setCanRenderTrees(bool $canRenderTrees)`

Set flag indicating whether or not we should render trees of view models. If set to true, the `Zend\View\View` instance will not attempt to render children separately, but instead pass the root view model directly to the `PhpRenderer`. It is then up to the developer to render the children from within the view script. This is typically done using the `RenderChildModel` helper: *$this->renderChildModel('child_name')*.

Additional Methods

Typically, you'll only ever access variables and *helpers* within your view scripts or when interacting with the `PhpRenderer`. However, there are a few additional methods you may be interested in.

render `render(string|Zend\View\Model $nameOrModel, $values = null)`

Render a template/view model.

If `$nameOrModel` is a string, it is assumed to be a template name. That template will be resolved using the current resolver, and then rendered. If `$values` is non-null, those values, and those values only, will be used during rendering, and will replace whatever variable container previously was in the renderer; however, the previous variable container will be reset when done. If `$values` is empty, the current variables container (see *setVars()*) will be injected when rendering.

If `$nameOrModel` is a `Model` instance, the template name will be retrieved from it and used. Additionally, if the model contains any variables, these will be used when rendering; otherwise, the variables container already present, if any, will be used.

resolver `resolver()`

Retrieves the `Resolver` instance.

vars `vars(string $key = null)`

Retrieve the variables container, or a single variable from the container..

plugin `plugin(string $name, array $options = null)`

Get a plugin/helper instance. Proxies to the broker's `load()` method; as such, any `$options` you pass will be passed to the plugin's constructor if this is the first time the plugin has been retrieved. See the section on *helpers* for more information.

addTemplate `addTemplate(string $template)`

Add a template to the stack. When used, the next call to `render()` will loop through all template added using this method, rendering them one by one; the output of the last will be returned.

CHAPTER 163

PhpRenderer View Scripts

Once you call `render()`, `Zend\View\Renderer\PhpRenderer` then `include()`s the requested view script and executes it "inside" the scope of the `PhpRenderer` instance. Therefore, in your view scripts, references to `$this` actually point to the `PhpRenderer` instance itself.

Variables assigned to the view – either via a View Model, Variables container, or simply by passing an array of variables to `render()` – may be retrieved in three ways:

- Explicitly, by retrieving them from the `Variables` container composed in the `PhpRenderer`: *$this->vars()->varname*.
- As instance properties of the `PhpRenderer` instance: *$this->varname*. (In this situation, instance property access is simply proxying to the composed `Variables` instance.)
- As local PHP variables: *$varname*. The `PhpRenderer` extracts the members of the `Variables` container locally.

We generally recommend using the second notation, as it's less verbose than the first, but differentiates between variables in the view script scope and those assigned to the renderer from elsewhere.

By way of reminder, here is the example view script from the `PhpRenderer` introduction.

```
<?php if ($this->books): ?>

    <!-- A table of some books. -->
    <table>
        <tr>
            <th>Author</th>
            <th>Title</th>
        </tr>

        <?php foreach ($this->books as $key => $val): ?>
        <tr>
            <td><?php echo $this->escapeHtml($val['author']) ?></td>
            <td><?php echo $this->escapeHtml($val['title']) ?></td>
        </tr>
        <?php endforeach; ?>
```

```
    </table>

<?php else: ?>

    <p>There are no books to display.</p>

<?php endif;?>
```

Escaping Output

One of the most important tasks to perform in a view script is to make sure that output is escaped properly; among other things, this helps to avoid cross-site scripting attacks. Unless you are using a function, method, or helper that does escaping on its own, you should always escape variables when you output them and pay careful attention to applying the correct escaping strategy to each HTML context you use.

The `PhpRenderer` includes a selection of helpers you can use for this purpose: `EscapeHtml`, `EscapeHtmlAttr` `EscapeJs`, `EscapeCss`, and `EscapeUrl`. Matching the correct helper (or combination of helpers) to the context into which you are injecting untrusted variables will ensure that you are protected against Cross-Site Scripting (XSS) vulnerabilities.

```
// bad view-script practice:
echo $this->variable;

// good view-script practice:
echo $this->escapeHtml($this->variable);

// and remember context is always relevant!
<script type="text/javascript">
    var foo = "<?php echo $this->escapeJs($variable) ?>";
</script>
```

CHAPTER 164

View Helpers

In your view scripts, often it is necessary to perform certain complex functions over and over: e.g., formatting a date, generating form elements, or displaying action links. You can use helper, or plugin, classes to perform these behaviors for you.

A helper is simply a class that implements the interface `Zend\View\Helper`. `Helper` simply defines two methods, `setView()`, which accepts a `Zend\View\Renderer` instance/implementation, and `getView()`, used to retrieve that instance. `Zend\View\PhpRenderer` composes a plugin broker, allowing you to retrieve helpers, and also provides some method overloading capabilities that allow proxying method calls to helpers.

As an example, let's say we have a helper class named `My\Helper\LowerCase`, which we map in our plugin broker to the name "lowercase". We can retrieve or invoke it in one of the following ways:

```
// $view is a PhpRenderer instance

// Via the plugin broker:
$broker = $view->getBroker();
$helper = $broker->load('lowercase');

// Retrieve the helper instance, via the method "plugin",
// which proxies to the plugin broker:
$helper = $view->plugin('lowercase');

// If the helper does not define __invoke(), the following also retrieves it:
$helper = $view->lowercase();

// If the helper DOES define __invoke, you can call the helper
// as if it is a method:
$filtered = $view->lowercase('some value');
```

The last two examples demonstrate how the `PhpRenderer` uses method overloading to retrieve and/or invoke helpers directly, offering a convenience API for end users.

A large number of helpers are provided in the standard distribution of Zend Framework. You can also register helpers by adding them to the plugin broker, or the plugin locator the broker composes. Please refer to the plugin broker documentation for details.

Included Helpers

Zend Framework comes with an initial set of helper classes. In particular, there are helpers for creating route-based *URL*s and *HTML* lists, as well as declaring variables. Additionally, there are a rich set of helpers for providing values for, and rendering, the various HTML <*head*> tags, such as `HeadTitle`, `HeadLink`, and `HeadScript`. The currently shipped helpers include:

- `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.

CHAPTER 165

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.

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">
        <?php echo $this->action('list',
                                 'comment',
                                 null,
                                 array('count' => 10)); ?>
    </div>
</div>
```

CHAPTER 166

BaseUrl Helper

While most *URL*s generated by the framework have the base *URL* prepended automatically, developers will need to prepend the base *URL* to their own *URL*s in order for paths to resources to be correct.

Usage of the BaseUrl helper is very straightforward:

```
/*
 * The following assume that the base URL of the page/application is "/mypage".
 */

/*
 * Prints:
 * <base href="/mypage/" />
 */
<base href="<?php echo $this->baseUrl(); ?>" />

/*
 * Prints:
 * <link rel="stylesheet" type="text/css" href="/mypage/css/base.css" />
 */
<link rel="stylesheet" type="text/css"
     href="<?php echo $this->baseUrl('css/base.css'); ?>" />
```

Note: For simplicity's sake, we strip out the entry *PHP* file (e.g., "`index.php`") from the base *URL* that was contained in `Zend_Controller`. However, in some situations this may cause a problem. If one occurs, use `$this->getHelper('BaseUrl')->setBaseUrl()` to set your own BaseUrl.

CHAPTER 167

Cycle Helper

The `Cycle` helper is used to alternate a set of values.

Cycle Helper Basic Usage

To add elements to cycle just specify them in constructor or use `assign(array $data)` function

```
<?php foreach ($this->books as $book):?>
  <tr style="background-color:<?php echo $this->cycle(array("#F0F0F0",
                                                            "#FFFFFF"))
                                                 ->next()?>">
  <td><?php echo $this->escape($book['author']) ?></td>
</tr>
<?php endforeach;?>

// Moving in backwards order and assign function
$this->cycle()->assign(array("#F0F0F0","#FFFFFF"));
$this->cycle()->prev();
?>
```

The output

```
<tr style="background-color:'#F0F0F0'">
   <td>First</td>
</tr>
<tr style="background-color:'#FFFFFF'">
   <td>Second</td>
</tr>
```

Working with two or more cycles

To use two cycles you have to specify the names of cycles. Just set second parameter in cycle method. `$this->cycle(array("#F0F0F0","#FFFFFF"),'cycle2')`. You can also use setName($name) func-

tion.

```
<?php foreach ($this->books as $book):?>
  <tr style="background-color:<?php echo $this->cycle(array("#F0F0F0",
                                                            "#FFFFFF"))
                                                  ->next()?>">
  <td><?php echo $this->cycle(array(1,2,3),'number')->next()?></td>
  <td><?php echo $this->escape($book['author'])?></td>
</tr>
<?php endforeach;?>
```

CHAPTER 168

Partial Helper

The `Partial` view helper is used to render a specified template within its own variable scope. The primary use is for reusable template fragments with which you do not need to worry about variable name clashes. Additionally, they allow you to specify partial view scripts from specific modules.

A sibling to the `Partial`, the `PartialLoop` view helper allows you to pass iterable data, and render a partial for each item.

Note: PartialLoop Counter

The `PartialLoop` view helper assigns a variable to the view named **partialCounter** which passes the current position of the array to the view script. This provides an easy way to have alternating colors on table rows for example.

Basic Usage of Partials

Basic usage of partials is to render a template fragment in its own view scope. Consider the following partial script:

```
<?php // partial.phtml ?>
<ul>
    <li>From: <?php echo $this->escape($this->from) ?></li>
    <li>Subject: <?php echo $this->escape($this->subject) ?></li>
</ul>
```

You would then call it from your view script using the following:

```
<?php echo $this->partial('partial.phtml', array(
    'from' => 'Team Framework',
    'subject' => 'view partials')); ?>
```

Which would then render:

```
<ul>
    <li>From: Team Framework</li>
```

```
    <li>Subject: view partials</li>
</ul>
```

Note: What is a model?

A model used with the `Partial` view helper can be one of the following:

- **Array**. If an array is passed, it should be associative, as its key/value pairs are assigned to the view with keys as view variables.
- **Object implementing toArray() method**. If an object is passed an has a `toArray()` method, the results of `toArray()` will be assigned to the view object as view variables.
- **Standard object**. Any other object will assign the results of `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_Rowsets` to `partialLoop()`, as you then have full access to your row objects within the view scripts, allowing you to call methods on them (such as retrieving values from parent or dependent rows).

Using PartialLoop to Render Iterable Models

Typically, you'll want to use partials in a loop, to render the same content fragment many times; this way you can put large blocks of repeated content or complex display logic into a single location. However this has a performance impact, as the partial helper needs to be invoked once for each iteration.

The `PartialLoop` view helper helps solve this issue. It allows you to pass an iterable item (array or object implementing **Iterator**) as the model. It then iterates over this, passing, the items to the partial script as the model. Items in the iterator may be any model the `Partial` view helper allows.

Let's assume the following partial view script:

```
<?php // partialLoop.phtml ?>
    <dt><?php echo $this->key ?></dt>
    <dd><?php echo $this->value ?></dd>
```

And the following "model":

```
$model = array(
    array('key' => 'Mammal', 'value' => 'Camel'),
    array('key' => 'Bird', 'value' => 'Penguin'),
    array('key' => 'Reptile', 'value' => 'Asp'),
    array('key' => 'Fish', 'value' => 'Flounder'),
);
```

In your view script, you could then invoke the `PartialLoop` helper:

```
<dl>
<?php echo $this->partialLoop('partialLoop.phtml', $model) ?>
</dl>
```

```
<dl>
    <dt>Mammal</dt>
    <dd>Camel</dd>

    <dt>Bird</dt>
    <dd>Penguin</dd>

    <dt>Reptile</dt>
    <dd>Asp</dd>

    <dt>Fish</dt>
    <dd>Flounder</dd>
</dl>
```

Rendering Partials in Other Modules

Sometime a partial will exist in a different module. If you know the name of the module, you can pass it as the second argument to either `partial()` or `partialLoop()`, moving the `$model` argument to third position.

For instance, if there's a pager partial you wish to use that's in the 'list' module, you could grab it as follows:

```
<?php echo $this->partial('pager.phtml', 'list', $pagerData) ?>
```

In this way, you can re-use partials created specifically for other modules. That said, it's likely a better practice to put re-usable partials in shared view script paths.

CHAPTER 169

Placeholder Helper

The `Placeholder` view helper is used to persist content between view scripts and view instances. It also offers some useful features such as aggregating content, capturing view script content for later use, and adding pre- and post-text to content (and custom separators for aggregated content).

Basic Usage of Placeholders

Basic usage of placeholders is to persist view data. Each invocation of the `Placeholder` helper expects a placeholder name; the helper then returns a placeholder container object that you can either manipulate or simply echo out.

```
<?php $this->placeholder('foo')->set("Some text for later") ?>

<?php
    echo $this->placeholder('foo');
    // outputs "Some text for later"
?>
```

Using Placeholders to Aggregate Content

Aggregating content via placeholders can be useful at times as well. For instance, your view script may have a variable array from which you wish to retrieve messages to display later; a later view script can then determine how those will be rendered.

The `Placeholder` view helper uses containers that extend `ArrayObject`, providing a rich 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'];
?>
```

Using Placeholders to Capture Content

Occasionally you may have content for a placeholder in a view script that is easiest to template; the `Placeholder` view helper allows you to capture arbitrary content for later rendering using the following *API*.

- `captureStart($type, $key)` begins capturing content.

  `$type` should be one of the `Placeholder` constants `APPEND` or `SET`. If `APPEND`, captured content is appended to the list of current content in the placeholder; if `SET`, captured content is used as the sole value of the placeholder (potentially replacing any previous content). By default, `$type` is `APPEND`.

  `$key` can be used to specify a specific key in the placeholder container to which you want content captured.

  `captureStart()` locks capturing until `captureEnd()` is called; you cannot nest capturing with the same placeholder container. Doing so will raise an exception.

- `captureEnd()` stops capturing content, and places it in the container object according to how `captureStart()` was called.

```
<!-- Default capture: append -->
<?php $this->placeholder('foo')->captureStart();
foreach ($this->data as $datum): ?>
<div class="foo">
```

```
    <h2><?php echo $datum->title ?></h2>
    <p><?php echo $datum->content ?></p>
</div>
<?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>

<?php echo $this->placeholder('foo') ?>
```

```
<!-- Capture to key -->
<?php $this->placeholder('foo')->captureStart('SET', 'data');
foreach ($this->data as $datum): ?>
<div class="foo">
    <h2><?php echo $datum->title ?></h2>
    <p><?php echo $datum->content ?></p>
</div>
 <?php endforeach; ?>
<?php $this->placeholder('foo')->captureEnd() ?>

<?php echo $this->placeholder('foo')->data ?>
```

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

CHAPTER 170

Doctype Helper

Valid *HTML* and *XHTML* documents should include a `DOCTYPE` declaration. Besides being difficult to remember, these can also affect how certain elements in your document should be rendered (for instance, CDATA escaping in **<script>** and **<style>** elements.

The `Doctype` helper allows you to specify one of the following types:

- `XHTML11`
- `XHTML1_STRICT`
- `XHTML1_TRANSITIONAL`
- `XHTML1_FRAMESET`
- `XHTML1_RDFA`
- `XHTML_BASIC1`
- `HTML4_STRICT`
- `HTML4_LOOSE`
- `HTML4_FRAMESET`
- `HTML5`

You can also specify a custom doctype as long as it is well-formed.

The `Doctype` helper is a concrete implementation of the Placeholder helper.

Doctype Helper Basic Usage

You may specify the doctype at any time. However, helpers that depend on the doctype for their output will recognize it only after you have set it, so the easiest approach is to specify it in your bootstrap:

```
1 $doctypeHelper = new Zend_View_Helper_Doctype();
2 $doctypeHelper->doctype('XHTML1_STRICT');
```

And then print it out on top of your layout script:

```
<?php echo $this->doctype() ?>
```

Retrieving the Doctype

If you need to know the doctype, you can do so by calling `getDoctype()` on the object, which is returned by invoking the helper.

```
$doctype = $view->doctype()->getDoctype();
```

Typically, you'll simply want to know if the doctype is *XHTML* or not; for this, the `isXhtml()` method will suffice:

```
if ($view->doctype()->isXhtml()) {
    // do something differently
}
```

You can also check if the doctype represents an *HTML5* document.

```
if ($view->doctype()->isHtml5()) {
    // do something differently
}
```

Choosing a Doctype to Use with the Open Graph Protocol

To implement the Open Graph Protocol, you may specify the XHTML1_RDFA doctype. This doctype allows a developer to use the Resource Description Framework within an *XHTML* document.

```
$doctypeHelper = new Zend_View_Helper_Doctype();
$doctypeHelper->doctype('XHTML1_RDFA');
```

The RDFa doctype allows XHTML to validate when the 'property' meta tag attribute is used per the Open Graph Protocol spec. Example within a view script:

```
<?php echo $this->doctype('XHTML1_RDFA'); ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:og="http://opengraphprotocol.org/schema/">
<head>
   <meta property="og:type" content="musician" />
```

In the previous example, we set the property to og:type. The og references the Open Graph namespace we specified in the html tag. The content identifies the page as being about a musician. See the Open Graph Protocol documentation for supported properties. The HeadMeta helper may be used to programmatically set these Open Graph Protocol meta tags.

Here is how you check if the doctype is set to XHTML1_RDFA:

```
<?php echo $this->doctype() ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      <?php if ($view->doctype()->isRdfa()): ?>
      xmlns:og="http://opengraphprotocol.org/schema/"
      xmlns:fb="http://www.facebook.com/2008/fbml"
      <?php endif; ?>
>
```

CHAPTER 171

HeadLink Helper

The *HTML* **<link>** element is increasingly used for linking a variety of resources for your site: stylesheets, feeds, favicons, trackbacks, and more. The `HeadLink` helper provides a simple interface for creating and aggregating these elements for later retrieval and output in your layout script.

The `HeadLink` helper has special methods for adding stylesheet links to its stack:

- `appendStylesheet($href, $media, $conditionalStylesheet, $extras)`
- `offsetSetStylesheet($index, $href, $media, $conditionalStylesheet, $extras)`
- `prependStylesheet($href, $media, $conditionalStylesheet, $extras)`
- `setStylesheet($href, $media, $conditionalStylesheet, $extras)`

The `$media` value defaults to 'screen', but may be any valid media value. `$conditionalStylesheet` is a string or boolean `FALSE`, and will be used at rendering time to determine if special comments should be included to prevent loading of the stylesheet on certain platforms. `$extras` is an array of any extra values that you want to be added to the tag.

Additionally, the `HeadLink` helper has special methods for adding 'alternate' links to its stack:

- `appendAlternate($href, $type, $title, $extras)`
- `offsetSetAlternate($index, $href, $type, $title, $extras)`
- `prependAlternate($href, $type, $title, $extras)`
- `setAlternate($href, $type, $title, $extras)`

The `headLink()` helper method allows specifying all attributes necessary for a **<link>** element, and allows you to also specify placement – whether the new element replaces all others, prepends (top of stack), or appends (end of stack).

The `HeadLink` helper is a concrete implementation of the Placeholder helper.

HeadLink Helper Basic Usage

You may specify a **headLink** at any time. Typically, you will specify global links in your layout script, and application specific links in your application view scripts. In your layout script, in the <head> section, you will then echo the helper to output it.

```
<?php // setting links in a view script:
$this->headLink()->appendStylesheet('/styles/basic.css')
                 ->headLink(array('rel' => 'icon',
                                  'href' => '/img/favicon.ico'),
                                  'PREPEND')
                 ->prependStylesheet('/styles/moz.css',
                                     'screen',
                                     true,
                                     array('id' => 'my_stylesheet'));
?>
<?php // rendering the links: ?>
<?php echo $this->headLink() ?>
```

CHAPTER 172

HeadMeta Helper

The *HTML* **<meta>** element is used to provide meta information about your *HTML* document – typically keywords, document character set, caching pragmas, etc. Meta tags may be either of the 'http-equiv' or 'name' types, must contain a 'content' attribute, and can also have either of the 'lang' or 'scheme' modifier attributes.

The `HeadMeta` helper supports the following methods for setting and adding meta tags:

- `appendName($keyValue, $content, $conditionalName)`
- `offsetSetName($index, $keyValue, $content, $conditionalName)`
- `prependName($keyValue, $content, $conditionalName)`
- `setName($keyValue, $content, $modifiers)`
- `appendHttpEquiv($keyValue, $content, $conditionalHttpEquiv)`
- `offsetSetHttpEquiv($index, $keyValue, $content, $conditionalHttpEquiv)`
- `prependHttpEquiv($keyValue, $content, $conditionalHttpEquiv)`
- `setHttpEquiv($keyValue, $content, $modifiers)`
- `setCharset($charset)`

The following methods are also supported with XHTML1_RDFA doctype set with the Doctype helper:

- `appendProperty($property, $content, $modifiers)`
- `offsetSetProperty($index, $property, $content, $modifiers)`
- `prependProperty($property, $content, $modifiers)`
- `setProperty($property, $content, $modifiers)`

The `$keyValue` item is used to define a value for the 'name' or 'http-equiv' key; `$content` is the value for the 'content' key, and `$modifiers` is an optional associative array that can contain keys for 'lang' and/or 'scheme'.

You may also set meta tags using the `headMeta()` helper method, which has the following signature: `headMeta($content, $keyValue, $keyType = 'name', $modifiers = array(), $placement = 'APPEND')`. `$keyValue` is the content for the key specified in `$keyType`, which should be either 'name' or 'http-equiv'. `$keyType` may also be specified as 'property' if the doctype has been set to

XHTML1_RDFA. `$placement` can be 'SET' (overwrites all previously stored values), 'APPEND' (added to end of stack), or 'PREPEND' (added to top of stack).

`HeadMeta` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a `stdClass` token, which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadMeta` helper is a concrete implementation of the Placeholder helper.

HeadMeta Helper Basic Usage

You may specify a new meta tag at any time. Typically, you will specify client-side caching rules or SEO keywords.

For instance, if you wish to specify SEO keywords, you'd be creating a meta name tag with the name 'keywords' and the content the keywords you wish to associate with your page:

```
// setting meta keywords
$this->headMeta()->appendName('keywords', 'framework, PHP, productivity');
```

If you wished to set some client-side caching rules, you'd set http-equiv tags with the rules you wish to enforce:

```
// disabling client-side cache
$this->headMeta()->appendHttpEquiv('expires',
                                   'Wed, 26 Feb 1997 08:21:57 GMT')
                 ->appendHttpEquiv('pragma', 'no-cache')
                 ->appendHttpEquiv('Cache-Control', 'no-cache');
```

Another popular use for meta tags is setting the content type, character set, and language:

```
// setting content type and character set
$this->headMeta()->appendHttpEquiv('Content-Type',
                                   'text/html; charset=UTF-8')
                 ->appendHttpEquiv('Content-Language', 'en-US');
```

If you are serving an *HTML5* document, you should provide the character set like this:

```
// setting character set in HTML5
$this->headMeta()->setCharset('UTF-8'); // Will look like <meta charset="UTF-8">
```

As a final example, an easy way to display a transitional message before a redirect is using a "meta refresh":

```
// setting a meta refresh for 3 seconds to a new url:
$this->headMeta()->appendHttpEquiv('Refresh',
                                   '3;URL=http://www.some.org/some.html');
```

When you're ready to place your meta tags in the layout, simply echo the helper:

```
<?php echo $this->headMeta() ?>
```

HeadMeta Usage with XHTML1_RDFA doctype

Enabling the RDFa doctype with the Doctype helper enables the use of the 'property' attribute (in addition to the standard 'name' and 'http-equiv') with HeadMeta. This is commonly used with the Facebook Open Graph Protocol.

For instance, you may specify an open graph page title and type as follows:

```
$this->doctype(Zend_View_Helper_Doctype::XHTML_RDFA);
$this->headMeta()->setProperty('og:title', 'my article title');
$this->headMeta()->setProperty('og:type', 'article');
echo $this->headMeta();

// output is:
//    <meta property="og:title" content="my article title" />
//    <meta property="og:type" content="article" />
```

CHAPTER 173

HeadScript Helper

The *HTML* **<script>** element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `HeadScript` helper allows you to manage both.

The `HeadScript` helper supports the following methods for setting and adding scripts:

- `appendFile($src, $type = 'text/javascript', $attrs = array())`
- `offsetSetFile($index, $src, $type = 'text/javascript', $attrs = array())`
- `prependFile($src, $type = 'text/javascript', $attrs = array())`
- `setFile($src, $type = 'text/javascript', $attrs = array())`
- `appendScript($script, $type = 'text/javascript', $attrs = array())`
- `offsetSetScript($index, $script, $type = 'text/javascript', $attrs = array())`
- `prependScript($script, $type = 'text/javascript', $attrs = array())`
- `setScript($script, $type = 'text/javascript', $attrs = array())`

In the case of the * `File()` methods, `$src` is the remote location of the script to load; this is usually in the form of a *URL* or a path. For the * `Script()` methods, `$script` is the client-side scripting directives you wish to use in the element.

Note: Setting Conditional Comments

`HeadScript` allows you to wrap the script tag in conditional comments, which allows you to hide it from specific browsers. To add the conditional tags, pass the conditional value as part of the `$attrs` parameter in the method calls.

Headscript With Conditional Comments

```
// adding scripts
$this->headScript()->appendFile(
    '/js/prototype.js',
    'text/javascript',
    array('conditional' => 'lt IE 7')
);
```

Note: Preventing HTML style comments or CDATA wrapping of scripts

By default `HeadScript` will wrap scripts with HTML comments or it wraps scripts with XHTML cdata. This behavior can be problematic when you intend to use the script tag in an alternative way by setting the type to something other then 'text/javascript'. To prevent such escaping, pass an `noescape` with a value of true as part of the `$attrs` parameter in the method calls.

Create an jQuery template with the headScript

```
// jquery template
$template = '<div class="book">{{:title}}</div>';
$this->headScript()->appendScript(
    $template,
    'text/x-jquery-tmpl',
    array('id='tmpl-book', 'noescape' => true)
);
```

`HeadScript` also allows capturing scripts; this can be useful if you want to create the client-side script programmatically, and then place it elsewhere. The usage for this will be showed in an example below.

Finally, you can also use the `headScript()` method to quickly add script elements; the signature for this is `headScript($mode = 'FILE', $spec, $placement = 'APPEND')`. The `$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.

Note: Use InlineScript for HTML Body Scripts

`HeadScript`'s sibling helper, InlineScript, should be used when you wish to include scripts inline in the *HTML* **body**. Placing scripts at the end of your document is a good practice for speeding up delivery of your page, particularly when using 3rd party analytics scripts.

Note: Arbitrary Attributes are Disabled by Default

By default, `HeadScript` only will render **<script>** attributes that are blessed by the W3C. These include 'type', 'charset', 'defer', 'language', and 'src'. However, some javascript frameworks, notably Dojo, utilize custom attributes in order to modify behavior. To allow such attributes, you can enable them via the `setAllowArbitraryAttributes()` method:

```
$this->headScript()->setAllowArbitraryAttributes(true);
```

HeadScript Helper Basic Usage

You may specify a new script tag at any time. As noted above, these may be links to outside resource files or scripts themselves.

```
// adding scripts
$this->headScript()->appendFile('/js/prototype.js')
                   ->appendScript($onloadScript);
```

Order is often important with client-side scripting; you may need to ensure that libraries are loaded in a specific order due to dependencies each have; use the various append, prepend, and offsetSet directives to aid in this task:

```
// Putting scripts in order

// place at a particular offset to ensure loaded last
$this->headScript()->offsetSetFile(100, '/js/myfuncs.js');

// use scriptaculous effects (append uses next index, 101)
$this->headScript()->appendFile('/js/scriptaculous.js');

// but always have base prototype script load first:
$this->headScript()->prependFile('/js/prototype.js');
```

When you're finally ready to output all scripts in your layout script, simply echo the helper:

```
<?php echo $this->headScript() ?>
```

Capturing Scripts Using the HeadScript Helper

Sometimes you need to generate client-side scripts programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the script and sprinkling in *PHP* tags. `HeadScript` lets you do just that, capturing it to the stack:

```
<?php $this->headScript()->captureStart() ?>
var action = '<?php echo $this->baseUrl ?>';
$('foo_form').action = action;
<?php $this->headScript()->captureEnd() ?>
```

The following assumptions are made:

- The script will be appended to the stack. If you wish for it to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- The script *MIME* type is assumed to be 'text/javascript'; if you wish to specify a different type, you will need to pass it as the second argument to `captureStart()`.
- If you wish to specify any additional attributes for the **<script>** tag, pass them in an array as the third argument to `captureStart()`.

CHAPTER 174

HeadStyle Helper

The *HTML* **<style>** element is used to include *CSS* stylesheets inline in the *HTML* **<head>** element.

Note: Use HeadLink to link CSS files

HeadLink should be used to create **<link>** elements for including external stylesheets. `HeadStyle` is used when you wish to define your stylesheets inline.

The `HeadStyle` helper supports the following methods for setting and adding stylesheet declarations:

- `appendStyle($content, $attributes = array())`
- `offsetSetStyle($index, $content, $attributes = array())`
- `prependStyle($content, $attributes = array())`
- `setStyle($content, $attributes = array())`

In all cases, `$content` is the actual *CSS* declarations. `$attributes` are any additional attributes you wish to provide to the `style` tag: lang, title, media, or dir are all permissible.

Note: Setting Conditional Comments

`HeadStyle` allows you to wrap the style tag in conditional comments, which allows you to hide it from specific browsers. To add the conditional tags, pass the conditional value as part of the `$attributes` parameter in the method calls.

Headstyle With Conditional Comments

```
// adding scripts
$this->headStyle()->appendStyle($styles, array('conditional' => 'lt IE 7'));
```

`HeadStyle` also allows capturing style declarations; this can be useful if you want to create the declarations programmatically, and then place them elsewhere. The usage for this will be showed in an example below.

Finally, you can also use the `headStyle()` method to quickly add declarations elements; the signature for this is `headStyle($content$placement = 'APPEND', $attributes = array())`. `$placement` should be either 'APPEND', 'PREPEND', or 'SET'.

`HeadStyle` overrides each of `append()`, `offsetSet()`, `prepend()`, and `set()` to enforce usage of the special methods as listed above. Internally, it stores each item as a `stdClass` token, which it later serializes using the `itemToString()` method. This allows you to perform checks on the items in the stack, and optionally modify these items by simply modifying the object returned.

The `HeadStyle` helper is a concrete implementation of the Placeholder helper.

Note: UTF-8 encoding used by default

By default, Zend Framework uses *UTF-8* as its default encoding, and, specific to this case, `Zend_View` does as well. Character encoding can be set differently on the view object itself using the `setEncoding()` method (or the the `encoding` instantiation parameter). However, since `Zend_View_Interface` does not define accessors for encoding, it's possible that if you are using a custom view implementation with this view helper, you will not have a `getEncoding()` method, which is what the view helper uses internally for determining the character set in which to encode.

If you do not want to utilize *UTF-8* in such a situation, you will need to implement a `getEncoding()` method in your custom view implementation.

HeadStyle Helper Basic Usage

You may specify a new style tag at any time:

```
1 // adding styles
2 $this->headStyle()->appendStyle($styles);
```

Order is very important with *CSS*; you may need to ensure that declarations are loaded in a specific order due to the order of the cascade; use the various append, prepend, and offsetSet directives to aid in this task:

```
1  // Putting styles in order
2
3  // place at a particular offset:
4  $this->headStyle()->offsetSetStyle(100, $customStyles);
5
6  // place at end:
7  $this->headStyle()->appendStyle($finalStyles);
8
9  // place at beginning
10 $this->headStyle()->prependStyle($firstStyles);
```

When you're finally ready to output all style declarations in your layout script, simply echo the helper:

```
1 <?php echo $this->headStyle() ?>
```

Capturing Style Declarations Using the HeadStyle Helper

Sometimes you need to generate *CSS* style declarations programmatically. While you could use string concatenation, heredocs, and the like, often it's easier just to do so by creating the styles and sprinkling in *PHP* tags. `HeadStyle` lets you do just that, capturing it to the stack:

```
<?php $this->headStyle()->captureStart() ?>
body {
    background-color: <?php echo $this->bgColor ?>;
}
<?php $this->headStyle()->captureEnd() ?>
```

The following assumptions are made:

- The style declarations will be appended to the stack. If you wish for them to replace the stack or be added to the top, you will need to pass 'SET' or 'PREPEND', respectively, as the first argument to `captureStart()`.
- If you wish to specify any additional attributes for the **<style>** tag, pass them in an array as the second argument to `captureStart()`.

CHAPTER 175

HeadTitle Helper

The *HTML* **<title>** element is used to provide a title for an *HTML* document. The `HeadTitle` helper allows you to programmatically create and store the title for later retrieval and output.

The `HeadTitle` helper is a concrete implementation of the Placeholder helper. It overrides the `toString()` method to enforce generating a **<title>** element, and adds a `headTitle()` method for quick and easy setting and aggregation of title elements. The signature for that method is `headTitle($title, $setType = null)`; by default, the value is appended to the stack (aggregating title segments) if left at null, but you may also specify either 'PREPEND' (place at top of stack) or 'SET' (overwrite stack).

Since setting the aggregating (attach) order on each call to `headTitle` can be cumbersome, you can set a default attach order by calling `setDefaultAttachOrder()` which is applied to all `headTitle()` calls unless you explicitly pass a different attach order as the second parameter.

HeadTitle Helper Basic Usage

You may specify a title tag at any time. A typical usage would have you setting title segments for each level of depth in your application: site, 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 -->
<?php echo $this->headTitle() ?>
```

CHAPTER 176

HTML Object Helpers

The *HTML* **<object>** element is used for embedding media like Flash or QuickTime in web pages. The object view helpers take care of embedding media with minimum effort.

There are four initial Object helpers:

- `htmlFlash()` Generates markup for embedding Flash files.
- `htmlObject()` Generates markup for embedding a custom Object.
- `htmlPage()` Generates markup for embedding other (X)HTML pages.
- `htmlQuicktime()` Generates markup for embedding QuickTime files.

All of these helpers share a similar interface. For this reason, this documentation will only contain examples of two of these helpers.

Flash helper

Embedding Flash in your page using the helper is pretty straight-forward. The only required argument is the resource *URI*.

```
<?php echo $this->htmlFlash('/path/to/flash.swf'); ?>
```

This outputs the following *HTML*:

```
<object data="/path/to/flash.swf"
        type="application/x-shockwave-flash"
        classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
        codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab
↪">
</object>
```

Additionally you can specify attributes, parameters and content that can be rendered along with the **<object>**. This will be demonstrated using the `htmlObject()` helper.

Customizing the object by passing additional arguments

The first argument in the object helpers is always required. It is the *URI* to the resource you want to embed. The second argument is only required in the `htmlObject()` helper. The other helpers already contain the correct value for this argument. The third argument is used for passing along attributes to the object element. It only accepts an array with key-value pairs. `classid` and `codebase` are examples of such attributes. The fourth argument also only takes a key-value array and uses them to create **<param>** elements. You will see an example of this shortly. Lastly, there is the option of providing additional content to the object. Now for an example which utilizes all arguments.

```
echo $this->htmlObject(
    '/path/to/file.ext',
    'mime/type',
    array(
        'attr1' => 'aval1',
        'attr2' => 'aval2'
    ),
    array(
        'param1' => 'pval1',
        'param2' => 'pval2'
    ),
    'some content'
);

/*
This would output:

<object data="/path/to/file.ext" type="mime/type"
    attr1="aval1" attr2="aval2">
    <param name="param1" value="pval1" />
    <param name="param2" value="pval2" />
    some content
</object>
*/
```

CHAPTER 177

InlineScript Helper

The *HTML* **<script>** element is used to either provide inline client-side scripting elements or link to a remote resource containing client-side scripting code. The `InlineScript` helper allows you to manage both. It is derived from HeadScript, and any method of that helper is available; however, use the `inlineScript()` method in place of `headScript()`.

Note: Use InlineScript for HTML Body Scripts

`InlineScript`, should be used when you wish to include scripts inline in the *HTML* **body**. Placing scripts at the end of your document is a good practice for speeding up delivery of your page, particularly when using 3rd party analytics scripts.

Some JS libraries need to be included in the *HTML* **head**; use HeadScript for those scripts.

CHAPTER 178

JSON Helper

When creating views that return *JSON*, it's important to also set the appropriate response header. The *JSON* view helper does exactly that. In addition, by default, it disables layouts (if currently enabled), as layouts generally aren't used with *JSON* responses.

The *JSON* helper sets the following header:

```
Content-Type: application/json
```

Most *AJAX* libraries look for this header when parsing responses to determine how to handle the content.

Usage of the *JSON* helper is very straightforward:

```
<?php echo $this->json($this->data) ?>
```

Note: Keeping layouts and enabling encoding using Zend_Json_Expr

Each method in the *JSON* helper accepts a second, optional argument. This second argument can be a boolean flag to enable or disable layouts, or an array of options that will be passed to `Zend_Json::encode()` and used internally to encode data.

To keep layouts, the second parameter needs to be boolean `TRUE`. When the second parameter is an array, keeping layouts can be achieved by including a `keepLayouts` key with a value of a boolean `TRUE`.

```
// Boolean true as second argument enables layouts:
echo $this->json($this->data, true);

// Or boolean true as "keepLayouts" key:
echo $this->json($this->data, array('keepLayouts' => true));
```

`Zend_Json::encode` allows the encoding of native *JSON* expressions using `Zend_Json_Expr` objects. This option is disabled by default. To enable this option, pass a boolean `TRUE` to the `enableJsonExprFinder` key of the options array:

```
<?php echo $this->json($this->data, array(
    'enableJsonExprFinder' => true,
    'keepLayouts'          => true,
)) ?>
```

CHAPTER 179

Navigation Helpers

The navigation helpers are used for rendering navigational elements from Zend_Navigation_Container instances.

There are 5 built-in helpers:

- Breadcrumbs, used for rendering the path to the currently active page.
- Links, used for rendering navigational head links (e.g. `<link rel="next" href="..." />`)
- Menu, used for rendering menus.
- Sitemap, used for rendering sitemaps conforming to the Sitemaps XML format.
- Navigation, used for proxying calls to other navigational helpers.

All built-in helpers extend `Zend_View_Helper_Navigation_HelperAbstract`, which adds integration with *ACL* and *translation*. The abstract class implements the interface `Zend_View_Helper_Navigation_Helper`, which defines the following methods:

- `getContainer()` and `setContainer()` gets and sets the navigation container the helper should operate on by default, and `hasContainer()` checks if the helper has container registered.
- `getTranslator()` and `setTranslator()` gets and sets the translator used for translating labels and titles. `getUseTranslator()` and `setUseTranslator()` controls whether the translator should be enabled. The method `hasTranslator()` checks if the helper has a translator registered.
- `getAcl()`, `setAcl()`, `getRole()` and `setRole()`, gets and sets *ACL* (`Zend\Permissions\Acl`) instance and role (`String` or `Zend\Permissions\Acl\Role\RoleInterface`) used for filtering out pages when rendering. `getUseAcl()` and `setUseAcl()` controls whether *ACL* should be enabled. The methods `hasAcl()` and `hasRole()` checks if the helper has an *ACL* instance or a role registered.
- `__toString()`, magic method to ensure that helpers can be rendered by echoing the helper instance directly.
- `render()`, must be implemented by concrete helpers to do the actual rendering.

In addition to the method stubs from the interface, the abstract class also implements the following methods:

- `getIndent()` and `setIndent()` gets and sets indentation. The setter accepts a `String` or an `Integer`. In the case of an `Integer`, the helper will use the given number of spaces for indentation. I.e.,

`setIndent(4)` means 4 initial spaces of indentation. Indentation can be specified for all helpers except the Sitemap helper.

- `getMinDepth()` and `setMinDepth()` gets and sets the minimum depth a page must have to be included by the helper. Setting `NULL` means no minimum depth.
- `getMaxDepth()` and `setMaxDepth()` gets and sets the maximum depth a page can have to be included by the helper. Setting `NULL` means no maximum depth.
- `getRenderInvisible()` and `setRenderInvisible()` gets and sets whether to render items that have been marked as invisible or not.
- `__call()` is used for proxying calls to the container registered in the helper, which means you can call methods on a helper as if it was a container. See example below.
- `findActive($container, $minDepth, $maxDepth)` is used for finding the deepest active page in the given container. If depths are not given, the method will use the values retrieved from `getMinDepth()` and `getMaxDepth()`. The deepest active page must be between `$minDepth` and `$maxDepth` inclusively. Returns an array containing a reference to the found page instance and the depth at which the page was found.
- `htmlify()` renders an **'a'** *HTML* element from a `Zend_Navigation_Page` instance.
- `accept()` is used for determining if a page should be accepted when iterating containers. This method checks for page visibility and verifies that the helper's role is allowed access to the page's resource and privilege.
- The static method `setDefaultAcl()` is used for setting a default *ACL* object that will be used by helpers.
- The static method `setDefaultRole()` is used for setting a default *ACL* that will be used by helpers

If a navigation container is not explicitly set in a helper using `$helper->setContainer($nav)`, the helper will look for a container instance with the key `Zend_Navigation` in the registry. If a container is not explicitly set or found in the registry, the helper will create an empty `Zend_Navigation` container when calling `$helper->getContainer()`.

Proxying calls to the navigation container

Navigation view helpers use the magic method `__call()` to proxy method calls to the navigation container that is registered in the view helper.

```
1 $this->navigation()->addPage(array(
2     'type' => 'uri',
3     'label' => 'New page'));
```

The call above will add a page to the container in the `Navigation` helper.

Translation of labels and titles

The navigation helpers support translation of page labels and titles. You can set a translator of type `Zend\I18n\Translator` in the helper using `$helper->setTranslator($translator)`.

If you want to disable translation, use `$helper->setUseTranslator(false)`.

The proxy helper will inject its own translator to the helper it proxies to if the proxied helper doesn't already have a translator.

Note: There is no translation in the sitemap helper, since there are no page labels or titles involved in an *XML* sitemap.

Integration with ACL

All navigational view helpers support *ACL* inherently from the class `Zend_View_Helper_Navigation_HelperAbstract`. A `Zend\Permissions\Acl` object can be assigned to a helper instance with *$helper->setAcl($acl)*, and role with *$helper->setRole('member')* or *$helper->setRole(new ZendPermissionsAclRoleGenericRole('member'))*. If *ACL* is used in the helper, the role in the helper must be allowed by the *ACL* to access a page's *resource* and/or have the page's *privilege* for the page to be included when rendering.

If a page is not accepted by *ACL*, any descendant page will also be excluded from rendering.

The proxy helper will inject its own *ACL* and role to the helper it proxies to if the proxied helper doesn't already have any.

The examples below all show how *ACL* affects rendering.

Navigation setup used in examples

This example shows the setup of a navigation container for a fictional software company.

Notes on the setup:

- The domain for the site is *www.example.com*.
- Interesting page properties are marked with a comment.
- Unless otherwise is stated in other examples, the user is requesting the *URL* *http://www.example.com/products/server/faq/*, which translates to the page labeled `FAQ` under *Foo Server*.
- The assumed *ACL* and router setup is shown below the container setup.

```
/*
 * Navigation container (config/array)

 * Each element in the array will be passed to
 * Zend_Navigation_Page::factory() when constructing
 * the navigation container below.
 */
$pages = array(
    array(
        'label'      => 'Home',
        'title'      => 'Go Home',
        'module'     => 'default',
        'controller' => 'index',
        'action'     => 'index',
        'order'      => -100 // make sure home is the first page
    ),
    array(
        'label'      => 'Special offer this week only!',
        'module'     => 'store',
        'controller' => 'offer',
        'action'     => 'amazing',
        'visible'    => false // not visible
    ),
    array(
        'label'      => 'Products',
        'module'     => 'products',
        'controller' => 'index',
        'action'     => 'index',
```

```
        'pages'      => array(
            array(
                'label'      => 'Foo Server',
                'module'     => 'products',
                'controller' => 'server',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'FAQ',
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'faq',
                        'rel'        => array(
                            'canonical' => 'http://www.example.com/?page=faq',
                            'alternate' => array(
                                'module'     => 'products',
                                'controller' => 'server',
                                'action'     => 'faq',
                                'params'     => array('format' => 'xml')
                            )
                        )
                    ),
                    array(
                        'label'      => 'Editions',
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'editions'
                    ),
                    array(
                        'label'      => 'System Requirements',
                        'module'     => 'products',
                        'controller' => 'server',
                        'action'     => 'requirements'
                    )
                )
            ),
            array(
                'label'      => 'Foo Studio',
                'module'     => 'products',
                'controller' => 'studio',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'Customer Stories',
                        'module'     => 'products',
                        'controller' => 'studio',
                        'action'     => 'customers'
                    ),
                    array(
                        'label'      => 'Support',
                        'module'     => 'prodcts',
                        'controller' => 'studio',
                        'action'     => 'support'
                    )
                )
            )
        )
    ),
```

```
    array(
        'label'      => 'Company',
        'title'      => 'About us',
        'module'     => 'company',
        'controller' => 'about',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'Investor Relations',
                'module'     => 'company',
                'controller' => 'about',
                'action'     => 'investors'
            ),
            array(
                'label'      => 'News',
                'class'      => 'rss', // class
                'module'     => 'company',
                'controller' => 'news',
                'action'     => 'index',
                'pages'      => array(
                    array(
                        'label'      => 'Press Releases',
                        'module'     => 'company',
                        'controller' => 'news',
                        'action'     => 'press'
                    ),
                    array(
                        'label'      => 'Archive',
                        'route'      => 'archive', // route
                        'module'     => 'company',
                        'controller' => 'news',
                        'action'     => 'archive'
                    )
                )
            )
        )
    ),
    array(
        'label'      => 'Community',
        'module'     => 'community',
        'controller' => 'index',
        'action'     => 'index',
        'pages'      => array(
            array(
                'label'      => 'My Account',
                'module'     => 'community',
                'controller' => 'account',
                'action'     => 'index',
                'resource'   => 'mvc:community.account' // resource
            ),
            array(
                'label' => 'Forums',
                'uri'   => 'http://forums.example.com/',
                'class' => 'external' // class
            )
        )
    ),
    array(
```

```
        'label'      => 'Administration',
        'module'     => 'admin',
        'controller' => 'index',
        'action'     => 'index',
        'resource'   => 'mvc:admin', // resource
        'pages'      => array(
            array(
                'label'      => 'Write new article',
                'module'     => 'admin',
                'controller' => 'post',
                'aciton'     => 'write'
            )
        )
    )
);

// Create container from array
$container = new Zend_Navigation($pages);

// Store the container in the proxy helper:
$view->getHelper('navigation')->setContainer($container);

// ...or simply:
$view->navigation($container);

// ...or store it in the reigstry:
Zend_Registry::set('Zend_Navigation', $container);
```

In addition to the container above, the following setup is assumed:

```
// Setup router (default routes and 'archive' route):
$front = Zend_Controller_Front::getInstance();
$router = $front->getRouter();
$router->addDefaultRoutes();
$router->addRoute(
    'archive',
    new Zend_Controller_Router_Route(
        '/archive/:year',
        array(
            'module'     => 'company',
            'controller' => 'news',
            'action'     => 'archive',
            'year'       => (int) date('Y') - 1
        ),
        array('year' => '\d+')
    )
);

// Setup ACL:
$acl = new Zend\Permissions\Acl\Acl();
$acl->addRole(new Zend\Permissions\Acl\Role\GenericRole('member'));
$acl->addRole(new Zend\Permissions\Acl\Role\GenericRole('admin'));
$acl->add(new Zend\Permissions\Acl\Resource\GenericResource('mvc:admin'));
$acl->add(new Zend\Permissions\Acl\Resource\GenericResource('mvc:community.account'));
$acl->allow('member', 'mvc:community.account');
$acl->allow('admin', null);

// Store ACL and role in the proxy helper:
```

```
$view->navigation()->setAcl($acl)->setRole('member');

// ...or set default ACL and role statically:
Zend_View_Helper_Navigation_HelperAbstract::setDefaultAcl($acl);
Zend_View_Helper_Navigation_HelperAbstract::setDefaultRole('member');
```

Breadcrumbs Helper

Breadcrumbs are used for indicating where in a sitemap a user is currently browsing, and are typically rendered like this: "You are here: Home > Products > FantasticProduct 1.0". The breadcrumbs helper follows the guidelines from Breadcrumbs Pattern - Yahoo! Design Pattern Library, and allows simple customization (minimum/maximum depth, indentation, separator, and whether the last element should be linked), or rendering using a partial view script.

The Breadcrumbs helper works like this; it finds the deepest active page in a navigation container, and renders an upwards path to the root. For *MVC* pages, the "activeness" of a page is determined by inspecting the request object, as stated in the section on Zend_Navigation_Page_Mvc.

The helper sets the *minDepth* property to 1 by default, meaning breadcrumbs will not be rendered if the deepest active page is a root page. If *maxDepth* is specified, the helper will stop rendering when at the specified depth (e.g. stop at level 2 even if the deepest active page is on level 3).

Methods in the breadcrumbs helper:

- *{get|set}Separator()* gets/sets separator string that is used between breadcrumbs. Defualt is ' > '.
- *{get|set}LinkLast()* gets/sets whether the last breadcrumb should be rendered as an anchor or not. Default is `FALSE`.
- *{get|set}Partial()* gets/sets a partial view script that should be used for rendering breadcrumbs. If a partial view script is set, the helper's `render()` method will use the `renderPartial()` method. If no partial is set, the `renderStraight()` method is used. The helper expects the partial to be a `String` or an `Array` with two elements. If the partial is a `String`, it denotes the name of the partial script to use. If it is an `Array`, the first element will be used as the name of the partial view script, and the second element is the module where the script is found.
- `renderStraight()` is the default render method.
- `renderPartial()` is used for rendering using a partial view script.

Rendering breadcrumbs

This example shows how to render breadcrumbs with default settings.

```
In a view script or layout:
<?php echo $this->navigation()->breadcrumbs(); ?>

The two calls above take advantage of the magic __toString() method,
and are equivalent to:
<?php echo $this->navigation()->breadcrumbs()->render(); ?>

Output:
<a href="/products">Products</a> > <a href="/products/server">Foo Server</a> > FAQ
```

Specifying indentation

This example shows how to render breadcrumbs with initial indentation.

```
Rendering with 8 spaces indentation:
<?php echo $this->navigation()->breadcrumbs()->setIndent(8);?>

Output:
        <a href="/products">Products</a> > <a href="/products/server">Foo Server</a> >
↪ FAQ
```

Customize breadcrumbs output

This example shows how to customze breadcrumbs output by specifying various options.

```
In a view script or layout:

<?php
echo $this->navigation()
          ->breadcrumbs()
          ->setLinkLast(true)                    // link last page
          ->setMaxDepth(1)                       // stop at level 1
          ->setSeparator(' ' . PHP_EOL); // cool separator with newline
?>

Output:
<a href="/products">Products</a>
<a href="/products/server">Foo Server</a>

////////////////////////////////////////////////////

Setting minimum depth required to render breadcrumbs:

<?php
$this->navigation()->breadcrumbs()->setMinDepth(10);
echo $this->navigation()->breadcrumbs();
?>

Output:
Nothing, because the deepest active page is not at level 10 or deeper.
```

Rendering breadcrumbs using a partial view script

This example shows how to render customized breadcrumbs using a partial vew script. By calling `setPartial()`, you can specify a partial view script that will be used when calling `render()`. When a partial is specified, the `renderPartial()` method will be called. This method will find the deepest active page and pass an array of pages that leads to the active page to the partial view script.

In a layout:

```
$partial = ;
echo $this->navigation()->breadcrumbs()
                        ->setPartial(array('breadcrumbs.phtml', 'default'));
```

Contents of *application/modules/default/views/breadcrumbs.phtml*:

```
echo implode(', ', array_map(
        create_function('$a', 'return $a->getLabel();'),
        $this->pages));
```

Output:

```
Products, Foo Server, FAQ
```

Links Helper

The links helper is used for rendering *HTML* `LINK` elements. Links are used for describing document relationships of the currently active page. Read more about links and link types at Document relationships: the LINK element (HTML4 W3C Rec.) and Link types (HTML4 W3C Rec.) in the *HTML*4 W3C Recommendation.

There are two types of relations; forward and reverse, indicated by the keyords *'rel'* and *'rev'*. Most methods in the helper will take a `$rel` param, which must be either *'rel'* or *'rev'*. Most methods also take a `$type` param, which is used for specifying the link type (e.g. alternate, start, next, prev, chapter, etc).

Relationships can be added to page objects manually, or found by traversing the container registered in the helper. The method `findRelation($page, $rel, $type)` will first try to find the given `$rel` of `$type` from the `$page` by calling *$page->findRel($type)* or *$page->findRel($type)*. If the `$page` has a relation that can be converted to a page instance, that relation will be used. If the `$page` instance doesn't have the specified `$type`, the helper will look for a method in the helper named *searchreltype* (e.g. `searchRelNext()` or `searchRevAlternate()`). If such a method exists, it will be used for determining the `$page`'s relation by traversing the container.

Not all relations can be determined by traversing the container. These are the relations that will be found by searching:

- `searchRelStart()`, forward 'start' relation: the first page in the container.
- `searchRelNext()`, forward 'next' relation; finds the next page in the container, i.e. the page after the active page.
- `searchRelPrev()`, forward 'prev' relation; finds the previous page, i.e. the page before the active page.
- `searchRelChapter()`, forward 'chapter' relations; finds all pages on level 0 except the 'start' relation or the active page if it's on level 0.
- `searchRelSection()`, forward 'section' relations; finds all child pages of the active page if the active page is on level 0 (a 'chapter').
- `searchRelSubsection()`, forward 'subsection' relations; finds all child pages of the active page if the active pages is on level 1 (a 'section').
- `searchRevSection()`, reverse 'section' relation; finds the parent of the active page if the active page is on level 1 (a 'section').
- `searchRevSubsection()`, reverse 'subsection' relation; finds the parent of the active page if the active page is on level 2 (a 'subsection').

Note: When looking for relations in the page instance (*$page->getRel($type)* or *$page->getRev($type)*), the helper accepts the values of type `String`, `Array`, `Zend_Config`, or `Zend_Navigation_Page`. If a string is found, it will be converted to a `Zend_Navigation_Page_Uri`. If an array or a config is found, it will be converted to one or several page instances. If the first key of the array/config is numeric, it will be considered to contain several pages, and each element will be passed to the page factory. If the first key is not numeric, the array/config will be passed to the page factory directly, and a single page will be returned.

The helper also supports magic methods for finding relations. E.g. to find forward alternate relations, call *$helper->findRelAlternate($page)*, and to find reverse section relations, call *$helper->findRevSection($page)*. Those calls correspond to *$helper->findRelation($page, 'rel', 'alternate');* and *$helper->findRelation($page, 'rev', 'section');* respectively.

To customize which relations should be rendered, the helper uses a render flag. The render flag is an integer value, and will be used in a bitwse and (&) operation against the helper's render constants to determine if the relation that belongs to the render constant should be rendered.

See the example below for more information.

- `Zend_View_Helper_Navigation_Link::RENDER_ALTERNATE`
- `Zend_View_Helper_Navigation_Link::RENDER_STYLESHEET`
- `Zend_View_Helper_Navigation_Link::RENDER_START`
- `Zend_View_Helper_Navigation_Link::RENDER_NEXT`
- `Zend_View_Helper_Navigation_Link::RENDER_PREV`
- `Zend_View_Helper_Navigation_Link::RENDER_CONTENTS`
- `Zend_View_Helper_Navigation_Link::RENDER_INDEX`
- `Zend_View_Helper_Navigation_Link::RENDER_GLOSSARY`
- `Zend_View_Helper_Navigation_Link::RENDER_COPYRIGHT`
- `Zend_View_Helper_Navigation_Link::RENDER_CHAPTER`
- `Zend_View_Helper_Navigation_Link::RENDER_SECTION`
- `Zend_View_Helper_Navigation_Link::RENDER_SUBSECTION`
- `Zend_View_Helper_Navigation_Link::RENDER_APPENDIX`
- `Zend_View_Helper_Navigation_Link::RENDER_HELP`
- `Zend_View_Helper_Navigation_Link::RENDER_BOOKMARK`
- `Zend_View_Helper_Navigation_Link::RENDER_CUSTOM`
- `Zend_View_Helper_Navigation_Link::RENDER_ALL`

The constants from `RENDER_ALTERNATE` to `RENDER_BOOKMARK` denote standard *HTML* link types. `RENDER_CUSTOM` denotes non-standard relations that specified in pages. `RENDER_ALL` denotes standard and non-standard relations.

Methods in the links helper:

- *{get|set}RenderFlag()* gets/sets the render flag. Default is `RENDER_ALL`. See examples below on how to set the render flag.
- `findAllRelations()` finds all relations of all types for a given page.
- `findRelation()` finds all relations of a given type from a given page.
- *searchRel{Start|Next|Prev|Chapter|Section|Subsection}()* traverses a container to find forward relations to the start page, the next page, the previous page, chapters, sections, and subsections.
- *searchRev{Section|Subsection}()* traverses a container to find reverse relations to sections or subsections.
- `renderLink()` renders a single *link* element.

Specify relations in pages

This example shows how to specify relations in pages.

```
$container = new Zend_Navigation(array(
    array(
        'label' => 'Relations using strings',
        'rel'   => array(
            'alternate' => 'http://www.example.org/'
        ),
        'rev'   => array(
            'alternate' => 'http://www.example.net/'
        )
    ),
    array(
        'label' => 'Relations using arrays',
        'rel'   => array(
            'alternate' => array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            )
        )
    ),
    array(
        'label' => 'Relations using configs',
        'rel'   => array(
            'alternate' => new Zend_Config(array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            ))
        )
    ),
    array(
        'label' => 'Relations using pages instance',
        'rel'   => array(
            'alternate' => Zend_Navigation_Page::factory(array(
                'label' => 'Example.org',
                'uri'   => 'http://www.example.org/'
            ))
        )
    )
));
```

Default rendering of links

This example shows how to render a menu from a container registered/found in the view helper.

```
In a view script or layout:
<?php echo $this->view->navigation()->links(); ?>

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="chapter" href="/products" title="Products">
```

```
<link rel="chapter" href="/company/about" title="Company">
<link rel="chapter" href="/community" title="Community">
<link rel="canonical" href="http://www.example.com/?page=server-faq">
<link rev="subsection" href="/products/server" title="Foo Server">
```

Specify which relations to render

This example shows how to specify which relations to find and render.

```
Render only start, next, and prev:
$helper->setRenderFlag(Zend_View_Helper_Navigation_Links::RENDER_START |
                       Zend_View_Helper_Navigation_Links::RENDER_NEXT |
                       Zend_View_Helper_Navigation_Links::RENDER_PREV);

Output:
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
```

```
Render only native link types:
$helper->setRenderFlag(Zend_View_Helper_Navigation_Links::RENDER_ALL ^
                       Zend_View_Helper_Navigation_Links::RENDER_CUSTOM);

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="chapter" href="/products" title="Products">
<link rel="chapter" href="/company/about" title="Company">
<link rel="chapter" href="/community" title="Community">
<link rev="subsection" href="/products/server" title="Foo Server">
```

```
Render all but chapter:
$helper->setRenderFlag(Zend_View_Helper_Navigation_Links::RENDER_ALL ^
                       Zend_View_Helper_Navigation_Links::RENDER_CHAPTER);

Output:
<link rel="alternate" href="/products/server/faq/format/xml">
<link rel="start" href="/" title="Home">
<link rel="next" href="/products/server/editions" title="Editions">
<link rel="prev" href="/products/server" title="Foo Server">
<link rel="canonical" href="http://www.example.com/?page=server-faq">
<link rev="subsection" href="/products/server" title="Foo Server">
```

Menu Helper

The Menu helper is used for rendering menus from navigation containers. By default, the menu will be rendered using *HTML UL* and *LI* tags, but the helper also allows using a partial view script.

Methods in the Menu helper:

- *{get|set}UlClass()* gets/sets the *CSS* class used in `renderMenu()`.

- *{get|set}OnlyActiveBranch()* gets/sets a flag specifying whether only the active branch of a container should be rendered.
- *{get|set}RenderParents()* gets/sets a flag specifying whether parents should be rendered when only rendering active branch of a container. If set to `FALSE`, only the deepest active menu will be rendered.
- *{get|set}Partial()* gets/sets a partial view script that should be used for rendering menu. If a partial view script is set, the helper's `render()` method will use the `renderPartial()` method. If no partial is set, the `renderMenu()` method is used. The helper expects the partial to be a `String` or an `Array` with two elements. If the partial is a `String`, it denotes the name of the partial script to use. If it is an `Array`, the first element will be used as the name of the partial view script, and the second element is the module where the script is found.
- `htmlify()` overrides the method from the abstract class to return *span* elements if the page has no *href*.
- `renderMenu($container = null, $options = array())` is the default render method, and will render a container as a *HTML UL* list.

  If `$container` is not given, the container registered in the helper will be rendered.

  `$options` is used for overriding options specified temporarily without rsetting the values in the helper instance. It is an associative array where each key corresponds to an option in the helper.

  Recognized options:

  - *indent*; indentation. Expects a `String` or an *int* value.
  - *minDepth*; minimum depth. Expcects an *int* or `NULL` (no minimum depth).
  - *maxDepth*; maximum depth. Expcects an *int* or `NULL` (no maximum depth).
  - *ulClass*; *CSS* class for *ul* element. Expects a `String`.
  - *onlyActiveBranch*; whether only active branch should be rendered. Expects a `Boolean` value.
  - *renderParents*; whether parents should be rendered if only rendering active branch. Expects a `Boolean` value.

  If an option is not given, the value set in the helper will be used.
- `renderPartial()` is used for rendering the menu using a partial view script.
- `renderSubMenu()` renders the deepest menu level of a container's active branch.

Rendering a menu

This example shows how to render a menu from a container registered/found in the view helper. Notice how pages are filtered out based on visibility and *ACL*.

```
In a view script or layout:
<?php echo $this->navigation()->menu()->render() ?>

Or simply:
<?php echo $this->navigation()->menu() ?>

Output:
<ul class="navigation">
    <li>
        <a title="Go Home" href="/">Home</a>
    </li>
    <li class="active">
        <a href="/products">Products</a>
```

```
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
                <ul>
                    <li class="active">
                        <a href="/products/server/faq">FAQ</a>
                    </li>
                    <li>
                        <a href="/products/server/editions">Editions</a>
                    </li>
                    <li>
                        <a href="/products/server/requirements">System Requirements</
↪a>
                    </li>
                </ul>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
                <ul>
                    <li>
                        <a href="/products/studio/customers">Customer Stories</a>
                    </li>
                    <li>
                        <a href="/prodcts/studio/support">Support</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
    <li>
        <a title="About us" href="/company/about">Company</a>
        <ul>
            <li>
                <a href="/company/about/investors">Investor Relations</a>
            </li>
            <li>
                <a class="rss" href="/company/news">News</a>
                <ul>
                    <li>
                        <a href="/company/news/press">Press Releases</a>
                    </li>
                    <li>
                        <a href="/archive">Archive</a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community">Community</a>
        <ul>
            <li>
                <a href="/community/account">My Account</a>
            </li>
            <li>
                <a class="external" href="http://forums.example.com/">Forums</a>
            </li>
        </ul>
```

```
    </li>
</ul>
```

Calling renderMenu() directly

This example shows how to render a menu that is not registered in the view helper by calling the `renderMenu()` directly and specifying a few options.

```
<?php
// render only the 'Community' menu
$community = $this->navigation()->findOneByLabel('Community');
$options = array(
    'indent'  => 16,
    'ulClass' => 'community'
);
echo $this->navigation()
          ->menu()
          ->renderMenu($community, $options);
?>
Output:
                <ul class="community">
                    <li>
                        <a href="/community/account">My Account</a>
                    </li>
                    <li>
                        <a class="external" href="http://forums.example.com/">Forums</
↪a>
                    </li>
                </ul>
```

Rendering the deepest active menu

This example shows how the `renderSubMenu()` will render the deepest sub menu of the active branch.

Calling `renderSubMenu($container, $ulClass, $indent)` is equivalent to calling `renderMenu($container, $options)` with the following options:

```
array(
    'ulClass'          => $ulClass,
    'indent'           => $indent,
    'minDepth'         => null,
    'maxDepth'         => null,
    'onlyActiveBranch' => true,
    'renderParents'    => false
);
```

```
<?php
echo $this->navigation()
          ->menu()
          ->renderSubMenu(null, 'sidebar', 4);
?>

The output will be the same if 'FAQ' or 'Foo Server' is active:
    <ul class="sidebar">
```

```
        <li class="active">
            <a href="/products/server/faq">FAQ</a>
        </li>
        <li>
            <a href="/products/server/editions">Editions</a>
        </li>
        <li>
            <a href="/products/server/requirements">System Requirements</a>
        </li>
    </ul>
```

Rendering a menu with maximum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setMaxDepth(1);
?>

Output:
<ul class="navigation">
    <li>
        <a title="Go Home" href="/">Home</a>
    </li>
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
            </li>
        </ul>
    </li>
    <li>
        <a title="About us" href="/company/about">Company</a>
        <ul>
            <li>
                <a href="/company/about/investors">Investor Relations</a>
            </li>
            <li>
                <a class="rss" href="/company/news">News</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community">Community</a>
        <ul>
            <li>
                <a href="/community/account">My Account</a>
            </li>
            <li>
                <a class="external" href="http://forums.example.com/">Forums</a>
            </li>
        </ul>
```

```
    </li>
</ul>
```

Rendering a menu with minimum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setMinDepth(1);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
        <ul>
            <li class="active">
                <a href="/products/server/faq">FAQ</a>
            </li>
            <li>
                <a href="/products/server/editions">Editions</a>
            </li>
            <li>
                <a href="/products/server/requirements">System Requirements</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/products/studio">Foo Studio</a>
        <ul>
            <li>
                <a href="/products/studio/customers">Customer Stories</a>
            </li>
            <li>
                <a href="/prodcts/studio/support">Support</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/company/about/investors">Investor Relations</a>
    </li>
    <li>
        <a class="rss" href="/company/news">News</a>
        <ul>
            <li>
                <a href="/company/news/press">Press Releases</a>
            </li>
            <li>
                <a href="/archive">Archive</a>
            </li>
        </ul>
    </li>
    <li>
        <a href="/community/account">My Account</a>
    </li>
    <li>
```

```
        <a class="external" href="http://forums.example.com/">Forums</a>
    </li>
</ul>
```

Rendering only the active branch of a menu

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
                <ul>
                    <li class="active">
                        <a href="/products/server/faq">FAQ</a>
                    </li>
                    <li>
                        <a href="/products/server/editions">Editions</a>
                    </li>
                    <li>
                        <a href="/products/server/requirements">System Requirements</
↪a>
                    </li>
                </ul>
            </li>
        </ul>
    </li>
</ul>
```

Rendering only the active branch of a menu with minimum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setMinDepth(1);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
        <ul>
            <li class="active">
                <a href="/products/server/faq">FAQ</a>
            </li>
            <li>
```

```
                <a href="/products/server/editions">Editions</a>
            </li>
            <li>
                <a href="/products/server/requirements">System Requirements</a>
            </li>
        </ul>
    </li>
</ul>
```

Rendering only the active branch of a menu with maximum depth

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setMaxDepth(1);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products">Products</a>
        <ul>
            <li class="active">
                <a href="/products/server">Foo Server</a>
            </li>
            <li>
                <a href="/products/studio">Foo Studio</a>
            </li>
        </ul>
    </li>
</ul>
```

Rendering only the active branch of a menu with maximum depth and no parents

```
<?php
echo $this->navigation()
          ->menu()
          ->setOnlyActiveBranch(true)
          ->setRenderParents(false)
          ->setMaxDepth(1);
?>

Output:
<ul class="navigation">
    <li class="active">
        <a href="/products/server">Foo Server</a>
    </li>
    <li>
        <a href="/products/studio">Foo Studio</a>
    </li>
</ul>
```

Rendering a custom menu using a partial view script

This example shows how to render a custom menu using a partial vew script. By calling `setPartial()`, you can specify a partial view script that will be used when calling `render()`. When a partial is specified, the `renderPartial()` method will be called. This method will assign the container to the view with the key *container*.

In a layout:

```
$partial = array('menu.phtml', 'default');
$this->navigation()->menu()->setPartial($partial);
echo $this->navigation()->menu()->render();
```

In application/modules/default/views/menu.phtml:

```
foreach ($this->container as $page) {
    echo $this->navigation()->menu()->htmlify($page), PHP_EOL;
}
```

Output:

```
<a title="Go Home" href="/">Home</a>
<a href="/products">Products</a>
<a title="About us" href="/company/about">Company</a>
<a href="/community">Community</a>
```

Sitemap Helper

The Sitemap helper is used for generating *XML* sitemaps, as defined by the Sitemaps XML format. Read more about Sitemaps on Wikpedia.

By default, the sitemap helper uses sitemap validators to validate each element that is rendered. This can be disabled by calling *$helper->setUseSitemapValidators(false)*.

Note: If you disable sitemap validators, the custom properties (see table) are not validated at all.

The sitemap helper also supports Sitemap XSD Schema validation of the generated sitemap. This is disabled by default, since it will require a request to the Schema file. It can be enabled with *$helper->setUseSchemaValidation(true)*.

Table 179.1: Sitemap XML elements

Element	Description
loc	Absolute URL to page. An absolute URL will be generated by the helper.
lastmod	The date of last modification of the file, in W3C Datetime format. This time portion can be omitted if desired, and only use YYYY-MM-DD. The helper will try to retrieve the lastmod value from the page's custom property lastmod if it is set in the page. If the value is not a valid date, it is ignored.
changefreq	How frequently the page is likely to change. This value provides general information to search engines and may not correlate exactly to how often they crawl the page. Valid values are: alwayshourlydailyweeklymonthlyyearlynever The helper will try to retrieve the changefreq value from the page's custom property changefreq if it is set in the page. If the value is not valid, it is ignored.
priority	The priority of this URL relative to other URLs on your site. Valid values range from 0.0 to 1.0. The helper will try to retrieve the priority value from the page's custom property priority if it is set in the page. If the value is not valid, it is ignored.

Methods in the sitemap helper:

- *{get|set}FormatOutput()* gets/sets a flag indicating whether *XML* output should be formatted. This corresponds to the *formatOutput* property of the native `DOMDocument` class. Read more at PHP: DOMDocument - Manual. Default is `FALSE`.
- *{get|set}UseXmlDeclaration()* gets/sets a flag indicating whether the *XML* declaration should be included when rendering. Default is `TRUE`.
- *{get|set}UseSitemapValidators()* gets/sets a flag indicating whether sitemap validators should be used when generating the DOM sitemap. Default is `TRUE`.
- *{get|set}UseSchemaValidation()* gets/sets a flag indicating whether the helper should use *XML* Schema validation when generating the DOM sitemap. Default is `FALSE`. If `TRUE`.
- *{get|set}ServerUrl()* gets/sets server *URL* that will be prepended to non-absolute *URL*s in the `url()` method. If no server *URL* is specified, it will be determined by the helper.
- `url()` is used to generate absolute *URL*s to pages.
- `getDomSitemap()` generates a DOMDocument from a given container.

Rendering an XML sitemap

This example shows how to render an *XML* sitemap based on the setup we did further up.

```
// In a view script or layout:

// format output
$this->navigation()
      ->sitemap()
      ->setFormatOutput(true); // default is false

// other possible methods:
// ->setUseXmlDeclaration(false); // default is true
// ->setServerUrl('http://my.otherhost.com');
// default is to detect automatically

// print sitemap
echo $this->navigation()->sitemap();
```

Notice how pages that are invisible or pages with *ACL* roles incompatible with the view helper are filtered out:

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/faq</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/editions</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/requirements</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio/customers</loc>
  </url>
  <url>
    <loc>http://www.example.com/prodcts/studio/support</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about/investors</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news/press</loc>
  </url>
  <url>
    <loc>http://www.example.com/archive</loc>
  </url>
  <url>
    <loc>http://www.example.com/community</loc>
  </url>
  <url>
    <loc>http://www.example.com/community/account</loc>
  </url>
  <url>
    <loc>http://forums.example.com/</loc>
  </url>
</urlset>
```

Render the sitemap using no *ACL* role (should filter out /community/account):

```
echo $this->navigation()
           ->sitemap()
           ->setFormatOutput(true)
           ->setRole();
```

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/faq</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/editions</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server/requirements</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio/customers</loc>
  </url>
  <url>
    <loc>http://www.example.com/prodcts/studio/support</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about/investors</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news/press</loc>
  </url>
  <url>
    <loc>http://www.example.com/archive</loc>
  </url>
  <url>
    <loc>http://www.example.com/community</loc>
  </url>
  <url>
    <loc>http://forums.example.com/</loc>
  </url>
</urlset>
```

Render the sitemap using a maximum depth of 1.

```php
echo $this->navigation()
          ->sitemap()
          ->setFormatOutput(true)
          ->setMaxDepth(1);
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.example.com/</loc>
  </url>
  <url>
    <loc>http://www.example.com/products</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/server</loc>
  </url>
  <url>
    <loc>http://www.example.com/products/studio</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/about/investors</loc>
  </url>
  <url>
    <loc>http://www.example.com/company/news</loc>
  </url>
  <url>
    <loc>http://www.example.com/community</loc>
  </url>
  <url>
    <loc>http://www.example.com/community/account</loc>
  </url>
  <url>
    <loc>http://forums.example.com/</loc>
  </url>
</urlset>
```

Note: UTF-8 encoding used by default

By default, Zend Framework uses *UTF-8* as its default encoding, and, specific to this case, `Zend_View` does as well. Character encoding can be set differently on the view object itself using the `setEncoding()` method (or the the `encoding` instantiation parameter). However, since `Zend_View_Interface` does not define accessors for encoding, it's possible that if you are using a custom view implementation with the Dojo view helper, you will not have a `getEncoding()` method, which is what the view helper uses internally for determining the character set in which to encode.

If you do not want to utilize *UTF-8* in such a situation, you will need to implement a `getEncoding()` method in your custom view implementation.

Navigation Helper

The Navigation helper is a proxy helper that relays calls to other navigational helpers. It can be considered an entry point to all navigation-related view tasks. The aforementioned navigational helpers are in the namespace `Zend_View_Helper_Navigation`, and would thus require the path *Zend/View/Helper/Navigation* to be added as a helper path to the view. With the proxy helper residing in the `Zend_View_Helper` namespace, it will always be available, without the need to add any helper paths to the view.

The Navigation helper finds other helpers that implement the `Zend_View_Helper_Navigation_Helper` interface, which means custom view helpers can also be proxied. This would, however, require that the custom helper path is added to the view.

When proxying to other helpers, the Navigation helper can inject its container, *ACL*/role, and translator. This means that you won't have to explicitly set all three in all navigational helpers, nor resort to injecting by means of `Zend_Registry` or static methods.

- `findHelper()` finds the given helper, verifies that it is a navigational helper, and injects container, *ACL*/role and translator.
- *{get|set}InjectContainer()* gets/sets a flag indicating whether the container should be injected to proxied helpers. Default is `TRUE`.
- *{get|set}InjectAcl()* gets/sets a flag indicating whether the *ACL*/role should be injected to proxied helpers. Default is `TRUE`.
- *{get|set}InjectTranslator()* gets/sets a flag indicating whether the translator should be injected to proxied helpers. Default is `TRUE`.
- *{get|set}DefaultProxy()* gets/sets the default proxy. Default is *'menu'*.
- `render()` proxies to the render method of the default proxy.

Registering Helpers

`Zend\View\Renderer\PhpRenderer` composes a plugin broker for managing helpers, specifically an instance of `Zend\View\HelperBroker`, which extends the base plugin broker in order to ensure we have valid helpers available. The `HelperBroker` by default uses `Zend\View\HelperLoader` as its helper locator. The `HelperLoader` is a map-based loader, which means that you will simply map the helper/plugin name by which you wish to refer to it to the actual class name of the helper/plugin.

Programmatically, this is done as follows:

```
// $view is an instance of PhpRenderer
$broker = $view->getBroker();
$loader = $broker->getClassLoader();

// Register singly:
$loader->registerPlugin('lowercase', 'My\Helper\LowerCase');

// Register several:
$loader->registerPlugins(array(
    'lowercase' => 'My\Helper\LowerCase',
    'uppercase' => 'My\Helper\UpperCase',
));
```

Within an MVC application, you will typically simply pass a map of plugins to the class via your configuration.

```
// From within a configuration file
return array(
    'di' => array('instance' => array(
        'Zend\View\HelperLoader' => array('parameters' => array(
            'map' => array(
                'lowercase' => 'My\Helper\LowerCase',
                'uppercase' => 'My\Helper\UpperCase',
            ),
        )),
    )),
);
```

The above can be done in each module that needs to register helpers with the `PhpRenderer`; however, be aware that another module can register helpers with the same name, so order of modules can impact which helper class will actually be registered!

Writing Custom Helpers

Writing custom helpers is easy. We recommend extending `Zend\View\Helper\AbstractHelper`, but at the minimum, you need only implement the `Zend\View\Helper` interface:

```
namespace Zend\View;

interface Helper
{
    /**
     * Set the View object
     *
     * @param  Renderer $view
     * @return Helper
     */
    public function setView(Renderer $view);

    /**
     * Get the View object
     *
     * @return Renderer
     */
    public function getView();
}
```

If you want your helper to be capable of being invoked as if it were a method call of the `PhpRenderer`, you should also implement an `__invoke()` method within your helper.

As previously noted, we recommend extending `Zend\View\Helper\AbstractHelper`, as it implements the methods defined in `Helper`, giving you a headstart in your development.

Once you have defined your helper class, make sure you can autoload it, and then *register it with the plugin broker*.

Here is an example helper, which we're titling "SpecialPurpose"

```
namespace My\View\Helper;

use Zend\View\Helper\AbstractHelper;

class SpecialPurpose extends AbstractHelper
```

```
{
    protected $count = 0;

    public function __invoke()
    {
        $this->count++;
        $output = sprintf("I have seen 'The Jerk' %d time(s).", $this->count);
        return htmlspecialchars($output, ENT_QUOTES, 'UTF-8');
    }
}
```

Then assume that when we *register it with the plugin broker*, we map it to the string "specialpurpose".

Within a view script, you can call the `SpecialPurpose` helper as many times as you like; it will be instantiated once, and then it persists for the life of that `PhpRenderer` instance.

```
// remember, in a view script, $this refers to the Zend_View instance.
echo $this->specialPurpose();
echo $this->specialPurpose();
echo $this->specialPurpose();
```

The output would look something like this:

```
I have seen 'The Jerk' 1 time(s).
I have seen 'The Jerk' 2 time(s).
I have seen 'The Jerk' 3 time(s).
```

Sometimes you will need access to the calling `PhpRenderer` object – for instance, if you need to use the registered encoding, or want to render another view script as part of your helper. This is why we define the `setView()` and `getView()` methods. As an example, we could rewrite the `SpecialPurpose` helper as follows to take advantage of the `EscapeHtml` helper:

```
namespace My\View\Helper;

use Zend\View\Helper\AbstractHelper;

class SpecialPurpose extends AbstractHelper
{
    protected $count = 0;

    public function __invoke()
    {
        $this->count++;
        $output  = sprintf("I have seen 'The Jerk' %d time(s).", $this->count);
        $escaper = $this->getView()->plugin('escapehtml');
        return $escaper($output);
    }
}
```

Registering Concrete Helpers

Sometimes it is convenient to instantiate a view helper, and then register it with the renderer. This can be done by injecting it directly into the plugin broker.

```
// $view is a PhpRenderer instance

$helper = new My_Helper_Foo();
// ...do some configuration or dependency injection...

$view->getBroker()->register('foo', $helper);
```

When registered, the plugin broker will inject the `PhpRenderer` instance into the helper.

CHAPTER 180

Introduction

From its home page, *XML-RPC* is described as a "...remote procedure calling using *HTTP* as the transport and *XML* as the encoding. *XML-RPC* is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned."

Zend Framework provides support for both consuming remote *XML-RPC* services and building new *XML-RPC* servers.

CHAPTER 181

Zend\XmlRpc\Client

Introduction

Zend Framework provides support for consuming remote *XML-RPC* services as a client in the `Zend\XmlRpc\Client` package. Its major features include automatic type conversion between *PHP* and *XML-RPC*, a server proxy object, and access to server introspection capabilities.

Method Calls

The constructor of `Zend\XmlRpc\Client` receives the *URL* of the remote *XML-RPC* server endpoint as its first parameter. The new instance returned may be used to call any number of remote methods at that endpoint.

To call a remote method with the *XML-RPC* client, instantiate it and use the `call()` instance method. The code sample below uses a demonstration *XML-RPC* server on the Zend Framework website. You can use it for testing or exploring the `Zend\XmlRpc` components.

XML-RPC Method Call

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

echo $client->call('test.sayHello');

// hello
```

The *XML-RPC* value returned from the remote method call will be automatically unmarshaled and cast to the equivalent *PHP* native type. In the example above, a *PHP* `String` is returned and is immediately ready to be used.

The first parameter of the `call()` method receives the name of the remote method to call. If the remote method requires any parameters, these can be sent by supplying a second, optional parameter to `call()` with an `Array` of values to pass to the remote method:

XML-RPC Method Call with Parameters

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$arg1 = 1.1;
$arg2 = 'foo';

$result = $client->call('test.sayHello', array($arg1, $arg2));

// $result is a native PHP type
```

If the remote method doesn't require parameters, this optional parameter may either be left out or an empty `array()` passed to it. The array of parameters for the remote method can contain native *PHP* types, `Zend\XmlRpc\Value` objects, or a mix of each.

The `call()` method will automatically convert the *XML-RPC* response and return its equivalent *PHP* native type. A `Zend\XmlRpc\Response` object for the return value will also be available by calling the `getLastResponse()` method after the call.

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 181.1: PHP and XML-RPC Type Conversions

PHP Native Type	XML-RPC Type
integer	int
Zend\Math\BigInteger\BigInteger	i8
double	double
boolean	boolean
string	string
null	nil
array	array
associative array	struct
object	array
DateTime	dateTime.iso8601
DateTime	dateTime.iso8601

Note: What type do empty arrays get cast to?

Passing an empty array to an *XML-RPC* method is problematic, as it could represent either an array or a struct. `Zend\XmlRpc\Client` detects such conditions and makes a request to the server's `system.methodSignature` method to determine the appropriate *XML-RPC* type to cast to.

However, this in itself can lead to issues. First off, servers that do not support `system.methodSignature` will log failed requests, and `Zend\XmlRpc\Client` will resort to casting the value to an *XML-RPC* array type. Additionally, this means that any call with array arguments will result in an additional call to the remote server.

To disable the lookup entirely, you can call the `setSkipSystemLookup()` method prior to making your *XML-RPC* call:

```
$client->setSkipSystemLookup(true);
$result = $client->call('foo.bar', array(array()));
```

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 181.2: 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
i8	Zend\XmlRpc\Value::XMLRPC_TYPE_I8	Zend\XmlRpc\Value\BigInteger
ex:i8	Zend\XmlRpc\Value::XMLRPC_TYPE_APACHEI8	Zend\XmlRpc\Value\BigInteger
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
nil	Zend\XmlRpc\Value::XMLRPC_TYPE_NIL	Zend\XmlRpc\Value\Nil
ex:nil	Zend\XmlRpc\Value::XMLRPC_TYPE_APACHENIL	Zend\XmlRpc\Value\Nil
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

Note: Automatic Conversion

When building a new `Zend\XmlRpc\Value` object, its value is set by a *PHP* type. The *PHP* type will be converted to the specified type using *PHP* casting. For example, if a string is given as a value to the `Zend\XmlRpc\Value\Integer` object, it will be converted using `(int)$value`.

Server Proxy Object

Another way to call remote methods with the *XML-RPC* client is to use the server proxy. This is a *PHP* object that proxies a remote *XML-RPC* namespace, making it work as close to a native *PHP* object as possible.

To instantiate a server proxy, call the `getProxy()` instance method of `Zend\XmlRpc\Client`. This will return an instance of `Zend\XmlRpc\Client\ServerProxy`. Any method call on the server proxy object will be forwarded to the remote, and parameters may be passed like any other *PHP* method.

Proxy the Default Namespace

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$service = $client->getProxy();           // Proxy the default namespace

$hello = $service->test->sayHello(1, 2);  // test.Hello(1, 2) returns "hello"
```

The `getProxy()` method receives an optional argument specifying which namespace of the remote server to proxy. If it does not receive a namespace, the default namespace will be proxied. In the next example, the 'test' namespace will be proxied:

Proxy Any Namespace

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$test  = $client->getProxy('test');     // Proxy the "test" namespace

$hello = $test->sayHello(1, 2);         // test.Hello(1,2) returns "hello"
```

If the remote server supports nested namespaces of any depth, these can also be used through the server proxy. For example, if the server in the example above had a method `test.foo.bar()`, it could be called as `$test->foo->bar()`.

Error Handling

Two kinds of errors can occur during an *XML-RPC* method call: *HTTP* errors and *XML-RPC* faults. The `Zend\XmlRpc\Client` recognizes each and provides the ability to detect and trap them independently.

HTTP Errors

If any *HTTP* error occurs, such as the remote *HTTP* server returns a **404 Not Found**, a `Zend\XmlRpc\Client\Exception\HttpException` will be thrown.

Handling HTTP Errors

```
$client = new Zend\XmlRpc\Client('http://foo/404');

try {
```

```
4
5       $client->call('bar', array($arg1, $arg2));
6
7   } catch (Zend\XmlRpc\Client\Exception\HttpException $e) {
8
9       // $e->getCode() returns 404
10      // $e->getMessage() returns "Not Found"
11
12  }
```

Regardless of how the *XML-RPC* client is used, the `Zend\XmlRpc\Client\Exception\HttpException` will be thrown whenever an *HTTP* error occurs.

XML-RPC Faults

An *XML-RPC* fault is analogous to a *PHP* exception. It is a special type returned from an *XML-RPC* method call that has both an error code and an error message. *XML-RPC* faults are handled differently depending on the context of how the `Zend\XmlRpc\Client` is used.

When the `call()` method or the server proxy object is used, an *XML-RPC* fault will result in a `Zend\XmlRpc\Client\Exception\FaultException` being thrown. The code and message of the exception will map directly to their respective values in the original *XML-RPC* fault response.

Handling XML-RPC Faults

```
1   $client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');
2
3   try {
4
5       $client->call('badMethod');
6
7   } catch (Zend\XmlRpc\Client\Exception\FaultException $e) {
8
9       // $e->getCode() returns 1
10      // $e->getMessage() returns "Unknown method"
11
12  }
```

When the `call()` method is used to make the request, the `Zend\XmlRpc\Client\Exception\FaultException` will be thrown on fault. A `Zend\XmlRpc\Response` object containing the fault will also be available by calling `getLastResponse()`.

When the `doRequest()` method is used to make the request, it will not throw the exception. Instead, it will return a `Zend\XmlRpc\Response` object returned will containing the fault. This can be checked with `isFault()` instance method of `Zend\XmlRpc\Response`.

Server Introspection

Some *XML-RPC* servers support the de facto introspection methods under the *XML-RPC* **system.** namespace. `Zend\XmlRpc\Client` provides special support for servers with these capabilities.

A `Zend\XmlRpc\Client\ServerIntrospection` instance may be retrieved by calling the `getIntrospector()` method of `Zend\XmlRpc\Client`. It can then be used to perform introspection operations on the server.

From Request to Response

Under the hood, the `call()` instance method of `Zend\XmlRpc\Client` builds a request object (`Zend\XmlRpc\Request`) and sends it to another method, `doRequest()`, that returns a response object (`Zend\XmlRpc\Response`).

The `doRequest()` method is also available for use directly:

Processing Request to Response

```
$client = new Zend\XmlRpc\Client('http://framework.zend.com/xmlrpc');

$request = new Zend\XmlRpc\Request();
$request->setMethod('test.sayHello');
$request->setParams(array('foo', 'bar'));

$client->doRequest($request);

// $client->getLastRequest() returns instanceof Zend_XmlRpc_Request
// $client->getLastResponse() returns instanceof Zend_XmlRpc_Response
```

Whenever an *XML-RPC* method call is made by the client through any means, either the `call()` method, `doRequest()` method, or server proxy, the last request object and its resultant response object will always be available through the methods `getLastRequest()` and `getLastResponse()` respectively.

HTTP Client and Testing

In all of the prior examples, an *HTTP* client was never specified. When this is the case, a new instance of `Zend\Http\Client` will be created with its default options and used by `Zend\XmlRpc\Client` automatically.

The *HTTP* client can be retrieved at any time with the `getHttpClient()` method. For most cases, the default *HTTP* client will be sufficient. However, the `setHttpClient()` method allows for a different *HTTP* client instance to be injected.

The `setHttpClient()` is particularly useful for unit testing. When combined with the `Zend\Http\Client\Adapter\Test`, remote services can be mocked out for testing. See the unit tests for `Zend\XmlRpc\Client` for examples of how to do this.

CHAPTER 182

Zend\XmlRpc\Server

Introduction

`Zend\XmlRpc\Server` is intended as a fully-featured *XML-RPC* server, following the specifications outlined at www.xmlrpc.com. Additionally, it implements the `system.multicall()` method, allowing boxcarring of requests.

Basic Usage

An example of the most basic use case:

```
1 $server = new Zend\XmlRpc\Server();
2 $server->setClass('My\Service\Class');
3 echo $server->handle();
```

Server Structure

`Zend\XmlRpc\Server` is composed of a variety of components, ranging from the server itself to request, response, and fault objects.

To bootstrap `Zend\XmlRpc\Server`, the developer must attach one or more classes or functions to the server, via the `setClass()` and `addFunction()` methods.

Once done, you may either pass a `Zend\XmlRpc\Request` object to `Zend\XmlRpc\Server::handle()`, or it will instantiate a `Zend\XmlRpc\Request\Http` object if none is provided – thus grabbing the request from `php://input`.

`Zend\XmlRpc\Server::handle()` then attempts to dispatch to the appropriate handler based on the method requested. It then returns either a `Zend\XmlRpc\Response`-based object or a

`Zend\XmlRpc\Server\Fault`object. These objects both have `__toString()` methods that create valid *XML-RPC XML* responses, allowing them to be directly echoed.

Anatomy of a webservice

General considerations

For maximum performance it is recommended to use a simple bootstrap file for the server component. Using `Zend\XmlRpc\Server` inside a Zend\Controller is strongly discouraged to avoid the overhead.

Services change over time and while webservices are generally less change intense as code-native *APIs*, it is recommended to version your service. Do so to lay grounds to provide compatibility for clients using older versions of your service and manage your service lifecycle including deprecation timeframes.To do so just include a version number into your *URI*. It is also recommended to include the remote protocol name in the *URI* to allow easy integration of upcoming remoting technologies. http://myservice.ws/**1.0/XMLRPC/**.

What to expose?

Most of the time it is not sensible to expose business objects directly. Business objects are usually small and under heavy change, because change is cheap in this layer of your application. Once deployed and adopted, web services are hard to change. Another concern is *I/O* and latency: the best webservice calls are those not happening. Therefore service calls need to be more coarse-grained than usual business logic is. Often an additional layer in front of your business objects makes sense. This layer is sometimes referred to as Remote Facade. Such a service layer adds a coarse grained interface on top of your business logic and groups verbose operations into smaller ones.

Conventions

`Zend\XmlRpc\Server` allows the developer to attach functions and class method calls as dispatchable *XML-RPC* methods. Via `Zend\Server\Reflection`, it does introspection on all attached methods, using the function and method docblocks to determine the method help text and method signatures.

XML-RPC types do not necessarily map one-to-one to *PHP* types. However, the code will do its best to guess the appropriate type based on the values listed in @param and @return lines. Some *XML-RPC* types have no immediate *PHP* equivalent, however, and should be hinted using the *XML-RPC* type in the PHPDoc. These include:

- **dateTime.iso8601**, a string formatted as 'YYYYMMDDTHH:mm:ss'
- **base64**, base64 encoded data
- **struct**, any associative array

An example of how to hint follows:

```
/**
* 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 'l' operator:

```
/**
* This is a sample function
*
* @param string|base64 $val1 String or base64-encoded data
* @param string|dateTime.iso8601 $val2 String or an ISO date
* @param array|struct $val3 Normal indexed array or an associative array
* @return boolean|struct
*/
function myFunc($val1, $val2, $val3)
{
}
```

Note: Allowing multiple signatures can lead to confusion for developers using the services; to keep things simple, a *XML-RPC* service method should only have a single signature.

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:

```
1 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:

```php
use Zend\XmlRpc\Server as XmlRpcServer;

// Register the "My\Services" namespace
$loader = new Zend\Loader\StandardAutoloader();
$loader->registerNamespace('My\Services', 'path to My/Services');
$loader->register();

$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';
$server = new XmlRpcServer();

if (!XmlRpcServer\Cache::get($cacheFile, $server)) {

    $server->setClass('My\Services\Glue', 'glue');   // glue. namespace
    $server->setClass('My\Services\Paste', 'paste'); // paste. namespace
    $server->setClass('My\Services\Tape', 'tape');   // tape. namespace

    XmlRpcServer\Cache::save($cacheFile, $server);
}

echo $server->handle();
```

The above example attempts to retrieve a server definition from `xmlrpc.cache` in the same directory as the script. If unsuccessful, it loads the service classes it needs, attaches them to the server instance, and then attempts to create a new cache file with the server definition.

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.

```php
/**
 * 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.

```php
require_once 'Services/Comb.php';

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb');
echo $server->handle();
```

Attaching a class with arguments

The following example illustrates how to attach a class' public methods and passing arguments to its methods. This can be used to specify certain defaults when registering service classes.

```php
class Services_PricingService
{
    /**
     * Calculate current price of product with $productId
     *
     * @param ProductRepository $productRepository
     * @param PurchaseRepository $purchaseRepository
     * @param integer $productId
     */
    public function calculate(ProductRepository $productRepository,
                              PurchaseRepository $purchaseRepository,
                              $productId)
    {
        ...
    }
}

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\PricingService',
                  'pricing',
                  new ProductRepository(),
                  new PurchaseRepository());
```

The arguments passed at `setClass()` at server construction time are injected into the method call `pricing.calculate()` on remote invokation. In the example above, only the argument `$purchaseId` is expected from the client.

Passing arguments only to constructor

`Zend\XmlRpc\Server` allows to restrict argument passing to constructors only. This can be used for constructor dependency injection. To limit injection to constructors, call `sendArgumentsToAllMethods` and pass `FALSE` as an argument. This disables the default behavior of all arguments being injected into the remote method. In the example below the instance of `ProductRepository` and `PurchaseRepository` is only injected into the constructor of `Services_PricingService2`.

```php
class Services\PricingService2
{
    /**
     * @param ProductRepository $productRepository
     * @param PurchaseRepository $purchaseRepository
     */
    public function __construct(ProductRepository $productRepository,
                                PurchaseRepository $purchaseRepository)
```

```
    {
        ...
    }

    /**
     * Calculate current price of product with $productId
     *
     * @param integer $productId
     * @return double
     */
    public function calculate($productId)
    {
        ...
    }
}

$server = new Zend\XmlRpc\Server();
$server->sendArgumentsToAllMethods(false);
$server->setClass('Services\PricingService2',
                  'pricing',
                  new ProductRepository(),
                  new PurchaseRepository());
```

Attaching a class instance

`setClass()` allows to register a previously instantiated object at the server. Just pass an instance instead of the class name. Obviously passing arguments to the constructor is not possible with pre-instantiated objects.

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
```

```php
Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb', 'comb');   // methods called as comb.*
$server->setClass('Services\Brush', 'brush'); // methods called as brush.*
$server->setClass('Services\Pick', 'pick');   // methods called as pick.*
echo $server->handle();
```

Utilizing custom request and response objects

Some use cases require to utilize a custom request object. For example, *XML/RPC* is not bound to *HTTP* as a transfer protocol. It is possible to use other transfer protocols like *SSH* or telnet to send the request and response data over the wire. Another use case is authentication and authorization. In case of a different transfer protocol, one need to change the implementation to read request data.

The example below instantiates a custom request object and passes it to the server to handle.

```php
require_once 'Services/Request.php';
require_once 'Services/Exception.php';
require_once 'Services/Comb.php';
require_once 'Services/Brush.php';
require_once 'Services/Pick.php';

// Allow Services_Exceptions to report as fault responses
Zend\XmlRpc\Server\Fault::attachFaultException('Services\Exception');

$server = new Zend\XmlRpc\Server();
$server->setClass('Services\Comb', 'comb');   // methods called as comb.*
$server->setClass('Services\Brush', 'brush'); // methods called as brush.*
$server->setClass('Services\Pick', 'pick');   // methods called as pick.*

// Create a request object
$request = new Services\Request();

echo $server->handle($request);
```

Specifying a custom response class

The example below illustrates specifying a custom response class for the returned response.

```php
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.*
```

```php

// Create a request object
$request = new Services\Request();

// Utilize a custom response
$server->setResponseClass('Services\Response');

echo $server->handle($request);
```

Performance optimization

Cache server definitions between requests

The example below illustrates caching server definitions between requests.

```php
use Zend\XmlRpc\Server as XmlRpcServer;

// Register the "Services" namespace
$loader = new Zend\Loader\StandardAutoloader();
$loader->registerNamespace('Services', 'path to Services');
$loader->register();

// Specify a cache file
$cacheFile = dirname(__FILE__) . '/xmlrpc.cache';

// Allow Services\Exceptions to report as fault responses
XmlRpcServer\Fault::attachFaultException('Services\Exception');

$server = new XmlRpcServer();

// Attempt to retrieve server definition from cache
if (!XmlRpcServer\Cache::get($cacheFile, $server)) {
    $server->setClass('Services\Comb', 'comb');   // methods called as comb.*
    $server->setClass('Services\Brush', 'brush'); // methods called as brush.*
    $server->setClass('Services\Pick', 'pick');   // methods called as pick.*

    // Save cache
    XmlRpcServer\Cache::save($cacheFile, $server);
}

// Create a request object
$request = new Services\Request();

// Utilize a custom response
$server->setResponseClass('Services\Response');

echo $server->handle($request);
```

Note: The server cache file should be located outside the document root.

Optimizing XML generation

`Zend\XmlRpc\Server` uses `DOMDocument` of *PHP* extension **ext/dom** to generate it's *XML* output. While **ext/dom** is available on a lot of hosts it is not exactly the fastest. Benchmarks have shown, that `XmlWriter` from **ext/xmlwriter** performs better.

If **ext/xmlwriter** is available on your host, you can select a the `XmlWriter`-based generator to leaverage the performance differences.

```
use Zend\XmlRpc;

XmlRpc\Value::setGenerator(new XmlRpc\Generator\XmlWriter());

$server = new XmlRpc\Server();
...
```

Note: Benchmark your application

Performance is determined by a lot of parameters and benchmarks only apply for the specific test case. Differences come from *PHP* version, installed extensions, webserver and operating system just to name a few. Please make sure to benchmark your application on your own and decide which generator to use based on **your** numbers.

Note: Benchmark your client

This optimization makes sense for the client side too. Just select the alternate *XML* generator before doing any work with `Zend\XmlRpc\Client`.

CHAPTER 183

ZendService\LiveDocx

Introduction to LiveDocx

LiveDocx is a *SOAP* service that allows developers to generate word processing documents by combining structured textual or image data from *PHP* with a template, created in a word processor. The resulting document can be saved as a *PDF*, *DOCX*, *DOC*, *HTML* or *RTF* file. LiveDocx implements mail-merge in *PHP*.

The family of `ZendService\LiveDocx` components provides a clean and simple interface to *LiveDocx Free*, *LiveDocx Premium* and *LiveDocx Fully Licensed*, authored by *Text Control GmbH*, and additionally offers functionality to improve network performance.

`ZendService\LiveDocx` is part of the official Zend Framework family, but has to be downloaded and installed in addition to the core components of the Zend Framework, as do all other service components. Please refer to GitHub (ZendServiceLiveDocx) for download and installation instructions.

In addition to this section of the manual, to learn more about `ZendService\LiveDocx` and the backend *SOAP* service LiveDocx, please take a look at the following resources:

- **Shipped demonstration applications**. There is a large number of demonstration applications in the directory `/demos`. They illustrate all functionality offered by LiveDocx. Where appropriate this part of the user manual references the demonstration applications at the end of each section. It is **highly recommended** to read all the code in the `/demos` directory. It is well commented and explains all you need to know about LiveDocx and `ZendService\LiveDocx`.
- LiveDocx in PHP.
- LiveDocx SOAP API documentation.
- LiveDocx WSDL.
- LiveDocx blog and web site.

Sign Up for an Account

Before you can start using LiveDocx, you must first sign up for an account. The account is completely free of charge and you only need to specify a **username**, **password** and **e-mail address**. Your login credentials will be dispatched to the e-mail address you supply, so please type carefully. If, or when, your application gets really popular and you require high performance, or additional features only supplied in the premium service, you can upgrade from the *LiveDocx Free* to *LiveDocx Premium* for a minimal monthly charge. For details of the various services, please refer to LiveDocx pricing.

Templates and Documents

LiveDocx differentiates between the following terms: 1) **template** and 2) **document**. In order to fully understand the documentation and indeed LiveDocx itself, it is important that any programmer deploying LiveDocx understands the difference.

The term **template** is used to refer to the input file, created in a word processor, containing formatting and text fields. You can download an example template, stored as a *DOCX* file. The term **document** is used to refer to the output file that contains the template file, populated with data - i.e. the finished document. You can download an example document, stored as a *PDF* file.

Supported File Formats

LiveDocx supports the following file formats:

Template File Formats (input)

Templates can be saved in any of the following file formats:

- DOCX- Office Open *XML* format
- DOC- Microsoft Word *DOC* format
- RTF- Rich text file format
- TXD- TX Text Control format

Document File Formats (output):

The resulting document can be saved in any of the following file formats:

- DOCX- Office Open *XML* format
- DOC- Microsoft Word *DOC* format
- HTML-*XHTML* 1.0 transitional format
- RTF- Rich text file format
- PDF- Acrobat Portable Document Format
- PDF/A- Acrobat Portable Document Format (ISO-standardized version)
- TXD- TX Text Control format
- TXT-*ANSI* plain text

Image File Formats (output):

The resulting document can be saved in any of the following graphical file formats:

- BMP- Bitmap image format
- GIF- Graphics Interchange Format
- JPG- Joint Photographic Experts Group format
- PNG- Portable Network Graphics format
- TIFF- Tagged Image File Format
- WMF- Windows Meta File format

`ZendService\LiveDocx\MailMerge`

`MailMerge` is the mail-merge object in the `ZendService\LiveDocx` family.

Document Generation Process

The document generation process can be simplified with the following equation:

Template + Data = Document

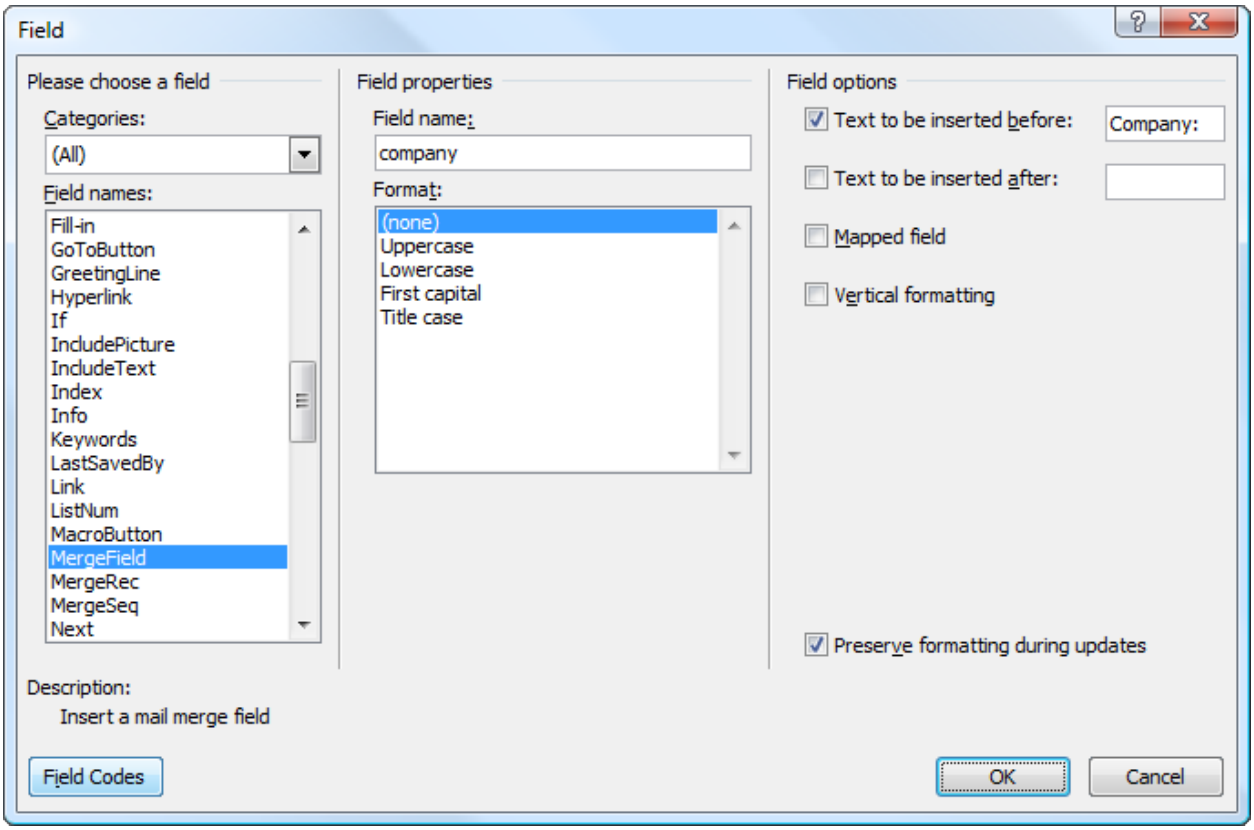
Or expressed by the following diagram:

Data is inserted into template to create a document.

A template, created in a word processing application, such as Microsoft Word, is loaded into LiveDocx. Data is then inserted into the template and the resulting document is saved to any supported format.

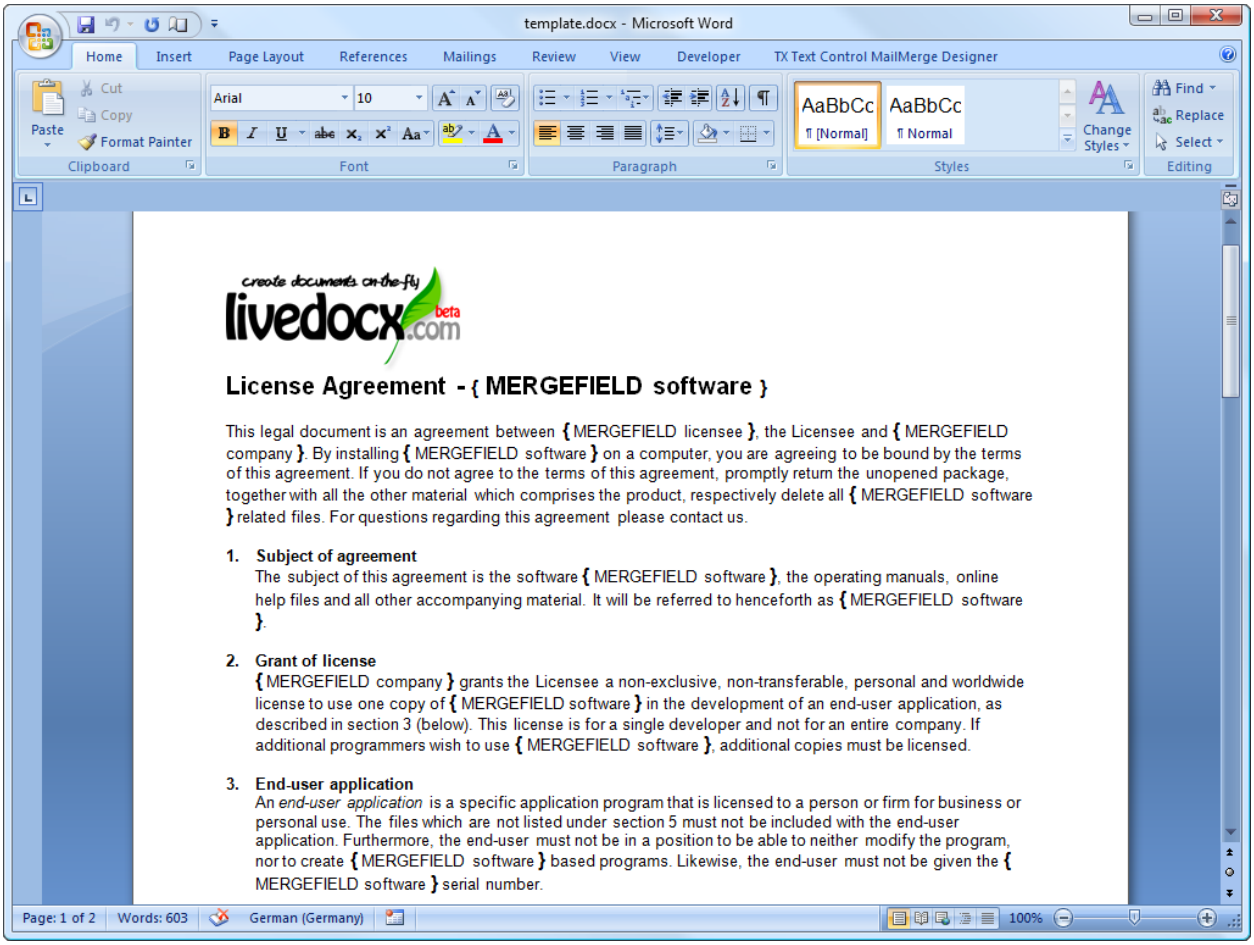
Creating Templates in Microsoft Word 2007

Start off by launching Microsoft Word and creating a new document. Next, open up the **Field** dialog box. This looks as follows:



Microsoft Word 2007 Field dialog box.

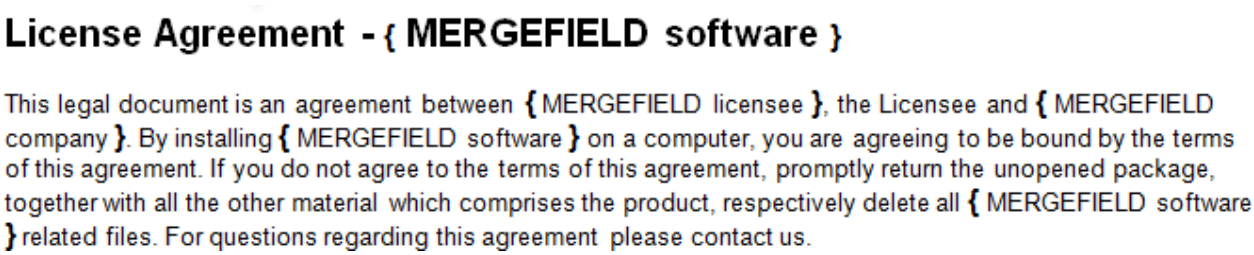
Using this dialog, you can insert the required merge fields into your document. Below is a screenshot of a license agreement in Microsoft Word 2007. The merge fields are marked as `{ MERGEFIELD FieldName }`:



Template in Microsoft Word 2007.

Now, save the template as **template.docx**.

In the next step, we are going to populate the merge fields with textual data from *PHP*.



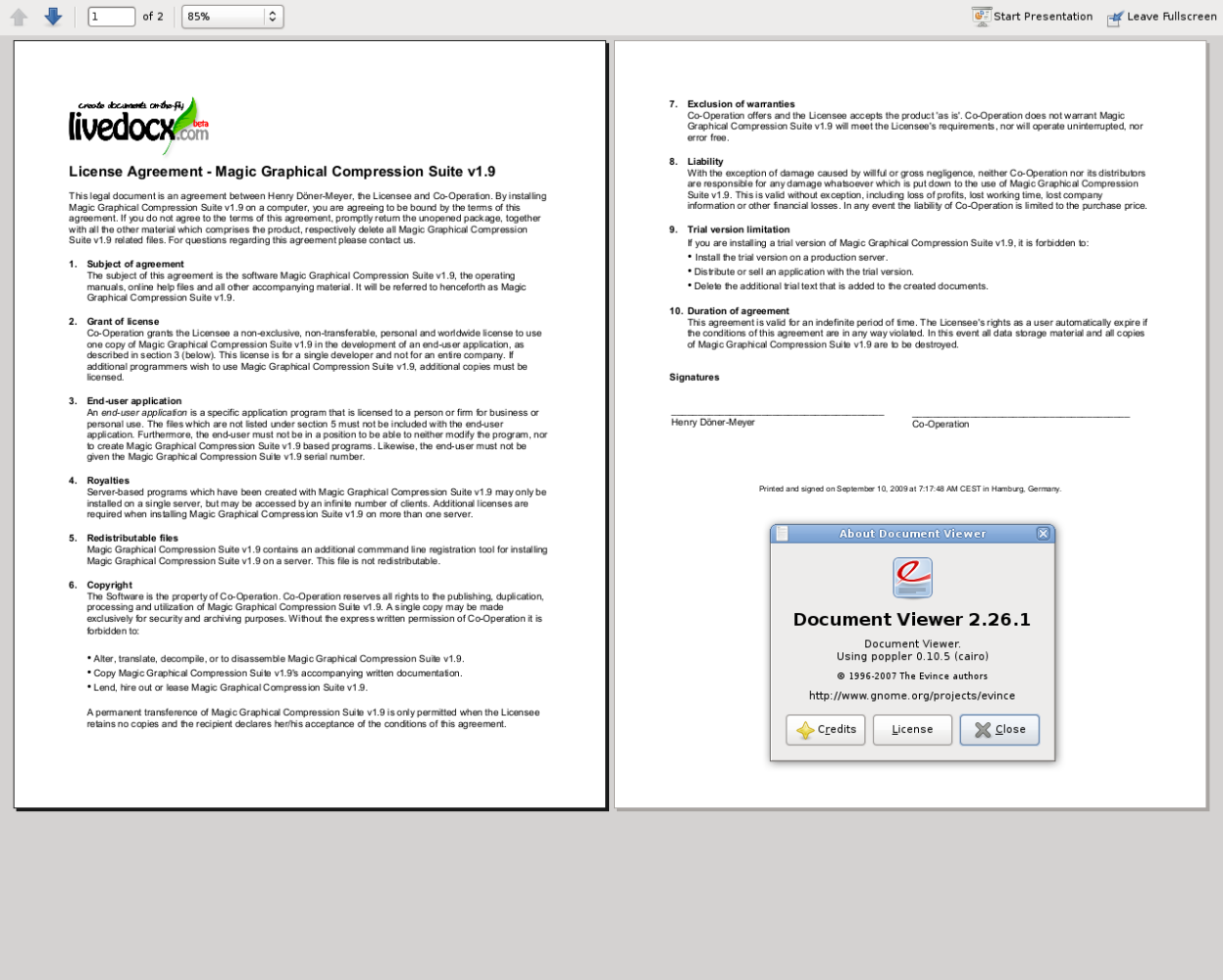
Cropped template in Microsoft Word 2007.

To populate the merge fields in the above cropped screenshot of the template in Microsoft Word, all we have to code is as follows:

```
use ZendService\LiveDocx\MailMerge;

$locale     = Locale::getDefault();
$timestamp = time();

$intlTimeFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::NONE, IntlDateFormatter::SHORT);
```

```php

$intlDateFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_PREMIUM

$mailMerge->setLocalTemplate('license-agreement-template.docx');

$mailMerge->assign('software', 'Magic Graphical Compression Suite v1.9')
          ->assign('licensee', 'Henry Döner-Meyer')
          ->assign('company',  'Co-Operation')
          ->assign('date',     $intlDateFormatter->format($timestamp))
          ->assign('time',     $intlTimeFormatter->format($timestamp))
          ->assign('city',     'Lyon')
          ->assign('country',  'France');

$mailMerge->createDocument();

$document = $mailMerge->retrieveDocument('pdf');

file_put_contents('license-agreement-document.pdf', $document);

unset($mailMerge);
```

The resulting document is written to disk in the file **license-agreement-document.pdf**. This file can now be post-processed, sent via e-mail or simply displayed, as is illustrated below in **Document Viewer 2.26.1** on **Ubuntu 9.04**:



Resulting document as *PDF* in Document Viewer 2.26.1. For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/license-agreement`.

Advanced Mail-Merge

`ZendService\LiveDocx\MailMerge` allows designers to insert any number of text fields into a template. These text fields are populated with data when **createDocument()** is called.

In addition to text fields, it is also possible specify regions of a document, which should be repeated.

For example, in a telephone bill it is necessary to print out a list of all connections, including the destination number, duration and cost of each call. This repeating row functionality can be achieved with so called blocks.

Blocks are simply regions of a document, which are repeated when `createDocument()` is called. In a block any number of **block fields** can be specified.

Blocks consist of two consecutive document targets with a unique name. The following screenshot illustrates these targets and their names in red:

blockStart_block1

Connection: { MERGEFIELD connection_number } { MERGEFIELD connection_duration }	{ MERGEFIELD fee }
Total net	{ MERGEFIELD total_net }

blockEnd_block1

The format of a block is as follows:

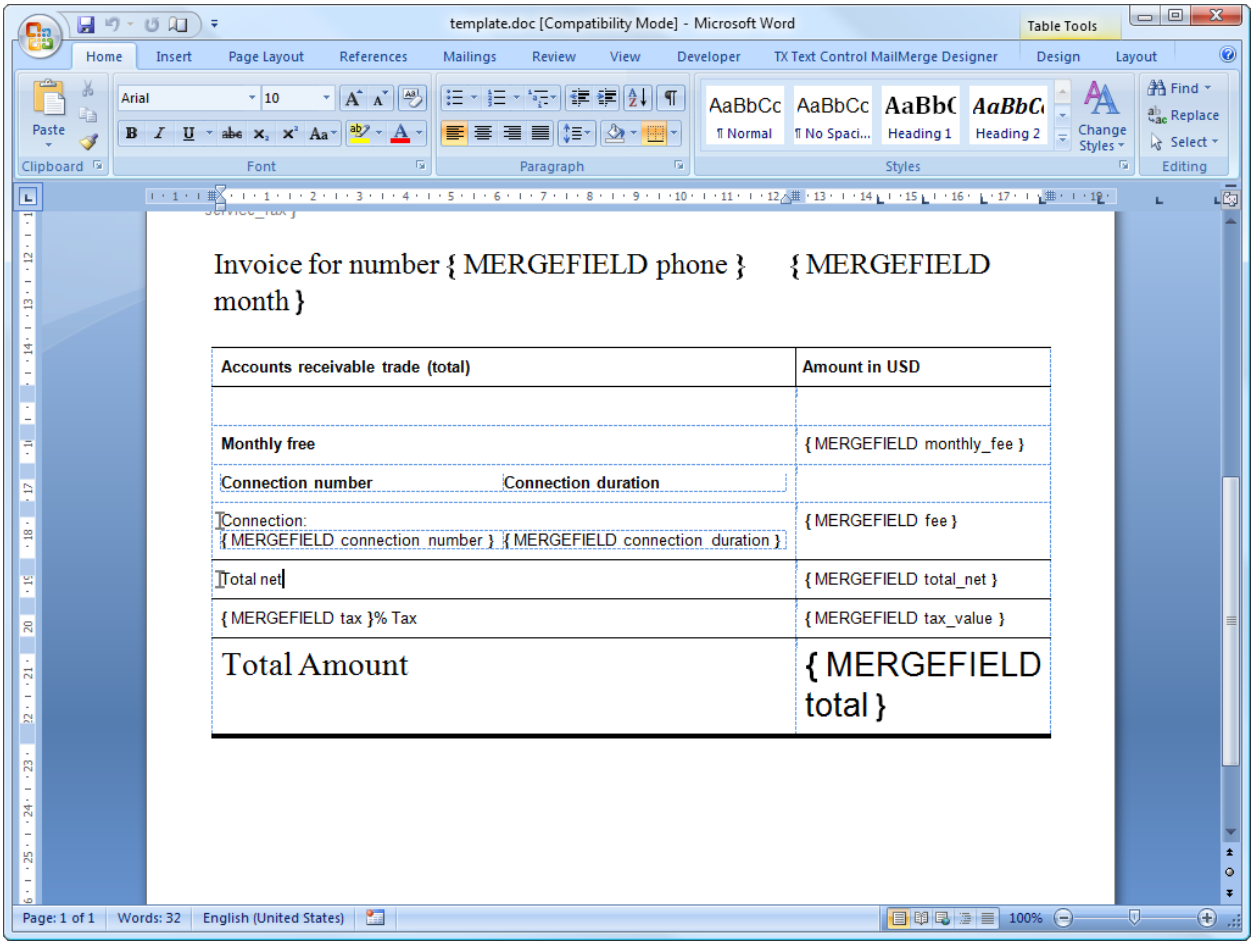
```
blockStart_ + unique name
blockEnd_ + unique name
```

For example:

```
blockStart_block1
blockEnd_block1
```

The content of a block is repeated, until all data assigned in the block fields has been injected into the template. The data for block fields is specified in *PHP* as a multi-assoc array.

The following screenshot of a template in Microsoft Word 2007 shows how block fields are used:



Template, illustrating blocks in Microsoft Word 2007.

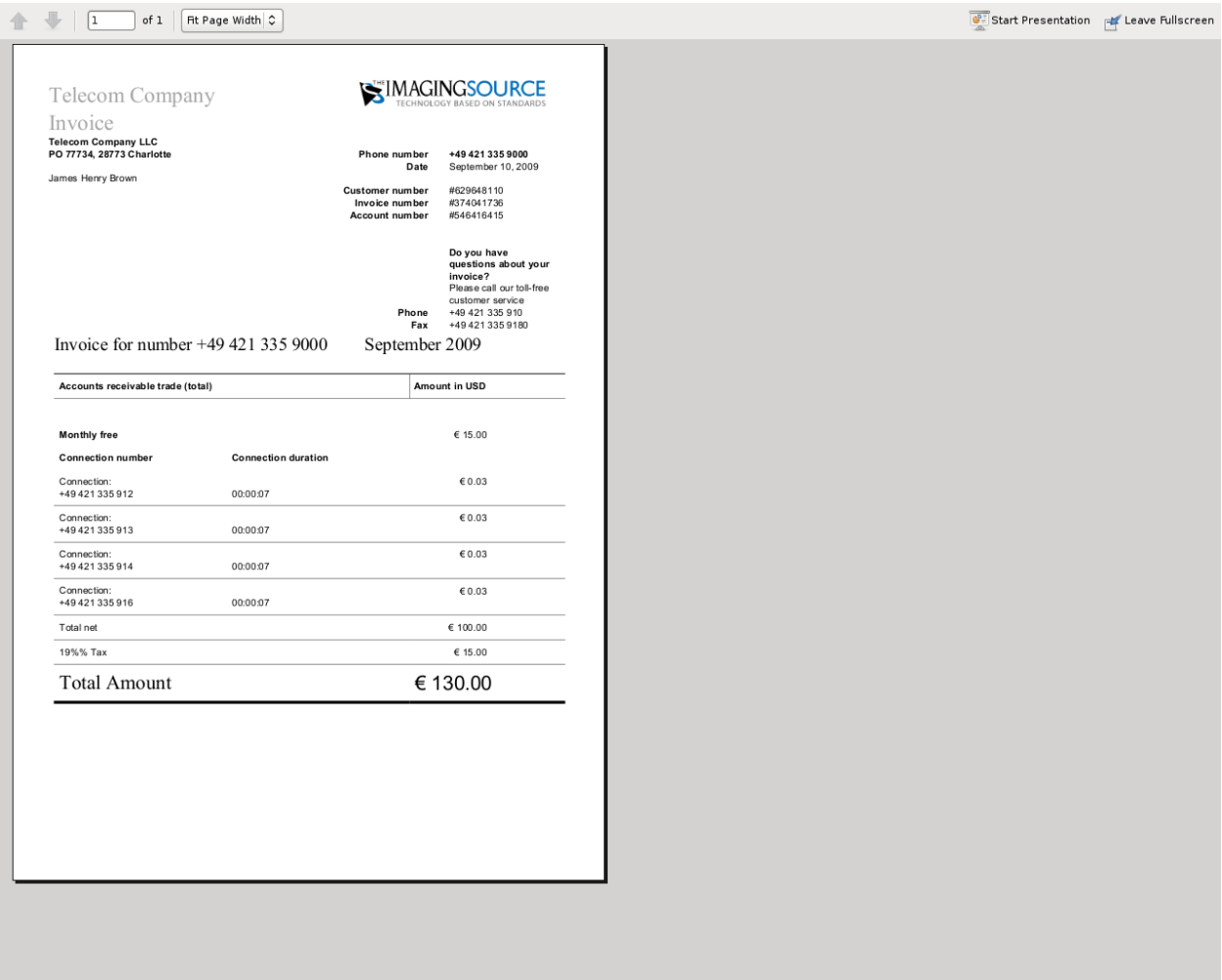
The following code populates the above template with data.

```php
use ZendService\LiveDocx\MailMerge;

$locale    = Locale::getDefault();
$timestamp = time();

$intlDateFormatter1 = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$intlDateFormatter2 = new IntlDateFormatter($locale,
        null, null, null, null, 'LLLL yyyy');

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use MailMerge::SERVICE_PREMIUM

$mailMerge->setLocalTemplate('telephone-bill-template.doc');

$mailMerge->assign('customer_number', sprintf("#%'10s", rand(0,1000000000)))
          ->assign('invoice_number',  sprintf("#%'10s", rand(0,1000000000)))
          ->assign('account_number',  sprintf("#%'10s", rand(0,1000000000)));

$billData = array (
    'phone'          => '+22 (0)333 444 555',
    'date'           => $intlDateFormatter1->format($timestamp),
    'name'           => 'James Henry Brown',
    'service_phone'  => '+22 (0)333 444 559',
    'service_fax'    => '+22 (0)333 444 558',
    'month'          => $intlDateFormatter2->format($timestamp),
    'monthly_fee'    => '15.00',
    'total_net'      => '19.60',
    'tax'            => '19.00',
    'tax_value'      =>  '3.72',
    'total'          => '23.32'
);

$mailMerge->assign($billData);

$billConnections = array(
    array(
        'connection_number'   => '+11 (0)222 333 441',
        'connection_duration' => '00:01:01',
        'fee'                 => '1.15'
    ),
    array(
        'connection_number'   => '+11 (0)222 333 442',
        'connection_duration' => '00:01:02',
        'fee'                 => '1.15'
    ),
    array(
        'connection_number'   => '+11 (0)222 333 443',
        'connection_duration' => '00:01:03',
        'fee'                 => '1.15'
    ),
    array(
        'connection_number'   => '+11 (0)222 333 444',
```

```
        'connection_duration' => '00:01:04',
        'fee'                 => '1.15'
    )
);

$mailMerge->assign('connection', $billConnections);

$mailMerge->createDocument();

$document = $mailMerge->retrieveDocument('pdf');

file_put_contents('telephone-bill-document.pdf', $document);

unset($mailMerge);
```

The data, which is specified in the array `$billConnections` is repeated in the template in the block connection. The keys of the array (`connection_number`, `connection_duration` and `fee`) are the block field names - their data is inserted, one row per iteration.

The resulting document is written to disk in the file **telephone-bill-document.pdf**. This file can now be post-processed, sent via e-mail or simply displayed, as is illustrated below in **Document Viewer 2.26.1** on **Ubuntu 9.04**:



Resulting document as *PDF* in Document Viewer 2.26.1.

You can download the *DOC* template file and the resulting PDF document.

NOTE: blocks may not be nested.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/telephone-bill`.

Merging Image Data into a Template

In addition to assigning textual data, it is also possible to merge image data into a template. The following code populates a conference badge template with the photo `dailemaitre.jpg`, in addition to some textual data.

The first step is to upload the image to the backend service. Once you have done this, you can assign the filename of the image to the template just as you would any other textual data. Note the syntax of the field name containing an image - it must start with `image:`

```
use ZendService\LiveDocx\MailMerge;

$locale    = Locale::getDefault();
$timestamp = time();

$intlDateFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use␣
↪MailMerge::SERVICE_PREMIUM

$photoFilename = __DIR__ . '/dailemaitre.jpg';
$photoFile     = basename($photoFilename);

if (!$mailMerge->imageExists($photoFile)) {          // pass image file *without* path
    $mailMerge->uploadImage($photoFilename);         // pass image file *with* path
}

$mailMerge->setLocalTemplate('conference-pass-template.docx');

$mailMerge->assign('name',         'Daï Lemaitre')
          ->assign('company',      'Megasoft Co-operation')
          ->assign('date',         $intlDateFormatter->format($timestamp))
          ->assign('image:photo', $photoFile);       // pass image file *without* path

$mailMerge->createDocument();

$document = $mailMerge->retrieveDocument('pdf');

file_put_contents('conference-pass-document.pdf', $document);

$mailMerge->deleteImage($photoFilename);

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/conference-pass`.

Generating Bitmaps Image Files

In addition to document file formats, `MailMerge` also allows documents to be saved to a number of image file formats (*BMP*, *GIF*, *JPG*, *PNG* and *TIFF*). Each page of the document is saved to one file.

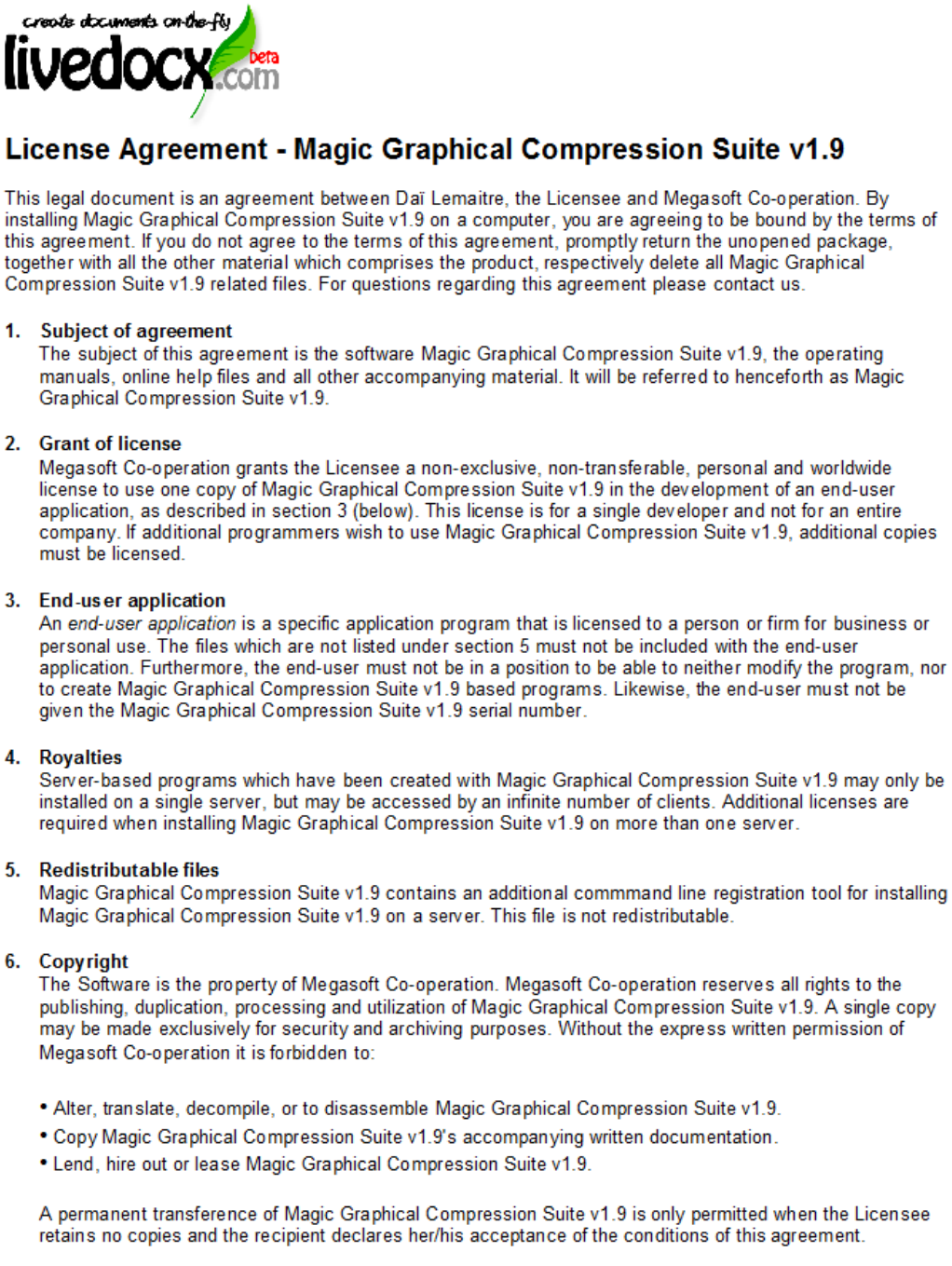
The following sample illustrates the use of `getBitmaps($fromPage, $toPage, $zoomFactor, $format)` and `getAllBitmaps($zoomFactor, $format)`.

`$fromPage` is the lower-bound page number of the page range that should be returned as an image and `$toPage` the upper-bound page number. `$zoomFactor` is the size of the images, as a percent, relative to the original page size. The range of this parameter is 10 to 400. `$format` is the format of the images returned by this method. The supported formats can be obtained by calling `getImageExportFormats()`.

```
use ZendService\LiveDocx\MailMerge;

$locale    = Locale::getDefault();
$timestamp = time();

$intlTimeFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::NONE, IntlDateFormatter::SHORT);

$intlDateFormatter = new IntlDateFormatter($locale,
        IntlDateFormatter::LONG, IntlDateFormatter::NONE);

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use␣
↪MailMerge::SERVICE_PREMIUM

$mailMerge->setLocalTemplate('license-agreement-template.docx');

$mailMerge->assign('software', 'Magic Graphical Compression Suite v1.9')
          ->assign('licensee', 'Henry Döner-Meyer')
          ->assign('company',  'Co-Operation')
          ->assign('date',     $intlDateFormatter->format($timestamp))
          ->assign('time',     $intlTimeFormatter->format($timestamp))
          ->assign('city',     'Lyon')
          ->assign('country',  'France');

$mailMerge->createDocument();

// Get all bitmaps
// (zoomFactor, format)
$bitmaps = $mailMerge->getAllBitmaps(100, 'png');

// Get just bitmaps in specified range
// (fromPage, toPage, zoomFactor, format)
//$bitmaps = $mailMerge->getBitmaps(2, 2, 100, 'png');

foreach ($bitmaps as $pageNumber => $bitmapData) {
    $filename = sprintf('license-agreement-page-%d.png', $pageNumber);
    file_put_contents($filename, $bitmapData);
}

unset($mailMerge);
```

This produces two files (`license-agreement-page-1.png` and `license-agreement-page-2.png`)

and writes them to disk in the same directory as the executable *PHP* file.



create documents on-the-fly
livedocx beta .com

License Agreement - Magic Graphical Compression Suite v1.9

This legal document is an agreement between Daï Lemaitre, the Licensee and Megasoft Co-operation. By installing Magic Graphical Compression Suite v1.9 on a computer, you are agreeing to be bound by the terms of this agreement. If you do not agree to the terms of this agreement, promptly return the unopened package, together with all the other material which comprises the product, respectively delete all Magic Graphical Compression Suite v1.9 related files. For questions regarding this agreement please contact us.

1. **Subject of agreement**
   The subject of this agreement is the software Magic Graphical Compression Suite v1.9, the operating manuals, online help files and all other accompanying material. It will be referred to henceforth as Magic Graphical Compression Suite v1.9.

2. **Grant of license**
   Megasoft Co-operation grants the Licensee a non-exclusive, non-transferable, personal and worldwide license to use one copy of Magic Graphical Compression Suite v1.9 in the development of an end-user application, as described in section 3 (below). This license is for a single developer and not for an entire company. If additional programmers wish to use Magic Graphical Compression Suite v1.9, additional copies must be licensed.

3. **End-user application**
   An *end-user application* is a specific application program that is licensed to a person or firm for business or personal use. The files which are not listed under section 5 must not be included with the end-user application. Furthermore, the end-user must not be in a position to be able to neither modify the program, nor to create Magic Graphical Compression Suite v1.9 based programs. Likewise, the end-user must not be given the Magic Graphical Compression Suite v1.9 serial number.

4. **Royalties**
   Server-based programs which have been created with Magic Graphical Compression Suite v1.9 may only be installed on a single server, but may be accessed by an infinite number of clients. Additional licenses are required when installing Magic Graphical Compression Suite v1.9 on more than one server.

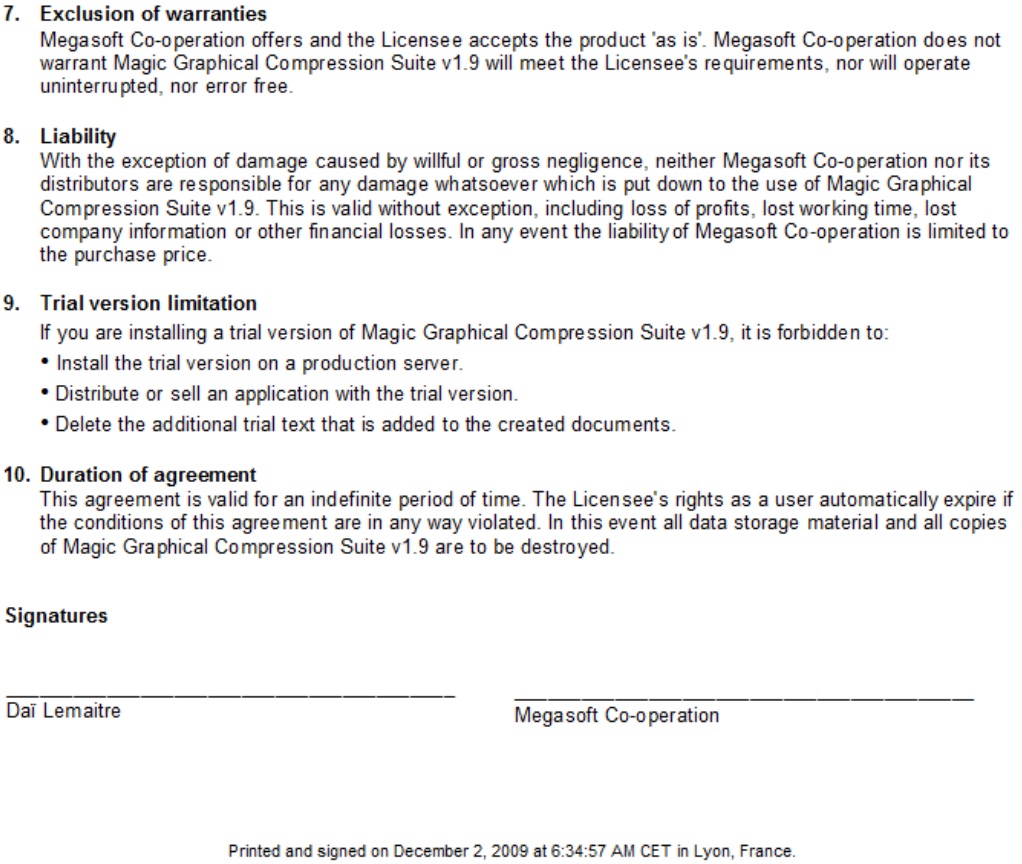
5. **Redistributable files**
   Magic Graphical Compression Suite v1.9 contains an additional commmand line registration tool for installing Magic Graphical Compression Suite v1.9 on a server. This file is not redistributable.

6. **Copyright**
   The Software is the property of Megasoft Co-operation. Megasoft Co-operation reserves all rights to the publishing, duplication, processing and utilization of Magic Graphical Compression Suite v1.9. A single copy may be made exclusively for security and archiving purposes. Without the express written permission of Megasoft Co-operation it is forbidden to:

   - Alter, translate, decompile, or to disassemble Magic Graphical Compression Suite v1.9.
   - Copy Magic Graphical Compression Suite v1.9's accompanying written documentation.
   - Lend, hire out or lease Magic Graphical Compression Suite v1.9.

   A permanent transference of Magic Graphical Compression Suite v1.9 is only permitted when the Licensee retains no copies and the recipient declares her/his acceptance of the conditions of this agreement.

license-agreement-page-1.png.



7. Exclusion of warranties

Megasoft Co-operation offers and the Licensee accepts the product 'as is'. Megasoft Co-operation does not warrant Magic Graphical Compression Suite v1.9 will meet the Licensee's requirements, nor will operate uninterrupted, nor error free.

8. Liability

With the exception of damage caused by willful or gross negligence, neither Megasoft Co-operation nor its distributors are responsible for any damage whatsoever which is put down to the use of Magic Graphical Compression Suite v1.9. This is valid without exception, including loss of profits, lost working time, lost company information or other financial losses. In any event the liability of Megasoft Co-operation is limited to the purchase price.

9. Trial version limitation

If you are installing a trial version of Magic Graphical Compression Suite v1.9, it is forbidden to:

- Install the trial version on a production server.
- Distribute or sell an application with the trial version.
- Delete the additional trial text that is added to the created documents.

10. Duration of agreement

This agreement is valid for an indefinite period of time. The Licensee's rights as a user automatically expire if the conditions of this agreement are in any way violated. In this event all data storage material and all copies of Magic Graphical Compression Suite v1.9 are to be destroyed.

Signatures

_____ _____

Daï Lemaitre Megasoft Co-operation

Printed and signed on December 2, 2009 at 6:34:57 AM CET in Lyon, France.

license-agreement-page-2.png. For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/bitmaps`.

Local vs. Remote Templates

Templates can be stored **locally**, on the client machine, or **remotely**, by LiveDocx. There are advantages and disadvantages to each approach.

In the case that a template is stored locally, it must be transfered from the client to LiveDocx on every request. If the content of the template rarely changes, this approach is inefficient. Similarly, if the template is several megabytes in size, it may take considerable time to transfer it to LiveDocx. Local template are useful in situations in which the content of the template is constantly changing.

The following code illustrates how to use a local template.

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use␣
↪MailMerge::SERVICE_PREMIUM

$mailMerge->setLocalTemplate('template.docx');

// assign data and create document

unset($mailMerge);
```

In the case that a template is stored remotely, it is uploaded once to LiveDocx and then simply referenced on all subsequent requests. Obviously, this is much quicker than using a local template, as the template does not have to be transfered on every request. For speed critical applications, it is recommended to use the remote template method.

The following code illustrates how to upload a template to the server:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use␣
↪MailMerge::SERVICE_PREMIUM

$mailMerge->uploadTemplate('template.docx');

unset($mailMerge);
```

The following code illustrates how to reference the remotely stored template on all subsequent requests:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use␣
↪MailMerge::SERVICE_PREMIUM

$mailMerge->setRemoteTemplate('template.docx');

```

```
// assign data and create document

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/templates`.

Getting Information

`ZendService\LiveDocx\MailMerge` provides a number of methods to get information on field names, available fonts and supported formats.

Get Array of Field Names in Template

The following code returns and displays an array of all field names in the specified template. This functionality is useful, in the case that you create an application, in which an end-user can update a template.

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use
↪MailMerge::SERVICE_PREMIUM

$templateName = 'template-1-text-field.docx';
$mailMerge->setLocalTemplate($templateName);

$fieldNames = $mailMerge->getFieldNames();
foreach ($fieldNames as $fieldName) {
    printf('- %s%s', $fieldName, PHP_EOL);
}

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/template-info`.

Get Array of Block Field Names in Template

The following code returns and displays an array of all block field names in the specified template. This functionality is useful, in the case that you create an application, in which an end-user can update a template. Before such templates can be populated, it is necessary to find out the names of the contained block fields.

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use
↪MailMerge::SERVICE_PREMIUM
```

```
8
9  $templateName = 'template-block-fields.doc';
10 $mailMerge->setLocalTemplate($templateName);
11
12 $blockNames = $mailMerge->getBlockNames();
13 foreach ($blockNames as $blockName) {
14     $blockFieldNames = $mailMerge->getBlockFieldNames($blockName);
15     foreach ($blockFieldNames as $blockFieldName) {
16         printf('- %s::%s%s', $blockName, $blockFieldName, PHP_EOL);
17     }
18 }
19
20 unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/template-info`.

Get Array of Fonts Installed on Server

The following code returns and displays an array of all fonts installed on the server. You can use this method to present a list of fonts which may be used in a template. It is important to inform the end-user about the fonts installed on the server, as only these fonts may be used in a template. In the case that a template contains fonts, which are not available on the server, font-substitution will take place. This may lead to undesirable results.

```
1  use ZendService\LiveDocx\MailMerge;
2  use Zend\Debug\Debug;
3
4  $mailMerge = new MailMerge();
5
6  $mailMerge->setUsername('myUsername')
7            ->setPassword('myPassword')
8            ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use␣
   ↪MailMerge::SERVICE_PREMIUM
9
10 Debug::dump($mailMerge->getFontNames());
11
12 unset($mailMerge);
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend\Cache\Cache`- this will considerably speed up your application.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-fonts`.

Get Array of Supported Template File Formats

The following code returns and displays an array of all supported template file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the input format of the documentation generation process.

```
1  use ZendService\LiveDocx\MailMerge;
2  use Zend\Debug\Debug;
3
4  $mailMerge = new MailMerge()
5
```

```
$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use
↪MailMerge::SERVICE_PREMIUM

Debug::dump($mailMerge->getTemplateFormats());

unset($mailMerge);
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend\Cache\Cache`- this will considerably speed up your application.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-formats`.

Get Array of Supported Document File Formats

The following code returns and displays an array of all supported document file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the output format of the documentation generation process.

```
use ZendService\LiveDocx\MailMerge;
use Zend\Debug\Debug;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use
↪MailMerge::SERVICE_PREMIUM

Debug::dump($mailMerge->getDocumentFormats());

unset($mailMerge);
```

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-formats`.

Get Array of Supported Image File Formats

The following code returns and displays an array of all supported image file formats. This method is particularly useful in the case that a combo list should be displayed that allows the end-user to select the output format of the documentation generation process.

```
use ZendService\LiveDocx\MailMerge;
use Zend\Debug\Debug;

$mailMerge = new MailMerge();

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);  // for LiveDocx Premium, use
↪MailMerge::SERVICE_PREMIUM

Debug::dump($mailMerge->getImageExportFormats());
```

```
11
12  unset($mailMerge);
```

NOTE: As the return value of this method changes very infrequently, it is highly recommended to use a cache, such as `Zend\Cache\Cache`- this will considerably speed up your application.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/supported-formats`.

Upgrading From LiveDocx Free to LiveDocx Premium

LiveDocx Free is provided by *Text Control GmbH* completely free for charge. It is free for all to use in an unlimited number of applications. However, there are times when you may like to update to LiveDocx Premium. For example, you need to generate a very large number of documents concurrently, or your application requires documents to be created faster than LiveDocx Free permits. For such scenarios, *Text Control GmbH* offers LiveDocx Premium, a paid service with a number of benefits. For an overview of the benefits, please take a look at LiveDocx pricing.

This section of the manual offers a technical overview of how to upgrade from LiveDocx Free to LiveDocx Premium.

All you have to do, is make a very small change to the code that runs with LiveDocx Free. Your instantiation and initialization of LiveDocx Free probably looks as follows:

```
1   use ZendService\LiveDocx\MailMerge;
2
3   $mailMerge = new MailMerge()
4
5   $mailMerge->setUsername('myUsername')
6             ->setPassword('myPassword')
7             ->setService (MailMerge::SERVICE_FREE);
8
9   // rest of your application here
10
11  unset($mailMerge);
```

To use LiveDocx Premium, you simply need to change the service value from `MailMerge::SERVICE_FREE` to `MailMerge::SERVICE_PREMIUM`, and set the username and password assigned to you for Livedocx Premium. This may, or may not be the same as the credentials for LiveDocx Free. For example:

```
1   use ZendService\LiveDocx\MailMerge;
2
3   $mailMerge = new MailMerge()
4
5   $mailMerge->setUsername('myPremiumUsername')
6             ->setPassword('myPremiumPassword')
7             ->setService (MailMerge::SERVICE_PREMIUM);
8
9   // rest of your application here
10
11  unset($mailMerge);
```

And that is all there is to it. The assignment of the premium WSDL to the component is handled internally and automatically. You are now using LiveDocx Premium.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/instantiation`.

Upgrading From LiveDocx Free or LiveDocx Premium to LiveDocx Fully Licensed

LiveDocx Free and Livedocx Premium are provided by *Text Control GmbH* as a service. They are addressed over the Internet. However, for certain applications, for example, ones that process very sensitive data (banking, health or financial), you may not want to send your data across the Internet to a third party service, regardless of the SSL encryption that both LiveDocx Free and Livedocx Premium offer as standard. For such scenarios, you can license LiveDocx and install an entire LiveDocx server in your own network. As such, you completely control the flow of data between your application and the backend LiveDocx server. For an overview of the benefits of LiveDocx Fully Licensed, please take a look at LiveDocx pricing.

This section of the manual offers a technical overview of how to upgrade from LiveDocx Free or LiveDocx Premium to LiveDocx Fully Licensed.

All you have to do, is make a very small change to the code that runs with LiveDocx Free or LiveDocx Premium. Your instantiation and initialization of LiveDocx Free or LiveDocx Premium probably looks as follows:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge()

$mailMerge->setUsername('myUsername')
          ->setPassword('myPassword')
          ->setService (MailMerge::SERVICE_FREE);
       // or
       // ->setService (MailMerge::SERVICE_PREMIUM);

// rest of your application here

unset($mailMerge);
```

To use LiveDocx Fully Licensed, you simply need to set the WSDL of the backend LiveDocx server in your own network. You can do this as follows:

```
use ZendService\LiveDocx\MailMerge;

$mailMerge = new MailMerge()

$mailMerge->setUsername('myFullyLicensedUsername')
          ->setPassword('myFullyLicensedPassword')
          ->setWsdl    ('http://api.example.com/2.1/mailmerge.asmx?wsdl');

// rest of your application here

unset($mailMerge);
```

And that is all there is to it. You are now using LiveDocx Fully Licensed.

For executable demo applications, which illustrate the above, please take a look at `/demos/ZendService/LiveDocx/MailMerge/instantiation`.

CHAPTER 184

Copyright Information

The following copyrights are applicable to portions of Zend Framework.

Copyright © 2005-Zend Technologies Inc. (http://www.zend.com)

CHAPTER 185

Introduction to Zend Framework 2

CHAPTER 186

User Guide

The user guide is provided to take you through a non-trivial example, showing you various techniques and features of the framework in order to build an application.

CHAPTER 187

Learning Zend Framework 2

CHAPTER 188

Zend Framework 2 Reference

Zend\Authentication

- *Introduction*
- *Database Table Authentication*
- *Digest Authentication*
- *HTTP Authentication Adapter*
- *LDAP Authentication*

Zend\Barcode

- *Introduction*
- *Barcode creation using Zend\Barcode\Barcode class*
- *Zend\Barcode\Barcode Objects*
- *Zend\Barcode Renderers*

Zend\Cache

- *Zend\Cache\Storage\Adapter*
- *Zend\Cache\Storage\Capabilities*
- *Zend\Cache\Storage\Plugin*
- *Zend\Cache\Pattern*

Zend\Captcha

- *Introduction*
- *Captcha Operation*
- *CAPTCHA Adapters*

Zend\Console

- modules/zend.console.introduction
- modules/zend.console.routes
- modules/zend.console.modules
- modules/zend.console.controllers
- modules/zend.console.adapter
- modules/zend.console.prompts

Zend\Config

- *Introduction*
- *Theory of Operation*
- *Zend\Config\Reader*
- *Zend\Config\Writer*
- *Zend\Config\Processor*

Zend\Crypt

- *Introduction*
- *Encrypt/decrypt using block ciphers*
- *Key derivation function*
- *Password secure storing*
- *Public key cryptography*

Zend\Db

- *Zend\Db\Adapter*
- *Zend\Db\ResultSet*
- *Zend\Db\Sql*
- *Zend\Db\TableGateway*

- *Zend\Db\RowGateway*
- *Zend\Db\Metadata*

Zend\Di

- *Introduction to Zend\Di*
- *Zend\Di Quickstart*
- *Zend\Di Definition*
- *Zend\Di InstanceManager*
- *Zend\Di Configuration*
- *Zend\Di Debugging & Complex Use Cases*

Zend\Dom

- *Introduction*
- *Zend\Dom\Query*

Zend\EventManager

- *The EventManager*

Zend\Form

- *Introduction to Zend\Form*
- *Form Quick Start*
- *Form Collections*
- *Form Elements*
- *Form View Helpers*

Zend\Http

- *Zend\Http*
- *Zend\Http\Request*
- *Zend\Http\Response*
- *Zend\Http\Headers And The Various Header Classes*
- *Zend_Http_Cookie and Zend_Http_CookieJar*
- *Zend\Http\Client*

- *Zend_Http_Client - Connection Adapters*
- *Zend_Http_Client - Advanced Usage*

Zend\I18n

- *Translating*
- *I18n View Helpers*
- *I18n Filters*

Zend\InputFilter

- *Introduction*

Zend\Ldap

- *Introduction*
- *API overview*
- *Usage Scenarios*
- *Tools*
- *Object oriented access to the LDAP tree using Zend\Ldap\Node*
- *Getting information from the LDAP server*
- *Serializing LDAP data to and from LDIF*

Zend\Loader

- *The AutoloaderFactory*
- *The PluginClassLoader*
- *The ShortNameLocator Interface*
- *The PluginClassLocator interface*
- *The SplAutoloader Interface*
- *The ClassMapAutoloader*
- *The StandardAutoloader*
- *The Class Map Generator utility: bin/classmap_generator.php*
- *The PrefixPathLoader*
- *The PrefixPathMapper Interface*

Zend\Log

- *Overview*
- *Writers*
- *Filters*
- *Formatters*

Zend\Mail

- *Zend\Mail\Message*
- *Zend\Mail\Transport*
- *Zend\Mail\Transport\SmtpOptions*
- *Zend\Mail\Transport\FileOptions*

Zend\Math

- *Introduction*

Zend\ModuleManager

- *Introduction to the Module System*
- *The Module Manager*
- *The Module Class*
- *The Module Autoloader*
- *Best Practices when Creating Modules*

Zend\Mvc

- *Introduction to the MVC Layer*
- *Quick Start*
- *Default Services*
- *Routing*
- *The MvcEvent*
- *Available Controllers*
- *Controller Plugins*
- *Examples*

Zend\Permissions\Acl

- *Introduction*
- *Refining Access Controls*
- *Advanced Usage*

Zend\ServiceManager

- *Zend\ServiceManager*
- *Zend\ServiceManager Quick Start*

Zend\Stdlib

- *Zend\Stdlib\Hydrator*

Zend\Uri

- *Zend\Uri*

Zend\Validator

- *Introduction*
- *Standard Validation Classes*
- *Validator Chains*
- *Writing Validators*
- *Validation Messages*

Zend\View

- *Zend\View Quick Start*
- *The PhpRenderer*
- *PhpRenderer View Scripts*
- *View Helpers*

Zend\XmlRpc

CHAPTER 189

Services for Zend Framework 2 Reference

ZendService\LiveDocx

CHAPTER 190

Copyright

CHAPTER 191

Indices and tables

- genindex
- modindex
- search