RAW RAW RAW

RAW RAW RAW

Quick Answers to common problems

Zend Framework 2.0 Cookbook: RAW

Over 80 highly focused practical development recipes to maximize your use of the Zend Framework

Nick Belhomme

[PACKT] open source*
PUBLISHING community experience distilled

Zend Framwork 2.0 Cookbook

RAW Book

Over 80 highly focused practical development recipes to maximize the Zend Framework.

Nick Belhomme

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Zend Framwork 2.0 Cookbook

Current RAW Publication: May 2011

RAW Production Reference: 3050511

Published by Packt Publishing Ltd.
32 Lincoln Road
Olton
Birmingham, B27 6PA, UK.

ISBN 1-904812-30-5

`www.packtpub.com`

About the Author

Nick Belhomme is a domain-driven design enthusiast and his consulting focus is on the design and construction of enterprise applications. With more than a decade in professional web application development / management, he has contributed in a number of ways to the PHP Community. He works as an independent contractor and consults PHP teams at various major Belgian Enterprises. He is an international conference speaker, authored several articles and when he is not sharing his knowledge he likes to travel, take hikes with his lovely wife and loves nature.

Table of Contents

Preface

Welcome to Zend Framework 2.0 Cookbook, the RAW edition. A RAW (Read As we Write) book contains all the material written for the book so far, but available for you right now, before it's finished. As the author writes more, you will be invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW!

Zend Framework 2.0 Cookbook offers a practical guide to the inner workings and hidden options of many of the Zend Framework components. Each component that is covered in the book provides usage and implementation information, as well as detailed code examples. You will learn how these important components fit into the software development process using the MVC design pattern implementation of Zend Framework and how you can leverage them to quickly solve your own design problems most efficiently.

What's in This RAW Book

In this RAW book, you will find these chapters:

In the first chapter the cookbook focuses on how to setup a Zend Framework application so you can start using it right away.

Chapter 2 gives the recipes needed for you to use the View from the Model-View-Controller implementation in Zend Framework and how to link the pages you have created together using the routing system and appropriate action and view helpers.

Chapter 3 will tell you how to optimize your application. You will be shown how to create a model, build configuration files for less coding and easy reuse. Creating and using cache to optimize for performance and build plug and play logic which you potentially can drop in each one of your ZF projects.

Chapter 4 shows that data can be stored and made available through different systems. Databases are one system, the file system or web services like SOAP and REST are others. This chapter focuses on handling databases and abstraction of the persistence layer.

Chapter 5 is all about handling Javascript and Ajax. It will give you guidance on how to integrate them into your Zend Framework application. It will also go deeper into the appropriate view helpers already discussed in chapter 2.

Chapter 6 will take a look at web services from a provider and a consumer point of view. It shows recipes on how to consume and make available SOAP, AMF and REST services with the use of various Zend Framework components.

Chapter 7 will cover debugging, exception handling and unit testing and will give you the foundation needed to build powerful and stable Zend Framework applications.

Chapter 8 will provide recipes on how to internationalize and localize your application. Thanks to this chapter you will be able to create internationalized applications and know which of the framework components are I18n and L10n aware.

Chapter 9 focuses on sending and receiving mail from within your application, which is crucial for reporting and customer service.

Chapter 10 will cover authentication - the process of verifying someone's identity - and authorization - the process of deciding whether that identity has access to a specific resource. Because authenticating and authorizing are two different processes Zend Framework provides you with two separate components and both will be covered in depth.

Chapter 11, Building Forms

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "It is assigned a `jQuery` object containing the results of the `$('.sticky')` query."

A block of code will be set as follows:

```
StickyRotate.init = function() {
  var stickies = $(".sticky");

  // If we don't have enough, stop immediately.
  if (stickies.size() <= 1 || $('#node-form').size() > 0) {
    return;
  }
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be made bold:

```
StickyRotate.init = function() {
  var stickies = $(".sticky");

  // If we don't have enough, stop immediately.
  if (stickies.size() <= 1 || $('#node-form').size() > 0) {
    return;
  }
```

Any command-line input and output is written as follows:

```
# cp /usr/src/asterisk-addons/configs/cdr_mysql.conf.sample
     /etc/asterisk/cdr_mysql.conf
```

New terms and important words are introduced in a bold-type font. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "clicking the Next button moves you to the next screen".

[Warnings or important notes appear in a box like this.]

[Tips and tricks appear like this.]

What Is a RAW Book?

Buying a Packt RAW book allows you to access Packt books before they're published. A RAW (Read As we Write) book is an eBook available for immediate download, and containing all the material written for the book so far.

As the author writes more, you are invited to download the new material and continue reading, and learning. Chapters in a RAW book are not "work in progress", they are drafts ready for you to read, use, and learn from. They are not the finished article of course—they are RAW! With a RAW book, you get immediate access, and the opportunity to participate in the development of the book, making sure that your voice is heard to get the kind of book that you want.

Is a RAW Book a Proper Book?

Yes, but it's just not all there yet! RAW chapters will be released as soon as we are happy for them to go into your book—we want you to have material that you can read and use straightaway. However, they will not have been through the full editorial process yet. You are receiving RAW content, available as soon as it written. If you find errors or mistakes in the book, or you think there are things that could be done better, you can contact us and we will make sure to get these things right before the final version is published.

When Do Chapters Become Available?

As soon as a chapter has been written and we are happy for it go into the RAW book, the new chapter will be added into the RAW eBook in your account. You will be notified that another chapter has become available and be invited to download it from your account. eBooks are licensed to you only; however, you are entitled to download them as often as you like and on as many different computers as you wish.

How Do I Know When New Chapters Are Released?

When new chapters are released all RAW customers will be notified by email with instructions on how to download their new eBook. Packt will also update the book's page on its website with a list of the available chapters.

Where Do I Get the Book From?

You download your RAW book much in the same way as any Packt eBook. In the download area of your Packt account, you will have a link to download the RAW book.

What Happens If I Have Problems with My RAW Book?

You are a Packt customer and as such, will be able to contact our dedicated Customer Service team. Therefore, if you experience any problems opening or downloading your RAW book, contact `service@packtpub.com` and they will reply to you quickly and courteously as they would to any Packt customer.

Is There Source Code Available During the RAW Phase?

Any source code for the RAW book can be downloaded from the Support page of our website (`http://www.packtpub.com/support`). Simply select the book from the list.

How Do I Post Feedback and Errata for a RAW Title?

If you find mistakes in this book, or things that you can think can be done better, let us know. You can contact us directly at `rawbooks@packtpub.com` to discuss any concerns you may have with the book.

Feedback from our readers is always welcome. Let us know what you think about this book, what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of. To send us general feedback, simply drop an email to `feedback@packtpub.com`, making sure to mention the book title in the subject of your message.

1
Setting up a Zend Framework Application

In this chapter, we will cover:

- Using Zend Framework 2.0
- Download a copy of Zend Framework
- Getting your system ready to work with a ZF project
- Creating your directory structure
- Setting up your bootstrap
- Creating your .htaccess file
- Setting up your application.ini
- Your first controller and action \Zend\Controller\Action
- Setting up the vhost and firing up the website
- Launching your first web page
- Using \Zend\Tool for easy project creation
- Getting input from the user with \Zend\Controller\Request\Http
- Filtering input with \Zend\Filter\StaticFilter
- Validating input with \Zend\Validator\StaticValidator
- Doing Filtering and Validating in one batch with \Zend\Filter\InputFilter
- How to set headers per action

Introduction

This chapter will show you the basics on how to implement a Zend Framework Project. To have a project running in no time, I suggest you implement the first eight recipes of this chapter. After that you can pick the recipe suited for your needs.

Using Zend Framework 2.0

If you used previous versions of Zend Framework a lot has been changed in the 2.0 branch. Components have been renamed, changed and/or deleted. This however should not stop you from using the latest version. Bug fixes and contributions are added all the time to this framework and when you decide not to update your version you have a potential stability or worse security leak. As with all open third party libraries updating is necessary. Migrating between minor versions is often unnoticed and requires no code change. But migrating between major versions as is with ZF1.* to ZF2.* you might expect some modifications needed to be done. This recipe will help you making the migration as smooth as possible.

Getting ready

Backup your project before making any modification.

How to do it...

One baby step at the time.

There's more...

More Info Section 1

More Info Section 2

More Info Section 3

See also

Obtaining the latest stable version of PHP or at the minimum 5.3: `http://www.php.net/`

Download a copy of Zend Framework

This recipe will teach you how to download the latest version of Zend Framework.

Getting ready

Zend Framework 2.* series requires PHP version 5.3 is running on your system.

How to do it...

1. Point your browser to `http://framework.zend.com/download/latest`
2. Click on free download "Zend Framework Minimal Package"
3. Choose your distribution Linux or Windows
4. Click on the format download link for "Zend Framework Minimal Package"
5. Complete the registration form or login
6. Automatic download should begin
7. Extract the file and place the Zend folder on your disk where you would like the library to be available. Probably to a path on your system where other libraries reside like: Library/Zend

How it works...

Downloading Zend Framework is straight forward. You download the code, place it in your favourite library folder which is included in the php `include_path` and you are done.

The big advantage of this complete download is that you get a lot of extras such as documentation but also translations for the validators amongst others.

There's more...

Using Pear

If you have `pear` enabled you can also use it for easy installation of the framework. This package is typically built within a day of an official Zend Framework release.

```
pear channel-discover pear.zfcampus.org
pear install zfcampus/zf
```

Using Subversion (svn)

If you already have `subversion` (svn) running you might want to consider getting the latest build from the publicly available repository. You can do a checkout from `http://framework.zend.com/svn/framework/standard/trunk/library` to have the minimum package available.

Using the repository will enable you to upgrade your version more often than "download" releases occur. On a production server you might want to consider exporting the repository to have a working copy without the `.svn` directories.

Using Git

As of Zend Framework 2.0 the standard version control system chosen is Git.

1. Setup a `https://github.com/signup/free` account, if you haven't yet
2. click on the fork icon at `http://github.com/zendframework/zf2`
3. open your local git command line and clone your fork locally:
4. `% git clone git@github.com:username/zf2.git`
5. enter your fork:
6. `% cd zf2`
7. use your ZF JIRA email address to configure git:
8. `% git config user.email <your email address>`
9. and last but not least add a remote to the GitHub mirror so you can keep your fork up-to-date:
10. `% git remote add zf2 git://github.com/zendframework/zf2.git`
11. `% git fetch zf2`

See also

- Recipe: Getting your system ready to work with a ZF project
- Installing Pear on your system: `http://pear.php.net/`
- Installing Subversion `http://subversion.apache.org/`
- Installing Git: `http://git-scm.com/`

Getting your system ready to work with a ZF project

For Zend Framework to be available to your project it has to be added to your php `include_path`. Also your application should be able to reach scripts outside your document root (this is your public folder). Sometimes your hosting provider has `open_basedir` in effect to only allow php to have access to your public and the temp folder. This recipe will show you how to change all that.

Getting ready

Have access to your Apache vhost configuration, if not jump to the "there's more" section for alternatives.

How to do it...

Create a virtual host for the website and set the document root directly to the public directory. An Apache Virtual host configuration could look like this:

```
<VirtualHost *:80>
    ServerName yourproject
    DocumentRoot "/path/to/yourproject/public"
    <Directory "/path/to/yourproject/public">
        php_value include_path "path/to/your/dir/containing/the/Zend/
dir"
        php_admin_value open_basedir "/path/to/your/projectname:path/
to/your/dir/containing/the/Zend/dir:/tmp"
        Options Indexes MultiViews FollowSymLinks
        AllowOverride All
        Order allow,deny
        Allow from all
    </Directory>
</VirtualHost>
```

How it works...

I will cover the vhost directives or settings related to a Zend Framework project,the `DocumentRoot` should point to the public directory of your application.

Create a `Directory` directive to enclose a group of directives that apply only to the project file-system directory and sub-directories.

`php_value include_path` will change the default include path to the path in which the Zend directory is installed.

`php_admin_value open_basedir` is only needed when you notice that your application cannot have access to scripts outside your document root. You should specify one level up from your public directory, the directory for the Zend library and a temporary directory. Admin directives can only be used in the httpd.conf or vhost file. Non-admins can also be used in `.htaccess`.

We need to make sure `.htaccess` files can be parsed. This is accomplished by setting `AllowOverride All`.

See also

- Recipe: Creating your `.htaccess`
- Understanding the Apache directives: `http://httpd.apache.org/docs/2.2/mod/core.html`

Creating your directory structure

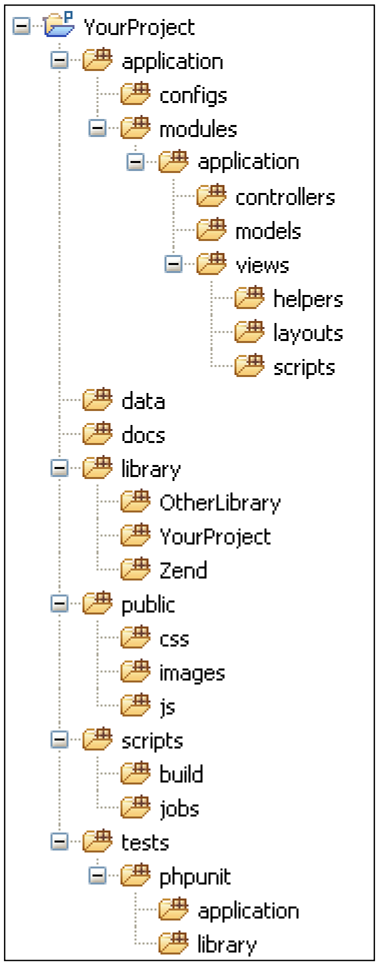
Because we want to use the Zend Framework as a full stack MVC solution we need to have a strict separation of code. On the most basic level this is accomplished by usage of the file system possibilities. A basic Zend Framework project should have the directories in this recipe available.

Getting ready

Make sure you have write permissions to the file system.

How to do it...

Create a directory structure as shown in the next screenshot.



```
YourProject
    application
        configs
        modules
            application
                controllers
                models
                views
                    helpers
                    layouts
                    scripts
    data
    docs
    library
        OtherLibrary
        YourProject
        Zend
    public
        css
        images
        js
    scripts
        build
        jobs
    tests
        phpunit
            application
            library
```

How it works...

Each directory has its specific use. We will take a look at each folder and explain the usage.

Application: This directory contains all application specific code.

Configs: The configuration directory for the entire application.

Modules: Because we might have the need for modules later on in every application, we already make full use of it and house all MVC logic in logically organized groups. By default one module is used `application`, in the past this was `default`. Separating the MVC from the default module outside of the module directory like often advocated only makes it less uniform, understandable and thus maintainable.

Controllers, models and views: These directories serve as the default MVC (`Model, View, Controller`) directories of a specific module.

Data: We want to store volatile and temporary data like cache, logs, session and upload data in this directory. It functions as an easy to find logically grouped data block, which isn't useful to put in a version control system. Cache would be a good place to put here.

Docs: Store documentation which is either generated or directly authored in this directory.

Library: Here we place all common libraries which are used by the application itself. As you can see it also contains the common library of the application itself under its own namespace. This entire directory should be on the PHP `include_path`. (We will later on show you how to do this through the `application.ini` file)

Public: The document root of your application. It will house all publicly available resources like images, js, css and so on.. It will also include an `index.php` which will dispatch all requests filtered by the `.htaccess` to the front controller of the MVC.

Scripts: As the name suggests this directory has the purpose to contain all maintenance scripts like command line, cron, patches amongst others which will guarantee a good functioning of the application. It is also the place to put your **continuous integration** build.

Tests: This directory contains all application tests, like Unit Tests.

Setting up your bootstrap

Zend Framework is an MVC implementation and all requests should be dispatched towards the Front Controller which will handle the request, route, dispatch and response objects. For this to work all php requests must go through a single point of entry namely your `bootstrap`. In this recipe you will see on how to set up a basic bootstrap.

How to do it...

Create a file called `index.php` into your public directory. (This single point of entry is called your bootstrap) and implement the file with the following code:

```
<?php
// Define path to project directory
defined('PROJECT_PATH')
    || define('PROJECT_PATH', realpath(dirname(__FILE__) .'/../'));
// Define path to application directory
defined('APPLICATION_PATH')
    || define('APPLICATION_PATH',
PROJECT_PATH.'/application');
// Define path to public directory
defined('PUBLIC_PATH')
```

```
    || define('PUBLIC_PATH', PROJECT_PATH.'/public');
// Define application environment
defined('APPLICATION_ENV')
    || define('APPLICATION_ENV', (getenv('APPLICATION_ENV') ?
getenv('APPLICATION_ENV') : 'production'));
/** \Zend\Application\Application */
require_once 'Zend/Application/Application.php';
// Create application, bootstrap, and run
$application = new Zend\Application\Application(
    APPLICATION_ENV,
    APPLICATION_PATH . '/configs/application.ini'
);
$application->bootstrap()
            ->run();
```

Create a secondary Bootstrap file application/Bootstrap.php which can function later as a place to load additional bootstrap information from outside of your public folder and in an object oriented way.

```
<?php
class Bootstrap extends \Zend\Application\Bootstrap
{
}
```

How it works...

We first define all the important application directories and the application environment as constants for easy access. After that we initiate `\Zend\Application\Application` which will run the application. As you can see there is also a command called `bootstrap()`. This will boot the application and dispatch to your real bootstrap if present and configured. We want to have a two level bootstrap for security reasons. `Index.php` should be implemented with an absolute minimum and all other bootstrap logic should go into the bootstrap class in your application directory. The next thing you will have to do is creating your `.htaccess` to rewrite all requests to use your public bootstrap file.

There's more...

Using an application.php file as an extra security measurement

We have now put a lot of code visibility into our `index.php`. A better way to do it is to put that same logic into `application/Application.php`. And implement your `index.php` to include that file. If something goes wrong with your server and access to the raw code is gained in your public folder, the only information that is published is an `include` statement. Visitors seeing this raw code will not be able to know if you use Zend Framework because the so well known setup has been moved to outside the document root.

This is how to do it:

implement `application/Application.php` with the current logic of `public/index.php`.

Change the implementation of `index.php` into

```
<?php

include '../application/Application.php';
```

Template methods in the application/Bootstrap.php

`application/Bootstrap.php` extends `\Zend\Application\Bootstrap`. `\Zend\Application\Bootstrap` defines a template method by reflection to automatically load user defined methods in the order that they are defined. These magic methods are found as soon as you define them in the following format `protected function _initSomeDescriptiveMethodName()`.

```
<?php
class Bootstrap extends \Zend\Application\Bootstrap
{
    protected function _initSystemLocale()
    {
        //first template method to be executed
        $this->bootstrap('locale');
        setlocale(LC_CTYPE, $this->getResource('locale')->toString() .
'.utf-8');
    }

    protected function _initPlugins()
    {
        //second template method to be executed
        $this->getPluginResource('frontcontroller')
             ->getFrontController()
             ->registerPlugin(new YourProject\Controller\Plugin\
Layout());
    }
}
```

In above code the two `_init()` methods will be executed in the order that they are declared in the bootstrap. This way you do not have to overwrite the `__construct()` and still have your own methods loaded. The implementations you see will be explained in more detail when we come to loading bootstrap resources and plugin resources. At the moment it is suffice to know that `_initSystemLocale()` initialises the locale bootstrap resource and `_initPlugins()` initialises the plugin resource Layout.

See also

- Recipe: creating your .htaccess.
- Recipe: setting up your application.ini

Creating your .htaccess

The most common way a Zend Framework project is setup to direct all requests to your public bootstrap file is through the usage of `.htaccess`. This recipe will show you how to implement this file and offer alternative solutions.

Getting ready

Before a `.htaccess` file can have an effect it should first be allowed to be parsed. And also which subset of the (vhosts) global configuration it is able to overwrite. We do this by allowing `.htaccess` in the Apache vhost configuration file and activating mod_rewrite into our Apache configuration file.

How to do it...

Open your vhost file and allow your `.htaccess` file to be parsed by creating a directive.

```
<directive>
    allow override all none
</directive>
```

Be sure your Apache httpd.conf file has mod_rewrite enabled in your module lists,

Create a `public/.htaccess` file with the following contents

```
SetEnv APPLICATION_ENV development

RewriteEngine On
RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ index.php [NC,L]
```

How it works...

`.htaccess` (hypertext access) is a powerful directive overwriting mechanism. It enables you to overwrite your global configuration with custom per directory directives. Each sub directory can hold its own `.htaccess`, each specifying rules for the related directory. If no `.htaccess` is found in the current sub directory it will fall back to the configuration settings from the `.htaccess` in the parent directory. If no parent is found it will take the global settings defined in your vhost or httpd.conf file.

In a real world application you will want to have different stages, environments on which your application will be deployed, testing, development and production. Your application should be aware on which environment it is currently running. One way to do this is by creating an environment variable `APPLICATION_ENV`. You define development because you are running the application on your development machine. Based on this `APPLICATION_ENV` we will have different settings in our application. Settings can include maximum error reporting, connect to a development database and many more.

Next we activate the `RewriteEngine` which is where `mod_rewrite` comes into play. The directives are very simple. The `RewriteCond` treats the requested filename as a pathname and tests whether or not it exists, and is a regular file with size greater than zero, symbolic link or a directory. If it didn't match it will direct to `index.php` else the requested file is served directly. This will make all your files placed in the public folder available for direct requests.

There's more...

Setting the environment in the VirtualHost itself

Instead of setting the environment variable in the `.htaccess` you should configure it directly on the system itself. This is much more safe because you cannot overwrite production `APPLICATION_ENV` with development by pushing the development .htaccess to production. Remove `SetEnv APPLICATION_ENV` development from `.htaccess` and place this inside your `VirtualHost` container which you have set up to serve the website.

Setting the mod_rewrite rules in the VirtualHost itself

Placing the rewrite rules inside your virtual host definition itself instead of placing it in `.htaccess` will give you a minor performance gain, but the focus is security. Setting Apache configurations asks for knowledge on the subject, if you work in team an inexperienced developer can mess with `.htaccess` or even delete it by mistake. In both cases your entire site goes down and you do not want that.

The directives look a little bit different inside an `.htaccess` compared to the `VirtualHost` ones. To enable this support remove the `RewriteEngine` rules from your `.htaccess` and place the following in your `VirtualHost`. If you apply this recipe together with the recipe on *setting the environment in the VirtualHost itself* you are able to remove the `.htaccess` file all together.

```
<Location />
        RewriteEngine On
        RewriteCond %{REQUEST_FILENAME} -s [OR]
        RewriteCond %{REQUEST_FILENAME} -l [OR]
        RewriteCond %{REQUEST_FILENAME} -d
        RewriteRule ^.*$ - [NC,L]
        RewriteRule ^.*$ /index.php [NC,L]
</Location>
```

See also

- Recipe: getting your system ready to work with ZF

Setting up your application.ini

The `application.ini` is where the application configuration takes place. This recipe will show you the very basics which should be configured for your application to work.

Getting ready

Make sure you have an `application/Bootstrap.php` file installed.

Make sure you have the directory structure as is defined in the recipe creating your directory structure.

How to do it...

Create a file `application/configs/application.ini` with the following code in it:

```
[production]
phpSettings.display_startup_errors = 0
phpSettings.display_errors = 0
phpSettings.date.timezone = "Europe/Brussels"

includePaths[] = PROJECT_PATH "/library"

bootstrap.path = APPLICATION_PATH "/Bootstrap.php"
```

```
resources.frontController.params.displayExceptions = 0
resources.frontController.moduleDirectory = APPLICATION_PATH "/
modules"

resources.view.strictVars = true
resources.view.encoding = "utf-8"
resources.view.content = "text/html"
resources.view.helperPath.Application\View\Helper = APPLICATION_PATH
"/modules/application/views/helpers"
resources.view.helperPath.YourProject_View_Helper = PROJECT_PATH "/
library/YourProject/View/Helper"

resources.locale.default = "en_US"
resources.locale.force = 1


autoloadernamespaces[] = "YourProject"

[development : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
resources.frontController.params.displayExceptions = 1

[testing : production]
phpSettings.display_startup_errors = 1
phpSettings.display_errors = 1
resources.frontController.params.displayExceptions = 1
```

How it works...

For each environment on which your application will be running you specify a section in the `application.ini`. We have defined production, development and testing, these three environments extend each other and by doing so they inherit their parents definition.
In production we do not want to see errors, but in development and testing we do. This is accomplished by overwriting these settings with the values needed for that specific environment.
A section name is defined between square brackets []. You can specify two environments separated with a colon (:) to indicate inheritance. Development and testing both extend production, but testing could just as easily extend development. A section definition will continue until the next section is found.

Let us take a closer look at each directive:
`phpSettings.` use this to overwrite or set php configuration settings.

`includePaths[]` adding paths to your include path on project initialisation.

`bootstrap.` Bootstrap settings

resources. \Zend\Application\Application uses resources to take repetitive logic out of your bootstrap.

Some resources come shipped with ZF. Initializing them is always the same format. You add the resource by adding it to the resources array. Then the resource name followed by constructor params with their values.

In our application we use the following resources: frontController, view, locale and modules. Unexpected params will be ignored.

autoloadernamespaces can be used to automatically add namespaces to the build in autoloader. This ensures that your classes with their own namespaces are found in the include_path.

There's more...

Using an array instead of a .ini file

Zend\Application\Application is initialised in your Bootstrap with configuration settings. The loading of the configuration can be either a string to the configuration file or the configuration itself in the form of an array or \Zend\Config\Config object. Using an array directly by passing the location or giving it as a parameter has some benefits in performance, but it makes maintenance more difficult because you do not have the luxury of inheritance while configuring your environments.

Using an XML format instead of a .ini file

With XML you still have access to inheritance but it is the slowest method of the three. XML files are bloated and are slower to be parsed. Both .xml and .ini configuration files can be store in a cached array for performance, but \Zend\Application\Application does not do this for you.

See also

- Recipe: creating your directory structure
- Recipe: setting up your bootstrap

Your first controller and action \Zend\Controller\Action

This recipe will explain what happens behind the scenes when a user request is made. And the minimum requirements needed to fulfil that request.

How to do it...

Create and implement `/application/modules/default/controllers/IndexController.php`

```
<?php
class IndexController extends \Zend\Controller\Action
{
    public function indexAction()
    {
        $this->view->name = 'Nick Belhomme';
    }
}
```

Implement the view script `/application/modules/default/views/scripts/index/index.phtml`

```
My very first view script, created by
<?php echo $this->escape($this->name); ?>
```

How it works...

The `Model View Controller` design is made completely automatic in Zend Framework. It starts with a `Bootstrap` which will initiate the `frontController`. This `\Zend\Controller\Front` is responsible for tying your entire application MVC together. It first takes a request, gives it to a `router` for handling, the router sets the `request` object with the matching `Module`, `Controller`, `Action` and hands it back to the frontController. The frontController then gives it to the dispatcher who will initiate the Module, Controller and Action set in the request. When no custom routes are set as in our example it falls back to the default route `application`. The default route maps a request URI to the following structure [/moduleName=default][/controllerName=indexController][/actionName=indexAction]

Each block [] is optional. If it is not given it will take the default value.

Thus the following request URI will map to the same default module, IndexController and indexAction.

```
/
/default
/default/index
/default/index/index
/index/
/index/index
```

The dispatched Action will, with the help of the Action Helper `ViewRenderer`, initiate the view component and load the corresponding view script. The corresponding view script is found in the same module view directory and maps to the directory bearing the same name as the controller and renders the script bearing the same action name. The Response object will be updated and given back to the frontController who will serve its output.

Let us have a deeper look at what we did in the `indexAction` and the corresponding view script `index.phtml`.

By default the ViewRenderer action helper is registered to the `helperBroker` and the only thing you have to do is assign values to the view. You assign a value to a property of the View and it will be available from the View script to use it. We assign the value NickBelhomme to a public property name.

It will then be available in our view script `index.phtml`

echoing the value is easy. `<?php echo $this->name; ?>` but all the good boyscouts live by the mantra filter input escape output. So we call the ViewHelper `escape()` method

It is that easy

There's more...

Implementing the ErrorController

If the dispatcher doesn't find a matching Module, Controller, Action or View script it will dispatch to the default `ErrorController`.

The registered errorHandler plugin will also do this when an exception is thrown from within your application. In this error Controller you can take the appropriate action inside the called `errorAction`. If this isn't found your application will throw an uncatched Exception. So it is wise to also implement the `ErrorController` with corresponding action and view script.

Create and implement `/application/modules/default/controllers/ErrorController.php`

```
<?php
class ErrorController extends \Zend\Controller\Action
{
    public function errorAction()
    {
        $errors = $this->_getParam('error_handler');
        switch ($errors->type) {
            case \Zend\Controller\Plugin\ErrorHandler::EXCEPTION_NO_
ROUTE:
            case \Zend\Controller\Plugin\ErrorHandler::EXCEPTION_NO_
CONTROLLER:
            case \Zend\Controller\Plugin\ErrorHandler::EXCEPTION_NO_
ACTION:
                $this->getResponse()
->setHttpResponseCode(404);
                $errorType = 'notFound';
                break;
            default:
                $this->getResponse()
->setHttpResponseCode(500);
                $errorType = 'applicationError';
                break;
        }
        $this->view->errorType = $errorType;
    }
}
```

Implement the view script `/application/modules/default/views/scripts/error/error.phtml`

```
<?php $this->headTitle('An error occured'); ?>
<h1>An error occurred</h1>
<?php if ('notFound' == $this->errorType): ?>
    Page not found.
<?php elseif ('applicationError' == $this->errorType): ?>
    An application error occurred. Please try later.
<?php endif; ?>
```

How it works is quite straight forward. If anything goes wrong with the dispatching it will display a "page not found" message, in all other cases it will be "An application error occurred". In a real world application you will want to have some kind of logging (See the Debugging, error handling and unit testing chapter).

See also

- Chapter: debugging, error handling and unit testing
- Recipe: Routing, optimizing for SEO

Using \Zend\Tool for easy project creation

This recipe will teach you on how to set up a project using \Zend\Tool through the command line.

Getting ready

Have the CLI tool available from your system path. The easiest way is to download the framework and copy the files from the bin/ directory to the same directory as the location of your php cli binary. The Pear installer does this automatically for you.

How to do it...

Navigate to the folder in which you want to create the project folder as `zf` will create the project in a directory called `./YourProject`.

1. `zf create project YourProject`
2. create a vhost as described in the "Getting your system ready to work with a ZF project"
3. launch your webpage

How it works...

`\Zend\Tool` is used to automatically create code for you. In this recipe we have created the basic site structure, including your initial controllers and views all with one cli command. This is a very easy tool to use, but as with all code generators I strongly advise not to use them. The tool generates code for you automagically and you do not have any control on what it generates. Also it blocks your understanding of your own project. And when your understanding of a Zend Framework project has evolved probably also your needs have evolved and you will want some tweaking. Very basic projects can be helped with these kind of systems, but what happens when your system gets bigger and bigger? Can you still rely on \Zend\Tool or the code it has generated? Maybe it does, maybe it doesn't, in which case you might have to write hacks or get conflicts.

There's more...

Adding Modules with \Zend\Tool Cli

1. enter your project directory.
   `Cd ./YourProject`
2. `zf create module YourModuleName`
3. add the following line to YourProject/application/configs/application.ini `resources.frontController.moduleDirectory = APPLICATION_PATH "/modules"`
4. fire up the website `http://yourproject/YourModuleName/`

How it works:

By going into your project directory zf can find the needed `.zfproject.xml` file which it uses to know the current state of your project.

Next you tell zf to create a module with the name of your choice. It will create the modules directory with the module inside. It will also create a default IndexController with the YourModuleName namespace prefixed to it, the index action and corresponding view script.

But it doesn't work out of the box. You have to tell the frontController where the moduleDirectory is. Once this is set you can fire up the website and you now have a nice module at your disposal.

In my opinion the third step should also have been done by `\Zend\Tool`.

Because it portraits the picture that it will generate all the code you need to have a working module but it simply doesn't, thus creating confusion. There is a recipe on modules in this book, if you want to know more, take a look at that.

Adding Controllers with \Zend\Tool cli

1. enter your project directory.
   `Cd ./YourProject`
2. `zf create controller YourControllerName`
   or
   `zf create controller YourControllerName -m YourModuleName`

both commands will create a controller with the name YourControllerName. The second will create a controller inside the module YourModuleName, hence the -m option. Again the module controller will be automatically prefixed with the module namespace.

Adding models with \Zend\Tool cli

1. enter your project directory.
   ```
   cd ./YourProject
   ```
2. `zf create Model YourModelName`
   or
   ```
   zf create model YourModelName -m YourModuleName
   ```

First we use a zf command to generate a model with the name `YourModelName`. The second will create a model inside the module `YourModuleName`, hence the -m option.

In both cases the model will have a namespace prefixed. If you do not specify a module it will take the default "Application" namespace. If you specify a module it will be automatically prefixed with the module namespace. But you cannot instantiate the module models because that namespace hasn't been registered yet. You have to do that manually by making use of the modules resource. Please take a look on the using modules recipe.

See also

- Recipe: Creating your directory structure

Getting input from the user with \Zend\Controller\Request\Http

Almost 99% of projects need some sort of user input. For PHP access to the user input comes from globals such as `$_GET`, `$_POST`, `$_COOKIES` and so on. Zend Framework has wrapped these values within the request object. After all they are a part of the user request. This recipe will show you on how to access these values.

Getting ready

Inside a controller you can gain access to the request object by usage of `$this->getRequest()` or simply `$this->_request`. I will be using this format throughout this recipe as we will get information from within a controller. Because we are working with the request object this recipe will work everywhere in your application where you have access to it.

How to do it...

```
$this->getRequest()->getParam('id', false);
```

How it works...

With this line of code we are getting the id value if it is set. The second optional parameter specifies the return value if the parameter is not found, this is by default null. The param is searched for in a specific stack order. First, the user parameters then $_GET and finally $_POST. The difference between the user params and the environment $_GET and $_POST parameters is that the latter are usually automatically set by the environment when creating the request. And the user params are set explicitly by the application by usage of the `setParam` or `setParams()` request methods.

There's more...

Accessing $_GET and $_POST

```
// $_GET

$this->getRequest()->getQuery('id', false);

// $_POST

$this->getRequest()->getPost('id', false);
```

How it works

These are the preferred way for accessing environment variables.

When no params are passed the entire $_GET or $_POST array will be returned.

Accessing the User Params

```
$this->getRequest()->getUserParam('id', false);
```

How it works

It will fetch the user parameter set by the application. These user params can come from the `router` or elsewhere from your application. Again this has preference above the `$request->getParam()` method, because it is more targeted to a specific source.

If you want all user parameters use the `$this->getRequest()->getUserParams()` method.

Accessing $_COOKIE

```
$this->getRequest()->getCookie('id', false);
```

When no params are passed the entire `$_COOKIE` array will be returned.

Accessing $_SERVER

```
$this->getRequest()->getServer('HTTP_HOST', false);
```

When no params are passed the entire `$_SERVER` array will be returned.

Accessing $_ENV

```
$this->getRequest()->getEnv('APPLICATION_ENV', 'production');
```

When no params are passed the entire `$_ENV` array will be returned.

Accessing Headers

```
$this->getRequest()->getHeader('Accept-Encoding');
```

A `HTTP` header name is required.

Accessing php://input

```
$this->getRequest()->getRawBody();
```

Getting the client his IP address

```
$this->getRequest()->getClientIp();
```

Accepts a boolean to specify if you want to try to retrieve the IP address from the client even when he is behind a proxy. If you specify false and the client is behind a proxy the ip address from the proxy will be returned.

See also

- See the source code for the Request object for more convenience methods as there are many more, especially header information.
- Recipe: Your first controller and action \Zend\Controller\Action

Filtering input with \Zend\Filter\StaticFilter

Filter input, escape output. We all love this mantra. Why? Because it makes our applications more secure and what a good feeling that is. This recipe will show you how to filter input unwanted input.

Filtering is the process of stripping (removing unwanted input, characters) or transforming (escaping input, characters) into something desirable.

Getting ready

There are already plenty of filters available in the framework. Some take parameters, others don't. I will show you how to call both. If you want to know which filters are available open the framework directory Zend/Filter. To see if a filter takes parameters take a look at the constructor of the filter. Alternatively take a look at the online documentation, but be aware that the latest and correct documentation is always the code itself.

How to do it...

We are going to filter a telephone number which is given in a different format than expected, but instead of forcing the user we filter the input.

```
echo \Zend\Filter\StaticFilter::execute('+32 16/888.01.02.03',
'pregReplace', array('match' => '#^\+#', 'replace' => '00'));
echo \Zend\Filter\StaticFilter::execute('+32 16/888.01.02.03',
'digits');
//0032 16/888.01.02.03
//3216888010203
```

How it works...

We are using the static method execute from `\Zend\Filter\StaticFilter` to automatically instantiate the appropriate filter for us. In the examples above it would be the same if you did it yourself by doing:

```
$filter = new \Zend\Filter\PregReplace(array('match' => '#^\+#',
'replace' => '00'));
echo $filter->filter('+32 16/888.01.02.03');
$filter = new \Zend\Filter\Digits();
echo $filter->filter('+32 16/888.01.02.03');
```

Both methods are equally good, it depends on your situation. If you need to use the filter multiple times in a loop or with different values then I would suggest to instantiate the filter yourself and reuse the same instance.

As you can see the first parameter to the `StaticFilter::execute` is the value to be filtered, the second the filter to be applied, the third its constructor params and optionally fourth a namespace for you custom filters.

There's more...

Writing your own custom filter

You can create your own filter for easy reuse. The only requirement is that you implement the `\Zend\Filter\Filter` interface which states that you must implement the filter() public method.

Let us uniform the phone number by adding two leading zeros when a leading + is used to indicate a country and secondly filter all other remaining non digits. This way we combine both the above used filters.

```
namespace YourProject\Filter
class PhoneNumber implements \Zend\Filter\Filter
{
    public function filter($value)
    {
        return preg_replace(array('#^\+#', '#[^\d]#'), array('00',
''), $value);
    }
}
```

Or if you want to reuse existing filters in a custom filter you can also do so.

```
namespace YourProject\Filter
class PhoneNumber implements \Zend\Filter\Filter
{
    public function filter($value)
    {
        $result = \Zend\Filter\StaticFilter::execute($value,
'pregReplace', array('match' => '#^\+#', 'replace' => '00'));
        $result = \Zend\Filter\StaticFilter::execute($result,
'digits');
        return $result;
    }
}
```

Then you can instantiate your filter like any other filter and use it.

If you want to be able to also call the filter statically you need to add the namespace as a fourth parameter.

```
echo \Zend\Filter\StaticFilter::execute('+32 16/888.01.02.03',
'PhoneNumber', array(), 'YourProject\Filter');
//003216888010203
```

\Zend\Filter\StaticFilter allows also to set default namespaces. This means that you can set them once in your bootstrap or resource plugin and not have to give them again for each call.

```
$loader = new \Zend\Loader\PluginLoader(
    array(
        '\YourProject\Filter' => 'YourProject/Filter/'
    )
);
\Zend\Filter\StaticFilter::setPluginLoader($loader);


echo \Zend\Filter\StaticFilter::execute('+32 16/888.01.02.03',
'PhoneNumber');
//003216888010203
```

Chaining multiple filters with \Zend\Filter\FilterChain

Often multiple filters should be applied to some value in a particular order, this can be achieved by creating a filter chain.

```
$filterChain = new \Zend\Filter\FilterChain();
$filterChain->addFilter(new \Zend\Filter\PregReplace(array('match' =>
'#^\+#', 'replace' => '00')))
            ->addFilter(new \Zend\Filter\Digits());
echo $filterChain->filter('+32 16/888.01.02.03');
```

Filters are executed in the order they are added.

Chaining multiple filters with \Zend\Filter\InputFilter

Another way of chaining filters is creating a \Zend\Filter\InputFilter instance and adding the filters to the filter stack.

```
$filters = array(
    'telephoneNumber' => array(
        array('pregReplace', array('match' => '#^\+#', 'replace' =>
'00')),
        'digits',
    ),
);

$filterChain = new \Zend\Filter\InputFilter($filters, array());
$filterChain->setData(array('telephoneNumber' => '+32
16/888.01.02.03'));
echo $filterChain->telephoneNumber;
//003216888010203
```

`\Zend\Filter\InputFilter` accepts an associative data array with values. Each value will be filtered using the assigned filters defined in the filter array. The first constructor parameter accepts this filters array.

The second parameter is also required and defines the validator stack. More about this in the Validator Recipe.

The third parameter is the data set, but this is optionally so we set it by example.

Once a data set has been set you can use the magic __get method to retrieve filtered input. Data set keys are transformed to filtered public properties and also passed trough an additional filter \Zend\Filter\HtmlEntities. If you want the unescaped value you need to use getUnescaped().

If you want to apply the filter to also the fax number you do not need to redefine the same filterchain for each data set key. Assigning to the wildcard symbol * will result in \Zend\Filter\InputFilter applying the filter to each value of the dataset.

```
$filters = array(
    'faxnumber' => array(
        'StringTrim'
    ),
    '*' => array(
        array('pregReplace', array('match' => '#^\+#', 'replace' =>
'00')),
        new \Zend\Filter\Digits(),
    ),
);

$filterChain = new \Zend\Filter\InputFilter($filters, array());
$filterChain->setData(array(
    'telephoneNumber' => '+32 16/888.01.02.03',
    'faxnumber' => '  +32 16/888.11.22.33  ',
    )
);
echo $filterChain->telephoneNumber;
echo $filterChain->faxnumber;
//003216888010203
//003216888112233
```

As you can see you can define the filter rules also by using instantiated filters as we did with \Zend\Filter\Digits. You can define extra filters for each input by creating an extra filter rule. Again as with filterchains filter rules are executed in the order that they are set. In this case the faxnumber will be first trimmed before applying the wild card rule.

See also

- Recipe: validating input with \Zend\Validator\StaticValidator
- Recipe: doing Filtering and Validating in one batch with \Zend\Filter\InputFilter

Validating input with \Zend\Validator \StaticValidator

A very important part in an application is input validation. This falls under the same mantra as Filter input, Escape output. If input is received we want to be able to check if it qualifies as expected input. Asking for an email address and receiving a string 'notneeded' should raise a warning. This is what `\Zend\Validator` namespace classes do.

Getting ready

There are already plenty of validators readily available in the framework. As with filters some take parameters, others don't. I will show you how to use both. If you want to know which validators are available open the framework directory `Zend/Validator`. To see if a validator takes parameters take a look at the constructor of the validator. Alternatively take a look at the online documentation, but be aware that the latest and correct documentation is always the code itself.

How to do it...

```
\Zend\Validator\StaticValidator::execute('ahà83dédsç', 'alnum');
\Zend\Validator\StaticValidator::execute('ahà83dédsç', 'stringLength',
array('min' => 12, 'max' => 16, 'encoding' => 'utf-8'));
```

How it works...

We are using the `\Zend\Validator\StaticValidator` static public method execute(). This is a convenience method to quickly asses if a value is "valid". Validators always give back a `boolean`, where true represents valid. In this example we are testing a string, for example a password. In the first test we test if it is alpha numeric. As you can see I use characters with accents. The validator internally uses `\Zend\Filter\Alnum` which uses your application locale to filter out all non language dependent characters. If the passed value is the same as the internally filtered value than it will pass. As you can see validators rely heavily on the filters. For my `locale` (be_NL) the alnum validation will pass, for the `locale` en_GB it won't. We also validate if the given password is between 2 and 16 characters long. For Unicode support you have to specify the encoding.

As you can see the first parameter to the static `is()` method is the value to be validated, the second the validator to be applied, the third its constructor params and optionally fourth a namespace for you custom validator.

The static method is the same as if you would instantiate the validators yourself.

```
$validator = new \Zend\Validator\Alnum();
$validator->isValid('ahà83dédsç');
$validator = new \Zend\Validator\StringLength(array('min' => 2, 'max'
=> 16, 'encoding' => 'utf-8'));
$validator->isValid('ahà83dédsç');
```

When using the static version, validation failure messages are not available. So it is best for quick checks where you only care about true or false scenario.

There's more...

Checking the error messages to see why input is not valid

Validators have a build in system to display a message on why a value is not valid. These messages are in the English format but your can translate them or even completely overwrite the messages with something more to your liking.

```
$validator = new \Zend\Validator\StringLength(array('min'=> 2, 'max'
=> 16, 'encoding' => 'utf-8'));
if (!$validator->isValid('a')) {
    var_dump($validator->getMessages());
}
//output
array
  'stringLengthTooShort' => string ''a' is less than 2 characters
long' (length=34)
```

You can use these error messaging for internal reporting or for feedback to the user who is most of the time responsible for a value that needs validating. \Zend\Form View Helpers actually use `getMessages()` to hint the user on the erroneous form input.If you want to display a more cocky message you can easily do so:

```
$validator = new \Zend\Validator\StringLength(array('min'=> 2, 'max'
=> 16, 'encoding' => 'utf-8'));
$validator->setMessage('Didn\'t I tell you to provide minimum %min%
charachters?!', \Zend\Validator\StringLength::TOO_SHORT);
if (!$validator->isValid('a')) {
    var_dump($validator->getMessages());
}
```

You can target messages by the use of their keys. In this case we want to overwrite the message for the key `stringLengthTooShort`. We will do this by using the class constant which is found at the top of each validator class. This makes our code more robust for future key changes. If you want to set multiple messages at once you can use `setMessages()`, which accepts an array.

As you can see you have special placeholders in a message. You can omit those placeholders or you can include them. In this example we have chosen to omit the `%value%` placeholder which will display the passed value but use the `%min%` placeholder to have our dynamic parameter value of 2. This way you can target your messages to your user base or screen space. You have to set the custom messages before calling the `isValid` method. If you want to reuse the custom messages in multiple places in your application I suggest you extend the validator with the custom messages set as default. But do not create custom messages because you want translated messages. You can use `\Zend\Translator\Translator` for that.

Using \Zend\Translator\Translator to translate the messages

If you want to serve your website in a different language than the default English you have to use a `\Zend\Translator\Translator` object for this. Once this is setup (see the recipe on translators) you can use this translate object to translate the messages a validator returns.

When a translation instance is available to `\Zend\Validator\Validator` it will try to use it for translating the failed validation messages. It does this by taking the `messageTemplates` values as the keys for your translation library.

As an example the translation key you should use is `'%value%' is less than %min% characters long` from the following `messageTemplate`

```
self::TOO_SHORT => "'%value%' is less than %min% characters long"
```

For your convenience the validators have been readily translated to a lot of languages. You can find them within the path /resources/languages in your Zend Framework Full Installation Path from the complete download. If the validators haven't been translated for your language you can use those as an example. You can translate all validators or only the ones needed. After you have a translation file copy it over to your translate directory as specified in the translation recipe and attach it to your translation object if it doesn't already automatically do so (depending on your setup, see the translation recipe).

Then you must make \Zend\Validator translation aware. This can be done in various ways:

1. By setting the default translation object for all validators:
   `\Zend\Validator\AbstractValidator::setDefaultTranslator($translator);`
2. By setting your translation object for the entire application so that translation aware components such as \Zend\Validator can use it:
   `\Zend\Registry::set('Zend_Translate', $translator);`

3. By adding it directly to the used validator:
   `$validator->setTranslator($translator);`

Writing your own Custom Validator

As with filters you can also create your custom validator for easy reuse. I am not going into detail as this is very well described in the documentation of ZF itself:

`http://framework.zend.com/manual/en/zend.validate.writing_validators.html`

An example implementation can also be found in the book accompanied code.

The only thing I want to explain is how to make it available to the `\Zend\Validator\StaticValidator::execute()` static method. Once you have created a custom validator you will want to add it to the default `namespaces` of `\Zend\Validator\AbstractValidator`. For the following validator \YourProject\Validator\MyPersonalValidator you would the namespace the following way

```
$loader = new \Zend\Loader\PluginLoader(array('\YourProject\
Validator' => 'YourProject/Validator/'));

\Zend\Validator\StaticValidator::setPluginLoader($loader);
```

This will give you access to:

```
\Zend\Validator\StaticValidator::execute('ahà83dédsç',
'MyPersonalValidator');
```

Chaining Validators with \Zend\Validator\ValidatorChain

Creating a validator chain is easy. You create a new `\Zend\Valdidator\ValidatorChain` instance and add validators to it in the order you want to see the value pass. By default all added validators will be run, even when the previous validator in the chain failed. This behavior can be changed when you pass true as the second parameter. Letting all validators do their magic builds the `getMessages()` array for feedback to the user. If no feedback is needed, than I would advise setting `$breakChainOnFailure` to true.

```
$validator = new \Zend\Validator\ValidatorChain();
$validator->addValidator(new \Zend\Validator\StringLength(array('min'
=> 14, 'max' => 16, 'encoding' => 'utf-8')), true)
          ->addValidator(new \Zend\Validator\Alnum());
$validator->isValid('ahà83dédsç');
```

Chaining multiple validators with \Zend\Filter\InputFilter

Another way of chaining validators is creating a `\Zend\Filter\InputFilter` instance and adding the validators to the validator stack.

```
$validators = array(
    'password' => array(
        array('StringLength', array('min' => 2, 'max' => 16,
'encoding' => 'utf-8')),
        new \Zend\Validator\Alnum(),
        'breakChainOnFailure' => true
    ),
    'username' => array(
        'emailAddress'
    )
);

$validatorChain = new \Zend\Filter\InputFilter(array(), $validators);
$validatorChain->setData(array('password' => 'ahà83dédsç',
'username'=> 'contact@nickbelhomme.com'));
$validatorChain->isValid('password');
$validatorChain->isValid('username');
$validatorChain->isValid();
//true
//true
//true
```

`\Zend\Filter\InputFilter` accepts an associative data array with values. Each value will be validated using the assigned validators defined in the validator array which needs to be set as the second param to `\Zend\Filter\InputFilter`. This validator array exists out of data set keys together with an array of validators needed to be executed in the order that they are added. You can use the validator names or use instantiated validators. Whichever is your preferred way of working. As with the Filter stack you can also add a validator wildcard `*` to target all input data. Even input data that isn't defined individually.

The first parameter is also required and defines the filter stack. More about this in the Filters Recipe.

The third parameter is the data set, but this is optionally so we set it by example.

You can validate all input params or only 1 param at the time using `isValid()`. Providing no input param name will do validation on all params and return one single `true` or `false`

See also

- Recipe: filtering input with \Zend\Filter\StaticFilter
- Recipe: doing Filtering and Validating in one batch with \Zend\Filter\InputFilter.
- Accompanied code chapter1: \YourProject\Validator\LocalIp
- `http://framework.zend.com/manual/en/zend.filter.input.html` for many more validator options

Doing Filtering and Validating in one batch with \Zend\Filter\InputFilter

This recipe will go into detail on how to bring the two previous recipes filtering and validating together.

Getting ready

Read the two recipes on validating and filtering. A good understanding of these two principles is needed. Take a good read on the `\Zend\Filter\InputFilter` sub chapter of each of those recipes, because this will build onto that principle.

How to do it...

```
$filters = array(
    '*' => array(
        'StringTrim'
    ),
    'faxnumber' => array(
        array('pregReplace', array('match' => '#^\+#', 'replace' =>
'00')),
        new \Zend\Filter\Digits(),
    ),
);

$validators = array(
    'password' => array(
        array('StringLength', array('min' => 2, 'max' => 16,
'encoding' => 'utf-8')),
        new \Zend\Validator\Alnum(),
    ),
    'username' => array(
        'emailAddress'
    )
);

$data = array(
    'username' => ' contact@nickbelhomme.com',
    'password' => 'ahà83dédsç',
    'faxnumber' => '+32 16/888.11.22.33',
);

$input = new \Zend\Filter\InputFilter($filters, $validators, $data);
$input->isValid();
//true
$input->faxnumber;
//null
$input->getUnescaped('faxnumber');
//null
$input->getEscaped('faxnumber');
//null
```

How it works...

The example above will strike odd after reading the previous recipes. Why does all the getters for faxnumber return null, we have provided the faxnumber input parameter. Yes we did, but as soon as you provide validator rules to \Zend\Filter\InputFilter it will use these rules to validate ALL input data. Input data that is not specified in the validator rules are ignored and you cannot get those values back, because they are considered unsafe. Providing a validator rule for faxnumber would solve this issue. In this case a 'Digits' validator should do just fine. The reason why the digits validator would pass is because `\Zend\Filter\InputFilter`always first executes the Filters prior to validating. Another solution could be to set a validator wildcard, even an empty rule. This would set a validator rule for all input and thus you can also retrieve it.

See also

- Recipe: getting input from the user with \Zend\Controller\Request\Http
- Recipe: filtering input with \Zend\Filter\StaticFilter
- Recipe: validating input with \Zend\Validator\StaticValidator
- `http://framework.zend.com/manual/en/zend.filter.input.html`

How to set headers per action

Because Zend Framework is 99.99% of the time used in Web applications an important feature is setting headers. This recipe will handle that.

How to do it...

```
public function downloadPdfAction()
{
    $this->_helper->viewRenderer->setNoRender(true);

    $fileName = 'onTheFly12547.pdf';
    $filePath = PROJECT_PATH.'/data/pdf/'.$fileName;

    $this->getResponse()->clearHeaders()
    ->setHeader('Content-Description', 'File Transfer')
    ->setHeader('Content-type', 'application/pdf')
    ->setHeader('Content-Disposition', 'attachment; filename="'.$file
Name.'"')
    ->setHeader('Content-Transfer-Encoding', 'binary')
    ->setHeader('Expires', '0')
    ->setHeader('Cache-Control', 'must-revalidate, post-check=0, pre-
check=0')
    ->setHeader('Pragma', 'public')
    ->setHeader('Content-Length', filesize($filePath));

    $this->getResponse()->clearBody();
    $this->getResponse()
         ->setBody(file_get_contents($filePath));
}
```

How it works...

Because you want to download a pdf file we have to first disable the layout and the ViewRenderer. This has no use and would corrupt the body of the response.

Because we want to serve a download we will have to set the specific headers. We do this by adding headers to the `Response` object. But first we have to remove all headers which could interfere with the correct settings. Next you can choose to set Raw Headers or key value pairs. I prefer the key value pair as shown above because that will give you the opportunity to replace previous set headers with the same key. This is done with the third parameter. The first is the header key, the second the value of the header. In this example we do not need the third parameter because we have cleared all headers and set the required headers only once.

After all headers are set we will have to set the body with the correct binary content. We use `\Zend\Controller\Response\AbstractResponse::clearBody()` in combination with `\Zend\Controller\Response\AbstractResponse::setBody()`. Together they will overwrite all previous body content.

There's more...

Setting the Unicode UTF-8 Header

You can set the Unicode UTF-8 header in the appropriate action overwriting all previous content-type headers with the third parameter.

```
public function someAction()
{
    $this->getResponse()
         ->setHeader('Content-Type', 'text/html; charset=utf-8',
true);
}
```

or you can choose to initialise this header for all actions by setting it in the bootstrap as a bootstrap resource.

```
protected function _initResponseHeaders()
{
    $response = new \Zend\Controller\Response\Http();
    $response->setHeader('Content-Type', 'text/html; charset=utf-8');

    $this->getPluginResource('frontcontroller')
         ->getFrontController()
         ->setResponse($response);
}
```

This second example is worth going deeper into it. You already know bootstrap resources and bootstrap plugins. And if you do not know them I would advise to take a look at the appropriate recipes. What basically happens here is that we get the frontController by getting it from the frontcontroller resource. By default the front controller does not yet have a response and request object added at initialization. So we have to create and set the response ourselves.

Because in this solution we decide which response object is added we have more control. If you want to let the front controller initialize the response object itself than I suggest moving this code to a bootstrap resource plugin where the initialized response will be available.

```
<?php
namespace YourNamespace\Controller\Plugin;
class Layout extends \Zend\Controller\Plugin\AbstractPlugin
```

```
{
    public function routeShutDown(\Zend\Controller\Request\
AbstractRequest $request)
    {
        $this->getResponse()->setHeader('Content-Type', 'text/html;
charset=utf-8', true);
    }
}
```

Setting a cookie

Every web application at some point needs to work with cookies. The way to do this in Zend Framework is easy. As you have seen in previous examples we could just add a complete header and be done with:

```
public function someAction()
{
    $this->getResponse()
         ->setHeader(
             'Set-Cookie',
             'gender=M'
            .'; expires=Saturday, 21-Aug-10 21:19:32 Europe/Brussels'
            .'; domain=.nickbelhomme.com'
            .'; path=/'
        );
}
```

This will set a cookie with the name gender and the value M. To create the expiry time of the cookie you can use \Zend\Date\Date. If for instance you want the cookie to have a lifetime of one month you would use the following:

```
\Zend\Date\Date::now()

->add(1, \Zend\Date\Date::MONTH)

->toString(\Zend\Date\Date::COOKIE);
```

Remember the `response::setHeader()` method supports a second parameter replace. Do not set it to true as this will not enable you to set more than one cookie per response and only the last one added will be sent.

Zend Framework doesn't natively support at the moment creation and setting of cookies in an oop way. Sure you can use `\Zend\Http\Cookie` to create a cookie but the magic method `__toString()` doesn't output a valid `Set-Cookie` header value. A way to go about this is to extend `\Zend\Http\Cookie` and override the `__toString()` method so that it returns a valid cookie string or by adding another method that will accept a response object and sets the cookie header in one go.

See also

- Recipe: your first controller and action \Zend\Controller\Action
- Recipe: setting up your bootstrap
- Recipe: easy date displaying with \Zend\Date\Date

Summary

In this chapter you have learned how to set up a Zend Framework application manually and by the use of `\Zend\Tool`. How a standard URI request from the user maps itself to the right module, controller, action. How to retrieve user input through the use of the `\Zend\Controller\Request\Http` object. Filter and validate this user input with the usage of `\Zend\Filter` and `\Zend\Validator` namespace classes. And eventually how to set the appropriate headers into the `\Zend\Controller\Response\Http` object to serve specific content to the user, whether it to set a cookie, set a download file or simply specify the charset used.

The one thing I really want you to take with you from this chapter is the good usage of `OOP`. You should always use the Request and Response instances registered to the frontController to retrieve input and set response information such as the body or the headers. Using `$_GET`, `$_POST`, `$_COOKIE` and such directly break this design. The same applies when you use `set_cookie()` and other response manipulating functions like `header()` directly.

For optimal understanding of this chapter I advise you to take a look at the sample code accompanying this book. It will give detailed instructions and real hands on examples on how to implement what you have learned so far.

2
Displaying information and linking pages together

In this chapter, we will cover:

- Using \Zend\View\View
- Creating reusable display parts with \Zend\View\Helper\Partial or render() method
- Using \Zend\View\Helper\Action for controller access from the View
- Using custom \Zend\View\Helper querying the model
- Creating layout scripts for global templating pages with \Zend\Layout\Layout
- Using \Zend\Paginator\Paginator to iterate over result sets
- Gluing pages together with url View Helper
- Routing, optimizing for SEO
- Easy redirecting with the redirector action helper
- Building menus with \Zend\Navigation\Navigation

Introduction

I want you to take this chapter from a hands-on-code point of view. That means taking the chapter 2 code and playing and trying while you read this chapter. You can just read the chapter and have a very good understanding, but nothing beats practice.

So let us begin with the view part of the MVC.

All the business logic goes into the models, the user input handling into the controllers and the presentation into the view. This chapter will give you some recipes on how to use that view in Zend Framework and how to link the pages you have created together using the routing system and appropriate action and view helpers.

Using \Zend\View\View

PHP is a very powerful template language, you should use the available functionality. Some PHP template languages are actually a wrapper around PHP trying to keep the language out of view scripts. This is done because the reasoning behind such template languages is that PHP has no business inside the view script. Yet nothing is wrong with allowing it in the scripts as long as you use it for outputting only and not start to manipulate the model from the Model View Controller. The view can and may make direct information requests to the model but not alter it. Alteration can only be done from the controller or models themselves. Keeping this separation in responsibilities is not the view its job, it is the job of the developer.

Using PHP as a template language also doesn't involve a learning curve, some template engines have their own language constructs. Also every IDE on the market knows how to do code syntax coloring and on the fly syntax validation. All these benefits should be used and that is why Zend Framework chose not to implement a wrapper around PHP as a template engine. And as you will see it will not hinder your webmaster from writing nice markup and or view scripts.

`Zend_View` accepts a view script (the template), optionally some variables and will render the script using simply php. It doesn't do more than that. The real magic in Zend Framework is inside the ViewRenderer. We will look at both.

How to do it...

By default you have to create a corresponding view script for each controller action you create. So let's say you want to have a hello world page with some user data in it. To make this work you have to go through a 5 step process.

1. Decide for which `module` you want to create the page
2. Decide for which `controller`
3. Decide on an action name and create this `action`
4. Create a `view script` corresponding to the location of the above decisions. For instance `YourProject/application/modules/application/views/scripts/index/hello-world.phtml` for a module `default` controller `IndexController` and finally the action `helloWorldAction()`.
5. Implement the view script with markup.

How it works...

Because Zend Framework is a full stack MVC framework, the framework architects have decided on automating the process of initiating the view and choosing which script to render. Ninety nine percent of all page requests should have a corresponding View rendered in a MVC application. This is done by the usage of a Controller Action Helper, namely the `ViewRenderer`. As the name suggests this action helper will try to render the view for each action that has been called. You can completely customize which script to render or you can choose to deactivate the `ViewRenderer` for the entire project or per action.

There's more...

Using the view resource

`\Zend\Application\Application` uses `Resources` to facilitate the bootstrapping of recurring initialization code. It has done the same for the `\Zend\View\View` initialization.

The only thing you have to do is add some lines in the `application.ini`. The resource accepts all of the `\Zend\View\View` constructor parameters and additionally also a doctype.

This would make the resource accept all these settings and more:

```
resources.view.helperPath = APPLICATION_PATH "/modules/application/
views/helpers"
resources.view.helperPathPrefix = "Application\View\Helper"
resources.view.escape = "htmlspecialchars"
resources.view.encoding = "UTF-8"
resources.view.scriptPath = APPLICATION_PATH "/modules/application/
views/scripts"
resources.view.strictVars = 1
resources.view.lfiProtectionOn = 1
```

Above I have showed you the most commonly used resource parameters.

You can specify a `helperPath` for your view. If you only specify one helper path you need to also give the `helperPathPrefix`. Alternatively you can also specify an array in which case you can add as many view helper paths as you want. The key is the prefix.

```
resources.view.helperPath.Application\View\Helper = APPLICATION_PATH
"/modules/application/views/helpers"
```

By default the `htmlspecialchars()` function is registered as the callback to escape content when you call `$view->escape()`. You can override the `escape` setting with another user defined function.

The view and view helpers need to be encoding aware and you can set it with the `encoding` parameter.

You can also specify a `scriptPath` as a string or multiple paths as an array in which case you would do

```
resources.view.scriptPath[] = path1

resources.view.scriptPath[] = path2
```

The `strictVars` is one of those params often overlooked but very handy. In production you want this to be set to false, but on development enabling this is a good tool. When it is true, undefined variables accessed in the view scripts will trigger notices. This way you can keep your scripts clean and also spot typing errors.

And last you have the `lfiProtectionOn` which stands for `Local File Inclusion Protection`, which is enabled by default. When enabled you cannot use `../` or `..\` to traverse paths with the `$view->render()` method. Disabling it could lead to rendering the contents of for instance `../../../../etc/passwd` because then `render()` allows for parent directory traversal.

Assigning variables to the view

Probably you are programming dynamic content and so you will need a way to pass this content to your view. The simplest way to do this is to assign the variables to a public property of the view.

```
$view->name = 'Nick Belhomme';
$view->age = $this->getAge();
```

Another way to do it is by assigning an array. This maps all of your keys to public properties and assign the values to them.

```
$view->assign(array(
    'name' => 'Nick Belhomme',
    'age' => $this->getAge(),
));
```

When inside a Controller you can access the view as a public property:

```
// some controller action
$this->view->age = $this->getAge();
```

And inside the view script that is being rendered you can access these same values by using `$this->registeredVariablename` for instance `<?php echo $this->name; ?>`. Be careful what you output in your view scripts. Never, ever output raw user input. Always use trustworthy model output and `escape` everything else before outputting it.

`<?php echo $this->escape($this->name); ?>` This will escape the name value with the by default registered callback function `htmlspecialchars`. This is needed to prevent XSS attacks and such. Remember the Mantra: Filter input, escape output. The `escape()` method is actually one of the many view helpers. View helpers are really useful and mandatory if you want to build a stable and well maintainable application.

Alternative syntax for control structures

PHP offers an alternative syntax for control structures such as `if`, `for`, `foreach`, `while` and `switch`. This alternative syntax makes it easy for you to build control structures in your view without using braces. Braces are very hard to read in the mids of all that markup. Simply replace the opening brace with a colon (:) and the ending brace with the corresponding end: `endif; endfor; endforeach; endwhile; endswitch;`.

```
<?php if ($this->success): ?>
    <div>The operation was a huge success</div>
<?php endif; ?>
```

Changing the automatically rendered view script

You can easily change the script that needs to be rendered from inside an action. You simply call the render method inside an action.

```
$this->render('some-view-script-name-without-extention');
```

The parameter is actually an action name that you should pass, but because the default behavior maps that action name to a corresponding view script you should use the script name.

Deactivate the automatic rendering of pages per application.

To deactivate the `viewRenderer` for an entire project you simply add one line of code in your `application.ini`

```
resources.frontController.params.noViewRenderer = 1
```

Now the frontController resource will initiate the frontController with the parameter `noViewRenderer` set to `true`. Which will effectively disable the automatic loading of the `viewRenderer` action helper plugin

Deactivating the automatic rendering of pages per action

By default the `viewRenderer` is loaded, but there are times you do not want to be forced to use a view script. For instance when you want to create a `downloadAction` or an `ajaxAction` which will echo a JSON encoded string. In such cases you can simply deactivate the rendering of the script by adding the following line of code in the Action.

```
$this->_helper->viewRenderer->setNoRender(true);
```

See also

- Recipe: Creating layout scripts for global templating pages with \Zend\Layout\Layout
- Chapter 1 Recipe: Filtering input with \Zend\Filter\StaticFilter

Creating reusable display parts with \Zend\View\Helper\Partial or render() method

As with business logic you also want your isolate and reuse some parts of your view templates.

How to do it...

You can do this using the view its `render()` method. Inside a `.phtml` script you can use `render()` to include other template scripts which are available in the views added script paths.

```
<div>
    <?php echo $this->render('_common/menu/main.phtml'); ?>
</div>
<h1>My wonderful page</h1>
```

How it works...

The render will tell the view (`$this`) to render a script and return the output. This means that the same view variables will be available inside the rendered script, because the same view is used and its current state.

If you want to render a script with the use of a clean view state, this is a view where all user set view variables have been unset, you should use `partial()`.

In the example above I have included a menu which I have stored outside our view controller directory and inside a `_common` directory to indicate it are view scripts that could be used by all scripts in that module. Again we organize our scripts orderly by using the file system.

There's more...

Rendering the included script in its own clean state using partial

With render the included script use the same view as the calling script. Sometimes this is unwanted and you want the script to run with its own view, to prevent variable name clashes. It could be because a certain view script needs in addition to the current view script another naming convention, whatever the reason. For this we have the view helper `partial` family available to us. Partial will clone the current view instance, reset all set variables and optionally accept an array to be used in an `assign` statement to set the variables you want to have available inside the scope.

```
<div id="articleList">
    <?php
        echo $this->partial(
            '_common/lists/headlines.phtml',
            array(
                'collection' => $articleList,
                'title' => 'my article list'
            )
        );
    ?>
</div>
<div id="latestNews">
    <?php
        echo $this->partial(
            '_common/lists/headlines.phtml',
            array(
                'collection' => $news,
                 'title' => 'Last minute news'
            )
        );
    ?>
</div>
```

In this example we have used a general template script (the partial) which will use `$this->collection` to build a list and put a title on top of it using the `$this->title` property. As a general rule you should not use partial when no variables are passed, because of the performance overhead in creating a new clean view state.

You can also call a script from another module by moving the second parameter (the model) to the third place, and setting the module name as the second param. I do not give examples on this as you should like mentioned before, put application shared view scripts outside of a module directory and include that path into the registered script paths of the view by adding the appropriate `scriptPath` line(s) in the view resource (see view resource recipe).

See also

- Using \Zend\View\View : Using the view resource

Using \Zend\View\Helper\Action for controller access from the View

A view should never make changes to the model. It can request from it but not alter it. Nor should it do any controller logic. So how are we able to create widgets which are able to process user input, make changes if needed to our model and keep the one liner plug and play abilities? We do that with the views action helper.

How to do it...

Create a controller action which processes some user input or manipulates the model in some way and let it output something.

Then call this action from the view script with

```
<?php echo $this->action(
    'some-action',
    'some-optional-controller',
    'some-optional-module',
    array (
        'some-optional-user-param' => 'some-value',
        'another-optional-user-param' => 'some-value',
    )
); ?>
```

How it works...

This view helper will call the controller action and will return its output which we will output using the echo statement. Together with modules and controllers this is a great way to create reusable widgets for your application. This could be for instance a login widget for which the action will process the user credentials and display if logged in a “you are logged in” message otherwise the login form.

See also

- Chapter 1 Recipe: Your first controller and action \Zend\Controller\Action

Using Custom \Zend\View\Helper querying the model

You do not every time have to pass through a controller to get some model information. To keep your view scripts clean, create re-usable logic and to use the DRY (Do No Repeat Yourself) principle you can use custom view helpers.

How to do it...

Imagine we have a blog and we want to display the latest n number of comments. This would be an excellent opportunity to use a custom view helper.

First add the following line to the `application.ini`

```
resources.view.helperPath.Application\View\Helper = APPLICATION_PATH
"/modules/application/views/helpers"
```

second create and implement:

`YourProject/application/modules/application/views/helpers/Comment.php`

```
<?php
namespace Application\View\Helper;
class Comments extends \Zend\View\Helper\AbstractHelper
{
    public function direct(array $args = array())
    {
        if (!isset($args['id'])) {
            throw new \Exception('plz pass a blog post id');
        }

        $postId = $args['id'];
        $qty = 5;
        $pageNumber = 1;

        if (isset($args['qty'])) {
            $qty = $args['qty'];
        }

        if (isset($args['pageNumber'])) {
            $pageNumber = $args['pageNumber'];
        }

        $service = new \YourProject\Service\Comments();
        $collection = $service->getComments($postId, $qty,
$pageNumber);

        $paginator = new \Zend\Paginator\Paginator(
            new \YourProject\Paginator\Adapter\Collection($collection)
        );
        $paginator->setCurrentPageNumber($pageNumber);
        $paginator->setItemCountPerPage($qty);

        return $paginator;
    }
}
```

And lastly use the custom created view helper in your view script.

```
<?php
$paginator = $this->comments(array(
    'id' => $this->blogPostId,
    'pageNumber' => $this->pageNumber,
    'qty' => 2
));
?>
<?php if (count($paginator)): ?>
    <ul>
        <?php foreach ($paginator as $entity): ?>
            <li>
                <?php echo $this->escape($entity->text); ?>
            </li>
        <?php endforeach ?>
    </ul>

    <?php
        echo $this->paginationControl($paginator, 'Sliding', '_common/
paginator.phtml')
    ?>
<?php endif; ?>
```

How it works...

If we want to use custom view helpers in our application we first have to let the application know where those are stored in the file system and which namespace they use. We do this in our `application.ini`. Once the helper path has been added, Zend Framework will search that added path in addition to the already registered paths.

Next we have to create the view helper. We have stored our view helper in the helpers folder of our default module. This would mean that those view helpers should only be used inside this module. You can call them from whatever view script, because the path has been added at initialization but it is good practice to keep dependencies together. If you want to reuse a view helper across multiple modules or in the layout file, I would suggest to create them in the `YourProject/library/YourProject/View/Helper/` path and register that path in your `application.ini`. For now this view helper will only be used in the default module.

The params for the `direct()` method must all have a default value to pass the abstract contract. If you do force params to be set, by not defining a default value, you will get a fatal error.

The view helper should never output (echo) results, only return it. Once this is done you are ready to use the view helper in any of your view scripts by just calling the class name camel cased.

In this example we get back a paginator which contains a collection of comment entities for which we echo the escaped comment text.

See also

Usage on other already defined view helpers.

- Recipe: Using \Zend\View\Helper\Action for controller access from the View
- Recipe: Creating reusable display parts with \Zend\View\Helper\Partial or render() method
- Recipe: Using \Zend\View\View : Using the view resource
- Recipe: Creating layout scripts for global templating pages with \Zend\Layout\Layout

Creating layout scripts for global templating pages with \Zend\Layout\Layout

In a web application almost all pages have the same recurring markup.

They all specify the <html>, <head>, ..., maybe a menu and </html> tags. You do not want to have to maintain this recurring markup each time in every script you render. Especially for this we can define a layout script which will be used by every page request to promote one central handling and code reuse of view output.

How to do it...

Implement the following file:

`/YourProject/application/modules/application/views/layouts/layout.phtml`

```
<?php echo $this->doctype(\Zend\View\Helper\Doctype::XHTML1_STRICT);
?>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<?php
    echo $this->headTitle().PHP_EOL;
    echo $this->headMeta().PHP_EOL;
    $this->headLink()
        ->prependStylesheet('/css/ie.css', 'screen', 'IE 7')
        ->prependStylesheet('/css/default.css');
    echo $this->headLink().PHP_EOL;
    echo $this->headStyle().PHP_EOL;

    $this->headScript()
->prependFile('/js/someJavascriptfile.min.js');
    echo $this->headScript();
?>
</head>
<body>
    <div id="content">
        <?php echo $this->layout()->content; ?>
    </div>
    <?php echo $this->inlineScript(); ?>
</body>
</html>
```

Than initialize the layout resource by adding the following line to your `application.ini`

```
resources.layout.layoutPath = APPLICATION_PATH "/modules/application/
views/layouts"
```

How it works...

Only two pieces of code are actually related to the layout itself. `<?php echo $this->layout()->content; ?>` in `layout.phtml` and `resources.layout.layoutPath = APPLICATION_PATH "/modules/application/views/layouts"` in the `application.ini`.

All others are actually view helpers which are commonly used in template scripts.

So let us start at the beginning. The place where you would like to put your layout script. As with all architectural decisions this is an important one. In the example above I have specified a layout file inside the module directory. This means that you should only use that layout script from within that module itself. If you would like to share it between modules I would recommend creating a layout directory outside the modules directory. For now we only have one module so we can readily load the application layout resource and set the `layoutPath` for the entire application inside our `application.ini`. As soon as you add modules I will show you how to switch layouts per module in the there is more section. The Layout resource will plug the Layout inside your MVC application. Which means that from then of on, all your controller actions will have the layout loaded by default. And all of their output will be returned with $this->layout()->content;

Because I am showing you a web application the following view helpers are really important.

`doctype()`
this view helper enables you to set the doctype easily by passing a constant. It facilitates the setting of the necessary doctype because you do not have to know its w3c specifications. In the above example we set and echo the doctype in the same layout script. The setting can either be done here, `application.ini` (in the view resource) or for instance in a bootstrap.

`headTitle()`

does what it says it will do, it will return a title tag. You will be using
this view helper to set the web page title. Setting the title from somewhere in your application is done by calling the view helper on a view instance: `$view->headTitle('Your Special Project')`

`headMeta()`

is used to set all the name and equiv settings in your head. Like content encoding, description, keywords etc. Setting some metadata is done by calling the view helper on a view instance.

`$view->headMeta()->appendName($keyValue, $content, $conditionalName)`
or

`$view->headMeta()->appendHttpEquiv($keyValue, $content, $conditionalHttpEquiv)`

For setting the charset in a HTML5 doctype you can use the convenience method `$view->headMeta()->setCharset($charset)`

`headLink()`

is used to aggregate the used external link files. These can be stylesheets or for instance favicons. If you have a widget or an action view script that needs additional stylesheets you can add them in that view script and they will be echoed automatically into your <head> tag.

`$view->headLink()->appendStylesheet($href, $media, $conditionalStylesheet, $extras)` Where you can specify an optional conditional for IE in example `'lt IE 7'`

```
headStyle()
```

HeadLink is used to create links towards external stylesheets and headStyle is used to include CSS stylesheets inline in the HTML <head> element with the help of <style> tags.

```
headScript()
```

is used to aggregate all the javascript that should be loaded in your <head>.

```
$view->headScript()->appendScript($script, $type = 'text/javascript',
$attrs = array())
```

to set inline scripts in your head or

```
$view->headScript()->appendFile($src, $type = 'text/javascript',
$attrs = array())
```

to set a external script file

```
inlineScript()
```

is used to aggregate all the javascript that can be loaded at the end of your <body> tag. For performance reasons it is always best to move as much js from the <head> to the <body> as you can. Has the same API as `headScript`

There's more...

Changing the layout script per action, module and or controller

One of the most commonly asked questions regarding the layout is how to change the loaded script file dynamically. Because the layout script is rendered at the last step you can easily switch the script during application execution with the same command you would use to set it in your `application.ini`. Namely the `layoutPath` option by calling `setLayoutPath()` on a layout instance.
In an action we can use the layout helper to access the layout:

```
$this->_helper->layout->setLayoutPath('script-path');
```

`$this->_helper->layout->setLayout('scriptname');`
To switch for an entire controller you can initialize this in the public function `init()` which will be loaded at the end of the constructor of that controller. Or make the switch really per controller or per module with a frontController plugin. This has the benefit you have one central place to place your layout switching options and also avoid repeating the same code in each and every controller of a specific module.

`/YourProject/library/YourProject/Controller/Plugin/Layout.php`

```
<?php

namespace YourProject\Controller\Plugin;
class Layout extends \Zend\Controller\Plugin\AbstractPlugin
{
    public function routeShutDown(\Zend\Controller\Request\
AbstractRequest $request)
    {
        \Zend\Layout\Layout::startMvc();
        $layout = \Zend\Layout\Layout::getMvcInstance();
        $layout->setLayoutPath(APPLICATION_PATH.'/modules/'.$request-
>getModuleName().'/views/layouts');
        $layout->setLayout('default');
    }
}
```

In the Bootstrap `YourProject/application/Bootstrap.php`

```
protected function _initPlugins()
{
    $plugin = new YourProject\Controller\Plugin\Layout();
    \Zend\Controller\Front::getInstance()
    ->registerPlugin($plugin);
}
```

This will load the `default.phtml` script inside the appropriate module. If you want to set it per controller you can also do this with by using `$request->getControllerName()` in the path, or layout script name.

Setting the content encoding

You have to set the `content encoding` in three places for each application.
You have already seen how to set the content encoding in your header. This is a very important practice as it will avoid unnecessary loading of your web page until the first `<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />` tag is found, after which the browser will re-render the page for a second time with the right encoding set.

The second place where to set the encoding is inside the document head itself:
`$view->headMeta()->appendHttpEquiv('Content-Type', 'text/html; charset=utf-8');`
And third but very important is in the view. The view must know which encoding is used. It has to know this so the view helpers can use this encoding to do their logic correctly. Setting it in the view has nothing to do with setting it in the headMeta. Both have a complete different purpose. HeadMeta is for the browsers and
`$view->setEncoding('utf8');` is for the inner working of the application.

See also

- Chapter 1 Recipe: How to set headers per action - Setting the Unicode UTF-8 Header

Using \Zend\Paginator\Paginator to iterate over datasets

Pagination is the process of splitting data sets in blocks and assigning a system to get the next and or previous block of data, optionally by means of page numbers.

Getting ready

Zend Framework comes packed with several adapters by default. Each paginator adapter is specifically designed so it knows how to handle different data sources. Array, Db, Iterator. To make you understand how pagination works I need to create a different data source and show you how to create an adapter for yourself.

How to do it...

Creating a paginator adapter is easy and I will guide you through this process so you get an understanding on how the adapters work.

First we are going to create some other type than the ones supported by the Paginator Adapters. An `ArrayObject` should do just fine.

- Implement `/YourProject/library/YourProject/Collection/AbstractCollection.php`

```
<?php
namespace YourProject\Collection;
class AbstractCollection extends \ArrayObject
{
    protected $_maxCount;

    public function getMaxCount()
    {
        if (null === $this->_maxCount) {
            return $this->count();
        }
        return $this->_maxCount;
    }

    public function setMaxCount($int)
    {
        if (!\Zend\Validator\StaticValidator::execute($int,
'Digits')) {
            throw new Exception('maxCount needs to be an positive
integer');
        }
        $this->_maxCount = (int) $int;
    }
}
```

- Implement `/YourProject/library/YourProject/Collection/Comments.php`

```
<?php
namespace YourProject\Collection;
use \YourProject\Entity;
use \YourProject\Collection\AbstractCollection;

class Comments extends AbstractCollection
{
    public function offsetSet($key, $val) {
        if (! $val instanceof Entity\Comment) {
            throw new Exception('passed entity must be of type
Comment');
        }
        return parent::offsetSet($key, $val);
    }
}
```

- Implement `/YourProject/library/YourProject/Entity/Comment.php`

```
<?php
namespace YourProject\Entity;
class Comment
{
    protected $_text;
```

```
        public function setText($text)
        {
            $this->_text = $text;
        }

        public function __get($name)
        {
            $property = '_'.$name;
            return $this->$property;
        }
    }
```

- And finally start implementing the adapter itself:
- `/YourProject/library/YourProject/Entity/Comment.php`

```
<?php
namespace YourProject\Paginator\Adapter;
use YourProject\Collection\AbstractCollection;
use \Zend\Paginator\Adapter;

class Collection implements Adapter
{
    protected $_collection;

    public function __construct(AbstractCollection $collection)
    {
        $this->_collection = $collection;
    }

    public function count()
    {
        return $this->_collection->getMaxCount();
    }

    public function getItems($offset, $itemCountPerPage)
    {
        return $this->_collection;
    }
}
```

- Using the adapter is easy, and the same for all other adapters. For this example we are going to use in inside a view helper which is demonstrated in the "Using Custom \Zend\View\Helper querying the model" recipe.

```
$paginator = new \Zend\Paginator\Paginator(
    new \YourProject\Paginator\Adapter\Collection($collection)
);
$paginator->setCurrentPageNumber($pageNumber);
$paginator->setItemCountPerPage($qty);
```

How it works...

`\YourProject\Collection\AbstractCollection` is our base class from which all our collections will extend. It has a setter and a getter for the `maxCount`. An `ArrayObject` has the method `count()` which will return all the public properties of the object. But if we want to use this object with pagination we will want to know how many results there are in total, not only in the object itself. When we fetch only 10 results (entities) out of 200 from our DAO, the collection will only hold those 10, and count() would return 10. Yet the paginator will want to know that 200 results are available. That is why we have chosen the ability to set that total amount in the object and you will see that we use the getter in our adapter.

The collection `YourProject\Collection\Comments` itself overrides the public method `offsetSet($key, $val)` to force the type on the collection. Don't worry about `ArrayObject::append()` as this just proxies to `offsetSet(null, $val);`

This way we are certain that all properties in the `ArrayObject` are of the type `\YourProject\Entity\Comment`. The entity `\YourProject\Entity\Comment` only defines a text property with a getter and a setter. This concludes the "Collections and Entities" part of the model, you will see more of this in the chapter "Database handling and layering your app"

The custom paginator adapter `\YourProject\Paginator\Adapter\Collection` which implements `\Zend\Paginator\Adapter` will be used whenever we want pagination on result sets that exists out of collections. Each adapter must implement two abstract methods `count()` and `getItems()`.

We set the result set which we want to iterate in the constructor and implement the count to call the `getMaxCount()` of the registered collection.

The adapter expects the `count()` method to return the maximum amount of results which can be made available, that is why we needed the property `maxCount` in our collection. You can implement the `getItems()` method to do the logic of retrieving the correct result set out of the total result set available, and the count to do a count on the total result set, but both implementations are not necessary as you can see. In this implementation I do not want the adapter to know how to limit the result set, we have to simply return the limited result set that has been passed to the constructor. In comparison the `array adapter` accepts a complete result set and will do a `count()` on the array itself. The `getItems()` will return a slice out of the array, thus limiting the result set. This is a good system for small sets, but not for big result sets, where you would have to pass thousands of results just to get a small portion from it. The adapter I have shown you works with all types by taking the result set limiting and counting out of its logic. This logic you would want in the service and not in the paginator adapter. If you would put the logic for counting and limiting the result set in the adapter it would only know how to handle one particular service API. The paginator adapter should stay general and not limit itself to only one service.

Now that we have our paginator available we do a service request to get a limited data result set. The service will return a collection which is prepared to handle paginating because the total number of results have been set with `setMaxCount()`. You simply pass the result set in the constructor, tell the paginator where you are in the set by passing a current page number and the items count per page and you are done. With the page number and item count information it will know how many pages to display and where you are in your result set so it can return previous and next options. You now have a fully configured paginator which you can use to display.

There's more...

Displaying the paginator in your view scripts

To display a paginator a view helper called `paginationControl` has been made available, which is configurable with up to four parameters: the paginator instance, a scrolling style, a view partial, and an array of additional parameters. The second parameter is the scrolling style which defines the behavior of the pagination and the third parameter is the partial which is used to display the pagination.

```
echo $this->paginationControl($paginator, 'Sliding', 'viewscript.
phtml');
```

The second parameter 'Sliding' is actually the default behavior style and it will display the current page number in the middle of the current page range. For more styles see the manual.

If you supply additional parameters as the fourth parameter to paginationControl those parameters are available in the view script as public properties of the view.

Creating a paginationControl partial

Zend Paginator doesn't know how to display the pages itself. It is the job of you to implement the correct view script.

```
<?php if ($this->pageCount): ?>
    <div class="paginationControl">
        <?php if (isset($this->previous)): ?>
            <a href="<?php echo $this->url(array('page' => $this-
>previous)); ?>">
            &lt; Previous
            </a> |
        <?php else: ?>
            <span class="disabled">&lt; Previous</span> |
        <?php endif; ?>

        <?php foreach ($this->pagesInRange as $page): ?>
            <?php if ($page != $this->current): ?>
                <a href="<?php echo $this->url(array('page' =>
$page)); ?>">
                <?php echo $page; ?>
                </a> |
            <?php else: ?>
                <?php echo $page; ?> |
            <?php endif; ?>
        <?php endforeach; ?>

        <?php if (isset($this->next)): ?>
            <a href="<?php echo $this->url(array('page' => $this-
>next)); ?>">
            Next &gt;
            </a>
        <?php else: ?>
            <span class="disabled">Next &gt;</span>
        <?php endif; ?>
    </div>
<?php endif; ?>
```

Creating the partial is like any other view script, you will use logical structures to loop and switch conditions. And all of these act on public properties of the view available in the script. The public properties that are available to the pagination control view partial are described in the manual, I used the following.

current	Current page
next	Next page
pageCount	Number of pages
pagesInRange	An array of pages. Where each page is a page number.
previous	Previous page

Gluing pages together with url View Helper

A website exists out of multiple web pages. This recipe will show you how to glue those pages together by the means of dynamic links that plug into your routing system.

The url() view helper builds URIs for you and the router defines which URIs are available for assembly and also matches them.

How to do it...

For each reference to a web page, whether it is as an href attribute of an anchor tag or some dynamic javascript variable, you will need to use the `url()` view helper.

```
<a href="<?php echo $this->url(array(), 'application'); ?>"
title="awesome page" >hello world</a>

var myAjaxUri = '<?php echo $this->url(array(), 'application'); ?>';
```

How it works...

Grasping this little piece of logic is vital for the maintainability of your application. Imagine you would hard code the links with something like `href="/index/hello"` because initially you wanted the page URI to be like that, and later on you would want to change the URI to `/index_hello` you would have to change all occurrences of this link. With the view helper `url()` you do this at a central place, your routing level, and do not care about it in your view scripts. The corresponding URI will be assembled on the fly. `url()` accepts 3 parameters of which only 1 is mandatory, the first array. This array specifies configuration parameters needed by your `router` for assembly. The second parameter is the name of the router you want to use, by default it falls back to `application`. And an optional third parameter "reset" specifies whether you want to automatically reuse the currently set route params, so you do not have to set them again. By default this third parameter is set to false, thus reusing all currently set values.

See also

- Recipe: Routing, optimizing for SEO

Routing, optimizing for SEO

By default Zend Framework has a default `\Zend\Controller\Router\Rewrite` route "`application`" registered. This route takes an URI in a specific format and translates it to the correct module, controller, action. If you want to use another URI format you will have to create another route. The router has one or multiple routes registered to itself. Each route defines a blueprint on how to match an URI and also how to build (assemble) one.

You probably will need three route types plus chaining in your application. These routes are `\Zend\Controller\Router\Route\Hostname`, `\Zend\Controller\Router\Route\StaticRoute`

`\Zend\Controller\Router\Route\Regex` and the chain itself

`\Zend\Controller\Router\Route\Chain`.

This recipe will guide you through the process of setting such routes.

How to do it...

The simplest to use of all routes and probably the one you will use often is the `StaticRoute`. This is useful for the URI that do not require dynamic parts in it. An example can be a `/contact-us` URI.

In you bootstrap you can plug in your routes, because at the bootstrap process the application hasn't gone through the routing process yet. This means we can make additional routes available for that process.

```
protected function _initRoutesStatic()
{
    $router = $this->getPluginResource('frontcontroller')
         ->getFrontController()->getRouter();
     $staticRoute = new \Zend\Controller\Router\Route\
StaticRoute('contact-us',
        array(
            'module' => 'application',
            'controller' => 'company',
            'action' => 'contact'
        )
     );
```

```
        $router->addRoute('contactus', $staticRoute);
    }
```

How it works...

First we get the default router registered in the frontController. This is needed because we will add routes to this router. In the example I only plug in one route but you can add as many routes as you want. We initiate a `\Zend\Controller\Router\Route\StaticRoute` with an URI path it will have to match and as second parameter the module, controller and action it should dispatch to. For each route you add you must specify a route name. This is needed to ensure uniqueness but also for assembly with the url view helper which accepts the route name as a second parameter.

There's more...

- Recipe: Gluing pages together with url View Helper

Creating a complete dynamic URI with the Regex Route

Often you will want dynamic URIs to display some metadata in it for SEO optimization and just to create user friendly URIs. This is easily done with the usage of Regex

Imagine an URI like http://yourproject/1245/this-is-an-awesome-blog-article

```
    protected function _initRoutesRegex()
    {
        $router = $this->getPluginResource('frontcontroller')
             ->getFrontController()->getRouter();
         $staticRoute = new \Zend\Controller\Router\Route\Regex(
            '(\d+)/([a-z-]+)',
            array(
                'module' => 'application',
                'controller' => 'article',
                'action' => 'index',
                'articleId' => '0',
                'articleName' => '',
            ),
            array(
                1 => 'articleId',
                2 => 'articleName',
            ),
            '%d/%s'
        );
        $router->addRoute('blogArticle', $staticRoute);
    }
```

This route will match all URIs which have the format of `'(\d+)/([a-z-]+)'` which translates to a digit followed by a slash and some alphabet string with a dash as the word separator. Because it is a simple `regex` we can take the groups and make them available for us later on in our application. First to be able to retrieve them we should set them in the second array. This second array is the position map. Here we map each regex group by position to a default user param name (which is set in the first array). Because we have mapped the position we will also be able to assemble an URI from the route itself by providing a fourth parameter, the `reverse`. This reverse is in a `printf` format and will use the map to feed the params in the correct order. Later in our application we will be able to retrieve the articleId and articleName by using

`$request->getUserParam().`

Creating and matching subdomains with Hostname Route

Subdomains are something wonderful. You can switch to complete new websites or subsections of a site depending on them. You can easily use them as SEO techniques (which is discouraged but heavily used by some pirate sites because of it ranking power).

So I know you will want to use them also. And Zend Framework naturally allows full control depending on the host.

```
    protected function _initRoutesHostnameAndShowChaining()
    {
        $router = $this->getPluginResource('frontcontroller')
            ->getFrontController()->getRouter();

        $hostnameRoute = new \Zend\Controller\Router\Route\Hostname(
            ':username.user.yourproject',
            array(
                'module' => 'specialmodule',
            )
        );

        $staticRoute = new Zend\Controller\Router\Route\
    StaticRoute('users',
            array(
                'controller' => 'user',
                'action' => 'index'
            )
        );

        $router->addRoute('usershost',
            $hostnameRoute->addChain($staticRoute));
    }
```

You can use hostname routes as is, without chaining them. But that would make little sense as all other routes are path routes and would always match even if they should not match for a particular subdomain. The solution is chaining.

Using a `\Zend\Controller\Router\Route\Hostname` is easy, you specify a host name and in that host name you can put an optional user param,which is indicated with a colon (:) followed by the param name. In the example `:username`, which would translate to

`$request->getUserParam('username')` for retrieval. In comparison with a regex you do not have the need to specify a reverse or a map because this is name based and not position based.

The chaining itself is done in order of matching. We first want the host to match before doing path matching, thus we chain the `staticRoute` to the `hostnameRoute`. A chain will try continuing to match until all added routes are matched. And only when all routes in a chain are matched does the route qualify as a match and will you be dispatched to the specified module, controller, action.

See also

- Recipe: Getting input from the user with \Zend\Controller\Request\Http - Accessing the User Params
- Recipe: Your first controller and action \Zend\Controller\Action
- Recipe: Gluing pages together with url View Helper
- Accompanied code chapter2: \YourProject\Application\Bootstrap.php

Easy redirecting with the redirector action helper

Redirecting is the process of pointing someone to another place. In the web this means towards another web page. We can have lots of different sort of redirects, permanent, temporary, and so on. In a Zend Framework project the action helper Redirector facilitates this process for you in. Remember we do not want to output the headers directly with the usage of PHP's `header()` function. In MVC this should be the responsibility of the response object. Also you might want to do some cleanup stuff before doing a redirect, so you do not want the header to be spit out immediately.

How to do it...

A `\Zend\Controller\Action\Helper\AbstractHelper` instance is mainly used from within an controller action. The redirector is no different. So let us implement an action with a 301 Moved Permanently redirect.

```
public function obsoletePageShouldRedirectToHomepageAction()
{
    $this->_helper->redirector
            ->setExit(false)
            ->gotoRoute(
            array(
                'action' => 'index',
                'controller' => 'index',
                'module' => 'application',
                'id' => 15,
            ),
            'application'
        );
}
```

How it works...

First we call the redirector action helper through the convenience overloading build in into the action controller. This is a shortcut to

`$this->getHelper('redirector')`, in analogy with `$this->getRequest()` or `$this->_request.` You can use whatever notation you or your editor likes the most. Next we set the response status code and then set the URI using one of the available set methods.

The `gotoRoute` is a method which will built the URI the same way the url view helper does so it has exactly the same interface. That is why this is my preferred method above the `gotoSimple` which might be convenient at times, but it also offers far less functionality. All methods like `gotoRoute()`, `gotoSimple()`, `gotoUrl()` force an immediate sending of headers and exiting the script with exit(). If you do not want the response object to immediately sent the headers and do an exit(), but allow for the entire the project to run through the entire process, you have to set the exit flag to false. (this way you can still do some application cleanup.)

There's more...

Redirecting with gotoSimple()

As previously mentioned Redirector has a gotoSimple method. This is a convenience method for assembling the URI based on the `application` route, which is the default from Zend Framework. The only thing you have to pass is an action name as a first parameter. All others are optional. If you do not pass the controller or module name as second or third parameter respectively it falls back to the default ones registered in the dispatcher. As the fourth parameter you can pass other optional userparams with which the route can assemble the URI.

```
public function obsoletePageShouldRedirectToHomepageAction()
{
    $this->_helper->redirector
        ->setCode(301)
        ->setExit(false)
        ->gotoSimple(
            array(
                'index',
                'index',
                'application',
                array(
                    'id' => 15
                )
            )
        );
}
```

You can also call the gotoSimple directly.

```
$this->_helper->redirector(
    array(
        'index',
        'index',
        'application',
        array(
            'id' => 15,
        )
    )
);
```

Be careful because this will not give you the option to set the status and exit flag. By default a 302 and true.

Redirecting with gotoUrl()

For all internal and external URIs which cannot be assembled by the registered router you should use `gotoUrl()`. These can be relative or absolute. The absolute URI setting doesn't require additional information, but there is some tweaking to be done when working with relative URIs.

```
$this->_helper->redirector
    ->setCode(301)
    ->setExit(false)
    ->setPrependBase(false)
    ->gotoUrl('/some/nice/relative/url');
```

A Zend Framework project can be configured to run under a path on your domain. Maybe you want use Zend Framework only for a part of your website and let it run on `yourproject.com/admin` instead of `yourproject.com`. Making yourproject.com a mixed technology website. For instance everything in `/admin` will bootstrap Zend Framework ,everything outside this folder might be legacy code. To make this work you have to set the `baseUrl` in the frontController. By default this is `null`. Yet once set to `/admin` everything in Zend Framework which has to do with routing will ignore this part for matching and use it for assembling. If you want to reach the legacy code you would have to tell the redirector not to prepend the relative URI with the base url `/admin`. If you would not set this flag to false it would return `/admin/some/nice/relative/url`.

Redirecting with the controller action shortcut _redirect

The `gotoUrl()` redirector usage from above has been simplified for your convenience by an added convenience method `$this->_redirect()` inside `\Zend\Controller\Action`.

```
$this->_redirect($url, array $options = array())
```

proxies to

```
$this->_helper->redirector->gotoUrl($url, $options);
```

See also

- Chapter 1 Recipe: How to set headers per action

Building menus with Navigation

`\Zend\Navigation\Navigation` will give you a central place to create all your menus, breadcrumbs, sitemaps and so on. Often menus are linked to some kind of ACL, `\Zend\Navigation\Navigation` has support built in. It is a very dynamic and easy to use system so lets take a look.

How to do it...

Menus are built out of pages (links towards them) and what we want to do is built a menu which contains submenus and display this in our view.

`/YourProject/libary/YourProject/Controller/Plugin/Navigation.php`

```
<?php
namespace YourProject\Controller\Plugin;

class Navigation extends \Zend\Controller\Plugin\AbstractPlugin
{
    public function dispatchLoopStartup(\Zend\Controller\Request\
AbstractRequest $request)
    {
        $nav = $this->_getNavigation();
        $view = \Zend\Controller\Action\HelperBroker::getStaticHelper(
'viewRenderer')->view;
        $view->navigation($nav);
    }

    protected function _getNavigation()
    {
        $nav = new \Zend\Navigation\Navigation();
        $zfPage = new \Zend\Navigation\Page\Mvc();
        $zfPage->setLabel('Zend Framework Navigation');
        $zfPage->setRoute('application');
        $zfPage->setModule('application');
        $zfPage->setController('zf');
        $zfPage->setAction('index');
        $nav->addPage($zfPage);

        $introductionPage = new \Zend\Navigation\Page\Uri();
        $introductionPage->setLabel('introduction');
        $introductionPage->setUri('http://framework.zend.com/manual/
en/zend.navigation.introduction.html');
        $zfPage->addPage($introductionPage);

        $dosAndDontsPage = new \Zend\Navigation\Page\Mvc();
        $dosAndDontsPage->setLabel('Do\'s and Don\'ts');
        $dosAndDontsPage->setRoute('application');
        $dosAndDontsPage->setModule('application');
        $dosAndDontsPage->setController('zf');
        $dosAndDontsPage->setAction('dos-and-donts');
        $zfPage->addPage($dosAndDontsPage);
```

```
            return $nav;
        }
    }
```

```
/YourProject/application/Bootstrap.php
```

```
    protected function _initPlugins()
    {
        $plugin = new YourProject\Controller\Plugin\Navigation();
        $this->getPluginResource('frontcontroller')
             ->getFrontController()
             ->registerPlugin($plugin);
    }
```

And in your random view script you can render the registered menu by adding the following line.

```
echo $this->navigation()->menu()->render();
```

How it works...

You have to create your pages container and assign it to the view. There are several places where you could initialize the container `\Zend\Navigation\Navigation`, as an application resource, bootstrap resource (_init method) , frontController plugin, as an action helper or in a view helper. Initialization all depends on how much configuration is needed. For a really simple menu, without any ACL or dynamics added you could put it in a view helper, bootstrap resource or application resource. As soon as you need access to the request object and such, I would advice to initialize it at a frontController plugin. In the above example I have showed the frontController plugin way. The actual navigation building is done in the `_getNavigation` method. Here we initialize a page container `$nav` on which you add all the pages themselves. You have two type of pages, Mvc and Uri. You should use `\Zend\Navigation\Page\Mvc` to create pages which are accessible in your Zend Framework MVC project, and `\Zend\Navigation\Page\Uri` to create pages outside of your application. The label is used to create the text of the <a> tag. You can also specify link attributes like class, id, title, rev, rel, target. All of these attributes can be set with their appropriate setter: setClass, setTitle and so on. Each page extends `\Zend\Navigation\Container` which allows you to create navigation tree branches by adding pages, for instance to built submenus. On `\Zend\Navigation\Page\Uri` you have to define a label and an URI and that is all. But `\Zend\Navigation\Page\Mvc` needs some more tweaking. Because the container will be used for breadcrumbs and menus within the application the navigation needs to know where you are when you are on a web page.

It does this by comparing the current module, controller and action to the ones specified in the page. When it finds a match it will add a class "`active`" to the displayed list of links and optionally only show that branch of the tree that is related to that web page. But the page also needs to know to assemble the URI for the href attribute. For this it will use the route that is set. If the route needs parameters you can use the `setParams()` method.

You can built your own view script which will iterate over the container and built the menu for you, but Zend Framework has created a view helper that does all that magic for you. The only thing you have to do is register the container to the navigation view helper and you can render the menu with one line of code. In our `dispatchLoopStartup` we get the view from our `viewRenderer` plugin, which is registered by default, and assign the navigation to it.

The rendering of the menu is done with one line of code, but behind this code you can put a lot of configuration. The example uses the default configuration. You can configure on two levels, `navigation()` and on `menu()` level. You can take a look at them in the next there is more recipes.

There's more...

Using the factory

The above `getNavigation()` code can be rewritten.

```
     protected function _getNavigation()
    {
        $menu = array(
            array(
                'label' => 'Zend Framework Navigation',
                'route' => 'application',
                'module' => 'application',
                'controller' => 'zf',
                'action' => 'index',
                'pages' => array(
                    array(
                        'label' => 'introduction',
                        'uri' => 'http://framework.zend.com/manual/en/
    zend.navigation.introduction.html'
                    ),
                    array(
                        'label' => 'Do\'s and Don\'ts',
                        'route' => 'application',
                        'module' => 'application',
                        'controller' => 'zf',
                        'action' => 'dos-and-donts',
                    ),
                ),
            ),
        );
        $nav = new \Zend\Navigation\Navigation($menu);
        return $nav;
    }
```

`\Zend\Navigation\Navigation` will accept the array and detect which type to use automatically. When you pass mvc params it will built an instance of `\Zend\Navigation\Page\Mvc` automatically, otherwise when it detects an URI param it will instantiate a `\Zend\Navigation\Page\Uri` instance.

Also a complete `\Zend\Config\Config` instance may be passed to the constructor which will enable you to build the container in an `.ini` or `.xml` format.

Understanding the navigation() view helper

Building views for breadcrumbs, links, menus and sitemaps can be done by calling their respective navigational view helpers. But initializing those view helpers can be done at one central place and that is in the navigation view helper. This is a proxy helper on which you can set an ACL, navigation and or translator object and it will inject these on all navigational helpers. This way you only have to set initialization only once and also do not care about loading the navigation helper namespace.

Translating your link texts aka labels

Setting labels is easy as shown above. Translating them is also a one liner. Make sure your translation object is registered in the `\Zend\Registry` with the key `Zend_Translate` or set it by using `setTranslator()` on the view helper directly or on the navigation proxy to set it for all helpers. By default `navigation()` will try to translate the labels, if you do not want this you can set the default to `false`.

```
$this->navigation()->breadcrumbs()->setUseTranslator(false)
```

or

```
$this->navigation()->setUseTranslator(false)
```

Using ACL with your view helpers

Because all navigational view helpers extend the `\Zend\View\Helper\Navigation\AbstractHelper` you can set ACL per helper or on the proxy.

```
$this->navigation()->menu()->setAcl($acl)->setRole($role);
```

or

```
$this->navigation()->setAcl($acl)->setRole($role);
```

For ACL to work you will also have to specify a role. So you have to specify both to navigation(). By default ACL is activated as soon as you set one. If for some reason you want to temporarily deactivate it you can call

```
$this->navigation()->setUseAcl(false);
```

Controlling the depth of the navigation container

Because navigation is a container which contain pages which are containers on itself, you have to be able to specify which branches you want to display and the depth of the branches.

These options are per navigational helper. So you cannot set them at the proxy level.

By default the `minDepth` is `null` (no minimum depth or 0). This is the top level of your container. With each embedded container the depth goes up with one. Only displaying the "introduction" and "Do's and Don'ts" pages which are at a depth of 1 can be done by using the appropriate setter.

```
$this->navigation()->menu()->setMinDepth(1)->render();
```

same applies for a `maxDepth`. If you only want to display the top level pages

```
echo $this->navigation()->menu()->setMaxDepth(0)->render();
```

For the `menu()` helper you also can specify that you only want the entire depth for the active branch of the tree (navigation container) displayed.

```
$this->navigation()->menu()->setOnlyActiveBranch(true)->render();
```

By default this is set to `false`.

Not using the markup auto generation

By default the navigational view helpers render HTML markup by default. If you want to render your own markup you can specify a partial and do the generation yourself for maximum flexibility. This would require you to get intimate with the entire API of `\Zend\View\Helper\Navigation\AbstractHelper` if you want the ACL, and other setters which have been passed to navigation() to work.

```
    $this->navigation()->menu()
    ->setPartial('main-navigation.phtml')->render();
```

See also

- Chapter 10 Recipe: Authorization with \Zend\Acl\Acl
- Chapter 8 Recipe: Making a multilingual website \Zend\Translator\Translator

Summary

This is one of the most important chapters in the book. It handles the correct usage of the view part of the MVC and how to keep it performable and maintainable with the correct usage of action(), render() and partial().

It also goes into depth on how to build your entire site navigation, with the help of routers and the view helper. Ensuring that your system stays flexible on the URI part also.

Together with the view and router other Zend components such as navigation and pagination are available and those components also help you keeping it DRY.

This has been a chapter with lots of code and abstract explanations, so I suggest that you get your hands dirty and take a look at the code that accompanies this chapter. It will make things really clear.

The most important thing I want you to take with you from reading this chapter and making your hands dirty is an eagerness for developing a good maintainable and flexible site with Zend Framework. A centralized routing system, and well categorized view scripts powered by view helpers can deliver that for you. See you in the next chapter.

3
Optimizing your application

In this chapter, we will cover:

- Creating a model
- Caching for performance
- Reading configuration files
- Extending controllers with action helpers
- Using application hooks with \Zend\Controller\Plugin\Abstract Plugin
- Using modules as widgets
- Using modules for a cleaner application structure
- Creating application resources for clean configuration
- Speed up your plugins

Introduction

In the first two chapters you have taken a look at how to get a Zend Framework project running and you already started to do some cool stuff. This chapter will show you how to do it correctly from an application point of view. It will show you how to create a model, build configuration files for less coding and easy reuse. Creating and using cache to optimize for performance and build plug and play logic which you potentially can drop in each one of your ZF projects.

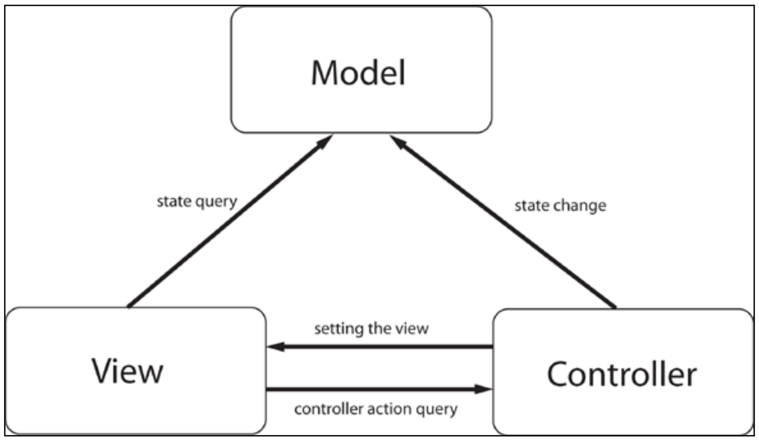
Creating a model

This recipe explains that a model is not data access, for instance from a database. Data access is part of the model. Zend Framework is often blamed for not really having a Model in the MVC stack because it doesn't come equipped with something that builds your objects and database code like ORM software. The object that represents data from a database, such as for instance a User Model is a model but not the Domain Model, which is what the M in MVC is all about. The domain Model exists out of many models which in total represent a system's data and behavior. It is thus the domain-specific representation of data upon which the application operates. Your Controller and View may be aware of this Model, but your Model should not be aware of them.

In short your Model actually is all the business logic that doesn't handle user input validation. User input validation and registering should be done by the controller. All the real domain logic and such should be done in the Model (with its own validation). Thus keeping your controllers really slim and your model potentially very fat.

Building a good model requires some thinking, but once you have done it a couple of times, it will feel second nature and layering will almost happen automatically. The model is the heart of the application and the time you invest in building this is well worth the effort.

How to do it...



How it works...

Your application exists out of your controllers, views and model.

Your controller is there to take user input and validate it, depending on that input, call the correct actions to be taken on the model and render the correct view. Not all data from the model should pass through a controller. If no manipulations are done on the model why should the controller be a pass-through-middle-man? So the view in itself can receive information from the controller and request information from your model, but not alter it. Because both the view and the controller rely heavily on the Model, the model is the heart of your application. Additionally the model in itself can also have input validation checks to make sure no unwanted operations from the controller or view are made.

Because the model is not aware of the view and controller, you could easily drop it into another framework.

Caching for performance

Caching is the art of serving stored information blocks. Building these information blocks are generally resource greedy and therefore are commonly serialized and stored as strings. Because building these blocks is so resource greedy we would want the building of these blocks as few times as possible. Once build their content is stored statically and thus also very fast to serve. The most known use case for caching is with database or web service results. Fetching these results is time and resource expensive and for every request to the page the same query would run and serve the same results. That is why it is often better to store the result and serve this saved version on future requests. It is faster and allows for more concurrent queries.

Zend Framework offers various caching front ends (use cases) and different back ends (where to store the information/cache).

How to do it...

```
$frontendOptions = array(
   'lifetime' => 3600,
   'automatic_serialization' => true
);
$backendOptions = array(
    'cache_dir' => PROJECT_PATH.'/data/cache'
);

$cache = \Zend\Cache\Cache::factory('Core',
                            'File',
                            $frontendOptions,
                            $backendOptions);

if (!$result = $cache->load('uniqueCacheKey')) {
    $result = rand(0,30);
    $cache->save($result);
}
if (!$anotherResult = $cache->load('anotherUniqueCacheKey')) {
    $anotherResult = rand(0,30);
    $cache->save($anotherResult);
}
echo $result.PHP_EOL.$anotherResult;
```

How it works...

Caching a block of information is done using a cache frontend and a backend. Zend Framework offers

`\Zend\Cache\Cache::factory($frontend, $backend, $frontendOptions = array(), $backendOptions = array(), $customFrontendNaming = false, $customBackendNaming = false, $autoload = false).`

You should always use this factory method to instantiate your cache. Except in the case of the cacheManager which will manage the instantiation of the cache for you through the means of templates (see there is more section).

In the example above we have instantiated a `\Zend\Cache\Frontend\Core` object with the backend `\Zend\Cache\Backend\File` set to it. For the frontend options we have specified a cache lifetime of 3600 seconds which means that after the `save` operation the cache will be valid for that amount of time, after which it will have to be rebuild. Automatic serialization has been enabled which will serialize everything that will be passed to the `save` method, saving you the effort of doing it yourself. Serialization is often used when caching arrays or objects but also other types benefit from it. The file backend needs to know where to store the cache files, you can specify this with the option `cache_dir`.

The frontend and backend options that are available depend per frontend and backend type, so it is important to take a look at the reference guide or the code itself to see all the options available. Once your `\Zend\Cache\Frontend\Core` object is available you can use the load method to return the cached information. When it is serialized it will automatically unserialize and when the cache for the passed cache key is not valid anymore it will return `false`. When it returns `false` we build the information to be cached and save it into the cache. For demonstration purposes I only save the result from a rand(0,30) which will give me a number between 0 and 30. This piece of code is plug and play and shows the cache in action by always echoing the same number for up to one hour. In a real life scenario you wouldn't want to cache such a thing but cache an expensive operation such as a call to a database, a web service or the result of another time or resource consuming operation. You also want to make the `uniqueCacheKey` truly unique and not reuse it for other save operations. Be aware that you have to be able to recreate the cache key for the load operation. As you can see we can reuse the cache object for multiple loading and saving operations. And this is the way the cache object should be used. There is no need to instantiate each time a new cache object. You can specify the lifetime on the save operation and not only during instantiation.

There's more...

Changing the lifetime per save operation

By setting the frontend `lifetime` option through the instantiation parameter you specify a lifetime for all save operations on that cache object. Sometimes however you would want to change the lifetime for a specific save operation. You can do this by setting the fourth parameter on the save method.

```
save($data, $id = null, $tags = array(), $specificLifetime = false,
$priority = 8)
```

Alternatively you can also change the lifetime for all future save operations on the cache object by using the `setLifetime($newLifetime)` method. This will overwrite the value you have set at instantiation.

Using tags for mass removal of your saved cache

Every cache-item has a unique key which you use for retrieval, but with each cache saving you can also specify tags.

```
save($data, $id = null, $tags = array(), $specificLifetime = false,
$priority = 8)
```

`$tags` is an array of strings under which you want a cache-item to be categorized. Later you will be able to remove all cache-items under a specific tag.

Remove cache items MATCHING the tags 'tagOne' AND 'tagTwo':

```
$cache->clean(

       \Zend\Cache\Cache::CLEANING_MODE_MATCHING_TAG,
       array('tagOne', 'tagTwo')
   );
```

Remove cache items NOT matching the tags 'tagOne' OR 'tagTwo':

```
$cache->clean(
    \Zend\Cache\Cache::CLEANING_MODE_NOT_MATCHING_TAG,
    array('tagOne', 'tagTwo')
);
```

Remove cache items MATCHING the tags 'tagOne' OR 'tagTwo':

```
$cache->clean(
    \Zend\Cache\Cache::CLEANING_MODE_MATCHING_ANY_TAG,
    array('tagOne', 'tagTwo')
);
```

alternatively you can also clean all records or only the outdated.

```
$cache->clean(\Zend\Cache\Cache::CLEANING_MODE_ALL);
$cache->clean(\Zend\Cache\Cache::CLEANING_MODE_OLD);
```

Using \Zend\Application\Resource\Cachemanager

Zend Framework lets you instantiate a `\Zend\Cache\Manager` automatically with an application resource `\Zend\Application\Resource\Cachemanager`. This resource takes configuration settings in your `application.ini` and from then on you can have access to different types of cache from your entire application.

Configuring your application resource.

`YourProject/application/configs/application.ini`

```
resources.cachemanager.service.frontend.name = Core
resources.cachemanager.service.frontend.options.automatic_
serialization = true
resources.cachemanager.service.frontend.options.lifetime = null
resources.cachemanager.service.backend.name = File
resources.cachemanager.service.backend.options.cache_dir = PROJECT_
PATH "/data/cache"
```

Using the resource from within your application

`YourProject/application/modules/application/controllers/IndexController.php`

```
public function showCacheUsingCacheManagerAction()
{
    $cache = $this->getInvokeArg('bootstrap')
                  ->getResource('cachemanager')
                  ->getCache('service');
```

```
        if (!$result = $cache->load('YetAnotherUniqueCacheKey')) {
            $result = rand(0,30);
            $cache->save($result);
        }
    }
```

In this block of code I grab the result from the cache manager resource init() and from that result (a `Zend\Cache\Manager` instance) I get the cache with the name I defined service. I have configured it in the `application.ini` to be push cache by setting the `lifetime` to `null`, which effectively configures the cache to have no expiry time.

Pull and push cache

There are two types of cache: pull and push.

The difference between the two is that with pull cache when the lifetime has expired a new cache is generated, pulled in existence. This means that when a certain number of users come to your site while the cache is invalid they all generate the new cache-item. This cache-item is identical for all users and should have been only requested to be made once. Using pull strategy is the easiest way to implement but has the drawback of making the site slow for all the users that are in the race condition of creating the cache. It also potentially makes your application exceed your API calls to the service which you are caching. With pull cache your cache-item is also invalid even when there is no new information available from your service.

A workaround on the race condition is to let the first failed `load` call `touch` your cache-item. Touching means setting the creation date to now. This means it will act as a freshly created item and all subsequent users will not enter a race condition but get the 'old' cache-item instead. The first user who receives `false` from `load()` will `touch()` first, then built and finally `save()`.

```
        if (!$result = $cache->load('uniqueCacheKey')) {
            $cache->touch('uniqueCacheKey',60);
           $result = rand(0,30);
            $cache->save($result);
        }
```

In the example above we make the old cache valid again for sixty seconds. Besides solving the race condition you also have the benefit of a more stable application. If something would go wrong in your failed load block (for instance a service is not reachable and an exception is thrown), old cache will be served, guaranteeing that content is served.

On the other hand you have push cache which is more difficult to implement, because you have to map somewhere outside the workflow of your application all the different queries used inside your application. This is good for very small sites, but in my experience using it in big complex applications gives you nightmares. However if you can handle it, you gain a very fine grained control on when and how a cache should be renewed. With push you define a lifetime to never expire. In ZF that means putting it to `0` in code or `null` in your `application.ini`. Doing this means that your cache will never be renewed. It is up to you to push new cache into existence. That can be done by an observer pattern or by a cron job, where the cron job is the lifetime of the cache-item.

You also have a mix where you still use the pull strategy but you push the cache to be recreated by deleting all existing cache. This can be on a call to action (for instance every time an editor makes a change in your CMS and presses the publish button) . This still generates a race condition but it does it only when you know new content is available.

Reading configuration files

Configuration files configure the initial settings for that which you want to configure. From a human perspective they are easy to read and maintain. Using configuration files give you access to a centralized place where configuration takes place for an entire application (`application.ini`) or to other more specific configurations such as available routes, navigational menus and so on.

How to do it...

Zend Framework currently supports out of the box `\Zend\Config\Config`, `\Zend\Config\Ini` and `\Zend\Config\Xml`, which let you respectively parse `array`, `.ini` and `.xml` configurations.

Reading a `.ini` configuration file.

```
$config = new \Zend\Config\Ini(APPLICATION_PATH.'/configs/application.
ini', 'development');
```

Reading a `.xml` configuration file.

```
$config = new \Zend\Config\xml(APPLICATION_PATH.'/configs/application.
xml', 'development');
Reading an array configuration
$config = new \Zend\Config\Config(
    array(
    'cachemanager' => array(
        'service' => array(
            'frontend' => array(
                'name' => 'Core',
                'options' => array(
                    'automatic_serialization' => true,
                    'lifetime' => null,
                ),
            ),
            'backend' => array(
                'name' => 'File',
                'options' => array(
                    'cache_dir' => PROJECT_PATH.'/data/cache',
                ),
            ),
        ),
    ),
);
```

How it works...

`public function __construct($xml, $section = null, $options = false)`

The constructor of `\Zend\Config\Xml` accepts a xml string or a filename. `$section` is the section you want to load. In case of our `application.ini` this would refer to `production`, `development`, `testing`. A section can inherit and override properties from the section it extends. If `$section` is null all sections are loaded.

You can also pass the following `$options` as an array:

`allowModifications`

By default this is `false` and specifies whether modifications to the configuration file are allowed at runtime.

`SkipExtends`

By default this is `false` and specifies whether the section specified should load the configuration setting from its parents section. If set to true only those settings specified in the section loaded will be available, effectively disabling inheritance.

If you do not care about `SkipExtends` you can pass a `boolean` to `$options` which will set `allowModifications`.

`public function __construct($filename, $section = null, $options = false)`

For `\Zend\Config\Ini` you have to provide a filename and also optionally a section. Other directive options are `allowModifications, skipExtends, nestSeparator.`

Again if you pass a `boolean` to `$options` it will set `allowModifications`, if you pass an array the three configuration directives may be set. `NestSeparator` is a `string` identifier that separates nesting levels of configuration data. By default this is a dot (.).

```
public function __construct(array $array, $allowModifications =
false)
```

Sometimes you have a configuration available in an array format instead of using the concrete adapters `\Zend\Config\Ini` and `\Zend\Config\Xml` you can use `\Zend\Config\Config` to build a configuration object. As you can see you do not have an option to pass the section which means that array configurations do not support inheritance.

There's more...

Using a \Zend\Config\Config object

A lot of Zend Framework components accept a config object to create and configure an instance of the component. This enables you to create configuration files, read them using the appropriate adapter and pass the `\Zend\Config\Config` object to `\Zend\Application\Application`, `\Zend\Navigation\Navigation`, `\Zend\Translator\Translator`, `\Zend\Controller\Router\Rewrite` and other components.

In the code accompanying this chapter I have included a lot of configuration files in different formats so you can see how they are built and used with the appropriate component.

`\Zend\Config\Config` also provides a nested object property based user interface for accessing this configuration data within application code.

```
$config->cachemanager->service
```

```
->frontend->options->automatic_serialization
```

Gives you the value of `automatic_serialization`.

Merging instances

You can also merge two `\Zend\Config\Config` instances if needed.

```
$config->merge($otherConfig)
```

The settings from `$otherConfig` will override the `$config` settings with the same name.

Pay attention that overriding configuration settings after creation is only allowed when `$allowModifications` on `$config` is set to `true` as a parameter in the constructor. After merging you can use `$config->setReadOnly()` to set it to a read only state. This method doesn't accept any params and will always set the configuration to not allow modifications. In the chapter 3 code you can see a use case of this to nicely group your routes over multiple files. This way you can keep your configuration files small enough to be modifiable and readable.

See also

- Chapter code for examples on different application, route and navigational configuration files.

Extending controllers with action helpers

With `\Zend\Controller\Action` you often have the need for common Action Controller functionality. You can use the action helpers automatically using the dispatch loop (`viewRenderer`) and/ or on-demand (`redirector`). Both `\Zend\Controller\Action\Helper` and `\Zend\View\Helper` helper systems help minimize the necessity to creating a common view or controller action base class packed with common functionality. This way keeping your code clean, lean fighting machines, that do not carry all their weapons on themselves which would slow them down, hinder their movements and eventually make them loose their fighting purpose. We want them clean and uncluttered but we want to have a rack available for them packed with weapons so they can pick the weapon of choice at the time they need it. Such a weapon rack is available in Zend Framework in the form of a brokerage system, which handles the details of registering helper objects and helper paths, as well as retrieving helpers on-demand.

How to do it...

Action helpers are really useful tools and the logic can be as complex as you want it to be. It can be a shortcut to your routing system (url), to your view (viewRenderer) or even to some custom created code that assists you in setting the meta data in your view. For this I have created a MetaData action helper `/NbeZf/Controller/Action/Helper/MetaData` which will set the meta data automatically and still let you override the default settings on the fly from within an action if needed.

`/YourProject/library/NbeZf/Controller/Action/Helper/MetaData.php`

```
<?php
namespace NbeZf\Controller\Action\Helper;
class MetaData extends  \Zend\Controller\Action\Helper\AbstractHelper
{
    protected $_metaDataConfig;
```

```
        protected $_currentMetaDataSettings;

        public function __construct(\Zend\Config\Config $metaData)
        {
            $this->_metaDataConfig = $metaData;
        }

        public function direct()
        {
            return $this->getCurrentMetaDataConfig();
        }

        public function postDispatch()
        {
            if ($this->getRequest()->isDispatched()
            && !$this->getResponse()->isRedirect()) {
                $config = $this->getCurrentMetaDataConfig();
                $view = \Zend\Controller\Action\HelperBroker::getStaticHel
    per('ViewRenderer')->view;
                $view->headTitle($config->title);
                $view->headMeta()->setName('Description', $config-
    >description);
                $view->headMeta()->setName('Keywords', $config->keywords);
            }
        }

        public function getCurrentMetaDataConfig()
        {
            $sectionParts = array(
                $this->getRequest()->getModuleName(),
                $this->getRequest()->getControllerName(),
                $this->getRequest()->getActionName()
            );

            do {
                $sectionName = implode('-', $sectionParts);
                if (null === array_pop($sectionParts)) {
                    return new \Zend\Config\Config(
                        array(
                            'title'=> '',
                            'description'=> '',
                            'keywords'=> '',
                        ),
                        true
                    );
                }
            } while (null === ($config = $this->_metaDataConfig-
    >$sectionName));
            return $config;
        }
    }

//inside a controller action

    public function indexAction()
    {
        $this->_helper->metaData()->title = 'this is awesome';
    }
```

How it works...

To create an action helper you have to extend the `\Zend\Controller\Action\Helper\AbstractHelper` which defines the interface for a helper. The interface doesn't force you to implement a single method (none are abstract) but gives you access to many.

`direct()` this is used for immediate access when you call the helper directly

`$this->_helper->metaData()`. In our case it returns a `\Zend\Config\Config` object which allows modifications.

Besides `direct()` `\Zend\Controller\Action\Helper\AbstractHelper` has other methods at its disposal for your usage. Some are getters and setters, others are hooks which plug into a specific time in your Controller life cycle. The methods are:

`setActionController()` is used to set the current action controller. Which will give you access to the controller from within the helper itself.

`init()` is a hook into the action controller initialization which can be used to initialize the helper or to set a specific state (reset) when the helper is used in multiple controllers.

`preDispatch()`, is a hook that is triggered prior to a dispatched action.

You could use this to set a specific state or a check whether the action should be run. The latter is the way it is used in the cache helper. Where it checks if cache exists and if so, it returns the cache and doesn't run the action.

`postDispatch()` is a hook which is triggered when a dispatched action is done. In the example above I use this to check if there is no other action to be dispatched. If not I request the current meta data, which might have been altered in the last action, and set it in the view.

`getRequest()` retrieves the current request object.

`getResponse()` retrieves the current response object.

`getName()` retrieves the helper name.

Remember each hook is called by the `helperBroker` even if you did not implement the hook. And hooks run always independent. They have no relationship on each other, only the relationship we give them. This means that `preDispatch` and `postDispatch` have no relationship and each will run, even if the other hook is not implemented. The only relationship they have is that they are both hooks and one is called prior and the other after an action has been dispatched.

My action helper accepts a meta data configuration file which is done when instantiating the helper for registration to the `helperBroker`.

Of course you can see this action helper in action in the example code from this chapter.

There's more...

Registering your own action helpers for easy access

Once you have created your own action helpers you will want to use them easily in your controllers.

```
$this->_helper->yourActionHelper

$this->getHelper('yourActionHelper')
```

For this to work you have to register your personal namespace to the plugin loader set in the helper broker.

```
\Zend\Controller\Action\HelperBroker::getPluginLoader()-
>addPrefixPath(

    '\YourProject\Controller\Action\Helper', 'YourProject/Controller/
Action/Helper/'

);
```

The helper broker already as a `\Zend\Loader\PluginLoader` registered with a default mapping `Zend\Controller\Action\Helper` to `Zend/Controller/Action/Helper/` attached to it. We add our own `prefixPath` to this.

The best place to do this is in a custom designed application resource or in a bootstrap resource method.

Getting registered action helpers from outside a controller action

Sometimes it is very useful to get a registered action helper from somewhere within your application. For instance in a front controller plugin where you could grab the view from the viewRenderer or set module based viewRenderer options.

You can get any registered action helper from everywhere in your application by using

```
\Zend\Controller\Action\HelperBroker::getStaticHelper($name);
```

Where `$name` can be any action helper name. For instance `viewRenderer, redirector`, `url`, etc.

If no action helper is already registered in the action stack with that name `getStaticHelper` will try to instantiate the helper for you. Finally if it cannot instantiate the helper it will throw an `\Zend\Controller\Action\Exception`.

Then you also have the `getExistingHelper` method

```
\Zend\Controller\Action\HelperBroker::getExistingHelper($name);
```

This will throw an `\Zend\Controller\Action\Exception` immediately if the helper is not registered to the stack.

Removing action helpers from the action stack

Sometimes you would want to remove a certain action helper from the stack because it introduces unwanted behavior.

```
if (\Zend\Controller\Action\HelperBroker::hasHelper('viewRenderer'
)) {
    \Zend\Controller\Action\HelperBroker::removeHelper('viewRender
er');
}
```

Here you check if the viewRenderer is registered to the action stack and when it is registered you remove it. This will have the same effect as when you would set the frontController application resource with the `noViewRenderer` param set to `true` in your `application.ini`

```
resources.frontController.params.noViewRenderer = 1
```

But doing it at run time and not in the configuration itself enables you to remove it based on a module or any other event.

See also

- Recipe: Using application hooks with \Zend\Controller\Plugin\Abstract Plugin

Using application hooks with \Zend\Controller\Plugin\Abstract Plugin

Front Controller plugins are used to listen for certain events in the front controller. With plugins you can perform specific actions on a specific event. Just as action helpers can be used in the dispatch process of controller actions, front controller plugins can be used to perform some logic when a certain event happens in your front controller process lifetime.

How to do it...

There are several use cases for you to use front controller plugins. For instance when using a module based application one of the most common use cases is to set a `layoutPath` and `errorHandlerModule` based on the module being dispatched to. I am going to show you how to change the layout loaded based per module.

`/YourProject/library/YourProject/Controller/Plugin/Layout.php`

```
<?php
namespace YourProject\Controller\Plugin;
class Layout extends \Zend\Controller\Plugin\AbstractPlugin
{
    public function routeShutDown(\Zend\Controller\Request\
AbstractRequest $request)
    {
        $layout = \Zend\Controller\Action\HelperBroker::getStaticHelpe
r('Layout')->getLayoutInstance();
        $layout->setLayoutPath(APPLICATION_PATH.'/modules/'.$request-
>getModuleName().'/views/layouts');
        $layout->setLayout('layout');
    }
}
```

And register this plugin to your front controller in the bootstrap.

```
protected function _initPlugins()
{
    $this->getPluginResource('frontcontroller')
         ->getFrontController()
         ->registerPlugin(new YourProject\Controller\Plugin\Layout());
}
```

or `application.ini`

```
resources.frontController.plugins[] = "\YourProject\Controller\Plugin\
Layout"
```

How it works...

Each plugin must implement `\Zend\Controller\Plugin\AbstractPlugin`. Which defines an interface for the following events.

`routeStartup()`: prior to routing the request, here you can still add to the router or manipulate the request prior to the routing.

`routeShutdown()`: after routing the request. From this point the request object is set with the matching module, controller, action.

`dispatchLoopStartup()`: prior to entering the dispatch loop

There is actually no other logic between this event and `routeShutdown()` but both are separated for semantic reasons, keeping your logic understable.

`preDispatch()`: prior to dispatching an individual action. Could be used for instance for handling `ACL`.

`postDispatch()`: after dispatching an individual action. For instance `\Zend\Layout` uses this event for putting its content into the response object.

`dispatchLoopShutdown()`: after completing the dispatch loop. Can be used for stuff that needs to be done at the end of your application run life cycle.

In the plugin I use the `routeShutdown()` event because this is the earliest event when the request object is configured with the dispatching information such as a module, controller and action. Because now I know which module is going to be dispatched I can set the `layoutPath` to reflect the path of that module. I could use `dispatchLoopStartup` but in my opinion that would be semantically incorrect.

There's more...

Requesting and changing the order of the plugins

You should create your own plugins that are not dependent on the loading of other plugins. Yet sometimes it is necessary to manipulate the order in which plugins are run. For instance if several plugins are registered which perform an action on `routeShutDown` you might want to know for which plugin the `routeShutDown` runs first. Once you know the order you can also change it at the registration process.

Setting the order is done at registering the plugin. It is the second parameter.

```
$front->registerPlugin(Plugin\AbstractPlugin $plugin, $stackIndex =
null)
```

or by using the front controller application resource.

```
resources.frontController.plugins.navigation.class = "\YourProject\
Controller\Plugin\Navigation"

resources.frontController.plugins.navigation.stackindex = -99
```

When you do not give a `stackIndex` to the `pluginBroker` it will use the next numeric index key starting at 0. By default Zend Framework's front controller registers a plugin `\Zend\Controller\Plugin\ErrorHandler` at stack index 100 at the beginning of its dispatch.

And when you use `\Zend\Layout\Layout` in MVC mode (which is the only mode described in this book) the layout also registers it's own plugin `\Zend\Layout\Controller\Plugin\Layout` at stack index 99. As you can see it is just prior to the `ErrorHandler`, in case something goes wrong in its `postDispatch` event.

Because front controller plugins can be registered at any time to the front controller it is hard to manually increment the stack index, it is better that you do not pass it. If however you want a certain plugin to be loaded real early or very late it is ok to use the stack index.

Because front controller plugins can be registered and removed at any time, retrieving the plugin order is rather difficult. You should know what happens in your application. A possible solution is to invoke the boostrap from somewhere in your application and get the front controller resource. From here you can get access to the front controller and do a dump of the plugins registered at that time.

`/YourProject/application/Application.php`

```
    $application->bootstrap()
                ->run();
    $front = $application->getBootstrap()
    ->getPluginResource('frontcontroller')
     ->getFrontController();
    \Zend\Debug::dump($front->getPlugins(), 'registered plugins');
```

See also

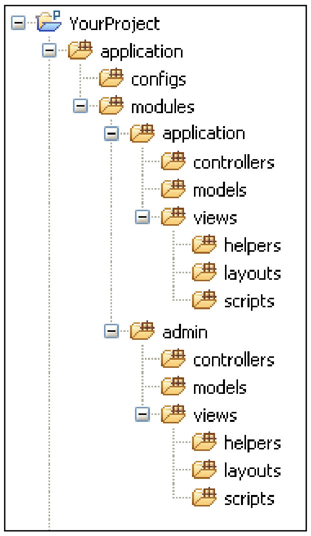
- Recipe: Using modules for a cleaner application structure

Using modules for a cleaner application structure

Modules are a great way to organize your code into logical blocks. It avoids that your bootstrap, controller and view directories are clutted with code from different sections from your website. The most common use case for this is for instance the two module setup: your application module and an admin module. Both probably probably need different controllers, views and maybe even a module specific model.

How to do it...

You can create an extra module by creating an extra folder with the same structure next to the application module.



Next the only thing you would have to do is add 2 lines to your `application.ini` configuration file.

```
resources.frontController.moduleDirectory = APPLICATION_PATH "/
modules"

resources.modules[] =
```

Now your module is available from everywhere in your application. If you use the default route from the framework itself it is just a matter of prefixing the URI path with the module name. For instance `/admin/index/dashboard`

How it works...

```
resources.frontController.moduleDirectory = APPLICATION_PATH "/
modules"
```

Zend framework will automatically find all modules listed under the folder `/YourProject/application/modules` and for each module add the controller directory to the frontController. Once the controller directories are added you have access to the modules from within your application. But modules can have their own bootstraps and namespace. We have to run the bootstraps and register the namespaces, so now that for each module the controller directory is added to the frontController we need to use the modules application resource. `resources.modules[]` does not accept any options so we pass an empty array (`resources.modules.foo = bar` would do exactly the same). This application resource will load for each module the bootstrap (if available) and register the namespace.

There's more...

Module bootstraps

A module can have an optional bootstrap. As soon as you use `resources.modules[] =` each bootstrap will be loaded for each registered `controllerDirectory`. And I really mean each. If you are in the module `admin`, the main bootstrap (`YourProject/application/Bootstrap.php`), the bootstrap from the application module (`YourProject/application/modules/application/Bootstrap.php`), the admin module (`YourProject/application/modules/admin/Bootstrap.php`) and other potential available bootstraps (`YourProject/application/modules/othermodule/Bootstrap.php`) will also be loaded.

A module bootstrap extends `\Zend\Application\Module\Bootstrap` instead of `\Zend\Application\Bootstrap`, has its own namespace (the module name) and gives you access to the same functionality as a `\Zend\Application\Bootstrap` bootstrap.

```
<?php
namespace Application;
class Bootstrap extends \Zend\Application\Module\Bootstrap
{
    protected function _initSomeInitFunction()
    {
        // do some logic here
    }
}
```

If you want module specific behavior you have to load it in a frontController plugin. Only once inside the dispatch loop and after the routing has shutdown you have access to the configured request object and you can get the module that is loaded: `$request->getModuleName()`.

Understanding the modules application resource

As you have seen, `module.resource[] =` takes no parameters and loads all the bootstraps of all available modules and registers their namespace. The modules are made available to the resource with the `resources.frontController.moduleDirectory` directive. It adds to the front controller all the controller directories for each module found in that module directory. Sometimes however you do not want each and every module to be loaded, even not the controller paths registered to the frontcontroller. In this case it is not a wise decision to use the convenience setting `moduleDirectory`, but instead you should manually add a controller directory for each module that you want to be loaded.

```
resources.frontController.controllerDirectory.application =
APPLICATION_PATH "/modules/application/controllers"

resources.frontController.controllerDirectory.admin = APPLICATION_
PATH "/modules/admin/controllers"
```

It builds an associative array of controller directories, where the key is the module name and the value the path. The front controller should have at least one controller directory registered to itself (preferably the application module).

In some projects I see `application.ini` settings such as

```
resources.modules[] = application

resources.modules[] = admin
```

which is absolute nonsense and clearly shows a misunderstanding of the module resource. If you want to load only some modules you should do as I suggest.

See also

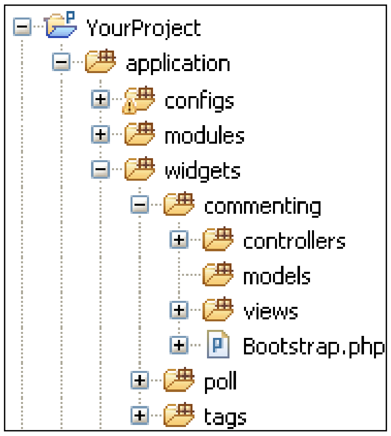
- Recipe: Using modules as widgets

Using modules as widgets

Modules are a great way to share common code between projects. Because a module can contain a complete MVC stack it is the ideal candidate for plug and play widgets such as a poll, commenting or tag cloud widget. Creating widgets per module level is really easy. Because you can drop it into an application and an optional extra bootstrap is loaded which can for instance load routes for your controllers, register additional front controller plugins and such. It also avoids you copy pasting a lot of controllers and views because those are also contained in the module package. Overall my preferred way of creating and reusing widgets.

How to do it...

Create an extra folder somewhere in your application code. For plug and play widgets I propose not to use the `/YourProject/application/modules` directory as your parent container but rather something like `/YourProject/application/widgets` or a path in your library. `/YourProject/application/modules` is used to use modules as a cleaner application structure system. This means keeping different sections such as an admin section in your application separated from the main (application) section. Of course this is a personal convention and Zend Framework doesn't force this kind of decisions.



Next the only thing you would have to do is add some lines to your `application.ini` configuration file.

```
resources.frontController.controllerDirectory.poll = APPLICATION_PATH
"/widgets/poll/controllers"

resources.frontController.controllerDirectory.poll = APPLICATION_PATH
"/widgets/tags/controllers"

resources.modules[] =
```

Now your module is available from everywhere in your application. You can inject a module controller action into your code as a widget by using the action view helper. I have added the poll and the tag cloud widget to be available in your system, but did not make available or load the commenting widget. You could create a front end gui (like WordPress) for configuring which widgets to load using the configuration writer `\Zend\Config\Writer`.

How it works...

```
resources.frontController.moduleDirectory = APPLICATION_PATH "/
modules"
```

Is used when you want to include all the *module sections* in your application. If including *module widgets* I suggest not to load a module directory but to add each and every widget individually using

```
resources.frontController.controllerDirectory.poll = APPLICATION_PATH
"/widgets/poll/controllers"
```

See also

- Recipe: Using modules for a cleaner application structure

Creating application resources for clean configuration

`\Zend\Application` uses a resource system for setting up your system. It provides bootstrap resources and plugin resources. You already have seen both but in this recipe I will discuss both in more detail, and also show some use cases.

How to do it...

A Bootstrap resource is nothing more than a protected function `_init*()` inside your derived `\Zend\Application\Bootstrap` bootstrap.

The * wildcard stands for any descriptive string of your liking. It is good practice to name your bootstrap resources descriptively.

```
protected function _initRegisterFrontControllerPlugins()
```

gives you more of a clue what happens inside the method than

```
protected function _initThirdBootstrapResourceToBeLoaded()
```

The latter name takes us to the subject of the loading sequence. All `_init` methods are run in a top to bottom order which is defined by the order in which they appear in your bootstrap. Sometimes however you want a bootstrap resource to be dependent on a resource plugin. You can force the loading of a resource prior to the implementation of the current resource by adding the bootstrap() method.

```
protected function _initMetaDataActionHelper()
{
    $metadata = $this->bootstrap('metadata')
```

```
                        ->getResource('metadata');
        //rest of the logic
    }
```

In this example we are forcing the custom application plugin resource metadata to be loaded prior to beginning the actual implementation of the bootstrap resource. In the example I also ask the return value of the resource. This is the return value of the plugin `init()` method (`\YourProject\Application\Resource\Metadata::init()`). If you do not care about the return value but rather want the instantiated resource `\YourProject\Application\Resource\Metadata` itself returned, you can use `$this->getPluginResource()`. You get back an instantiated resource but the instance has not been initialized (the `init()` hasn't run yet), so you still have to run the `bootstrap()` method which will proxy to the application plugin resource's `init()`.

You can also get back the return values of previously run bootstrap resources from within a bootstrap resource. You simply use the method `getResource($name)`, which is the same method that you would use to get the `init()` return value from a plugin resource.

```
    protected function _initResponseInFrontController()
    {
        $response = new \Zend\Controller\Response\Http();

        $this->getPluginResource('frontcontroller')
             ->getFrontController()
             ->setResponse($response);
        return $response;
    }

    protected function _initResponseHeaders()
    {
        $response = $this->getResource('ResponseInFrontController');
        $response->setHeader('language', 'en_US')
                 ->setHeader('content-language', 'en_US')
                 ->setHeader('Content-Type', 'text/html; charset=utf-8');
    }
```

How it works...

During a `\Zend\Application\Application` initialization the following happens:

1. It will load the configuration file passed to the constructor
2. use the bootstrap path set in the options to load your main bootstrap.
3. Use a get_class_methods to build a list with all available methods in that bootstrap

4. check for each config option if there is a set method available and run it
5. if no set is available it will try to register the option as a plugin resources (instantiate and register it, not run or initialize it)

Then when you run `$application->bootstrap()`

1. It will proxy to your bootstrap bootstrap() method
2. Execute (run) all bootstrap resources
3. Execute (run) all plugin resources (calls the init() method on the resource)

On each execution it will check if the resource has been executed already, if not it gets executed. If during execution you call your bootstrap `bootstrap()` method with a resource name as parameter it will try to load that specific resource.

There's more...

Other bootstrap configuration tools, the magic set() method

You have seen plugin and bootstrap resources. At first sight it seems that you can only have the power of plugin resources available to you if you want to have configuration options available in your boostrap code. Yet there are two other ways of injecting the configuration options from your `application.ini` into your bootstrap.

```
public function _initAvailablelanguages()

   {
       \Zend\Debug::dump($this->getOptions(), 'settings from
   application.ini');
   }
```

The most used is the bootstrap `getOptions()` method where you get all the settings from your `application.ini` in one `array` the other way is using a magic method set.

Indeed for really small one time usage code blocks very specific to your project you can use the set* methods in your bootstrap.

```
/YourProject/application/configs/application.ini
availablelanguages[] = nl
availablelanguages[] = en
```

```
/YourProject/application/Bootstrap.php
public function setavailablelanguages($options)
{
    \Zend\Debug::dump($options, 'available languages');
}
```

`\Zend\Debug` will dump an array containing nl and en. This means the method is run automatically and you have access to the options related to that method, because those where injected as a parameter.

Be aware however that these magic set methods are always lowercased and that you do not have access to bootstrap and plugin resources as they are initialized later in the process (see the how it works section).

Creating a plugin resource

Plugin resources are real classes which are used and reused over several projects. If you have common bootstrap resources between projects it is probably a wise decision to put that code in a plugin resource. When it is a plugin you can easily plug it into multiple applications. In several of my projects I have a metadata plugin that makes my life much easier regarding the head meta data like the `<title>` tag and `<meta>` tags. I can set it up in my configuration file and forget about it. The resource plugin loads a configuration file and register a custom made controller action helper. Because I do not want to copy, paste and configure that initialization method each time in my bootstraps this plugin is created.

Now I only have to think about configuration.

```
pluginPaths.YourProject\Application\Resource = PROJECT_PATH "/
library/YourProject/Application/Resource"
```

```
resources.metadata.configfile = APPLICATION_PATH "/configs/metaData.
xml"
```

You can see this resource stripped to the basics in the example code from this chapter.

See also

Speed up your plugins

Plugin loading is a fairly expensive task. The loading of plugins is done with the use of a plugin loader which maps the plugin prefix to an actual path in your filesystem or include path. Each prefix can have multiple paths attached to it and these paths on your filesystem are checked for existence and readability when trying to load a plugin. As multiple components use the pluginloader it can become a lot of stat calls on the file system.

To greatly improve the performance of your servers you can use a build-in caching system. When enabled, `\Zend\Loader\PluginLoader` will create a file that contains all successful includes which you can then call early from within your bootstrap.

How to do it...

`/YourProject/application/configs/application.ini`

```
pluginloadercache.cacheFile = PROJECT_PATH "/data/pluginLoaderCache.
php"
pluginloadercache.enabled = 1
```

`/YourProject/application/Bootstrap.php`

```
public function setpluginloadercache($options)
{
    if ($options['enabled']) {
        if (file_exists($options['cacheFile'])) {
            include_once $options['cacheFile'];
        }
        \Zend\Loader\PluginLoader::setIncludeFileCache($options['cach
eFile']);
    }
}
```

How it works...

The magic happens here `\Zend\Loader\PluginLoader::setIncludeFileCache($options['cacheFile']);`

It tells the `pluginLoader` to write the path for each successful include into a PHP cache file. Opening this cache file you will see it contains regular php `include_once` statements. These statements should be run as early as possible in your application, that is why I have opted to use the combination of `pluginloadercache` configuration options in the `application.ini` and a `set` method in the bootstrap `setpluginloadercache()`, this way ensuring that it is called prior to any bootstrap or resource plugin. Using the `application.ini` also gives me the flexibility to override the enabled option per section. For instance not to enable cache on development.

Summary

This chapter must give you a deeper insight in how `\Zend\Application\Application` works and how you can reap its benefits by using bootstrap resources, plugin resources and configuration setters. It also shows you that you can switch between different configuration formats and you do not have to code everything because a lot of functionality is made available through the use of configuration directives.

It also shares my ideas on the usage of modules and that it could potentially generate a widget library online, such as you have seen for Wordpress.

Overall I consider this chapter a very important one, and I would really want to press you to really grasp everything that I have showed you. Study how to use cache, front controller plugins and action helpers, all these tools are available and a great asset in your toolbox.

Of course you can see the sample code from this chapter and remember nothing beats practice. So start taking a look at the example code.

4
Database handling and abstraction

In this chapter, we will cover:

- Connecting to your database with a database adapter
- \Zend\Db\Table
- \Zend\Db\Select
- Optimize with \Zend\Db\Profiler\Firebug
- Building a DAO
- Building Entities and Collections

Introduction

Ninety percent of PHP applications connect to a database. This means your application or your future application will most likely need a database connection and a way to do some CRUD (Create, Read, Update and Delete) operations. This chapter will first introduce the Zend Framework's way to connect to a database adapter. After that it will build up slowly and show you the abstractions the framework will offer you, on which we will build our own abstractions making it easier to switch between persistence solutions. Data can be stored in different systems for which a database is one, the filesystem or web services like SOAP or REST are others. It is a very exciting chapter so let's get on with it.

Connecting to your database with a database adapter

Getting ready

Get yourself comfortable with the vendor's database driver Mysqli, Pgsql, SqlServer, Oracle and so on. Because we are going to connect to them to do some database operations. If you know which database you want to use, but you have no experience with it from a PHP point of view, I suggest you take a look at the specific database extension chapter in the PHP manual at `php.net`. The difference between PDO (PHP Data Objects) and the native drivers is that PDO is PHP's abstraction answer to accessing data. It tries to give you a common frontend API between different databases, thus abstracting the underlying used driver. In theory this makes switching between databases easier, because you are using the same functions to issue queries and fetch data.

How to do it...

Use the database resource with the adapter of your choice:

`/YourProject/application/configs/application.ini`

```
resources.db.adapter = "PdoMysql"
resources.db.params.host = "localhost"
resources.db.params.username = "nick"
resources.db.params.password = "password"
resources.db.params.dbname = "test"
resources.db.isDefaultTableAdapter = true
```

Then it is a matter of getting the resource `init()` result and you are good to go.

Getting the adapter from your bootstrap:

```
protected function _initPlugins()
{
    $db = $this->bootstrap('db')->getResource('db');
    $this->getPluginResource('frontcontroller')
         ->getFrontController()
         ->registerPlugin(new \YourProject\Controller\Plugin\SomePlugi
nThatNeedsTheDbAdapter($db));
}
```

Getting the adapter from within a controller.

```
public function indexAction()
{
    $db = $this->getInvokeArg('bootstrap')
               ->getResource('db');
    $stmt = $db->query('select 1');
}
```

How it works...

First we use the db application resource plugin, which will instantiate the required adapter for us and then you can access that adapter from within your code. First I demonstrate how to get it from the bootstrap to, for instance, set it in a front controller plugin that needs database access. And secondly I demonstrate how to get it through the `getInvokeArg()` method from within a controller. Then I perform a query.

Different available adapters:

- IBM DB2 ⟶ resources.db.adapter = "Db2"
- IBM DB2 PDO ⟶ resources.db.adapter = "Pdo\Ibm\Db2"
- Informix Dynamic Server ⟶ resources.db.adapter = "Pdo\Ibm\Ids"
- MySQLi ⟶ resources.db.adapter = "Mysqli"
- MySQL PDO ⟶ resources.db.adapter = "PdoMysql"
- Oracle ⟶ resources.db.adapter = "Oracle"
- Oracle PDO ⟶ resources.db.adapter = "Pdo\Oci"
- Microsoft Sql Server ⟶ resources.db.adapter = "Sqlsrv"
- Microsoft Sql Server PDO ⟶ resources.db.adapter = "Pdo\Mssql"
- PostgreSQL PDO ⟶ resources.db.adapter = "Pdo\Pgsql"
- SQLite PDO ⟶ resources.db.adapter = "Pdo\Sqlite"

There's more...

Instantiating the adapter without Db application resource

If you do not want to use the database application plugin resource you can manually create an instance.

```
$db = Zend_Db::factory('PdoMysql', array(
    'host'     => 'localhost',
    'username' => 'nick',
    'password' => 'password',
    'dbname'   => 'test'
));
```

Here you can substitute the adapter name `PdoMysql` to any of the strings defined in the above recipe: `Pdo\Pgsql`, `Oracle`, ...

Creating an adapter instance manually or through the plugin resource does not make a connection to the database. Lazy loading is implemented so the adapter instance saves the connection params and only when you make a query or use the connection object directly do you connect.

Querying the database using the connection object directly

I want to start by retrieving and using the connection object directly. I follow this order in the book so that you will have a better understanding of what really happens and that you understand that you can always fall back to this object when `\Zend\Db\Adapter\AbstractAdapter` cannot provide what you need.

Using the connection object might be useful in the following cases:

- When the adapter cannot provide you with some extension features.
- When some SQL statements should not be prepared such as CREATE, ALTER. Zend\Db always prepares and executes the statements.

Retrieving the connection object is done with the `getConnection()` method from the adapter.

```
$databaseExtension = $db->getConnection();
```

`$databaseExtension` will be a vendor specific database extension connection object.

Take a look at the php manual to see what kind of methods are available for each type:

`http://www.php.net/manual/en/refs.database.php`

Sometimes you have the need to execute a string which consists out of multiple queries.

For instance from a patch file:

```
    CREATE SEQUENCE patch_id_seq START 1;

    CREATE TABLE patch(
       id INT NOT NULL DEFAULT nextval('patch_id_seq') PRIMARY KEY,
       patch CHAR(13) NOT NULL,
       status VARCHAR(255) NOT NULL,
       execution_date TIMESTAMP NOT NULL DEFAULT now()
    );
```

You would not be able to do this from the adapter itself, because `$db->query()` builds a prepared statement and that doesn't allow multiple queries. You would need to do it from the connection object. The method to execute SQL statements without preparing them in PDO is `exec()`.

```
try {
    $db->getConnection()->beginTransaction();
    $db->getConnection()->exec($sql);
    $db->getConnection()->commit();
} catch (\Exception $e) {
    $db->getConnection()->rollBack();
    throw $e;
}
```

If you do not use `PDO` you have to check your vendor specific database extension manual for the correct methods. If `getConnection()` would return a `Mysqli` object the code above would differ greatly. As you can see depending on the interface returned by `getConnection()` you would need to call different methods and implement different logic. So using the connection object directly makes you lose the abstraction and binds the code more to a specific vendor. Sometimes however there is no other way. You should use Zend Framework's abstraction as much as you possibly can by using the adapter instead of the connection object directly.

Querying the database using Zend Framework's abstraction

As PDO provides you with a PHP supported data-access abstraction, so does Zend Framework provide adapters with a common interface for all popular database brands, even the ones who do not provide a PDO driver. Like with PDO drivers, the ZF adapters all have their own common interface `\Zend\Db\Adapter\AbstractAdapter` defined which makes your life easier if you ever want to switch between database drivers. Even between PDO and non PDO drivers.

The most commonly used interface methods are the following:

```
beginTransaction()

insert($table, array $bind)

update($table, array $bind, $where = '')

delete($table, $where = '')

commit()

rollBack()

query($sql, $bind = array())

fetchAll($sql, $bind = array(), $fetchMode = null)
```

```
fetchRow($sql, $bind = array(), $fetchMode = null)
fetchOne($sql, $bind = array())
lastInsertId($tableName = null, $primaryKey = null);
```

There are more methods in the interface, some are discussed in the `\Zend\Db\Select` recipe, some aren't so I suggest you inspect the interface code to see what other options there are. But for now you probably will have enough with the above functionality. Let us begin with database transactions, these are logical units of work for which all of them need to be successful executed or none of them. You define the beginning of a transaction with `beginTransaction()` and then you perform database operations you want executed and succeeded as a batch after which you `commit()` (confirm) them all. If for some reason a certain `query()` doesn't succeed and throws an exception the `rollback()` (undo) will be executed for all executed units of work in that transaction.

Next I want to focus on the `insert` (create), `update` and `delete` from the CRUD operations. All three need a database table (`string`) to work. Some have an optional where param which is a classical SQL string such as '`id = 54`', and if created dynamically still allows SQL-Injection. Others force you to specify a `bind` parameter which is the data you want to set in the table. It is an array of key, value pairs, where the keys are the table column names and for which the value will be quoted automatically for insertion into the table to prevent SQL injection. These three operations perform in the back a `query()` operation. The `query()` method is a general way to prepare and execute an SQL statement of a concrete database implementation of `\Zend\Db\Statement\AbstractStatement`. It will return an executed statement which you can re-execute optionally with different bind params or fetch results from it.

```
$stmt = $db->query('SELECT name FROM customer WHERE id = ?',
array(54));
$name = $stmt->fetchColumn(1);
$stmt->execute(array(16));
$name = $stmt->fetchColumn(1);
```

Initially in the above code `$stmt` will be a prepared and executed statement for which you fetch the name value for `id=54`. Then you re-execute the statement with a different id value of 16 and fetch that associated name. This way you only have to write your SQL once and have it executed with different values which are quoted automatically. That is the power of prepared statements. If you do not want to execute the same statement over and over Zend Framework facilitates preparing, executing and fetching results for one statement with the other options such as `fetchAll()` from the list above.

```
$name = $db->fetchOne('SELECT name FROM customer WHERE id = ?',
array(54));
```

Will return the value of the first column from the first row. `fetchAll()` will return all rows and `fetchRow()` will return the next row. Take caution that only the bind params are quoted, that means that SQL-injection is still possible if the `$sql` part is dynamically created.

If an insert operation has been executed you should retrieve the last inserted primary key value inserted into the database with `lastInsertId()`. You only need to specify parameters if your RDBMS works with sequences but in that case I suggest you use `lastSequenceId($ sequenceName)`.

\Zend\Db\Table\Table

An object oriented way to database tables is `\Zend\Db\Table\Table`. Performing table operations for retrieving information is made easy with this class. All information returned is in an object oriented way as it is an implementation of the Table Data Gateway pattern, which as it's name suggest is an access point (gateway) to the data from a certain table. Actions on row level for the table use the Row Data Gateway pattern which provides you an object-oriented way to manipulate rows.

Getting ready

For each table in your database for which you want to use a table class you must specify a primary key. `\Zend\Db\Table\Table` needs a way to identify a unique row.

How to do it...

```
$commentTable = new \Zend\Db\Table\Table('comment');
$comment = $commentTable->createRow();
$comment->text = 'another comment inserted in the database';
$comment->article_id = 4;
$id = $comment->save();

$comment = $commentTable->find(1)->getRow(0);
$comment->text = 're-edited comment '.rand(0,13);
$comment->save();
```

How it works...

In the above example code you instantiate a `\Zend\Db\Table\Table` class which will provide you with an object-oriented way to create rows and save those same rows completely automatic. In the first block you create such a `\Zend\Db\Table\Row` which will have the table column names set as object properties. You can then alter/set them and store the new row in the database with the `save()` method. From then of on the row is available for retrieval.

In the second block you search for a row in the comment table with a primary key set to 1. This will return you a row set (`\Zend\Db\Table\Rowset`) because you can also give an array with multiple PK values. In this example you know you only ask one row, and you also know at least one row is available because you previously inserted one. From the returned row set you get the first row (position 0). As you can see you do not have to specify the column name of the primary key, because on instantiation of the table object, the table metadata is extracted behind the scene with the adapter's public `describeTable()` method and thus the table object is primary key aware. Again you use the `save()` method, but this time to update the row. Internally `\Zend\Db\Table\Row` knows whether it is an `insert` or an `update` that has to be performed.

Of course instantiating a new `\Zend\Db\Table\Table` class on the fly for each table is not that good practice and should be avoided. One of those reasons is the reading of the table metadata information on each `insert()`, `find()` and `info()` operation. Other reasons might be specific functionality you want to add to the concrete table class or configuration options.

There's more...

Caching the metadata information

You need a cache object (see previous chapter) and pass that to each instance constructor or set it as the default cache for all table instances.

Setting it per instance:

```
$commentTable = new \Zend\Db\Table\Table(
    array(
        'name' => 'comment',
        'metadataCache' => $cache
    )
);

$commentTable = new \YourProject\Db\Table\Comment(
    array(
        'metadataCache' => $cache
    )
);
```

Or setting it for all table classes:

```
\Zend\Db\Table\AbstractTable::setDefaultMetadataCache($cache)
```

Another option is to hardcode the metadata. But this means you will have to write a script that updates this information for you each time you change your table structure or do it manually. This only works when you instantiate a concrete custom implementation of `\Zend\Db\Table\AbstractTable`.

```
<?php
namespace YourProject\Db\Table;
class Comment extends \Zend\Db\Table\AbstractTable
{
    protected $_name = 'comment';

    protected $_metadata = array (
        'id' => array (
          'SCHEMA_NAME' => NULL,
          'TABLE_NAME' => 'comment',
          'COLUMN_NAME' => 'id',
          'COLUMN_POSITION' => 1,
          'DATA_TYPE' => 'int',
          'DEFAULT' => NULL,
          'NULLABLE' => false,
          'LENGTH' => NULL,
          'SCALE' => NULL,
          'PRECISION' => NULL,
          'UNSIGNED' => NULL,
          'PRIMARY' => true,
          'PRIMARY_POSITION' => 1,
          'IDENTITY' => true,
        ),
        'text' => array (
          'SCHEMA_NAME' => NULL,
          'TABLE_NAME' => 'comment',
          'COLUMN_NAME' => 'text',
          'COLUMN_POSITION' => 2,
          'DATA_TYPE' => 'text',
          'DEFAULT' => NULL,
          'NULLABLE' => true,
          'LENGTH' => NULL,
          'SCALE' => NULL,
          'PRECISION' => NULL,
          'UNSIGNED' => NULL,
          'PRIMARY' => false,
          'PRIMARY_POSITION' => NULL,
          'IDENTITY' => false,
        ),
        'article_id' => array (
          'SCHEMA_NAME' => NULL,
          'TABLE_NAME' => 'comment',
          'COLUMN_NAME' => 'article_id',
          'COLUMN_POSITION' => 3,
          'DATA_TYPE' => 'int',
          'DEFAULT' => NULL,
          'NULLABLE' => false,
          'LENGTH' => NULL,
          'SCALE' => NULL,
          'PRECISION' => NULL,
          'UNSIGNED' => NULL,
          'PRIMARY' => false,
          'PRIMARY_POSITION' => NULL,
          'IDENTITY' => false,
        ),
      );
}
```

Creating this yourself would give you headaches so the following trick will help you in creating the array.

```
$commentTable = new \YourProject\Db\Table\Comment();

var_export($commentTable->getAdapter()->describeTable('comment'));
```

This will dump the array in copy and paste code to your screen. You basically instantiate the table from which you get the database adapter and do a `describeTable` operation with the table name as a parameter. Of course you do not need to instantiate the table itself if you have the correct adapter readily available.

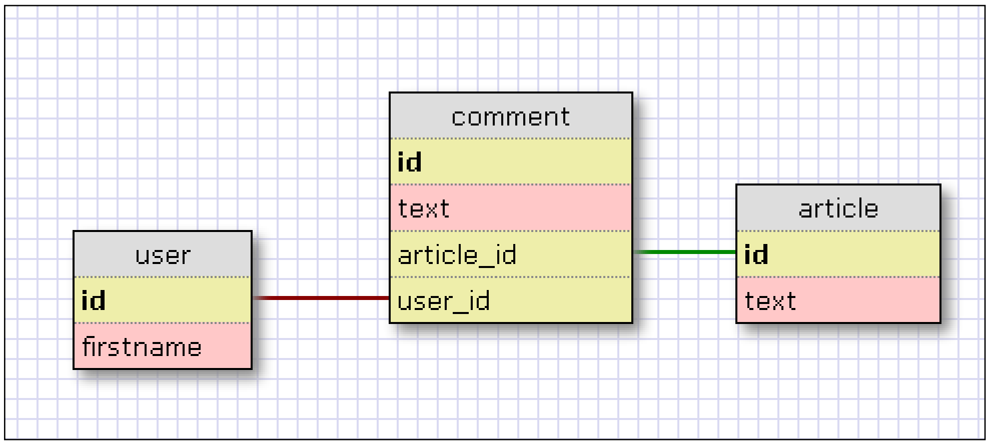
Extending \Zend\Db\Table\AbstractTable

As you saw in "caching the metadata information" extending the `AbstractTable` is very useful. The concrete implementation should have as a minimum the property `$_name` to identify the database table for which you want an object-oriented interface. Your primary key or unique key should be identified with the `$_primary` property. This is not necessary but gives you more flexibility if you want to specify multiple column compound keys.

One of the best practices in Zend Framework is the usage of the `init()` template methods to avoid overriding of the constructor. Again this is also advocated for table classes.

Defining table relationships

In a RDBMS (Relational database management system) there are relationships between tables. `\Zend\Db\Table\AbstractTable` allows you to declare the referential integrity constrains between tables.



Article	
id	**text**
1	this is a post about tables

User	
id	**firstname**
1	Nick

Comment			
id	**text**	**article_id**	**user_id**
1	I like this post	1	1
2	I hate this post	1	1

```
/YourProject/library/YourProject/Db/Table/Article.php
<?php
namespace YourProject\Db\Table;
class Article extends \Zend\Db\Table\AbstractTable
{
    protected $_name = 'article';

    protected $_dependentTables = array('Comment');
}
```

```
/YourProject/library/YourProject/Db/Table/User.php
<?php
namespace YourProject\Db\Table;
class User extends \Zend\Db\Table\AbstractTable
{
    protected $_name = 'user';

    protected $_dependentTables = array('Comment');
}

/YourProject/library/YourProject/Db/Table/Comment.php
<?php
namespace YourProject\Db\Table;
class Comment extends \Zend\Db\Table\AbstractTable
{
    protected $_name = 'comment';

    protected $_referenceMap = array(
        'User' => array(
            'columns' => array('user_id'),
            'refTableClass' => '\YourProject\Db\Table\User',
            'refColumns' => array('id'),
            'onUpdate' => self::CASCADE,
            'onDelete' => self::RESTRICT,
        ),
        'Article' => array(
            'columns' => array('article_id'),
            'refTableClass' => '\YourProject\Db\Table\Article',
            'refColumns' => array('id'),
            'onUpdate' => self::CASCADE,
            'onDelete' => self::RESTRICT,
        ),
    );
}
```

Because the comment table has a foreign key towards the article and user table I define the dependency in those classes. In the `Article` class I say that the comment class is dependent on `Article` and I also declare that same dependency from the `User` class. And in the dependent table class `Comment` I define those foreign keys with the `$_referenceMap` property. Each reference map is defined by a rule for which I suggest to be descriptive and respect method naming conventions. A rule exists out of the following information:

columns, refTableClass, refColumns and optionally onDelete and onUpdate.

In SQL code this would translate to

```
CONSTRAINT comment_article_id_fkey FOREIGN KEY (article_id)
REFERENCES article(id) ON UPDATE CASCADE ON DELETE RESTRICT

CONSTRAINT comment_user_id_fkey FOREIGN KEY (user_id) REFERENCES
user(id) ON UPDATE CASCADE ON DELETE RESTRICT
```

After setting the dependencies and integrities you can use the following code:

```
$articleTable = new \YourProject\Db\Table\Article();
$article = $articleTable->fetchRow('id=1');
foreach ($article->findDependentRowset('\YourProject\Db\Table\
Comment') as $comment) {
    $user = $comment->findParentRow('\YourProject\Db\Table\User');
}
```

First you retrieve the article with the id set to 1 and for that article you find all comments through the `findDependentRowset()` method. This method is used because you go from your primary key `article(id)` to the table with the foreign key `comment(article_id)`. During the loop over the entire rowset you retrieve for each comment the user who posted it. For this you use `findParentRow()` because this time you start from the foreign key `comment(user_id)` towards the parent table with the primary key `user(id)`. This block of code would result in the following queries being sent to the database.

```
'SELECT `article`.* FROM `article` WHERE (id=1) LIMIT 1'

'SELECT `comment`.* FROM `comment` WHERE (`article_id` = 1)'

'SELECT `user`.* FROM `user` WHERE (`id` = 1) LIMIT 1'

'SELECT `user`.* FROM `user` WHERE (`id` = 1) LIMIT 1'
```

As you can see everything is really atomic, no joins are being made and the wildcard selector is used. You also could potentially make duplicate calls to the database waisting resources. In the above example the same user has posted 2 comments on the article which results in two identical separate DB calls. The wildcard is needed for the gateway pattern and the ease to update rows but most applications consist of selects and not updates, again waisting a lot of resources because you fetch data you will never need.

The \Zend\Db\Table namespace can be used to build an admin tool, because speed and memory is not that important, but should be avoided in a high traffic site. In a database you often have helper columns strictly for where clauses (populated by triggers) and those will also be retrieved, which is useless and might even corrupt data.

The idea behind forcing the integrity in the classes is because that would enable you to do deletes and updates and force integrity from a application point of view. However all RDBMS support those integrities themselves and you should set them at database level. And this is where everything really gets weird, you can use cascading UPDATE in `\Zend\Db\Table\AbstractTable` only if your database does not enforce that referential integrity constraint. So it forces you to remove the integrity at database level, which would make your database corrupt if you also use another interface (in the database directly or another application) to perform CRUD operations. If however your database doesn't allow integrities then it might be a good (the only) solution.

\Zend\Db\Select

`\Zend\Db\Select` is an object-oriented way to build your SQL SELECT statement in a piece-by-piece manner. Because it is a database independent abstraction you are probably able to reuse (possibly with a minimum on refactoring) the SQL whenever you switch between databases. Almost all db adapter, table, row and rowset methods which accept a SQL string also accept a `\Zend\Db\Select` which makes a little going deeper into this class worthwhile.

How to do it...

Instantiate manually passing a database adapter:

```
$select = new \Zend\Db\Select($db);
```

Instantiate from an adapter:

```
$select = $db->select();
```

from a table:

```
$select = $table->select(true);
```

from a row:

```
$select = $row->select();
```

How it works...

When creating a Select object it needs a database adapter (`$db`) to have access to the following adapter methods:

```
quoteInto($text, $value, $type = null, $count = null)
```

To prevent SQL Injection and or accidental usage of special database characters the values passed to the query need to be escaped / quoted. Especially if those values come from PHP variables.

```
$userName = "nick";
$sql = $db->quoteInto("SELECT * FROM user WHERE firstname = ?",
$userName);
//SELECT * FROM user WHERE firstname = 'nick'

$userName = "nick' OR 1 = 1";
$select = $db->quoteInto("SELECT * FROM user WHERE firstname = ?",
$userName);
//SELECT * FROM user WHERE firstname = 'nick\' OR 1 = 1'

$id = '1';
$select = $db->quoteInto("SELECT * FROM user WHERE id = ?", $id);
//SELECT * FROM user WHERE id = '1'

$id = '1';
$select = $db->quoteInto("SELECT * FROM user WHERE id = ?", $id,
'INTEGER');
//SELECT * FROM user WHERE id = 1

$id = 1;
$select = $db->quoteInto("SELECT * FROM user WHERE id = ?", $id);
//SELECT * FROM user WHERE id = 1

$id = '1';
$select = $db->quoteInto("SELECT * FROM user WHERE id = ? and id = ?",
$id, 'INTEGER', 1);
//SELECT * FROM user WHERE id = 1 and id = ?
```

As you can see the third parameter will let you specify a database column type. All but numeric datatypes are quoted. When you pass a PHP integer no quoting will be applied and the third param is not needed. The fourth param `$count` let you define how many from left to right placeholders are going to be quoted into.

`quoteIdentifier($ident, $auto=false)`

Sometimes you need to build SQL statements with database reserved words such as USER, ORDER, etc. These are called identifiers. The above statements may fail in certain databases because of the reserved word user. That is why you need to quote them so you can still use these delimited identifiers as for instance table and column names.

```
$tableName = $db->quoteIdentifier('user');
$select = $db->quoteInto("SELECT * FROM ".$tableName." WHERE id = ?",
1);
//SELECT * FROM `user` WHERE id = 1
```

`quoteTableAs($ident, $alias = null, $auto = false)`

Internally this method uses the protected method `quoteIdentifierAs()` which uses in its turn `quoteIdentifier()`.

```
$tableName = $db->quoteTableAs('user', 'u');
$select = $db->quoteInto("SELECT * FROM ".$tableName." WHERE id = ?",
1);
SELECT * FROM `user` AS `u` WHERE id = 1
```

`quoteColumnAs($ident, $alias, $auto=false)`

same as the method as above but then specifically for column names

`query($sql, $bind = array())`

Returns an executed prepared SQL statement. This method has been discussed thoroughly in the "Querying the database using Zend Framework's abstraction" recipe

`limit($sql, $count, $offset = 0)`

Will append the SQL string with the database adapter LIMIT OFFSET clause

`getFetchMode()`

Returns the fetch mode set in the adapter. An adapter can have different fetch modes. By default a fetch returns a row as an associative array (`\Zend\Db\AbstractAdapter::FETCH_ASSOC`), where the column names or aliases are the keys. Your adapter can be set to return positional arrays, where the position of the column name will be represented by an integer in the row array (`\Zend\Db\AbstractAdapter::FETCH_NUM`). Or it can be set to return objects with `\Zend\Db\AbstractAdapter::FETCH_OBJ`.

The select's own `query()` method which proxies to the adapter and returns an executed prepared statement uses this mode to set it in the statement.

Now that you have seen why `\Zend\Db\Select` needs an adapter object in the constructor we can go back to the different ways on how to instantiate a select object.

If you instantiate it from the adapter itself, the adapter will instantiate a select object and registers itself to it before returning it.

Instantiating it from a table object will return you a `\Zend\Db\Table\Select` object with the table configured into it. You can pass a `boolean` to automatically set the from part of the SELECT based on the table. By default this is set to `false`.

Instantiating it from a row just proxies to the select method from the table embedded in the row object.

There's more...

Setting the FROM in your $select object

`$select->from($name, $cols = '*', $schema = null)`

`$name` specifies the table name, this can be the table name as a string, or a table name with its correlation as an associative array.

```
echo $select = $db->select()
                  ->from('article');
//SELECT `article`.* FROM `article`
echo $select = $db->select()
                  ->from(array('a' => 'article'));
//SELECT `a`.* FROM `article` AS `a`
```

In the `$cols` parameter you can specify which columns to include in your `SELECT` statement. By default it takes `*` which is of course not the optimal situation, because you fetch a lot of information that you will never use or is even intended for internal database handling. Adding columns is done by passing an associative array. When you pass a key it will be set as the alias. The values can be column names or expressions. Expressions are detected automatically when parentheses are used and transformed to `\Zend\Db\Expr`. If you do not use parentheses it is your responsibility to pass a `\Zend\Db\Expr` yourself, failing to do so will result in a database error.

```
$db->select()
   ->from(
       array('a' => 'article'),
       array(
           'text',
           'total' => 'count(1)',
           'published' => new \Zend\Db\Expr("CASE WHEN 1>0 THEN 'true'
ELSE 'false' END")
       )
   );
// SELECT
// `a`.`text`, count(1) AS `total`, CASE WHEN 1>0 THEN 'true' // ELSE
'false' END AS `published`
// FROM
// `article` AS `a`
```

Alternatively you can also set the columns after the `from()` method was used. This can be useful if you dynamically want to add extra columns.

```
columns($cols = '*', $correlationName = null)
```

```
    $db->select()
        ->from(
            array('a' => 'article'),
            null
        )
        ->columns(
            array(
                'text',
                'total' => 'count(1)'
            )
        )
        ->columns(array('published' => new \Zend\Db\Expr("CASE WHEN 1>0
    THEN 'true' ELSE 'false' END")));
```

This will result in exactly the same SQL statement as the previous select object, this time however you add columns dynamically. One thing to pay attention to is that you also set the initial `$cols` param from the `from()` method to an empty `array` or something that equals `false` when you convert it to a `boolean`. I chose to use a `null` value because for me this is semantically speaking most correct. The reason is that internally `array_filter` is used with no callback set. If you forget to set the second parameter the default will be used which is a SQL column wildcard `*`.

Adding Joins

```
joinInner($name, $cond, $cols = self::SQL_WILDCARD, $schema = null)

joinInnerUsing($name, $cond, $cols = self::SQL_WILDCARD, $schema =
null)

joinLeft($name, $cond, $cols = self::SQL_WILDCARD, $schema = null)

joinLeftUsing($name, $cond, $cols = self::SQL_WILDCARD, $schema =
null)

joinRight($name, $cond, $cols = self::SQL_WILDCARD, $schema = null)

joinRightUsing($name, $cond, $cols = self::SQL_WILDCARD, $schema =
null)

joinFull($name, $cond, $cols = self::SQL_WILDCARD, $schema = null)

joinFullUsing($name, $cond, $cols = self::SQL_WILDCARD, $schema =
null)
```

```
joinCross($name, $cols = self::SQL_WILDCARD, $schema = null)

joinNatural($name, $cols = self::SQL_WILDCARD, $schema = null)
```

I am not going to explain which join to use and when or how to use them. That is SQL and this is a book on Zend Framework, so I will stick to that part. `$name` is the table name and the `$cols` parameter you already know from the previous recipes. So what is the `$cond` parameter? Well the `$cond` parameter tells the Select object how to link the join to a table previously set in the select object.

```
$db->select()
   ->from(
       array('u' => 'user'),
       array('firstname')
   )
   ->joinInner(
       array('c' => 'comment'),
       'c.user_id=u.id',
       array('text')
   );
```

When using `USING` joins you must specify the column name to join and not an expression.

Now that you have attached another table to your query you can also dynamically add extra columns (with correlation) to your query using the `columns()` method described above.

Adding a where clause

What would be the use of a database if you would not be able to fetch a selection out of the millions of rows. This is where the `where` clause kicks in. From a Zend Framework point of view these are strings you will have to build manually and ZF will attach it for you to the select statement using binding if you want. You have two different kind of where clauses. An "and" where clause and an "or" where clause.

```
where($cond, $value = null, $type = null)
orWhere($cond, $value = null, $type = null)
```

`$cond` is the where clause with a possible placeholder (recommended). When you use a placeholder you can set the `$value` param and optionally the database column type as you already saw in the quoteInto part of the main recipe.

```
$db->select()
   ->from(
       'user',
       array('firstname')
   )
   ->where('firstname like ?', 'ni%')
   ->orWhere('id = ?', 1);
```

Binding can also be done using named bindings which is preferable when setting the value during the fetch.

```
$select = $db->select()
   ->from(
       'user',
       array('firstname')
   )
   ->where('firstname like :firstname')
   ->orWhere('id = :id');
$result = $db->fetchAll(
    $select,
    array(
        'firstname' => 'ni%',
        'id' => 1
    )
);
```

You can add as many `where()` and `orWhere()` methods to the select object as needed to build your entire SQL where clause. With SQL the order of the conditions in your where clause greatly impacts the performance of your query. In ZF it is equally important.

Other parts of a SQL Query

A SQL query often consists out of more than some columns, a table name, possible joins and a where clause. It can also contain a group by, having, distinct, order by, limit and so on. The most important have special methods with the following signature:

```
distinct($flag = true)

group($spec)

having($cond)

limit($count = null, $offset = null)

order($spec)
```

Optimize with \Zend\Db\Profiler\Firebug

Performing SQL can have a serious performance impact on your application. To profile your database connection and spot the slow queries a wonderful tool has been implemented in Zend Framework. This is the `\Zend\Db\Profiler\Firebug`.

Getting ready

Make sure you have firefox running with the Firebug and FirePHP add-ons installed. If you haven't already installed these plugins as a PHP developer shame on you. Once both are installed make sure the firebug Net functionality is enabled.

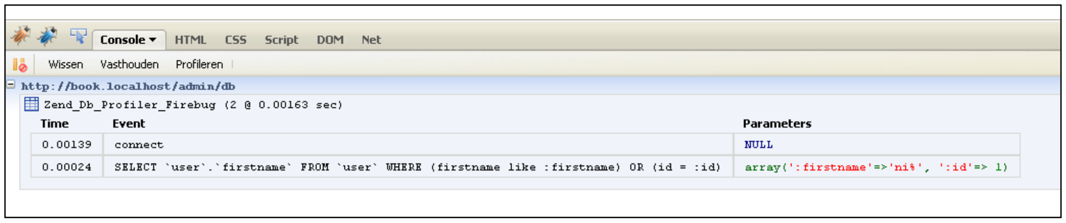
How to do it...

Add the following lines to your `application.ini`

```
resources.db.params.profiler.enabled = "true"

resources.db.params.profiler.class = "\Zend\Db\Profiler\Firebug"
```

run your application and take a look at your firebug console tab.



```
http://book.localhost/admin/db
Zend_Db_Profiler_Firebug (2 @ 0.00163 sec)
```

Time	Event	Parameters
0.00139	connect	NULL
0.00024	SELECT `user`.`firstname` FROM `user` WHERE (firstname like :firstname) OR (id = :id)	array(':firstname'=>'ni%', ':id'=> 1)

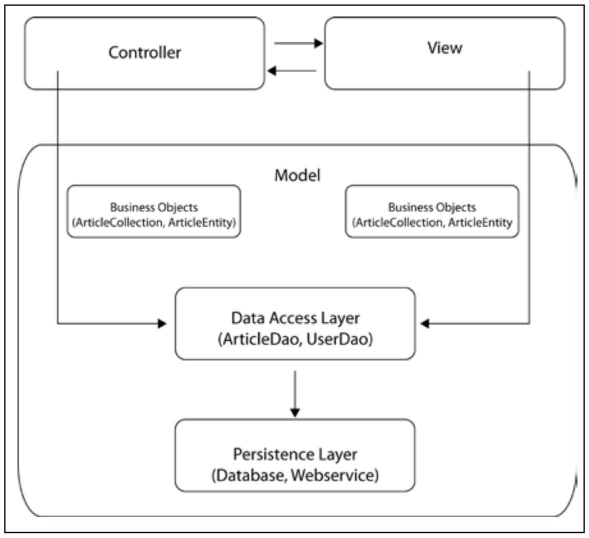
How it works...

As you can see it tells you in an unobtrusive way the queries that have been run, how much time it took and possible parameters passed. You are able to use this profiling information to optimize your queries and also see which expensive SQL calls possibly need caching if optimizing the SQL fails to give you a usable time.

Building a DAO

The sole purpose of a Data Access Object (DAO) is to access data from one central place. The theory is when you encapsulate all the access logic in one object it makes it easier if you ever want to switch a storage system. It also promotes reuse of code such as queries which are reused over several controllers or models.

Zend Framework already supports excellent database abstraction making a switch between databases a breeze, but what if you want to switch to webservices? If you did not setup your application the right way you could end up refactoring almost all your code because you are too dependant on the `\Zend\Db` namespace. That is why it is best to have your own objects (entities and collections), let the DAO retrieve and set the information from and into the persistence layer.


Controller
View
Model
Business Objects (ArticleCollection, ArticleEntity)
Business Objects (ArticleCollection, ArticleEntity
Data Access Layer (ArticleDao, UserDao)
Persistence Layer (Database, Webservice)

How to do it...

From your Presentation layer (view) and Business layer (can be controllers and also part of the Model itself) create a DAO instance using a specific DAO factory and use that object to communicate with your persistence layer.

```
public function someAction()
{
    $articleDao = \YourProject\Dao\Article\Factory::get();
    $paramsForFindById = $articleDao->getParamsContainer('findById');
    $paramsForFindById->id = 1;
    $article = $articleDao->findById($paramsForFindById);
}
```

`/YourProject/Library/YourProject/Dao/Article/Factory.php`

```
<?php
namespace YourProject\Dao\Article;
class Factory
{
    static protected $type = 'mysql';
```

```
        static public function get()
        {
            switch ( strtolower(self::$type) ) {
                case 'mysql':
                    $dao = new \YourProject\Dao\Article\DbMysqlPdo();
                    break;
                case 'soap':
                    $dao = new \YourProject\Dao\Article\Soap();
                    break;
                default:
                    throw new \YourProject\Dao\Exception();
                    break;
            }
            return $dao;
        }
    }
```

/YourProject/Library/YourProject/Dao/Article/DbMysqlPdo.php

```
    <?php
    namespace YourProject\Dao\Article;
    use \Nbe\Dao\AbstractDao;

    class DbMysqlPdo extends AbstractDao implements Article
    {
        protected $tableName = 'article';

        public function findById(Param\FindById $param)
        {
            $rowSet = $this->getAdapter()->fetchAll("Select * FROM
    ".$this->tableName." WHERE id = ?", array($param->id));
            $articles = new \YourProject\Collection\Articles();
            foreach ($rowSet as $row) {
                $article = new \YourProject\Entity\Article();
                $article->id = $row['id'];
                $article->text = $row['text'];
                $articles->append($article);
            }
            return $articles;
        }

        protected function getAdapter()
        {
            return \YourProject\Db\Factory::get('mysql');
        }
    }
```

/YourProject/Library/YourProject/Dao/Article/Article.php

```
<?php
namespace YourProject\Dao\Article;

interface Article
{
    public function findById(Param\FindById $param);
}
```

/YourProject/Library/Nbe/Dao/AbstractDao.php

```
<?php
namespace Nbe\Dao;

abstract class AbstractDao
{
    public function getParamsContainer($methodName)
    {
        $classPath = explode('\\', get_class($this));
        \array_pop($classPath);
        $classPath[] = 'Param';
        $classPath[] = \ucfirst($methodName);
        $className = \implode('\\', $classPath);
        return new $className();
    }
}
```

/YourProject/Library/YourProject/Db/Factory.php

```
<?php
namespace YourProject\Db;
class Factory
{
    static protected $adapters = array();

    protected function __construct() {}

    static public function get($type = 'mysql')
    {
        if (!isset(self::$adapters[$type])) {
            switch ($type) {
                case 'mysql':
                    self::$adapters[$type] = self::
getMysqlPdoAdapter();
                break;
            }
        }
        return self::$adapters[$type];
    }

    static protected function getMysqlPdoAdapter()
    {
        return new \Zend\Db\Adapter\PdoMysql(
            array(
                'host'     => 'localhost',
                'username' => 'nick',
                'password' => 'password',
                'dbname'   => 'test'
            )
        );
    }
}
```

How it works...

You now have a complete decoupling between your Persistence layer and your Business layer, through the use of a Data Access Layer.

From the Business layer you get a concrete Data Access Object through the factory. You do not instantiate a concrete DAO manually but let the factory do this for you. This way you do not have to change the instantiation everywhere in our code if you switch from Mysql to Soap. You only have to make the switch in one place and that is changing the static `$type` variable in the factory. The factory will return us a concrete DAO which knows how to talk to that specific data storage. In this case it knows how to talk to a Mysql Pdo adapter which it retrieves from a DB adapter factory. This factory is also responsible to only instantiate one adapter of the same kind per request. This way this DAO and other DAOs which make use of the Mysql PDO Adapter know they will always work on the same adapter instance, saving resources. It is no use to create a new adapter instance (and thus underlying connection object) per DAO.

Each Article Dao must implement the Article Interface to ensure that polymorphism will work. A uniform interface guarantees when you switch the concrete DAO the business layer is not impacted because it isn't aware where the data comes from, only that it can use that interface to set and retrieve data from the persistence layer.

A little extra I always implement in my DAO is a sort of `\Zend\Config` idea. I hate long parameter lists, especially when those have default values and such. It also limits flexibility in the order you want to set them and when you want to add a parameter to the list it is almost impossible. That is why with each DAO Interface I also implement a params object specifically for that method. In this case you only need to set the `id` in the param object but what happens if later you also want to add a parameter `isPublished` with a default to `true` for `findById()`? You only have to add that property to the param object and all DAOs have immediate access to that flag property which they can choose to implement or not (of course they should).

And finally all results are not the results from the persistence layer but mapped to our business objects such as collections and entities, further decoupling the persistence layer from the business layer. To save book space the mapping itself is in this example done directly in the `findById()` method itself. You should extract it to its own method or even to its own mapper object.

Building Entities and Collections

You cannot have a proper DAO without the need of your own business objects. I chose to implement Entities and Collections. An entity is an object that is not represented by its attributes (set as properties) but by its identity (often identified with an Id) . You could have an user entity for the user John, and one for Daisy, but also an entity for each article in your application. Collections are nothing more than an aggregate of entities in one object. You can have an article collection containing twenty article entities. The big advantage of using collections instead of traditional arrays is that you can implement additional logic in the collection classes and you now have type forcing at your disposal.

How to do it...

```
<?php
namespace Nbe\Entity;
abstract class AbstractEntity
{
    private $protectedProperties;

    public function __set($name, $value)
    {
        $setter = 'set'.ucfirst($name);
        $this->$setter($value);
        return $this;
    }
    public function __get($name)
    {
        $getter = 'get'.ucfirst($name);
        return $this->$getter();
    }

    public function define(array $properties)
    {
        foreach ($properties as $property => $value) {
            if (!empty($value) && $this->propertyExists($property)) {
                $method = 'set'.ucfirst($property);
                $this->$method($value);
            }
        }
    }

    public function __call($method, $args)
    {
        $type = substr($method, 0, 3);
        $property = strtolower($method[3]) . substr($method, 4);

        if ('get' === $type) {
            if (!$this->propertyExists($property)) {
                $this->throwNotExisting($property);
            }
            return $this->$property;
        } elseif ('set' === $type) {
            if (!$this->propertyExists($property)) {
                $this->throwNotExisting($property);
            }
            $this->$property = $args[0];
            return;
        }

        throw new Exception('Invalid method call: ' . get_class($this)
.'::'.$method.'()');
    }
}


<?php
namespace YourProject\Entity;
use \Nbe\Entity\AbstractEntity;
```

```
class Article extends AbstractEntity
{
    protected $id;
    protected $text;
    protected $date;

    public function setDate(\Zend\Date\Date $value)
    {
        $this->date = $value;
    }
}


$article = new \YourProject\Entity\Article();
$article->define(array('id' => 4)); // will set attribute id
$article->date = 'hello'; // will fail, not a \Zend\Date\Date
```

How it works...

You first implement an abstract Entity. For my implementation I have it in the `Nbe namespace` , because it is not application specific. In the code from the chapter you will see a complete implementation, here in this recipe I only show the basic stuff, like a setter and a getter system. If needed this system allows you to easily implement a specific setter and getter method for a property. You can find such a setter in the concrete Article Entity for the `$date` property. The Article Entity is application specific so it is found in the `YourProject namespace`. Finally you use the entity somewhere in your code, you can set the attributes and then pass that object along in your business layer or even the view. Remember the view should never alter the state of the object, but it may read information from it.

There's more...

Implementing a Collection

Once you have entities you will want to group them at a certain point. This is done in a collection. I have chosen to implement my collections through the use of the class `ArrayObject`. You could also choose to do the implementation completely yourself with the help of SPL (Standard PHP Library). The class `ArrayObject` already implements the SPL `IteratorAggregate`, `ArrayAccess` and `Countable` interfaces which is all the functionality needed to have an object that acts like an array but still is an object. So you get the best from two worlds.

```
<?php
namespace Nbe\Collection;
```

```
class AbstractCollection extends \ArrayObject
{
    protected $entityName = 'stdClass';

    public function offsetSet($key, $val) {
        if (!is_object($val) || $this->entityName !== get_class($val))
{
            throw new Exception('passed entity must be of type
'.$this->entityName);
        }
        return parent::offsetSet($key, $val);
    }
}

<?php
namespace YourProject\Collection;
use \YourProject\Entity;
class Articles extends \Nbe\Collection\AbstractCollection
{
    protected $entityName = 'YourProject\Entity\Article';
}
```

Somewhere in the model or controller.

```
$articles = new \YourProject\Collection\Articles();
for ($x=0; $x<5; $x++) {
    $article = new \YourProject\Entity\Article();
    $article->define(array('id' => rand(100,200)));
    $article->date = \Zend\Date\Date::now();
    $articles->append($article);
}
```

In a view script

```
foreach ($articles as $article) {
    echo $article->date->toString('yyyy MMMM dd');
}
```

First we instantiate an Article Collection on which we are going to append Article Entities. For demonstration purposes I do a loop to make sure 5 random articles are added to the collection. Because of the offsetSet check in the AbstractCollection we are certain that only Article Entities can be added to the collection. Passing any other value will result in an Exception thrown. Because you are certain that each entity in the collection has a date attribute of the type `\Zend\Date\Date` you can safely use this to output the date in the view. The type forcing helps us to guarantee expected results. You could for instance guarantee that the Article Entity can only accept a Comments Collection for its Comments attribute. So the type forcing works both ways. Finally implementing your own collections and entities gives you to more flexibility over for instance using `\Zend\Db\Table\Rowset`, `\Zend\Db\Table\Row` or any native webservice result, such as a soap result which is a simple `stdClass`.

Summary

Doing table manipulations is easy with `\Zend\Db\Table\Table`, it really provides a nice interface and the table and row gateway implementations are great but it has its flaws. Giving the option to put integrities in `\Zend\Db\Table\Table` where it doesn't belong (it belongs in the database) and using those integrities to do lookups on an atomic level (no joins and duplicate queries to the database) is really bad practice and should be avoided. That is why generally speaking I am not really in favour for using `\Zend\Db\Table\TableAbstract` and prefer to implement my own logic to deal with the database. I do love the ease in which you can instantiate a database adapter with the application plugin resource and the interface it provides. It also has a powerful profiler in place of which I am a big fan. Then you have `\Zend\Db\Select` with its strong on the fly building SQL statements capabilities which are very impressive. It still requires you to know what you are doing because it still allows for SQL injection with `group()` or `having()` amongst others, but it does a nice abstraction between databases. In the end I always tend to use the Data Access Object pattern. Where I have full control over data retrieval and how it is given to the business layer. In my opinion your business layer should never come in contact with concrete persistence layer output or output that is really hard coupled. In this optic using a `\Zend\Db\Table\Row` in your controllers is hard coupling and very bad practice. It should only be used within your DAO and converted to your own implementations like the Collections and Entities I showed you. So if you ever want to use the `\Zend\Db\Table` namespace it should only be used between your DAO and the persistence layer.

5
Handling JavaScript and Ajax

In this chapter, we will cover:

- \Zend\Json\Json for encoding and decoding Json datastrings
- Making existing controllers ajax compatible with the Context switch
- Using headScript and inlineScript view helpers
- \Zendx\JQuery and \Zend\Dojo

Introduction

All modern web applications perform `AJAX` (Asynchronous Javascript And XML) calls and have some kind of Javascript library that helps building a nice GUI (graphical user interface). This chapter will not go into the depths of those libraries but will give you guidance on how to integrate them into your Zend Framework. It will also go deeper into the appropriate view helpers already discussed in chapter 2.

\Zend\Json\Json for encoding and decoding JSON datastrings

First of all we have to take a look at the `JSON` (Javascript Object Notation) which has become the de facto standard for communication between different programming languages (and Javascript). It is a very lightweight data interchange format which is easy to build and read. It would however be nice if encoding and decoding is completely automated. This functionality is available as of PHP 5.2.0.. The JSON extension is bundled and compiled into PHP by default. Zend Framework tries to use this extension and when it is not available or when you explicitly choose not to use it, it will use its own encoder and decoder to do it's magic. Using the Zend Framework component provides the same functionality as the extension and more. It also offers a nice convenient interface to convert XML to JSON with a simple method call. Caution is however advised with the format because it confuses Internet Explorer when attributes are used.

How to do it...

Encoding a PHP value to JSON data:

```
\Zend\Json\Json::encode($valueToEncode, $cycleCheck = false,
$options = array())
```

Decoding a JSON data string to an array:

```
\Zend\Json\Json::decode($encodedValue, $objectDecodeType = self::
TYPE_ARRAY)
```

How it works...

Several options are available for encoding.

`$valueToEncode` represents the value (string, integer, object, ...) you want to encode as a JSON string.

`$cycleCheck` The JSON format does not allow object references. This means your object may not have object cycles during encoding. A cycle is when your object contains a reference to itself. When you use the PHP extension a cycle will return a recursive warning (Warning: json_encode() [function.json-encode]: recursion detected) and an incorrect / incomplete JSON object. Using the Zend Framework built-in encoding will result by default in a fatal error "Fatal error: Maximum function nesting level reached, aborting! " If you want to catch this error you have to set `$cycleCheck` to `true`, which will then throw a `\Zend\Json\Exception`.

To illustrate a cycle:

```
class Observable {
    public $observer;
    public $attributeToBeObserved = 4;
}
class Observer {
    public $observable;
}

$a = new Observable();
$b = new Observer();
$a->observer[] = $b;
```

```
$b->observable = $a;
\Zend\Json\Json::$useBuiltinEncoderDecoder = true;
try {
    \Zend\Json\Json::encode($b, true);
} catch (\Zend\Json\Exception $e) {
    echo $e->getMessage();
}
```

The above code example is a deviation of the observer pattern to illustrate. Have a look at the SPL (Standard PHP Library) for a correct interface. If you execute this code the exception will be thrown with the following message: "Cycles not supported in JSON encoding, cycle introduced by class "Observer".

`$options` is an array which can hold the following key value pairs:

`$options['enableJsonExprFinder'] = false` You can encode native Javascript expressions using `\Zend\Json\Expr` and tell the encoder to find them by setting this option to `true`.

`$options['silenceCyclicalExceptions'] = false` If you do not care about an incomplete JSON output when you have recursion you can force the encoder to output anyhow and not throw an exception by setting this option to true in combination with `$cycleCheck=true`. For the above example it would output something like: `{"__className":"Observer","observable":{"__className":"Observable","observer":["* RECURSION (Observer) *"],"attributeToBeObserved":4}}`

Several options are available for decoding.

`$encodedValue` is a JSON string you want to decode to a PHP array or object.

`$objectDecodeType` by default decoding will be done to an array and is controlled by this parameter. Passing `\Zend\Json\Json::TYPE_ARRAY` will convert JSON to an array, `\Zend\Json\Json::TYPE_OBJECT` to an StdClass.

Using the Zend Framework encoder and decoder has its benefits. The encoder adds the class name of each object enabling the client to use that information to do mappings to his own objects. It also has cycle check support embedded with several behaviors from which you can choose as you saw from the examples above.

The price you pay for this added feature list is that the Zend Framework encoder and decoder is slower than the extension. If you have the JSON PHP extension enabled but you want to make use of the built in encoder and decoder of Zend Framework, you alter a static property.

```
\Zend\Json\Json::$useBuiltinEncoderDecoder = true;
```

If you not set this explicitly, the encoder will try to proxy to the `json_encode` function.

There's more...

Using the fromXml() method

```
\Zend\Json\Json::fromXml ($xmlStringContents, $ignoreXmlAttributes=tr
ue)
```

This method is self explanatory. It will accept an XML string and convert it internally with the `simpleXML` extension to an object and convert that object to an array stripped from all XML attributes. This array will then be encoded to a JSON string. If you want the attributes also available in your JSON string you can set the `$ignoreXmlAttributes` to `false`. Be careful with this setting if you want to use your JSON string from within Internet Explorer because attributes are packed together in an `@attributes` object. In IE @ is considered a conditional statement for Javascript. And it will throw a "conditional compilation is turned off" error. However turning it on will surely break, because you did not intend the@ to be a condition. To bypass this error in IE you will have to replace @attributes with something else like a double underscore __attributes after encoding has occurred.

Debugging with prettyPrint()

```
\Zend\Json\Json::prettyPrint($json, $options = array())
```

Reading a JSON object can be a pain because it doesn't have nice formatting. Well this static function is the `\Zend\Debug::dump()` for JSON. It only accepts one `$option` value:

`$option['indent'] = "\t"`. By default it is a TAB. However you can specify whatever indentation format.

Implement your own to Json method in your classes

(Business) Objects that need to be given to the browser and deviate from the normal way `\Zend\Json\Json` handles encoding can implement their own `toJson` method. The beauty is that `\Zend\Json\Json` has support for this method and will proxy to this method when performing a `\Zend\Json\Json::encode()`. This is useful when your attributes (properties from your business objects) are declared non-public. Only the public properties are encoded, others are ignored. However good practice teaches you to encapsulate attributes and control that information with the use of mutators (setters) and accessors (getters). That is why often you also want to encode protected (or even private) properties and make them available in your `JSON` object, which you can do by implementing a toJson method.

Making existing controllers ajax compatible with the ContextSwitch action helper

Keeping it DRY with a nice KISS is what the context switch is all about. Why would you need to re-implement the same logic to handle a user request when only the output formatting should change? The answer is you don't need to (remember MVC?). You will be able to reuse a normal controller action for your XML and AJAX requests. Depending on the format requested by the client, `Zend\Controller\Action\Helper\ContextSwitch` will help you return the correct headers and body output.

Getting ready

Have a controller action for which you want to have a different output format available. In a web application this often is an action that will process user input from a form. To enhance the user experience you now want that form to be handled by AJAX. When pressing submit the entire form should be posted and the success or error messages will be displayed to the user without a page refresh. To make this work your controller action should still process the form and return the success or error messages but it should return this information as a JSON string with appropriate headers instead of the normal response with a layout and view script rendered.

How to do it...

// some controller

```
public function init()
{
    $contextSwitch = $this->_helper->getHelper('contextSwitch');
    $contextSwitch->addActionContext('index', 'json')
                  ->initContext();
}

public function indexAction()
{
    if ($this->_request->isPost()) {
        if (18 == $this->_request->getPost('equation')) {
            $this->view->success = true;
        } else {
            $this->view->error = 'not correct result';
        }
    }
}
```

//corresponding index.phtml

```
<div style="margin-bottom: 5em;">
    <form method="post" action="<?php echo $this->url(array());?>">
        <label for="math">What is the equation of 6x3?</label>
        <input type="text" value="" id="math" name="equation" />
        <input type="submit" value="check" id="submit"/>
    </form>
    <div id="message">
        <?php if (isset($this->success)): ?>
            WOOOW You are a math wizard!
        <?php elseif (isset($this->error)):?>
            <?php echo $this->error; ?>
        <?php endif; ?>
    </div>
</div>

<script>
</script>


<?php $this->headScript()->appendFile('http://ajax.googleapis.com/
ajax/libs/jquery/1.4/jquery.min.js'); ?>
<?php $this->inlineScript()->captureStart(); ?>
$('#submit').click(function() {
    $.post(
        '<?php echo $this->url(array());?>',
        {equation:$('#math').val(), format:'json'},
        function(result) {
            if (undefined !== result.success) {
                $('#message').text('WOOOW You are a math wizard!');
            } else if (undefined !== result.error) {
                $('#message').text(result.error);
            }
            $('#message').show();
            $('#message').fadeOut(3000);
        },
        'json'
    );
    return false;
});
<?php $this->inlineScript()->captureEnd(); ?>
```

How it works...

Let us have a look at the indexAction first. This is a normal action which checks if a POST request has been executed and whether the equation equals 18. If it is equal it will set the success variable in the view and otherwise it will set an error message.

In our view we then have a simple form, still statically built in HTML, because the focus in this chapter is on AJAX. See coming chapter for Zend Framework Forms. This form asks you for the outcome of six times three (=18). When you would submit this form it would post to the current indexAction and on the next page load you will receive a success message or an error message. This work flow has worked for over two decades. Nowadays user interfaces are much approved and AJAX is added. First you load the jQuery library from Google CDN (Content Delivery Network) which guarantees a very fast response. Maybe another site has used that same library which would make it even faster because then the jQuery library is already stored in the client his browser cache. Once the library is loaded with the view helper headScript you start your Javascript coding. What you basically did was put an onclick event on the submit button. This event would make a POST AJAX call to the same indexAction as the form and retrieve a JSON string which will be transformed automatically to a Javascript object. Then it is just a matter of checking if it is a success or an error, and update the message placeholder. The reason why a POST request to the indexAction would return a JSON response and not a full page in HTML containing the layout and view script result is twofold.

1. Because of the ContextSwitch action helper
2. You passed an additional format parameter with the value 'json' in the POST request.

The context switch is set up in the `init()` method from your controller. You grab the context switch from the helper broker and add contexts to it.

`addActionContext($action, $context)` is called for each action for which you want to create a different context. `$context` indicates in which environment your action will be used. In our case we will add a context for a JSON aware environment. After adding the contexts you initialize them with `initContext()`. This will have the default result that when the action is executed the `application/json` header is set, the layout and view script will be disabled and all the public variables of the view will be JSON encoded and returned to the JSON client.

Variables assigned to the view will be converted to `JSON` automatically. So do not forget to implement the `toJson()` method for the objects which have to be encoded. Because the objects that are going to be JSON encoded will only have their public properties available in the JSON string. The view properties to JSON auto convert feature can be disabled before initializing the context.

```
$contextSwitch->setAutoJsonSerialization(false)

->initContext();
```

There's more...

Using another request param to activate the context

`$contextSwitch->setContextParam($name)`

Sets the request parameter to check to determine if a context switch has been requested. The default value is 'format'.

Using AjaxContext action helper

In the above example you have seen how to use the `ContextSwitch` action helper, in this chapter code you will see that the `AjaxContext` is used instead of the `ContextSwitch`. This action helper extends the `ContextSwitch` and has all the same functionality but extends that functionality with its own. It also adds another context "html". This context will render a specific view script with the suffix `.ajax.phtml`. This gives you the opportunity to write HTML specifically targeted to the AJAX requests. Take a look at the code to see it in action.

Using headScript and inlineScript view helpers

When you write code which can load different view scripts some design issues may arise. For some responses you will want to add a script tag in the head of your HTML document, but you do not want to include this for all responses. The solution for this is to use the headScript view helper in all the view scripts which require that certain tags be available. Scripts that might go well in a HTML head tag are for instance JavaScript libraries. Another problem that may arise is that for some responses you want to include JavaScript that needs to be loaded at the end of your document load. For this the view helper inlineScript can be used.

Getting ready

Have a layout loaded with the following lines of code in it.

Between the `<head>` tags <?php echo $this->headScript(); ?>

Just before the closing `</body>` tag: <?php echo $this->inlineScript(); ?>

How to do it...

For this example I will re-visit the view script from the "Making existing controllers ajax compatible with the ContextSwitch action helper" recipe.

How it works...

You first make certain the jQuery library is loaded from the Google Content Delivery Network.

```
<?php $this->headScript()->appendFile('http://ajax.googleapis.com/
ajax/libs/jquery/1.4/jquery.min.js'); ?>
```

This will append the head script file stack with that external javascript source location and makes sure that the library will be available in the response. Because you use Google CDN you make sure the response is quick and it might even already be loaded in the client his browser cache.

Next we want to aggregate all our inline scripts to the bottom of the page document. This way me make sure our entire DOM is loaded and it gives us the possibility to keep the HTML and the JS cleanly separated.

```
<?php $this->inlineScript()->captureStart(); ?>
$('#submit').click(function() {
    $.post(
        '<?php echo $this->url(array());?>',
        {equation:$('#math').val(), format:'json'},
        function(result) {
            if (undefined !== result.success) {
                $('#message').text('WOOOW You are a math wizard!');
            } else if (undefined !== result.error) {
                $('#message').text(result.error);
            }
            $('#message').show();
            $('#message').fadeOut(3000);
        },
        'json'
    );
    return false;
});
<?php $this->inlineScript()->captureEnd(); ?>
```

To aggregate the javascript you use the `captureStart()` and `captureEnd()` of the `inlineScript` view helper. Everything between those two commands will be captured and echoed in the layout just prior to the `<body>` end.

Another example is:

```
<?php $this->inlineScript()->appendFile($this->baseUrl().'/js/poll.
js'); ?>
<?php $this->inlineScript()->captureStart(); ?>
    var pollInstance = new pollHandler();
    pollInstance.init();
<?php $this->inlineScript()->captureEnd(); ?>
```

You can also include external files to be loaded at the end of the body tag.

```
$this->inlineScript()->appendFile($file)
```

In the above example you make sure the external javascript script is loaded prior to outputting the captured content. The order in which you append files and capture content is important.

\Zendx\JQuery and \Zend\Dojo

Sometimes you will have to provide JavaScript to partners which do not have a specific Javascript library loaded and you have to write the code in standard Javascript catching all the differences between the browsers yourself. Handling Javascript without levering one of the many libraries out there is probably making it yourself very difficult. Especially when AJAX, events and effects are involved. Libraries have made it easy to provide cross browser functionality. Zend has built in abstractions for two very popular Javascript libraries: jQuery and Dojo.

Getting ready

`\Zend\Dojo` is available with the main Zend Framework library. You can start using everything that is described in the manual to leverage Dojo, but jQuery is not available in the standard Zend namespace. The abstraction for jQuery is available in the ZendX namespace which is available when you download the full Zend Framework package. The ZendX folder resides in the extras folder. The only thing to do is make your application ZendX namespace aware by dropping it in the library folder and setting the appropriate settings in application.ini. It also has a application resource plugin. How to make your application namespace aware and load application resources can be found in the previous chapters.

There's more...

Loading the libraries manually without using the abstractions

From my perspective Javascript is a complete different language and if you want to fully appreciate and use its power you should not use the `\Zend\Dojo` or `\ZendX\JQuery` components. The abstractions have some nice ideas and I suggest you take a look at the manual and the code but be careful when you use it.

One of those great ideas is to load the libraries from Google CDN. You can use the headScript view helper:

```
<?php $this->headScript()->appendFile('JAVASCRIPT SOURCE LOCATION AT
GOOGLE CDN HERE'); ?>
```

jQuery JS locations:

- the library: http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.js
- minified library: http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js
- UI library: http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.5/jquery-ui.js
- minified UI library: http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.5/jquery-ui.min.js
- localization: http://ajax.googleapis.com/ajax/libs/jqueryui/1.8.5/i18n/jquery-ui-i18n.min.js

jQuery UI CSS location (to be used with headLink):

- http://ajax.googleapis.com/ajax/libs/jqueryui/1.8/themes/base/jquery-ui.css

```
<?php $this->headScript()->appendFile('http://ajax.googleapis.com/
ajax/libs/jquery/1.4/jquery.min.js'); ?>
```

Here you specify 1.4, which means you want the latest 1.4 release. If you specify 1 you request the latest major release for the 1 branch. Likewise requesting 1.4.2 will retrieve that minor version 2. Using the version number gives you a fine grained control on which version you want to have loaded for your application. Another benefit is that Google will always serve the file from the closest available Google server, which is almost certain to be faster than your own host server and you save bandwidth. :)

Dojo JS location:

- the library: http://ajax.googleapis.com/ajax/libs/dojo/1.5/dojo/dojo.xd.js

```
<?php $this->headScript()->appendFile('http://ajax.googleapis.com/
ajax/libs/dojo/1.5/dojo/dojo.xd.js'); ?>
```

Summary

Because PHP is "still" mainly used for web application JavaScript is an exciting and very important part. Knowing how to use the appropriate view helpers to include the libraries, aggregate your inline code and being able to have tools like the ContextSwitch / `AjaxContext` action helpers and automatic JSON encoding and decoding makes your life much more easy. I truly love Zend Framework support for JavaScript by providing the `headScript` and `inlineScript` view helpers, the context switch and the JSON handling. It has made my life much more easy and I thank it for doing so.

6
Web services

In this chapter, we will cover:

- \Zend\Soap
- \Zend\Rest
- \Zend\Amf

Introduction

With the web getting bigger and bigger, companies and communities sharing their data and consumers wanting centralized ways to access all their online information, it is time to take a look at web services. The Word Wide Web Consortium (W3C in short) describes a Web Service as a provider of standard means of inter-operating between different software applications, running on a variety of platforms and/or frameworks. In short it is a software system designed to support interoperable machine-to-machine interaction over a network. In this chapter you will take a look at web services from a provider and a consumer point of view.

\Zend\Soap

Today SOAP (originally Simple Object Access Protocol, now just a word) might be the most popular communication technology between applications. This used to be XML-RPC (Xml Remote Procedure Call) but new service implementations favor SOAP as its successor. This is due to the standardisation of the XML format and the ease this XML is automatically generated with the use of another XML document, the WSDL. Today each programming language has its WSDL generator which makes using SOAP child's play.

Setting up your own SOAP server and client is a very simple job in PHP. When the SOAP extension is enabled you can use `\SoapServer` and `\SoapClient`. Zend Framework has made wrappers around the SOAP extension. I will show you how to use those.

How to do it...

Setting a SOAP server is straight forward. You need two things. A class which is fully documented with DocBlocks specifying a description for each public method, the params it takes and the return values it will provide. Next you have to load that class as a server and make it handle incoming requests.

Set up your class

```
//YourProject/Library/YourProject/Service/Poll.php
```

```
<?php
namespace YourProject\Service;
class Poll
{
  protected $dao;

  public function __construct(\YourProject\Dao\Poll\Poll $dao)
  {
    $this->dao = $dao;
  }

  /**
   * get all polls
   * @return array
   */
  public function findAll()
  {
    $polls =  $this->dao->findAll();
    return $polls->toArray();
  }

  /**
   * get poll by id
   * @param integer $id
   * @return array
   */
  public function find($id)
  {
    $poll = $this->dao->prepareAndExecuteFind(
      array(
        'id' => $id
      )
    );
    return $poll->toArray();
```

```
}

/**
 * Find the active poll
 * @return array
 */
public function findActive()
{
  $poll =  $this->dao->findActive();
  return $poll->toArray();
}


/**
 *
 * Create a poll
 * @param string $question
 * @param array $answers
 * @param boolean $active
 * @return boolean false on failure
 * @return integer id on success
 */
public function create($question, $answers, $active)
{
  $inputFilter = new \YourProject\Filter\Input\Poll(
    array(
      'question' => $question,
      'answer' => $answers,
      'active' => $active
    )
  );

  if ($inputFilter->isValid()) {
    $poll = new \YourProject\Entity\Poll\Poll();
    $poll->question = $inputFilter->question;
    $poll->active = $inputFilter->active;
    foreach ($inputFilter->answer as $value) {
      $answer = new \YourProject\Entity\Poll\Answer();
      $answer->value = $value;
      $poll->answers->append($answer);
    }
```

```
          $param = $this->dao->getParamsContainer('SavePoll');
          $param->poll = $poll;
          if ($id = $this->dao->savePoll($param)) {
            return $id;
          }
        }
        return false;
      }
    }
```

Set up your Controller to provide the above class as a Service

```
//YourProject/Application/modules/services/controllers/Soap.php
```

```
    <?php
    class SoapController extends \Zend\Controller\Action
    {
      public function init()
      {
        $this->_helper->layout->disableLayout(true);
        $this->_helper->viewRenderer->setNoRender(true);
        $this->_response->clearAllHeaders();
        $this->_response->clearBody();
      }

      public function indexAction()
      {
        if (array_key_exists('WSDL',$this->_request->getQuery())) {
          // provide your WSDL here, see there is more...
        } else {
          try {
            $server = new \Zend\Soap\Server(
              'http://yourproject/soap?WSDL'
            );
            $server->setClass(
              '\YourProject\Service\Poll',
              \YourProject\Dao\Poll\Factory::get()
            );
            $server->handle();
          } catch (Exception $e) {
            echo $e->getMessage();
          }
        }
      }
    }
```

How it works...

The Poll Service Class in itself is very simple, it will provide the client with a some operations fully described with the use of DocBlocks. You have to specify a WSDL (Web Services Description Language) when you make a server instance. Technically you do not have to work with a WSDL but if you don't you defy the entire purpose of using SOAP. That said the DocBlocks are used to generate this WSDL. More on how to make it available is found in the *there is more* section of this recipe. Initiating a SOAP Server can be done outside of the MVC (preferred way) or in any controller action. When using in the MVC you have to make sure that the ViewRenderer will not render any view scripts and the Layout is disabled. I also optionally clear the existing response. These three actions make certain that only output generated by the Server will be send back to the client. If there are action helpers or front controller plugins registered which manipulate the response, it might be a good idea to unregister them or perform an exit after `$server->handle()`. But as I said, the preferred way is to setup a server outside the MVC.

Creating the Server is done by creating a `\Zend\Soap\Server` instance with the WSDL location passed as the first parameter.

```
\Zend\Soap\Server::__construct($wsdl = null, array $options = null)
```

As a second parameter you can pass the following options:

'`actor`', '`classmap`', '`encoding`', '`soapVersion`', '`uri`', '`wsdl`', '`features`', '`cache_wsdl`'.

Alternatively you can also instantiate the class without passing options and use the appropriate setters afterwards, `setActor()`, `setSoapVersion()`, and so on.

After it is instantiated you pass it the class on which it needs to operate.

```
\Zend\Soap\Server::setClass($class);
```

Any optional arguments (second, third, and so on) will be passed to that class constructor when instantiated with the use of `func_get_args`. The actual method definition doesn't reflect this, so do not get confused when your IDE gives you a `$namespace` and `$argv` parameter as second and third, they mean nothing. In the example above I pass the DAO as a second argument.

And finally tell the server to handle the request and response.

```
\Zend\Soap\Server::handle($request = null)
```

When a request is set it must be of the type `DOMDocument`, `DOMNode`, `SimpleXMLElement`, `stdClass` or `string`. When you use a string it must be valid XML and when you pass a custom object the magic `__toString()` method should be implemented to return a valid XML. If no request is set it will take the request from the `php://input` stream. Passing a request is useful if you want to do some pre-processing on the `$request` before letting it be handled by the server. The `handle()` echoes the SOAP response by default. If you want to do some response processing before outputting it to the client you can indicate this before calling `handle()` with `setReturnResponse(true)`.

There's more...

Using an instance instead of a class to prepare the server

You can use `setClass()` to instantiate and setup your server request handler. But what if your instance needs more tweaking not available through the use of constructor options. Well you can set preconfigured object instances as request handlers instead of a class.

```
$pollService = new \YourProject\Service\Poll();
$pollService->setDao(\YourProject\Dao\Poll\Factory::get());
$server = new \Zend\Soap\Server(
  'http://yourproject/soap?WSDL'
);
$server->setObject($pollService);
$server->handle();
```

Making available a WSDL for the server and the client

Making available the WSDL is child's play in Zend Framework. The only thing you need is a well documented class (all your classes are well documented right?!) and `\Zend\Soap\AutoDiscover` does the rest.

```
$wsdl = new \Zend\Soap\AutoDiscover();
$wsdl->setClass('\YourProject\Service\Poll');
$wsdl->handle();
```

In the main recipe above we have used the same action to serve the WSDL and the server itself. This is done because requesting a WSDL for SOAP services is often just a matter of appending `?WSDL` to the SOAP URI. We do not want to break that best practice by placing it in a separate action and thus also different URI.

Optimize the server by not using the MVC

Bootstrapping and executing the entire ZF MVC is overkill when you simply want to make a `\Zend\Soap\Server` available. Not using the MVC can give you much faster response times which is critical when you have a SOAP service which is heavily consumed. The faster the response the more requests the service can handle. The SOAP Server is already breaking encapsulation rules by not using the \Zend\Controller\Request and \Zend\Controller\Response components, so there is really no need to use the MVC. Basically the SoapServer API provides you the V and the C of the MVC Pattern.

Setting up the server stays the same, but this time we do not want the MVC to run. We can easily achieve this by using `\Zend\Application`.

```
//YourProject/public/soap.php
```

```
<?php
$doNotBootstrapMvc = true;
include '../application/Application.php';

if (array_key_exists('WSDL', $_GET)) {
    $wsdl = new \Zend\Soap\AutoDiscover();
    $wsdl->setClass('\YourProject\Service\Poll');
    $wsdl->handle();
} else {
    $server = new \Zend\Soap\Server(
      'http://yourproject/soap?WSDL'
    );
    $server->setClass(
  '\YourProject\Service\Poll',
  \YourProject\Dao\Poll\Factory::get());
    $server->handle();
}
```

```
//YourProject/application/Application.php
```

```
// define constants
...
// Create application
$application = new Zend\Application\Application(
    APPLICATION_ENV,
    APPLICATION_PATH . '/configs/application.ini'
);

// do not bootstrap and run depending on a flag
if (!isset($doNotBootstrapMvc)) {
    $application->bootstrap()
                ->run();
}
```

The easiest solution is to re-use your `Application.php` file to avoid code duplication and depending on the `$doNotBootstrapMvc` flag you would bootstrap the application and run it. For your SOAP service you do not need to bootstrap everything and do not need to run the MVC. If you do need to bootstrap some resources before handling the server, you can do so right after the include.

`//YourProject/public/soap.php`.

```
$doNotBootstrapMvc = true;
include '../application/Application.php';
$application->bootstrap('db');
if (array_key_exists('WSDL', $_GET)) {
```

Using \Zend\Soap\Client

The chance that you will need to consume some SOAP service nowadays is big, very big. Web 2.0 is all about sharing data and more and more mashups rise every day. Zend Framework has many clients readily available for the most popular public APIs from services such as Twitter, SlideShare, Amazon, and so on. But not all services have their counterpart in ZF. If the web service is of the type SOAP like our poll service from this recipe it is as easy as pointing towards the WSDL and calling methods like from any local service.

```
$client = new \SoapClient('http://yourproject/soap?WSDL');
$result = $client->findActive();
```

The above code will read the WSDL of our previously created SOAP service and from then on you can do operations on the client as if the service was locally.

With `\Zend\Soap\Client` you have access to all the same methods and functionality as what the PHP SOAP extension delivers with `\SoapClient`.

Using the SoapUI desktop application

A very good SOAP client is the desktop application SoapUI from Eviware (`http://www.eviware.com/soapui/`). It has an excellent free basic version which will allow you to do SOAP requests to your service. It is very easy to use and it will provide you with information whether your service is working. I generally always start with this client when first consuming a SOAP service. By using this tool I have more confidence in the PHP Client I am implementing. It parses the WSDL and gives a nice overview of the operations available and the parameters they require. Because it is an excellent client it is also the ideal candidate for testing your own SOAP services that you are building or providing.

\Zend\Rest

SOAP is currently the most important web service protocol in use at the moment, but REST is gaining ground. REST is an architectural style and the acronym stands for Representational State Transfer. It is used as an architecture which allows machines to talk to each other in a machine-understandable format. REST has no fixed communication language and it can be XML, JSON, or any other language. Also its communication protocol is not fixed but HTTP it the most common for its transfer protocol. What is important in REST is that you understand that everything is a resource. A resource would translate itself in human language as a noun. A glass, a camera, a doll, and so on. On each of these resources you can perform verbs such as GET, POST, PUT, DELETE. You can ask for a glass by issuing the command GET glass, you can throw away the glass by saying DELETE glass, create a glass by POST glass and changing the existing glass by PUT glass (in a different state). In HTTP you have worked with GET and POST already, PUT and DELETE are less known but are valid HTTP commands. REST uses these 4 commands to get or manipulate resources. And each resource is represented with a unique URI. A unique resource identifier such as `api/rest/glass`, `api/rest/camera`, `api/rest/doll` and so on on which you will perform these commands. Zend Framework supports setting up REST resources and a way to consume REST services.

How to do it...

```
//YourProject/library/YourProject/Controller/Action/RestController.
php
```

```
<?php
namespace YourProject\Controller\Action;
abstract class RestController extends \Zend\Rest\Controller
{
  public function init()
  {
    $this->_helper->viewRenderer->setViewSuffix('xml.phtml');
    $this->_response->setHeader('Content-type', 'text/xml');
  }

  protected function getPutOrDeleteInput()
  {
    $requestParams = array();
    if ($rawBody = $this->_request->getRawBody()) {
      parse_str($rawBody, $requestParams);
    }
    return $requestParams;
  }
}
```

//YourProject/application/modules/rest/controllers/AnswerController.
php

```
<?php
namespace Rest;
class AnswerController
extends \YourProject\Controller\Action\RestController
{
  public function indexAction()
  {
      // expects no parameters, handle in view
  }

  public function getAction()
  {
     // expects only an resource ID, handle that resource in view
      $id = $this->_request->getParam('id');
      if ($this->isValidId($id)) {
        $this->view->id = $id;
      }
  }

  public function postAction()
  {
    // expect params or information
    // needed to create the resource, accepting params is
    // done in the controller, state change is done in the model
    // with the help of the controller.
    // feedback to consumer in the view.
    $input = $this->_request->getPost();
  }

  public function putAction()
  {
     // needs a resource ID to know which resource to alter, edit
     $input = $this->getPutOrDeleteInput();
     if ($this->isValidId($input['id'])) {
         // change resource state
     }
  }
```

```
    public function deleteAction()
    {
       // needs a resource ID to know which resource to delete
       $input = $this->getPutOrDeleteInput();
       if ($this->isValidId($input['id'])) {
           // delete resource
       }
    }

    protected function isValidId($id)
    {
      if (\Zend\Validator\StaticValidator::execute(
        $id,
        'digits'
      )) {
        return $id;
      }
      throw new \Exception('invalid answer Id');
    }
  }
```

//YourProject/application/modules/rest/Bootstrap.php

```
<?php
namespace Rest;
class Bootstrap extends \Zend\Application\Module\Bootstrap
{
  protected function _initRestRoute()
  {
    $this->bootstrap('frontController');
    $front = $this->getPluginResource('frontcontroller')
                  ->getFrontController();
    $restRoute = new \Zend\Rest\Route(
      $front,
      array(),
      array('rest')
    );
    $front->getRouter()
          ->addRoute('rest', $restRoute);
  }
}
```

How it works...

The first thing you want to do is put a base controller in place. This to avoid duplication of code for every resource. Each resource will have its own controller. So that means that you normally would have to implement each controller's `init()` method to set the XML content-type and set the viewRenderer's view suffix. Setting the suffix is optional but I like to do so because I am using XML as the output format and not html. You could also implement a context switch here, if you want for instance to also support JSON. The correct way to go about this in REST is to inspect the request headers for the `Accept` header. Another thing that needs to happen in the base controller is implement a method that can handle the raw body of the request. You need the raw body to extract the PUT and DELETE params from the request. The request has getPost and getQuery methods but no getDelete and getPut methods. Thus you need to implement that for each REST controller. Indicating what kind of content (XML or JSON) is being provided in the request is done with the request header Content-Type. Another reason why this base controller is special is because it is extending a unique controller from the Zend Framework library. `\Zend\Rest\Controller` is an abstract action controller defining the four REST verbs (GET, POST, PUT, DELETE) plus a list action. The indexAction will act as the list action of the specific resource. This way the consumer will know which resource-items are available and where to find them. The list can be accessed at the following resource location `http://yourproject/rest/answer` and the output could look something like the response below.

```
<?xml version="1.0" encoding="UTF-8"?>
<response xmlns:r="http://yourproject/rest"
          xmlns:xlink="http://www.w3.org/1999/xlink">
  <r:items>
    <item id="1"
          xlink:href="http://yourproject/rest/answer/id/1" />
    <item id="2"
          xlink:href="http://yourproject/rest/answer/id/2" />
    <item id="3"
          xlink:href="http://yourproject/rest/answer/id/3" />
  </r:items>
</response>
```

The list is represented in XML with a minimum of information. The list should not return detailed information about each resource-item (answer). If the consumer wants more information about a specific item it should follow the resource location (the xlink). This resource location points to the same location (there is only one resource location, only the verbs change) but with an id param passed as a query param. This makes it effectively a GET operation for the resource. It could also have been `http://yourproject/rest/answer?id=1` but we use Zend Frameworks pretty URIs. When doing a GET request to the resource location passing an ID it should return detailed information.

```
<?xml version="1.0" encoding="UTF-8"?>
<response xmlns:r="http://yourproject/rest"
          xmlns:xlink="http://www.w3.org/1999/xlink">
```

```
  <r:item>
    <id>1</id>
    <value>11-15</value>
    <votes>0</votes>
  </r:item>
</response>
```

The reason why the `getAction()` is executed instead of `indexAction()` is because you passed an id parameter with the GET request method. This mapping is done with the help of `\Zend\Rest\Route`. This is a special route which you have to add to your router registered with the front controller. This can best be done in the module specific bootstrap. The constructor takes three parameters:

```
\Zend\Rest\Route::__construct(
    \Zend\Controller\Front $front,
    array $defaults = array(),
    array $responders = array()
)
```

The first param is the front controller, the second are optional route default values (see the chapter two) and last but very important is the `$responders` array. This will accept module names as values or module-controller as key-value pairs. In the above example you want the entire module rest to be REST. This route will check the request method and set the corresponding action. Thus using POST, DELETE or PUT method will execute the corresponding controller action. Because the GET, PUT and DELETE operations need an id to identify the resource I have created an helper method `isValidId()` to check for the validity of that specific input. Alternatively you can also implement a `\Zend\Filter\InputFilter`. If it is not valid I throw an exception which will be handled by the module specific error handler. This rest error handler will output in XML and could resemble something like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<response xmlns:r="http://yourproject/rest"
          xmlns:xlink="http://www.w3.org/1999/xlink">
  <r:msg>invalid answer Id</r:msg>
</response>
```

If you want to see a REST service in action you can take a look at the example code joining this book. But the basic thing you have to understand is that you use a special router which will forward to the correct controller action depending on the verb used and that each resource should have one corresponding controller. There is no real REST language forced and REST communicates with XML, JSON or another language. You can use a module specific layout and error handler to guarantee this in all circumstances, even in the errorController.

There's more...

Consuming a REST service

The REST client of ZF is a bit different from other web service clients. ZF doesn't have a REST client because it is simply a HTTP request and no uniform response can be parsed as with SOAP or AMF. You need to use the excellent `\Zend\Http\Client` to consume a REST service. `\Zend\Http\Client` is able to execute requests using the GET, POST, PUT and DELETE method.

```
$client = new \Zend\Http\Client(
  'http://yourproject/rest/answer',
  array(
    'adapter'   => '\Zend\Http\Client\Adapter\Curl',
    'curloptions' => array(CURLOPT_FOLLOWLOCATION => true),
  )
);
$requestParams = array(
  'id' => 1,
  'value' => 'a car',
  'votes' => 0,
);

$client->setRawData(http_build_query($requestParams));
$response = $client->request('PUT');
if (200 == $response->getStatus()) {
    $body = $response->getBody();
    // do something with the response body which could be in XML or
JSON
}
```

In the above example you create a `\Zend\Http\Client` and pass the request params as an URL-encoded string to the client using the `setRawData` method. This gives the service access to these params easily because it is in the same format as in which a POST from a form is given. Because the communication language is not fixed the service could alternatively require a XML body.

```
$client = new \Zend\Http\Client(
  'http://somehost/restapi/resource',
  array(
    'adapter'   => '\Zend\Http\Client\Adapter\Curl',
    'curloptions' => array(
        CURLOPT_FOLLOWLOCATION => true,
        CURLOPT_HTTPHEADER => 'Content-Type: text/xml',
    ),
    )
);
$requestBody = '<request><id>2</id></request>';

$client->setRawData($requestBody);
```

After performing the `$client->request($method)` you are returned a `Zend\Http\Response` from which you can inspect the response status code and the body.

Response status codes are important in REST architecture.

- 200 - OK
- 201 - Created (useful with a POST)
- 204 - No Content (useful with DELETE)
- 401 - Unauthorized
- 404 - Not Found
- 500 - Internal Server Error (application error)

The body itself can be XML, JSON or any other language. Depending on the request header `Accept` you get back a response header `Content-Type` which will indicate which kind of body you get back.

\Zend\Amf

With the internet quick evolving to provide various Rich Internet Applications (RIAs) with tools such as Flex, there was a need for a uniform optimized message format. Adobe created this for their products and the Action Message Format (AMF) was born. AMF is the equivalent to what JSON means to Javascript. It can be used for consumption other than through Adobe products, but most likely it will be used for Flash/Flex applications. The AMF protocol is used for client applications such as Flash, Flex to be able to communicate with server-side languages such as PHP. To facilitate this process the `\Zend\Amf` component was introduced.

How to do it...

```
$server = new \Zend\Amf\Server();
$server->setClass('\YourProject\Service\Poll', 'poll');
echo $server->handle();
```

How it works...

`\Zend\Json\Server\Server`, `\Zend\Soap\Server`, `\Zend\XmlRpc\Server` and `\Zend\Amf\Server` all share a similar style of setting up the server. They have similar methods to set classes and adding functions. Yet you do have to take a look at the code itself to know the differences. You already saw in `\Zend\Soap\Server` that the `setClass` method doesn't really follow it's own parameters making the function definition useless. The same applies to `\Zend\Amf\Server`.

```
setClass($class, $namespace = '', $argv = null)
```

The above method indicates it accepts arguments as a third parameter. Yet these are ignored (that is to say the parameter is set to null immediately inside the function).

The `$namespace` parameter is something you will use, because it will indicate to the consumer which service to call. In the above example all the public methods from the class `\YourProject\Service\Poll` will be mapped to the service `poll`. By default it will map your class its name as a namespace, but why would you want to expose your inner structure. There is no `setObject` for `\Zend\Amf\Server` but the documentation states you can simply pass an object to setClass. This is not entirely true, you can pass an object to `setClass` but it will use it as the method is named, to set a class (not an instance). It will perform a `get_class()` on the instance to get the class name. Thus passing a fully prepared instance or a class that needs constructor params is not possible with `Zend\Amf\Server`.

```
echo $server->handle()
```

Finally you echo the response of the `handle()` method. `\Zend\Amf\Server` doesn't have a flag to control the return of the response. As you see there are indeed many subtle differences between services but if you take a look at the code itself with your favorite IDE you will quickly spot them. All the functionality is there to have a complete working AMF service, while implementing I suggest you use your custom FLEX tool, or the excellent PINTA application.

For AMF services I would also follow the same performance settings as for SOAP, there is no need to bootstrap the entire MVC.

There's more...

Using the Pinta desktop application

When consuming or implementing an AMF Service I tend to use the AIR desktop client Pinta. It is a utility that allows developers to make custom AMF service calls and view its output in detail. The creators like to label it an AMF service test and debug utility and it does that. It can be found on Google Code: `http://code.google.com/p/pinta/`

Downloading the \Zend\Amf component standalone

If you do not want to have the full Zend Framework library installed but still want to have AMF capabilities, you can download Zend AMF standalone:

`http://framework.zend.com/download/amf`

Summary

Web services are great, they can be used to create mashups and desktop clients. Web 2.0 is evolving more and more towards open APIs sharing data and this means more freedom for everybody. You are not restricted anymore to display your content on one website, you can share it on any platform attracting even more consumers and increase the market value of your application. Good examples are the major players of today such as Twitter, Facebook, Slideshare. If you are a wholesale you can easily share your stock with your retailers and if you are a retailer you can easily place an order at the wholesale service fully automatically when a customer purchases a backorder from your site. Another great thing about web services is internal scaling of the application. Not all web services should be open to the world. You can also use a service as your persistence layer, which allows you to easily scale and optimize that layer independently. The reasons why you would need to provide or consume a service are vast and in the future you probably are going to work with one, if you haven't already. The most known "one-way" services are feeds (RSS or ATOM). One-way because it allows you to retrieve information but you cannot perform state changes. If you need retrieving and updating then it might be a good idea to switch to SOAP or REST. REST has the big advantage that it scales very easily by just dropping an additional resource in place when needed. The disadvantage is that it is more difficult to consume. And that is where SOAP comes in, it is more easily consumed with it's fixed XML language definition but doesn't scale that easy. It all depends on what you need, both systems have proven their use. Regarding REST and Zend Framework it works perfectly but you still have to set or retrieve special rawBody data from your Request.

With `\Zend\Amf` you can provide Flash applications all the information they need from your application. This enables you to build RIAs which gives you wonderful creative tools with no design limits and natural GUIs like you are used to with desktop applications.

7
Debugging, Exception handling and unit testing

In this chapter, we will cover:

- Replacing `\var_dump` with `\Zend\Debug`
- Log information for debugging
- Exception Handling
- Introduction to PHPUnit testing a ZF
- Zend\Test\PHPUnit\ControllerTestCase
- Unit Testing Models

Introduction

Creating a stable application requires a lot of hard thinking about architecture and logic. Any strange exceptional logic that happens when the application is running should be caught by Exceptions and the Catch mechanism should be used to log information to the system and to provide a fall back such as displaying a nice error message to the user. Logging enables you to see when and how something went wrong. When developing an application you have to test the stability and to guarantee future stability by doing unit tests. This chapter will cover all those grounds and will give you the foundation needed to build powerful and stable Zend Framework applications.

Replacing var_dump with \Zend\Debug

Who hasn't used `var_dump` during the implementation and debugging phase of an application? That is correct, everybody has used it and the truth is probably still do. I use Stepping (step-by-step debugging method of executing one line of code at a time) and love it for code inspection, comprehension, unobtrusiveness, and so on. I have used loggers, on which you will see more in this chapter, to write to the "screen" with the use of FirePHP or to a permanent storage for later analysis. But sometimes I still will want to use `var_dump` during development. Zend Framework acknowledges that and gives you a wrapper for it.

How to do it...

```
$variableToInspect = array('id' => 10, 'text' => '<h2>string</
h2>');
\Zend\Debug::dump($variableToInspect, 'var to inspect');
//output
<pre>var to inspect array(2) {
  [&quot;id&quot;] =&gt; int(10)
  [&quot;text&quot;] =&gt; string(15) &quot;&lt;h2&gt;string&lt;/
h2&gt;&quot;
}
</pre>
```

How it works...

```
\Zend\Debug::dump($var, $label=null, $echo=true)
```

The `$var` will be proxied to the `var_dump` function and the output will be captured but because `$var` is part of a params list of `Zend\Debug::dump` you cannot pass multiple variables for inspection like you were able with

```
\var_dump('a string', array('id'=>10), 8).
```

The `$label` option will prefix the dump with a descriptive label of you choice. This way when you have multiple dumps you know which is what. Debug will always `return` the dump but by default also `echo` it. If you don't want to echo it simply pass `false`. This is handy for storage and later analysis. When storing you probably do not want the output to be processed by `htmlspecialchars` or by `xdebug` (if activated) and wrapped with a `<pre>` tag. To fall back to the normal `var_dump` output you need to use the Server API (SAPI) setter to command line.

```
\Zend\Debug::setSapi('cli');
```

If you are executing the PHP script from CLI, it will use the constant `PHP_SAPI` to detect this setting automatically for you.

Log information for debugging

Where were we if we didn't have logging. Indeed logging makes our lives so much easier, and again it has become that much easier with Zend Framework. In Zend Framework logging is split up in several components: a log, writer(s), filter(s) and formatter(s).

How to do it...

```
$logger = new \Zend\Log\Logger();

$fireBugWriter = new \Zend\Log\Writer\Firebug();
$logger->addWriter($fireBugWriter);

$streamWriter = new \Zend\Log\Writer\Stream(PROJECT_PATH.'/data/error.
log');
$logger->addWriter($streamWriter);

$logger->log('this is awesome, no?', \Zend\Log\Logger::INFO);
```

How it works...

You first have to instantiate a logger object on which you can attach one or multiple writers, filters and or formatters. If you only plan on using only one writer you can pass it to the constructor, but for this example you have attached two writers with the use of the `addWriter` method. One for real time analysis `\Zend\Log\Writer\Firebug` and one for later analysis `\Zend\Log\Writer\Stream`. No matter how much you love the speed of Google Chrome, you cannot deny Firefox for its many very useful addons. The FireBug + FirePHP combination is one of them. `\Zend\Log\Writer\Firebug` is very useful because it allows you to see the logs for a specific web page live from your console. No need to open / query any file or other storage backend (writer) you have chosen. In the above example you log the string 'this is awesome, no?' towards the firebug console and the file system. When you log something a log event is created. A log event is an associative array containing the message, timestamp, priority and priorityName. This event will then be formatted by the formatter and written by the writer.

Imagine something goes wrong in your production server and you need immediate feedback. If your architecture is correctly setup you can have the Exceptions readily available right from your browser window indicating the problem on the spot. At the same time the stream writer will write to the file system for you to retrieve the messages at a later stage. There are several built-in 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

Each one of them can be used as a `\Zend\Log\Logger` constant or through their respective convenience method such as `$logger->info('this is awesome, no?')`.

If the above priorities are not sufficient you can add custom ones:

```
$logger->addPriority('audit', 8)
```

These priorities can also be used to log only to specific writers, which brings us to the next topic: filters.

There's more...

Using filters to do selective writing

A filter blocks a message from being written to the log. You can attach a filter to a logger or to a writer. Both make available the addFilter() method to attach a filter. At this time three types of filters are readily available to you:

`\Zend\Log\Filter\Message`

Accepts a regular expression to filter internally using a `preg_match`. Only those messages that are a match will be logged.

`\Zend\Log\Filter\Priority`

Depending on a priority you can let a message go through to the writer(s) or not. Priorities are integers, so you can also pass a comparison operator as the second parameter.

`\Zend\Log\Filter\SupressFilter`

This is a boolean filter and once attached will de-activate the writer(s) and can be used to deactivate all writers at once because of a certain condition in your application logic. Individual de-activating at a later stage is not supported.

Using a filter is easy.

```
$fireBugWriter = new \Zend\Log\Writer\Firebug();
$firebugFilter = new \Zend\Log\Filter\Priority(Zend\Log\Logger::INFO,
'>=');
$fireBugWriter->addFilter($firebugFilter);
```

This will make sure that only informational and debug messages are written to the firebug console. This filter system allows you to easily use multiple writers all with their targeted messages being logged. A very popular use case is to separate your messages into multiple log files `info.log`, `error.log`, `alert.log` and so on. For each log file you create a writer and attach a specific priority filter with the second parameter set to `==`.

By default the second parameter of the Priority constructor defaults to the operator '<='.

Possible case sensitive operators are `<`, `lt`, `<=`, `le`, `>`, `gt`, `>=`, `ge`, `==`, `=`, `eq`, `!=`, `<>`, `ne`

Using formatters to have a specific format style

Logs are only as useful as their readability. This is where you can specify formats.

Formatters are used to convert the logger internal `event array` into a readable format.

The logger internal event looks like:

```
return array_merge(array(
  'timestamp' => date($this->_timestampFormat),
  'message' => $message,
  'priority' => $priority,
  'priorityName' => $this->_priorities[$priority]
  ),
  $this->_extras
);
```

Which means you can easily attach extra information to the array with the method `setEventItem($name, $value)`

```
$logger->setEventItem(
  'uri',
  $request->getRequestUri()
);

$logger->setEventItem(
  'post',
  http_build_query($request->getPost())
);

$logger->setEventItem(
  'cookie',
  http_build_query($request->getCookie())
);
```

By default `\Zend\Log\Formatter\Simple` is used with the following format:

```
'%timestamp% %priorityName% (%priority%): %message%' . PHP_EOL;
```

All event items can be included with the `%eventitemName%` format and formatters are attached to writers. Except in the case of the Database writer which does not allow this.

```
$logger->setEventItem(
  'uri',
  $request->getRequestUri()
);

$logger->setEventItem(
  'post',
  http_build_query($request->getPost())
);

$logger->setEventItem(
  'cookie',
  http_build_query($request->getCookie())
);

$streamWriter->setFormatter(
  new \Zend\Log\Formatter\Simple(
    '%timestamp% %priorityName% (%priority%): %message%'.PHP_EOL
    .'uri: %uri%'.PHP_EOL
    .'post: %post%' .PHP_EOL
    .'cookie: %cookie%'.PHP_EOL.PHP_EOL
  )
);
```

Using \Zend\Application\Resource\Log for easy but limited setup

Because each application needs a logger the application resource plugin Log has been developed. Internally it takes the configuration params from `application.ini` and forwards it to `\Zend\Log\Logger::factory()` which accepts a `\Zend\Config\Config`. Later you will be able to retrieve this generated logger from the bootstrap ready to be used.

```
//YourProject/application/configs/application.ini
```

```
resources.log.error.writerName = "Stream"
resources.log.error.writerParams.stream = PROJECT_PATH "/data/logs/
err.log"
resources.log.error.writerParams.mode = "a"
resources.log.error.filterName = "Priority"
resources.log.error.filterParams.priority = 3
resources.log.error.filterParams.operator = "=="
```

```
    resources.log.critical.writerName = "Stream"
    resources.log.critical.writerParams.stream = PROJECT_PATH "/data/logs/
    crit.log"
    resources.log.critical.writerParams.mode = "a"
    resources.log.critical.filterName = "Priority"
    resources.log.critical.filterParams.priority = 2
    resources.log.critical.filterParams.operator = "=="

    resources.log.debug.writerName = "Firebug"
    resources.log.debug.filterName = "Priority"
    resources.log.debug.filterParams.priority = 7

//In some controller like the error controller

    public function errorAction()
    {
      $errors = $this->_getParam('error_handler');
      $bootstrap = $this->getInvokeArg('bootstrap');
      if ($bootstrap->hasPluginResource('Log')) {
        $logger = $bootstrap->getResource('Log');
        $logger->log(
          $errors->exception->getMessage(),
          $errors->exception->getCode()
        );
      }
    }
```

Why is this a limited setup? Because it doesn't allow you to do some real tweaking. You cannot set formatters and you only can attach writers and filters PER environment not per user input. Imagine you only want a writer only available when a certain query param is set. If you need that kind of tweaking you will need to implement it yourself as a bootstrap or plugin resource.

The errorController

Under normal circumstances the ErrorController should never be reached by Exceptions thrown from your domain model. All exceptions thrown should be caught in the logic that provoked them. Only uncaught exceptions should reach this fallback controller.
The ErrorController its job is to set the view to give a nice user message and like with all exceptions log it. You know how to log it, now let us take a look at how Exception Handling has changed in ZF2.0.

Exception Handling

Most of the languages have a construct which allows the programmer to handle the occurrence of exceptions, special conditions that change the normal flow of a program its execution. In PHP this language construct was added in version 5 and over time SPL Exceptions were added and best practices emerged. Because Zend Framework 2 is built upon +5.3 it's Exception Handling has also changed for the better. It now uses the SPL library and nested model. Let us take a look at that.

How to do it...

```
//Zend/Translate/Adapter.php
```

```
<?php
namespace Zend\Translator;

use Zend\Log,
  Zend\Locale,
  Zend\Translator\Exception\InvalidArgumentException;

abstract class Adapter
{
  public function addTranslation($options = array())
  {
    // some logic
    try {
      $options['locale'] = Locale\Locale::findLocale(
        $options['locale']
      );
    } catch (Locale\Exception $e) {
      throw new InvalidArgumentException(
        "The given Language '{$options['locale']}'"
        ." does not exist",
        0,
        $e
      );
    }
    // some more logic
  }
}
```

How it works...

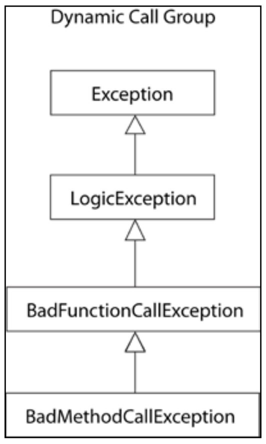
The above class is stripped from all irrelevant implementation details because here you can see Zend Framework Exception Handling in action. Let me show you step by step.

1. The adapter tries to find whether the local option is available to `\Zend\Locale\Locale`.
2. If not the latter will throw an exception of the type `\Zend\Locale\Exception`
3. This Exception will be caught by the client (the adapter)
4. The `addTranslation` method will then throw it's own specific `InvalidArgumentException`. This Exception tells the client (your code) the following three things:
    1. You passed an invalid argument
    2. error code of 0
    3. More information available through the original `\Zend\Locale\Exception` nested as a third parameter.

Now let me explain the steps in greater detail. I am positive you know about Exception Handling, so catching the first `\Zend\Locale\Exception` is nothing new. This is the same as in version 1.* of this wonderful framework. The fun stuff begins at step 4.

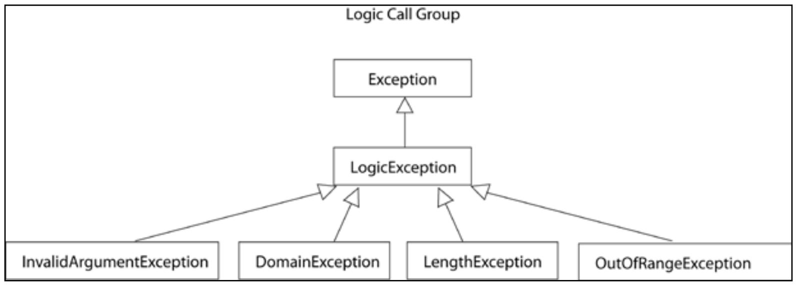
You as a client of the `\Zend\Translator` component expects a `Zend\Translator\Exception` and not a `\Zend\Locale\`Exception so it is caught and re-thrown. But instead of having a general component exception thrown, the code throws a more specific component Exception. `Zend\Translator\Exception\InvalidArgumentException` is a SPL Exception of which there are many, each with their use case.

SPL defines 2 base types and 13 new exceptions in total. All exceptions can be grouped in three categories.



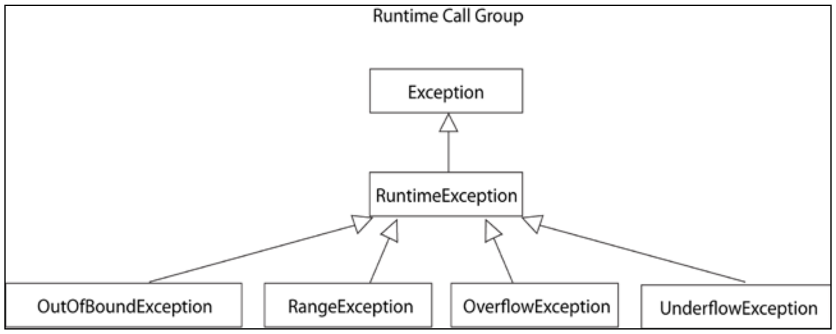
Dynamic call group: These exceptions should be thrown when a method or function is not callable.

1. `BadMethodCallException`: this generally arises as a result from an unresolvable __call
2. `BadFunctionCallException`: should be thrown when a check to `is_callable()` return false.



Logic call group: Should be thrown when there is an exceptional condition that arises when you pass wrong arguments or incorrect mutation of state.

1. `LogicException`: Should be thrown when there is an exception in the program logic and the more specific logic exceptions below or dynamic call group exceptions do not fit the purpose.
2. `InvalidArgumentException`: should be thrown when an invalid argument has been passed such as in our example code from above.
3. `DomainException`: should be thrown when your domain model cannot perform some kind of operation. For example when it tries to guess the operating system being used, but cannot.
4. `LengthException`: should be thrown when a parameter exceeds the allowed length. This can be used for strings length, array size, file size, and so on.
5. `OutOfRangeException`: should be thrown when an illegal index was request.



Runtime call group: these exceptions should be thrown when an exceptional situation arises during the runtime of a method or function call.

1. `RuntimeException`: should be thrown when an error can only be detected at runtime and doesn't fit any of the more specific runtime exceptions. Zend Framework uses this one for example when "No default controller directory registered with front controller ", which can only be detected at runtime...
2. `OutOfBoundException`: should be thrown when an illegal index was requested that cannot be detected at compile time.
3. `RangeException`: is the runtime version of `DomainException` and should be thrown when there are range errors during program execution. Most likely an arithmetic error other than overflow or underflow. An example could be when an illegal index is requested but during runtime `$index >= count($array)`
4. `OverflowException`: should be thrown when an arithmetic overflow has occurred. For example: dynamically you create a gameboard map which can only hold five tiles. When your try to add a sixth tile, this Exception should be thrown.
5. `UnderflowException`: should be thrown when an arithmetic underflow has occurred. In the gameboard program of above you did not add all 5 tiles to the map.

Thanks to all these different Exceptions you can catch Exceptions at different levels. If for example a `InvalidArgumentException` was thrown the client could easily catch it by 3 levels of granularity: `\Exception`, `\LogicException` or `\InvalidArgumentException`. When using a library the client also wants to be able to catch on component level also, this is where the marker interface comes into play. A marker interface hooks the above Exceptions to a ZF Component. Take a look at the following implementation.

`//Zend/Translator/Exception.php`

```
<?php
// marker interface
namespace Zend\Translator;
interface Exception
{}
```

`//Zend/Translator/Exception/InvalidArgumentException.php`

```
<?php
namespace Zend\Translator\Exception;

class InvalidArgumentException
  extends \InvalidArgumentException
  implements \Zend\Translator\Exception
{}
```

This means that besides being able to have the three levels of granularity you have a fourth and fifth: `\Zend\Translator\Exception\InvalidArgumentException` and `\Zend\Translator\Exception`. The latter being the general component Exception. The interface in itself doesn't extend a `\Zend\Exception` because ZF2.0 does not want a dependency to the general library.

This granularity has the following benefits:

- enables catching per Exception type and not catching a general one and then inspecting the message
- Uniform Exceptions which are defined by SPL
- Better understandable for non English users.

Another thing PHP 5.3 supports and ZF uses is nesting of Exceptions.

In the above example a `\Zend\Translator\Exception\InvalidArgumentException` is thrown and more information is available to the client through the original `\Zend\Locale\Exception` nested as a third parameter. Retrieval of the Exception originating from the Locale component can be done with the following code

```
$localeException = $e->getPrevious()
```

There's more...

Best Practices

Try to adhere to the following rules when writing your application code:

1. Use SPL Exceptions and marker Interfaces
2. Because there is no uniform code parameter best practice, I suggest to use the same ones as you use for logging: 1 alert, 2 critical and so on... This makes it easier to log exceptions and to know how to categorize them.
3. If an Exception was caught and you need to re-throw another Exception resulting from that, always nest the original Exception in the new Exception.
4. An Exception is a situation which should never arise so you should always log it. An exception is often an indicator of unstable code and ignoring it can bring forth doomsday.

Introduction to PHPUnit testing a ZF

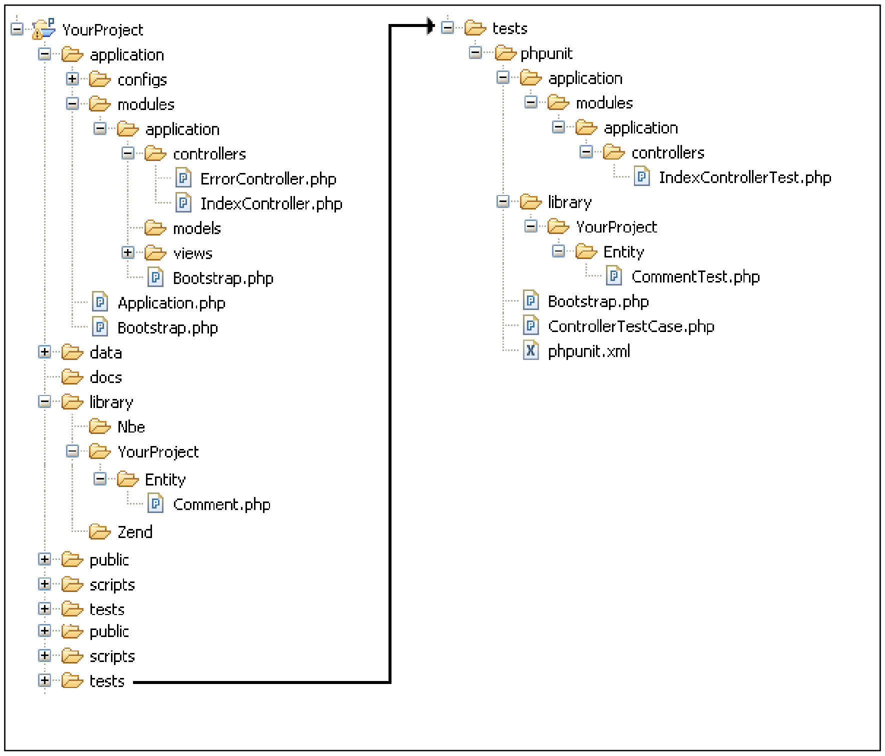
Unit Testing is the testing of units. The tests are done to ensure that each unit is working like it is intended to work.

The tests should be automatic and performed on regular intervals performing validation and verification on the correct working of the units tested. Ideally each test should be independent of another. A unit to be tested is a small piece of software code. Today in Object oriented programming languages these small pieces of software code are generally individual methods from a specific class. Unit testing gives you a way to test the implementations, the design and the behavior of the classes you write.

Today we can claim that Unit testing is a fundamental part of quality modern software development. That is why Zend Framework has Unit tested all their components and recommends you do the same for your models and application. It even provides tools to do functional and integration testing.

How to do it...

First define your directory structure. This should mirror your project structure.



```
YourProject
  application
    configs
    modules
      application
        controllers
          ErrorController.php
          IndexController.php
        models
        views
        Bootstrap.php
    Application.php
    Bootstrap.php
  data
  docs
  library
    Nbe
    YourProject
      Entity
        Comment.php
    Zend
  public
  scripts
  tests
  public
  scripts
  tests
```

```
tests
  phpunit
    application
      modules
        application
          controllers
            IndexControllerTest.php
    library
      YourProject
        Entity
          CommentTest.php
    Bootstrap.php
    ControllerTestCase.php
    phpunit.xml
```

How it works...

If you look closely you will see that the test directory has a PHPUnit directory. This is because your application can also use other testing methodologies. Inside this folder you mimic the existing project structure. This means you create an application folder and library folder. For each class which is native to this project you should create unit tests. You should not create unit tests for external libraries. Zend already has it's tests, so does my library Nbe. Those tests are not included in the project because you should not alter these libraries from within the project itself. However the YourProject library does need extensive testing.

If you haven't done unit testing before or have no prior expertise in PHPUnit I suggest you take a look at the PHPUnit manual or my `PHPUnit training course excerpt`. For those who know how to do it, read the following recipes on how to apply your knowledge to a Zend Framework project.

\Zend\Test\PHPUnit\ControllerTestCase

At the moment of this writing Zend Framework offers tools to facilitate unit testing of your Zend Framework MVC application. The TestCase is build to provide functional testing where you test the behaviour of the system as a whole. This means testing the integration between all the units of code that make up the system.

How to do it...

First you need to setup a PHPUnit Bootstrap. I like to keep to the PHPUnit configuration options naming convention and call mine Bootstrap.php, others call it TestHelper.php. In the end it doesn't really matter because I will show you how to setup a phpunit.xml configuration file which will indicate the bootstrap.

Let us start with that.

`//YourProject/tests/phpunit/phpunit.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit bootstrap="./Bootstrap.php" colors="true">
  <testsuites>
    <testsuite name="YourProject Test Suite">
      <directory>./</directory>
    </testsuite>
  </testsuites>

  <filter>
    <whitelist>
      <directory suffix=".php">../../library</directory>
      <directory suffix=".php">../../application</directory>
    </whitelist>
  </filter>

  <logging>
    <log highlowerbound="80" lowupperbound="50"
    highlight="true" yui="true" charset="UTF-8"
    target="../../data/logs/report" type="coverage-html" />
  </logging>
</phpunit>
```

Next step the `Bootstrap.php` which will setup the constants and take care of autoloading.

`//YourProject/tests/phpunit/Bootstrap.php`

```
<?php
// Define path to project directory
defined('PROJECT_PATH')
    || define('PROJECT_PATH', realpath(__DIR__ .'/../../'));
// Define path to application directory
defined('APPLICATION_PATH')
    || define('APPLICATION_PATH', PROJECT_PATH .'/application');
// Define test path to application directory
defined('TEST_PATH')
    || define('TEST_PATH', __DIR__);

// Define application environment
define('APPLICATION_ENV',  'testing');

// Ensure library/ is on include_path
set_include_path(implode(PATH_SEPARATOR, array(
    realpath(PROJECT_PATH .'/library'),
    get_include_path(),
)));

require_once 'Zend/Loader/StandardAutoloader.php';
$loader = new \Zend\Loader\StandardAutoloader();
$loader->register();
```

With our bootstrapping in place, we need to start setting up a common base class for all our controller tests.

`//YourProject/tests/phpunit/ControllerTestCase.php`

```
<?php
abstract class ControllerTestCase
extends \Zend\Test\PHPUnit\ControllerTestCase
{
    public function setUp()
    {
        $application = new \Zend\Application\Application(
            APPLICATION_ENV,
            APPLICATION_PATH . '/configs/application.ini'
        );

        $this->bootstrap = $application;
        return parent::setUp();
    }
}
```

And finally a real TestCase with several tests and assertions.

```
//YourProject/tests/phpunit/application/modules/application/
controllers/IndexControllerTest.php
```

```
<?php
require_once TEST_PATH . '/ControllerTestCase.php';

class IndexControllerTest extends ControllerTestCase
{
  public function testHomePageShouldPullFromIndexAction()
  {
    $this->dispatch('/');
    $this->assertModule('application');
    $this->assertController('index');
    $this->assertAction('index');
    $this->assertResponseCode(200);
  }

  public function testHomePageMetaData()
  {
    $this->dispatch('/');
    $this->assertQueryContentContains('title', 'my project');
    $this->assertXpath(
      '//meta[@content="text/html; charset=UTF-8"]'
    );
  }

  public function testXhrRequestReturnsJsonForIndexAction()
  {
    $this->getRequest()
       ->setHeader('X-Requested-With', 'XMLHttpRequest')
       ->setQuery('format', 'json');
```

```
        $this->dispatch('/');
        $this->assertHeaderContains(
          'Content-Type',
          'application/json'
        );
      }

      public function testValidLoginShouldAuthAndRedirectToHP()
      {
        $this->request
           ->setMethod('POST')
           ->setPost(array(
             'username' => 'contact@nickbelhomme.com',
             'password' => 'nick'
           ));
        $this->dispatch('/index/login');
        $auth = new \Zend\Authentication\AuthenticationService();
        $this->assertTrue($auth->hasIdentity());
        $this->assertRedirectTo('/');
      }

    }
```

Now your system is set up to run phpunit from the command line.

How it works...

PHPUnit supports a configuration file for setting the run options. The first thing you do is define the bootstrap for PHPUnit to run. Secondly a test suite to define which tests to run, in this case we want all tests recursively found from the current directory. The filter option is used to create your code coverage report. It does not tell you which tests to run, that is handled by the testsuite option. In this case you want a code coverage indication on your .php files from your library and application directories. Next you define to which directory the code coverage report will be saved to and with what options. If you want more detail on this take a look at the PHPUnit manual `http://www.phpunit.de/manual/3.6/en/index.html` or to the free training course excerpt http://blog.nickbelhomme.com/php/phpunit-training-course-for-free_282.

Once this configuration file is set up, PHPUnit will run it automatically when you run PHPUnit from that same directory.

Inside the `Bootstrap.php` you define the same constants as before, but this time you also set the environment to testing for your `application.ini` settings and you register a `\Zend\Loader\StandardAutoloader` to the SPL provided `__autoload` stack. This to ensure you do not need to worry about requiring the appropriate classes.

Zend Framework provides you with `\Zend\Test\PHPUnit\ControllerTestCase`, a TestCase for ZF MVC applications that contains assertions for testing against a variety of responsibilities. One of the things you do not want to repeat in each controller is the setup process. So it makes sense that you extend this class with your own implementation. `\Zend\Test\PHPUnit\ControllerTestCase` relies heavily on the public `$bootstrap` property. There are several ways to setup the bootstrap.

1. As a `\Zend\Application\Application`, this is the way the above example has been setup. Internally the `bootstrap()` method of `\Zend\Application\Application` will be executed.
2. As a callback `array($className, $functionName)`. Will internally be executed with `call_user_func`
3. As a string pointing towards a file, which will include the code to setup the system. Internally this is done with an `include`.

Once the `$bootstrap` property is set, you need to call the `parent::setup()`, which will check the `$bootstrap` property and act accordingly. By extending this `ControllerTestCase` you can begin the functional or system (application) testing.

`TestHomePageShouldPullFromIndexAction` dispatches the `'/'` request and checks if

1. indexAction was the last action used
2. IndexController was the last controller used
3. Application was the last module used
4. The http response code is 200

`testHomePageMetaData` dispatches the same request but this time checks the returned response body. Because you know it is HTML, you can perform checks using the DOM. You can check the DOM with `assertQuery*` and `assertXpath*` assertions, both use `\Zend\Dom` for their magic. In the example I demonstrate both for you, but Xpath is recommended.

Checks if a `<title>` tag exists and is set to 'my project'. Query assertions are done with the help of CSS2 selectors.

```
$this->assertQueryContentContains('title', 'my project');
```

All `assertQuery*` assertions are being transformed behind the scenes to Xpath. During this conversion sometimes things get not converted correctly and for this reason it is best to use the `assertXpath*` assertion family.

```
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"
/>
$this->assertXpath(
  '//meta[@content="text/html; charset=UTF-8"]'
);
```

The Firefox addon Firebug is a great tool to find the Xpath of a DOM element. Inspect the element and select "copy xpath". Despite this great addon you still need to get familiar in defining your own Xpaths for maximum flexibility.

`assertXpath($path, $message = '')`

Checks if the element exists in the DOM

`assertXpathContentContains($path, $match, $message = '')`

checks if the element exists in the DOM and has the correct value

`assertXpathContentRegex($path, $pattern, $message = '')`

Checks if the element exists in the DOM and matches a regex pattern.

`assertXpathCount($path, $count, $message = '')`

Checks if a certain DOM element occurs an exact number of times

`assertXpathCountMin($path, $count, $message = '')`

Checks if a certain DOM element occurs at least this number of times

`assertXpathCountMax($path, $count, $message = '')`

Checks if a certain DOM element occurs maximum this number of times

`testXhrRequestReturnsJsonForIndexAction`

In this test you test for the action helper AjaxContext to kick in. You prepare the request with the correct headers and query params before dispatching. Then you assert the response headers for a specific value.

`assertHeader($header, $message = '')`

Checks if the header exists.

`assertHeaderContains($header, $match, $message = '')`

Checks if the header exists in the DOM and has the correct value

`assertHeaderRegex($header, $pattern, $message = '')`

Checks if the header exists and matches a regex pattern.

`testValidLoginShouldAuthAndRedirectToHP`

This one sets the request to be a POST with the username and password set to a value that should always successfully authenticate. You can set these input parameters and the method because `\Zend\Test\PHPUnit\ControllerTestCase` uses `Zend\Controller\Request\HttpTestCase` and `Zend\Controller\Response\HttpTestCase` for its request and response objects. The application is again dispatched and checked if the identity is available in the auth storage and if a redirect has been set.

`assertRedirectTo($url, $message = '')`

Assert that response redirects to given URL

`assertRedirectRegex($pattern, $message = '')`

Asserts that a redirect is set with a location that matches a regex pattern

Using the above manipulation of the request object, dispatching it and then inspecting the response you can also test your form handling in the application.

`\Zend\Test\PHPUnit\ControllerTestCase` extends `\PHPUnit_Framework_TestCase`, which means if the assertions are not sufficient you can always get back the response, front controller and so on, and use PHP and PHPUnit assertions to test the application.

Unit Testing Models

If you would perform any type of testing it should be this type. These tests are targeted to test the domain and business logic at a much lower level than integration tests such as ControllerTestCase and DatabaseTestCase. Testing your models is exactly as how you would test normal classes with PHPUnit. There should be no need to run an entire application to test a single model. Testing is thus done with `PHPUnit_Framework_TestCase`. The same applies for your custom Zend helpers and plugins.

How to do it...

Testing a Front Controller Plugin.

First you need to add your project namespaces to the autoloader. You have to do this manually because `\Zend\Application` will not be used in `setUp()`.

```
//YourProject/tests/phpunit/Bootstrap.php

    require_once 'Zend/Loader/StandardAutoloader.php';
    $loader = new \Zend\Loader\StandardAutoloader();
    $loader->setOptions(
        array(
            'namespaces' => array(
                'YourProject' => PROJECT_PATH .'/library/YourProject',
            )
        )
    );
    $loader->register();
```

//YourProject/tests/phpunit/library/YourProject/Controller/Plugin/LayoutTest.php

```
<?php
namespace YourProject\Controller\Plugin;
class LayoutTest
extends \PHPUnit_Framework_TestCase
{
  public function setUp()
  {
    $this->plugin = new Layout();
    //we reset because another test can interfere. This
    // is the drawback of a singleton in a single process...
    \Zend\Layout\Layout::resetMvcInstance();
  }

  public function testPluginType()
  {
    $this->assertType(
      '\Zend\Controller\Plugin\AbstractPlugin',
      $this->plugin
    );
  }

  public function testLayoutMvcStartInRouteShutDown()
  {
    $layout = \Zend\Layout\Layout::getMvcInstance();
    $this->assertNull($layout);

    $moduleName = 'application';
    $request = new \Zend\Controller\Request\Http();
    $request->setModuleName('application');
    $this->plugin->routeShutDown($request);
    $layout = \Zend\Layout\Layout::getMvcInstance();

    $this->assertType('\Zend\Layout\Layout', $layout);
  }

  public function testSettingOfLayoutPathInRouteShutDown()
  {
    \Zend\Layout\Layout::startMvc();
    $layout = \Zend\Layout\Layout::getMvcInstance();
```

```
        $this->assertNull($layout->getLayoutPath());

        $moduleName = 'application';
        $request = new \Zend\Controller\Request\Http();
        $request->setModuleName('application');
        $this->plugin->routeShutDown($request);
        $this->assertEquals(
          APPLICATION_PATH
          .'/modules/'
          .$moduleName
          .'/views/layouts',
          $layout->getLayoutPath()
        );
      }

      public function testSettingOfLayoutInRouteShutDown()
      {
        \Zend\Layout\Layout::startMvc();
        $layout = \Zend\Layout\Layout::getMvcInstance();
        $this->assertEquals('layout', $layout->getLayout());

        $moduleName = 'application';
        $request = new \Zend\Controller\Request\Http();
        $request->setModuleName('application');
        $this->plugin->routeShutDown($request);
        $this->assertEquals('default', $layout->getLayout());
      }
    }
```

How it works...

Alter the autoloader setup in the Bootstrap file to include the namespace you want to test. Next Implement a TestCase as you normally would with `PHPUnit_Framework_TestCase`. The only problem is that you will need to depend on some classes and instances from Zend Framework itself and this could also mean singletons. If a singleton survived into the Zend Framework 2.0 branch it will have a reset method, such as seen on the `\Zend\Layout\Layout` component. This instance has to be reset between tests and TestCases, so that is why it is included in the `setUp` and not the `tearDown`. The first test checks if the plugin really extends `AbstractPlugin`. This is critical to be used as a Front Controller plugin. The next test checks if the Layout will be instantiated in the plugin. This is needed for the functionality described in the third test. In this test we check if the plugin `routeShutdown` method alters the `layoutPath`. And finally we also check the setting of the `Layout` in the fourth test.

There's more...

Using \Zend\Test\PHPUnit\DatabaseTestCase

Creating tests for your models which perform CRUD on a database is made easy with `\Zend\Test\PHPUnit\DatabaseTestCase`. This is again a form of integration testing but at a lower level than a ControllerTestCase. The reason why it is integration testing is because it uses several different application layers for testing purposes. Please take a look at the Zend Framework manual `http://framework.zend.com/manual/en/zend.test.phpunit.db.html` for all the info you need to setup a DatabaseTestCase.

Because `\Zend\Test\PHPUnit\DatabaseTestCase` extends `\PHPUnit_Extensions_Database_TestCase` you also might want to take an extra look at the official PHPUnit documentation: `http://www.phpunit.de/manual/current/en/database.html`

Summary

Zend Framework offers you all the tools needed to build and debug robust applications. For debugging it provides you with an enhanced var_dump, which can also be used in combination with logging when the $echo parameter is set to false. The Exception Handling in this version has improved immensely supporting SPL. PHPUnit unit testing has always been available to test your models but Zend Framework adds additional functional and integration TestCases with `\Zend\Test\PHPUnit\ControllerTestCase` and `\Zend\Test\PHPUnit\DatabaseTestCase`.

8 Internationalization and Localization

In this chapter, we will cover:

- Manage the `Locale` setting
- Making a multilingual application with \Zend\Translator
- Easy date displaying with \Zend\Date

Introduction

Learn how to make your website multilingual and local format conventions aware. You know there are a lot of languages in the world, but did you know that date or currency representations differ between regions as well? You do not want to hard code all these differences into your code, instead you will want to use Internationalization (i18n) and Localization (L10n). I18n is the process of designing an application so that it can be adapted to various regions without the need for engineering changes. Once software is internationalized you can localize it. This means plug in all the translations and formats for a specific region. Several components of the ZF library are locale aware so creating an internationalized application is a breeze. Let's take a look.

Manage the locale setting

A locale is what defines which language to use, how to display dates, numbers, currencies and so on. The framework offers `\Zend\Locale` to detect this setting automatically. It can of course still be manually set for maximum flexibility.

How to do it...

```
//YourProject/application/configs/application.ini

    resources.locale.default = "nl_BE"
    resources.locale.force = 1
    resources.locale.registry_key = "Zend_Locale"
```

How it works...

The Locale application plugin resource is used and offers you the most basic setup for an MVC application. The resource accepts three optional parameters.

default: set your own fallback locale which will be used in the case that the locale could not be detected automatically.

force: disable the auto-detection. When set to true you need to set the fallback locale. Default is set to `false`.

registry_key: Make your locale available but do not impact all the locale aware components by setting this option to something else than the default: `Zend_Locale`.

```
resources.locale =
```

When no parameters are set, the resource plugin will try to auto-detect the locale and set it in the registry for all locale aware components.

The following Zend Framework Components are locale aware

\Zend\Currency: Localization of currencies. Injection

\Zend\Date: Localization of dates, times. Injection + Registry

\Zend\Locale\Data: Retrieve localized standard strings as country names, language names and more from the CDLR. Injection

\Zend\Locale\Format: Parsing and generating localized numbers such as amongst others 58.938,21. Locale awareness by injection.

\Zend\Measure: Parsing and generating localized measurements such as amongst others meters to yards. Injection + Registry

\Zend\Translator: Translating of strings to the locale language. Injection + Registry

There's more...

Implement your own \Zend\Locale handling

Often the locale has to be dynamically set. You can use the auto-detection, but sometimes you might want to set the locale from a route parameter or some input (cookie) that is not yet available during bootstrapping.

If this is the case the application resource plugin will not suffice and you will have to create your own front controller plugin, which also means instantiating your own locale object.

```
<?php
namespace YourProject\Controller\Plugin;
use \Zend\Controller\Plugin\AbstractPlugin as AbstractPlugin,
    \Zend\Controller\Request\AbstractRequest as Request;

class Locale extends AbstractPlugin
{
  public function dispatchLoopStartup(Request $request)
  {
      $language = $request->getUserParam('language', 'nl');
      $locale = new \Zend\Locale\Locale($language);

      \Zend\Registry::set('Zend_Locale', $locale);
  }
}
```

The constructor of `\Zend\Locale\Locale` optionally accepts a valid locale string such as `nl_BE` or `nl`.

If no locale is given or `\Zend\Locale\Locale::BROWSER` an automatic search is done and the most probable locale will be automatically set. The search order in this case is

1. HTTP Client, the HTTP_ACCEPT_LANGUAGE header sent by the browser.
2. Server Environment, the systems standard locale
3. Framework Standard, a degraded default "en" locale is available.

Search order when passing `\Zend\Locale\Locale::ENVIRONMENT`

1. Server Environment, also available through static method getEnvironment()
2. HTTP Client, also available through static method getBrowser()
3. Framework Standard, also available through static method getDefault()

Search order when passing `\Zend\Locale\Locale::ZFDEFAULT`

1. Framework Standard
2. Server Environment
3. HTTP Client

Set a new Framework Standard Locale with the static method:

```
\Zend\Locale\Locale::setDefault('nl_BE');
```

Getting Language and Region from the Locale

When you have a Locale instance, you can query it for the language or region through the following methods:

`$locale->getLanguage()` //returns nl for nl_BE

`$locale->getRegion()` // returns BE for nl_BE

You can also translate locale values to human readable values with the static `getTranslation` method

```
getTranslation($value = null, $path = null, $locale = null);
```

$value is the Locale part you want to have information about.

$path will indicate which information you want to get back. There is a huge list and not all of them should be used directly, because you have convenience classes for those such as date and currency. To see the list of paths I suggest you take a look at the manual.

$locale indicates in which language the information for the given `$value` should be returned. If not set it will return it in the language of the corresponding locale set with `$value`.

`\Zend\Locale\Locale::getTranslation('nl', 'language');`

// returns Nederlands

`\Zend\Locale\Locale::getTranslation('BE', 'territory');`

// returns België

Making a multilingual application with \Zend\Translator

`Zend\Translator` is Zend Frameworks solution for multilingual applications. This component is locale aware, which means it can detect the user his preferred language and return the correct translated text. The only thing you need to do is create the translation source files, plug them into your application and return the translations back to the end user.

How to do it...

Make sure the Front Controller plugin is registered and available to the dispatch process.

```
//YourProject/application/Bootstrap.php
```

```
<?php
use \YourProject\Controller\Plugin\Translator as PluginTranslator;

class Bootstrap extends \Zend\Application\Bootstrap
{
  protected function _initPlugins()
  {
    $broker = new \Zend\Application\ResourceBroker();
    $broker->load('frontcontroller')
           ->getFrontController()
           ->registerPlugin(new PluginTranslator());
  }
}
```

The plugin itself:

```
//YourProject/library/YourProject/Controller/Plugin/Translator.php
```

```
<?php
namespace YourProject\Controller\Plugin;
use \Zend\Controller\Plugin\AbstractPlugin as AbstractPlugin,
    \Zend\Controller\Request\AbstractRequest as Request;

class Translator extends AbstractPlugin
{
  public function dispatchLoopStartup(Request $request)
  {
    $translator = new \Zend\Translator\Translator(
      array(
        'adapter' => \Zend\Translator\Translator::AN_ARRAY,
        'content' => array(
          'shopping cart' => 'winkelwagentje',
          'buy' => 'koop',
        ),
        'locale' => 'nl',
        'disableNotices' => true,
      )
    );
```

```
        $translator->setLocale(null);

        \Zend\Registry::set('Zend_Translate', $translator);
      }
    }
```

And finally using the view helper to display the content in different languages in some random view script.

```
    echo $this->broker('translate')->direct('shopping cart');
```

How it works...

First you register the front controller plugin, for more information see chapter three *"Using application hooks with \Zend\Controller\Plugin\AbstractPlugin"*. The reason to implement the translator as a plugin is the Locale component. In the example above the locale is manually set to '`nl_BE`'. You need to set the Locale with setLocale. If you don't it will use the locale from the constructor as the currently used one. If you specify `null` it will try to find the locale to be used in the registry or with auto detection.

The constructor from `Zend\Translator\Translator` only needs a string indicating the adapter to be used. It is however best to also provide the content and a corresponding locale param, if not a notice will be outputted or logged when notices are enabled. This is because by default it will try to load a content for the locale found. Passing params can be done as a parameter list or an options array. If passing multiple options always prefer the options array, this provides more flexibility. In contrast with a parameter list, when a parameter is added, or parameter positions change, your code is not impacted.

The options are

adapter: `\Zend\Translator` works with adapters. By using an adapter the interface is consistent and the client can use on of the many translation source implementations. Available are `ArrayAdapter`, `Csv`, `Gettext`, `Ini`, `Qt`, `Tbx`, `Tmx`, `Xliff` and `XmlTm` and can be indicated by using their respective constant. Because you have to indicate which adapter to use when instantiating a translator object you cannot mix content types in the same instance.

Content: This is the actual translation for a specific locale. An array is passed in the example above, but almost always you will want to pass a string indicating the file or directory location containing the translation source(s). When using a file location with the ArrayAdapter the file has to return the array. `<?php return array(...);`

Locale: Each content must be linked to a specific locale. This locale doesn't have to be fully qualified. Translator only needs the language not the region. You can of course pass a not degraded locale string such as `nl_BE`, internally the language will then be extracted. It is important to realize that when you pass this option to the constructor the locale will be set as the adapter default, effectively ignoring a possible `Zend_Locale` key within the registry and the `\Zend\Locale` autodetection feature. Use the `setLocale` method to use another Locale.

disableNotices: when `true`, omits notices from being displayed or logged. If `false` and a log is made available it will be logged else it will be outputted. Default `false`.

ignore: a prefix for files and directories which should not be added when specifying a file or directory as content.

log: a instance of `\Zend\Log` where logs are written to. Is used in combination with logUntranslated and disableNotices. This is where all the untranslated strings are being logged to. Can be useful for optimization / debugging.

logMessage: message format to be logged. By default `"Untranslated message within '%locale%': %message%"`

logUntranslated: when true, untranslated messages are logged, default false.

clear: boolean, when `true`, then translations for the specified language will be replaced. When false they will be added. In the constructor it doesn't make much sense, but adding this option in the `addTranslation()` method does.

reload: reloads the cache by reading the content again, default `false`.

scan: searches for translation files and directories. Accepts `\Zend\Translator\Adapter::LOCALE_FILENAME` and `\Zend\Translator\Adapter::LOCALE_DIRECTORY` and will cause the adapter to validate each filename or directory for a valid locale format such as `en_XX`. If valid it will be included in the content search.

tag: tag to use for the cache, so it can be deleted by tag (if the backend supports it, see *"chapter 3 - Caching for performance"*)

Languages can be added at a later stage using:

```
$translator->addTranslation($options)
```

`$options` accepts the same parameters as the constructor.

A static locale can be set through the constructor. To dynamically set a locale use the method `setLocale($locale)`. `$locale` can be a locale, 'auto' for autodetection or `null` to first try to retrieve it from the Registry, if not found fall back to auto-detection.

After setting up the translator you have to make the Zend Framework components aware of this instance. The view helper Translate is one of those components that take a look at the Registry to see if a translate instance is registered. Amongst others also \Zend\Form.

Use the view helper to translate your string placeholders.

```
    echo $this->broker('translate')->direct('shopping cart');
```

Results in 'winkelwagentje'.

There's more...

Using the Gettext Adapter

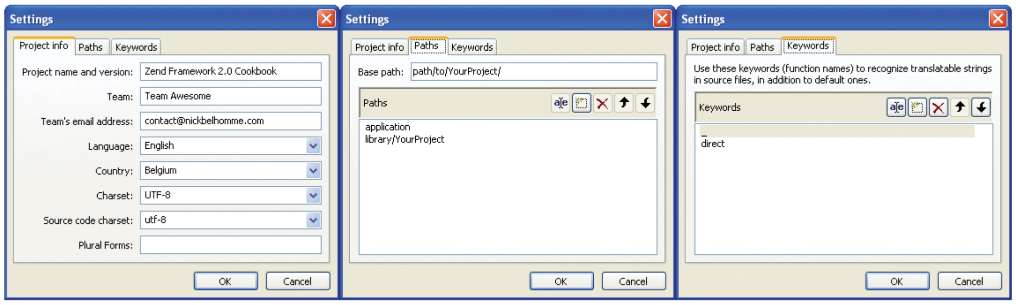
Gettext is a translation source format and together with a tool called Poedit professional translation is very simple.

First setup Poedit

1. Download and install Poedit from `http://www.poedit.net/download.php`
2. Start Poedit and identify yourself as the user of the program
3. In Poedit go to File → Preferences → Parsers
4. Here select PHP and press Edit
5. set the list of extensions to: *.php;*.phtml
6. Alter the Parser command to: xgettext –force-po -o %o %C %K %F -L php
7. Confirm your changes with OK

Secondly create a Project Catalog file.

1. Create a new Catalog: File → New Catalog



Project name and version: Zend Framework 2.0 Cookbook
Team: Team Awesome
Team's email address: contact@nickbelhomme.com
Language: English
Country: Belgium
Charset: UTF-8
Source code charset: utf-8
Plural Forms:

Base path: path/to/YourProject/
Paths
application
library/YourProject

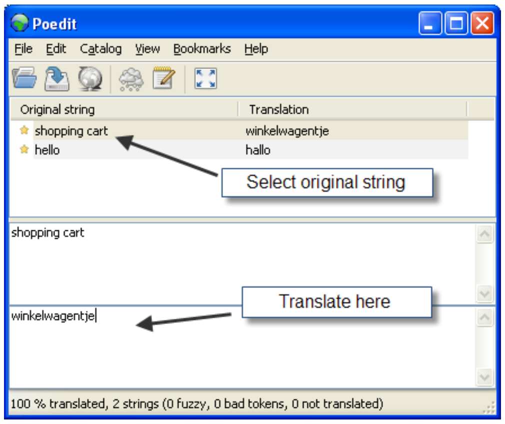
Use these keywords (function names) to recognize translatable strings in source files, in addition to default ones.
Keywords
_
direct

2. At the Project info tab be sure to fill out all the information. You do not need to specify a Language or plural forms. You do however have to set the charset and source code charset to utf-8.
3. In the Keywords tab, add keywords `translate` and _
4. At the tab Paths make sure you have the absolute file system path towards the application folder (/full/path/to/YourProject/application) of the project added and the project library (/full/path/to/YourProject/library/YourProject) itself. Poedit will search in these paths for the files with the extensions defined in step 5. If it finds within these files keywords defined in step 3, the translations identified by those keywords will be added in the catalog.

5. Confirm all changes
6. Update the catalog:
   Catalog → update from sources. You need to do this each time when you add translation strings to your project. Poedit will parse your project and add them to the list.
7. save the catalog to /path/to/YourProject/languages/default.po
8. close Poedit and rename the above default.po file to default.pot, remove the .mo file. This is to create a .pot file we will be using for updating our language sources.

Third, create a (dutch) language file.

1. Start Poedit again and select File → New catalog from POT file and select the default.pot file.
2. Go to the Paths tab and remove the base path and other paths registered. This will have the effect that you can only update this language source from a POT file and not by parsing the code.
3. Save the file to /path/to/YourProject/languages/nl/LC_MESSAGES/default.po for a dutch (nl) language file.
4. Translate all the original strings and save again.



Original string	Translation
shopping cart	winkelwagentje
hello	hallo

shopping cart

winkelwagentje

100 % translated, 2 strings (0 fuzzy, 0 bad tokens, 0 not translated)

Finally setup your Gettext adapter to use the automatically saved .mo files.

```
$translator = new \Zend\Translator\Translator(
  array(
    'adapter' => \Zend\Translator\Translator::AN_GETTEXT,
    'content' => PROJECT_PATH .
      '/languages/nl/LC_MESSAGES/default.mo',
    'locale' => 'nl',
    'disableNotices' => true,
  )
);
```

And add more sources

```
$translator->addTranslation(
  array(
    'content' => PROJECT_PATH .
   '/languages/fr/LC_MESSAGES/default.mo',
    'locale' => 'fr',
  )
);
```

Cache your Translations to speed things up

You can easily speed things up with caching. For this you instantiate your `\Zend\Cache\Frontend` and pass it to the `setCache` Method.

```
\Zend\Translator\Translator::setCache($cache)
```

Easy date displaying with \Zend\Date

For years you have used the various PHP date and time functions and hopefully starting PHP 5.2 you have started using the `DateTime` class which gives you an object oriented way to handle dates and times. Zend Framework `\Zend\Date` offers the same functionality and more. It offers a simple API, is completely internationalized, supports an unlimited timestamp range, ISO-8601 date specification support and can calculate sunrise and sunset times.

How to do it...

First make sure you set your project default timezone (see there is more section).

After this you can use `\Zend\Date` to do date, time manipulations and formatting.

```
$date = new \Zend\Date\Date();

echo $date;

// outputs 28 nov. 2010 19:51:06, when the locale nl_BE is set in the
\Zend\Registry registered Locale.
```

How it works...

First an instance is created with the constructor its default settings:

```
__construct($date = null, $part = null, $locale = null)
```

Passing no `$date` will create an instance with the current time. `$date` could be a unix timestamp, localized date, string, integer, array even parts of dates or time are supported. Very flexible indeed. Let's take a look at how `\Zend\Date\Date` accepts the various inputs which are not so clear in how they should be passed.

localized date: A string such as "28 nov. 2010 19:51:06" . This is tricky because the string parsing relies on the locale. "28 Nov. 2010 19:51:06" will for instance fail for the locale 'nl_BE', which it retrieves from the `\Zend\Registry`. Passing the `$locale` parameter set to 'en_US' solves the problem because in the English language month names start with a capital. But as you can see this is very error prone. But if you can trust the source to be correct, it might be a way to go.

string or integers: By supplying the `$part`, strings can also be used to indicate a part of a date (a month)

```
new \Zend\Date\Date('05', \Zend\Date\Date::MONTH_SHORT);
```

or a complete date formatted with a fixed known format (databases often output in ISO 8601 format).

```
new \Zend\Date\Date(

     '2010-11-28T19:51:06+01:00',
     \Zend\Date\Date::ISO_8601
   );
```

array: You can also pass an array containing all values

```
$date = new \Zend\Date\Date(

     array(
       'year' => 2010,
       'month' => 11,
       'day' => 28,
       'hour' => 19,
       'minute' => 51,
       'second' => 06,
     )
   );
```

When the `Date` instance is created you can simply echo it thanks to the __toString implementation which simply proxies to the toString() method.

```
   public function __toString()
   {
       return $this->toString(null, $this->_locale);
   }
```

There's more...

Controlling the output of \Zend\Date\Date

The default implementation of the magic `__toString` method proxies to

```
toString($format = null, $type = null, $locale = null)
```

which also supports the following interfaces

```
toString($format = null, $locale)
```

```
toString($locale)
```

When no `$format` is given it will return the date in the default locale its date and time format. When no `$locale` is given it will try to find it in the Registry and fall back to autodetection as a last resort.

`\Zend\Date\Date` has to know in which kind of format-type you specify your format. Date uses 2 types iso (the default) and php, and both are very different. Take a look at the following formats:

iso: Y - Year according to ISO 8601, at least one digit

php: Y - Year, at least four digit

The php format type is what you probably are most accustomed to and what is used in PHP functions such as date().

Thanks to the `toString` method you can output in any custom or pre-defined format. When using a pre-defined format be aware that this can differ depending on which locale you use.

//output to a cookie acceptable format :

```
$date->toString(\Zend\Date\Date::COOKIE);
```

//output to a custom defined string 01 January2011 using the default type

```
$date->toString('dd MMMM yyyy');
```

In most cases It is critical to use the real year y (lowercase) and not ISO year Y (uppercase). If you are using `dd MMMM yyyy` you will get '01 January 2011' but if you use `dd MMMM YYYY` you will get '01 January2010' for the date 2011-01-01.

To see all constants and available format specifiers for both types take a look at the documentation: `http://framework.zend.com/manual/en/zend.date.constants.html`

Getting a part of the Date

`ToString()` is used for retrieving a string representation of the date for displaying or storage purposes. To retrieve a specific part of the Date for logical purposes two options are available

`$date->toArray()` will return array representation of the selected date according to the conventions of the object's locale.

`$date->get()` will return the date its `timestamp`.

`$date->get($part, $locale)` will return the part of the date in the locale its format. `$part` can again be a constant or a format string. Specify a `$locale` if you want the output in another locale format than the `Date` it's registered locale.

```
$date->get(Zend\Date\Date::MONTH);
```

Speed up Date with Caching

You can easily speed things up with caching. For this you instantiate your `\Zend\Cache\Frontend` and pass it to the static `setOptions` Method.

Before creating a single Zend\Date\Date instance. (in the bootstrap or a plugin)

```
\Zend\Date\Date::setOptions(array('cache' => $cache))
```

Internally it will cache the Locale data.

Set the project default timezone

If access you should always set the default timezone in your php.ini file.

```
date.timezone = "Europe/Brussels"
```

Alternatively when using the MVC implementation add the configuration to your `application.ini` file.

```
phpSettings.date.timezone = "Europe/Brussels"
```

Or do it manually with `date_default_timezone_set('Europe/Brussels');`

The available timezones can be found here: `http://php.net/manual/en/timezones.php`

Summary

Zend Framework offers a very powerful and easy to use Internationalization and Localization. Setting the i18n and L10n components and making them interact requires the use of the Registry. Generally try not to use the `\Zend\Registry`, except to make the Zend Framework components aware of each other in the MVC implementation. A Registry is dangerous because it is just some kind of super super-global. From a coding point of view you do not know where something was set or overwritten. Injection is the better way to go, where you pass the value through the constructor or through a specifically designed setter.

During your application initialization you set the Locale to be used and all locale aware components will be immediately localized. Zend\Translator offers a very consistent and easy to use API and creating translations for you application can be easily maintained from a array point of view or from a Poedit setup. Smaller websites can use arrays but from a certain point it might be better to start using the Poedit setup because it separates the technical expertise (touching arrays) from editing content (translations).

Be aware however that Poedit is a binary format and that you will loose the easy merging possibilities. Ensuring that no more than 1 person can work at the time on such files is no luxury.

In this Chapter we have scratched the surface of \Zend\Date, this means only the creation and outputting, but not the manipulations. Zend\Date offers various date manipulation methods such as add and substract dates and date parts.

I love the i18n and l10n of ZF that much that I always internationalize my applications from the start, even if I probably know only 1 region and language will be supported. It is that easy to setup and use.

9
Handling mail with \Zend\Mail

In this chapter, we will cover:

- How to send mail
- How to retrieve mail

Introduction

Handle mail from within your application. Send and retrieve mail with \Zend\Mail.

How to send mail

Often – almost always – your application must send some mail towards an e-mail address. Often used for feedback to or from your users, in some applications for monitoring. Setting up a mail and sending it will be explained now.

How to do it...

```
$mail = new \Zend\Mail\Mail('utf-8');
$mail->setBodyText('here you set the text body of the mail.')
     ->setFrom('contact@nickbelhomme.com', 'Nick Belhomme')
     ->addTo('fanmail@nickbelhomme.com', 'Awesome Nick')
     ->setSubject('Wanted to say hi in the mail subject')
     ->send();
```

How it works...

First you instantiate a `\Zend\Mail\Mail` with the required charset (default iso-8859-1*),* then you specify at least one recipient, a sender, if not set with `\Zend\Mail\Mail::setDefaultFrom($email, $name = null)` , a message body and a subject. Sending the mail is done with send(). The latter method also accepts a transport adapter to be used. By default `\Zend\Mail\Transport\Sendmail` is used and no adapter needs to be passed. Passing an adapter can be useful if you wish to use a different adapter for different recipients.

As always with Zend Framework the architects tried to be as descriptive as possible. In the code above you see set* methods and an add* method. With setting you can only set one configuration per instance, where with add, you add to an existing list of configurations. This means that you can call addTo as many times as you want, or you can do it in one go by passing an array.

```
$mail->addTo(
  array(
    'Awesome Nick' => 'fanmail@nickbelhomme.com',
    'Cookie the Dreadful' => 'cookie@nickbelhomme.com',
  )
);
```

If needed you can also differentiate between recipients with the use of `addCc()` and `addBcc()`, both have the same interface as `addTo()`. But be aware when using Bcc with the SendMail adapter because on Windows it will act as Carbon Copy (cc) and not Blind Carbon Copy (Bcc). If you need Bcc on Windows switch to `\Zend\Mail\Transport\Smtp`.

There's more...

Configuring the transportation system

By default a `\Zend\Mail\Transport\Sendmail` is registered to `\Zend\Mail\Mail` as the default transportation adapter. This adapter is actually a wrapper for the PHP `mail()` function. To send mail via SMTP, an instance of \Zend\Mail\Transport\Smtp needs to be registered with `\Zend\Mail\Mail` before the `send()` method is called.

Configuration is best done once per application. That means in a bootstrap resource or by using the available application resource plugin. The plugin can be used to instantiate a transport for `\Zend\Mail\Mail` or set the default name and address, as well as the default replyto- name and address.

```
//YourProject/application/configs/application.ini

resources.mail.transport.type = smtp
resources.mail.transport.host = "smtp.nickbelhomme.com"
resources.mail.transport.auth = login
resources.mail.transport.username = "Nick"
resources.mail.transport.password = "1234"
resources.mail.defaultFrom.email = contact@nickbelhomme.com
resources.mail.defaultFrom.name = "Nick Belhomme"
resources.mail.defaultReplyTo.email = "no_reply@nickbelhomme.com"
resources.mail.defaultReplyTo.name = "no-reply"
```

The above configuration will effectively instantiate the transport adapter, register it as the default tranport method to the mail class and set the mail defaults. It is the equivalent to:

```
$config = array(
  'auth' => 'login',
  'username' => 'Nick',
  'password' => '1234'
);
$transport = new \Zend\Mail\Transport\Smtp('smtp.nickbelhomme.com',
$config);
\Zend\Mail\Mail::setDefaultTransport($transport);

\Zend\Mail\Mail::setDefaultFrom(
  'contact@nickbelhomme.com',
  'Nick Belhomme'
);
\Zend\Mail\Mail::setDefaultReplyTo(
  'no_reply@nickbelhomme.com',
  'no-reply'
);
```

Testing your application

You have build a cool application and tested the mail functionality with your own mail address, but that only covers it so far. You want to test it with all email addresses exactly like your application would normally run. You want to do integration tests, well you can, for this a special adapter is available `\Zend\Mail\Transport\File`.

If you use the application resource plugin from above, simply override the transport setting in the development section.

Create the directory: `/YourProject/data/mail`

Then modify your application.ini by adding the following configuration.

```
//YourProject/application/configs/application.ini

[development : production]

resources.mail.transport.type = file

resources.mail.transport.path = PROJECT_PATH "/data/mail"
```

Instead of sending any real emails the adapter simply dumps the email's body and headers to a file in the filesystem. That is why you need to specify a path inside your project. Everything inside the data folder is considered volatile so this is the best place to store those mail files.

By default this adapter will create filenames with the format

```
'ZendMail_' . time() . '_' . mt_rand() . '.tmp'
```

File names such as `ZendMail_1291555126_668143206.tmp` make no sense at all when you are searching for a specific mail. You can however setup your own filename format. For this the adapter its constructor accepts a callback option. The callback is a function (lambda) or a function name and is for creating unique custom names such as `fanmail@nickbelhomme.com_2010-12-05_14-37-48_837889509.tmp`

```
$callback = function ($transport) {
  return $transport->recipients
         . '_' .date('Y-m-d_H-i-s')
         . '_' . mt_rand()
         . '.tmp';
};

$adapter = new \Zend\Mail\Transport\File(
  array(
      'path' => PROJECT_PATH . '/data/mail',
      'callback' => $callback
  )
);
```

Using HTML instead of text or using both

You can easily use HTML instead of text or simply use both at the same time.

```
$mail->setBodyText('here you set the text body of the mail.')
     ->setBodyHtml('<b>this is <u>HTML</u></b>');
```

Adding attachments

Often you will want to add attachments to your mails. This can be done in two ways.

With the `\Zend\Mail\Mail::createAttachment` method, which registers and returns a `\Zend\Mime\Part` so you can fine tune it, if necessary.

```
$mail = new \Zend\Mail\Mail();
$mail->setBodyText('Please find attached your order receipt')
  ->setFrom('contact@nickbelhomme.com', 'Nick Belhomme')
  ->addTo('contact@nickbelhomme.com', 'Nick')
  ->setSubject('Receipt of your order at nickbelhomme.com');

$part = $mail->createAttachment(
  $text,
  \Zend\Mime\Mime::TYPE_TEXT,
  \Zend\Mime\Mime::DISPOSITION_ATTACHMENT,
  \Zend\Mime\Mime::ENCODING_8BIT,
  'onTheFly12547.txt'
);
$mail->send();
```

Or with the `\Zend\Mail\Mail::addAttachment` method.

```
$pdf = file_get_contents(PROJECT_PATH . '/data/pdf/onTheFly12547.
pdf');
$part = new \Zend\Mime\Part($pdf);
$part->type = 'application/pdf';
$part->filename = 'onTheFly12547.pdf';
$part->disposition = \Zend\Mime\Mime::DISPOSITION_INLINE;
$part->encoding = \Zend\Mime\Mime::ENCODING_BASE64;

$mail = new \Zend\Mail\Mail();
$mail->setBodyText('Please find attached your order receipt')
  ->setFrom('contact@nickbelhomme.com', 'Nick Belhomme')
  ->addTo('contact@nickbelhomme.com', 'Nick')
  ->setSubject('Receipt of your order at nickbelhomme.com')
  ->addAttachment($part)
  ->send();
```

How to retrieve mail

There are various use cases why someone would build an application to retrieve mail and parse or display it.

Create a ticket system, where everything is grouped by ticket. For this your mail needs to be parsed and inserted into a database automatically.

For filtering, when you send your newsletter to thousands of mails some of those addresses are bound to be expired. All mailer-deamons that are returned can be parsed and the subscriber can be put inactive or removed from the database.

To create an online mail client.

In short many reasons. `\Zend\Mail\Storage` to the rescue.

How to do it...

```
$mail = new \Zend\Mail\Storage\Pop3(
  array(
    'host' => 'pop3.live.com',
    'user' => 'random_pop_account@hotmail.com',
    'password' => '******',
    'port' => '995',
    'ssl' => true,
  )
);
echo $mail->countMessages();
foreach ($mail as $i => $message) {
  echo '<hr>MESSAGE '.$i.' Unique Id '
       .$mail->getUniqueId($i).'<hr>';
  echo '<pre>';
  foreach ($message->getHeaders() as $name => $value) {
    if (is_string($value)) {
      echo "$name: $value\n";
      continue;
    }
    foreach ($value as $entry) {
      echo "$name: $entry\n";
    }

  }
  if ($message->isMultipart()) {
    foreach ($message as $part) {
      echo 'body: '.quoted_printable_decode($part->getContent());
```

```
        }
      } else {
          echo 'body: '.quoted_printable_decode($message->getContent());
      }
      echo "</pre>";
    }
```

How it works...

In the above code sample everything is outputted, so you can simply run this code and see what happens kind of info is returned. Until then let me explain the above code. Let's start from the connection to a pop3 service, in this case a hotmail account. For this we use pop3 from the 4 different storage adapters. The other available mail storage adapters are Mbox, Maildir and IMAP. In the table below you can see which parameters are required and which are optional and have a default value.

Pop3 parameters					
user	**host**	**password**	**port**	**ssl**	
REQUIRED	localhost	''	995	false	
MBox parameters					
filename					
REQUIRED					
Maildir parameters					
dirname					
REQUIRED					
Imap parameters					
user	**host**	**password**	**port**	**ssl**	folder
REQUIRED	localhost	''	995	false	INBOX

Each storage adapter extends AbstractStorage which implements \Countable, \ArrayAccess and \SeekableIterator. This basically means you can use it as an array.

$mail->countMessages() could thus also be written as count($mail) and both will tell you how many mails are available.

There's more...

Implement your own \Zend\Locale handling

Summary

Zend Framework offers a very powerful and easy to use Internationalization and Localization. Setting the i18n and L10n components and making them interact requires the use of the Registry. Generally try not to use the `\Zend\Registry`, except to make the Zend Framework components aware of each other in the MVC implementation. A Registry is dangerous because it is just some kind of super super-global. From a coding point of view you do not know where something was set or overwritten. Injection is the better way to go, where you pass the value through the constructor or through a specifically designed setter.

During your application initialization you set the Locale to be used and all locale aware components will be immediately localized. Zend\Translator offers a very consistent and easy to use API and creating translations for you application can be easily maintained from a array point of view or from a Poedit setup. Smaller websites can use arrays but from a certain point it might be better to start using the Poedit setup because it separates the technical expertise (touching arrays) from editing content (translations).

Be aware however that Poedit is a binary format and that you will lose the easy merging possibilities. Ensuring that no more than 1 person can work at the time on such files is no luxury.

In this Chapter we have scratched the surface of \Zend\Date, this means only the creation and outputting, but not the manipulations. Zend\Date offers various date manipulation methods such as add and substract dates and date parts.

I love the i18n and l10n of ZF that much that I always internationalize my applications from the start, even if I probably know only 1 region and language will be supported. It is that easy to setup and use.

10
Authentication and Authorization

In this chapter, we will cover:

- Authenticating with \Zend\Authentication
- Authorization with \Zend\Acl

Introduction

This chapter will cover authentication - the process of verifying someones identity - and authorization - the process of deciding whether that identity has access to a specific resource.

Because authenticating and authorizing are two different processes Zend Framework provides you with two separate components

`\Zend\Authentication` and `\Zend\Acl` (Access Control List - which defines a resource list to check whether a specific role has access).

Authenticating with \Zend\Authentication

Authentication is often mixed with authorization. The reason is simple. Authentication is often used to shield a user from a specific part of the application. If you are not logged in you are redirected to a login page or receive the message that you need to login to view the page. While this approach works it has some problems.

a lot of duplicate code: checking whether the user is authenticated on each page you want to restrict access

does not provide you with the flexibility to differentiate even further between logged in users.

makes refactoring difficult if you ever need that differentiation.

That is why you need to see Authentication as what it is: is the user who he claims he is. The next step controlling to what that user has access to is done with authorization.

Authenticating is done by providing credentials of some sort. This can be any input you can think of. The most common use case is a username or email address in combination with a password. With facebook reaching millions of users, providing a facebook auth is also part of the common use cases. `\Zend\Authentication` works with an adapter system making it possible for you to implement all possible use cases. Zend Framework ships with digest, HTTP, Ldap, OpenId and a database adapter. Once the adapter is set authentication is only a 3 line process (handling an invalid authentication takes some more).

How to do it...

First define the namespaces to be used:

```
use Zend\Authentication,
      Zend\Authentication\Result;
```

next use the component.

```
$authService = new Authentication\AuthenticationService();
$result = $authService->authenticate($authAdapter);
if ($result->isValid()) {
  // User is successfully logged in,
  // do some stuff like getting the identity
  // or doing a redirect...
  $identity = $result->getIdentity();
} else {
  switch ($result->getCode()) {
    case Result::FAILURE_IDENTITY_NOT_FOUND:
      // identity not found
      break;
    case Result::FAILURE_CREDENTIAL_INVALID:
      // invalid credential
      break;
    default:
      // other reasons, you could output
      // the reasons why during debugging
      foreach ($result->getMessages() as $message) {
        echo "$message\n";
      }
      break;
  }
}
```

How it works...

Authentication happens with the help of an authentication service. All instances share by default the same storage - `\Zend\Authentication\Storage\Session` - unless you set a different `\Zend\Authentication\Storage` during initialisation or with the setter method. Because `\Zend\Authentication\AuthenticationService` is not a singleton it is possible to use multiple auth instances at the same time without the storage conflicting. A storage is embedded in the authentication because you only want your users to give their credentials once and not on every request.

Once an instance is created you can authenticate with the help of the adapter. The authenticate method will return a `\Zend\Authentication\Result` on which you can do various operations. One of them is checking whether it was a valid authentication - in which case the user gave the correct credentials. The other operations are used to get feedback on why the authentication was not valid. Depending on the error code you can return different error messages to the user or take appropriate action such as adding a captcha after 5 invalid authentications.

There's more...

Using the DbAdapter

```
$authAdapter = new \Zend\Authentication\Adapter\DbTable(
  $dbAdapter,
  'users',
  'username',
  'password',
  'md5(?)'
);

$authAdapter
  ->setIdentity('yourUsername')
  ->setCredential('yourPassword');
```

First you configure the instance with constructor parameters. These parameters are

1. a database adapter `\Zend\Db\Adapter\AbstractAdapter` so the Auth Adapter can connect to your database. Optional, if not set will get the default adapter registered in the system.
2. the table name for where to find the validation fields. Optional, can also be set with the special method: `setTableName`
3. The table field for the identity. Optional, can also be set with the special method: `setIdentityColumn`
4. the table field for the credential. Optional, can also be set with the special method: `setCredentialColumn`
5. Credential treatment. Optional, can also be set with the special method: `setCredentialTreatment`. In a good design all user credentials are stored securely. This means they are not retrievable and are often stored with some one way encryption such as md5.

Secondly you set the actual values needed for validation. These are the identity, it is who the user claims he is. And the credential – the proof the user is who he claims is. In real life when a police officer asks for your name, you identify yourself and when he asks for credentials you give your passport.

Sometimes you want an additional check in your table. Such a check can be whether the user has already activated his account. It could be set next to the credential treatment. `setCredentialTreatment('md5(?) AND active = 1')`

But a far better way is retrieving the select object from the adapter.

```
$adapter->getDbSelect()->where('active = 1');
```

Implementing your own Authentication Adapter

Facebook is being used by more than 500 million users. Why not use that power to offer a one click authentication to your users. This can be done by implementing your own Auth Adapter specifically designed for Facebook authentication. For this to work you will need to set up an Facebook application linked to your domain. This can be done at `http://developers.facebook.com/setup`. Facebook will then give you a client id and a secret key. Only once during the first authentication your users will have to accept your Facebook application.

```
//YourProject/library/YourProject/Authentication/FacebookAdapter.php
```

```
<?php
namespace YourProject\Authentication;
use Zend\Authentication\Result,
      Zend\Authentication\Adapter;
```

```
class FacebookAdapter implements Adapter
{
  protected $facebookId;
  protected $clientId = '12345678910147852';
  protected $clientSecret = '4bc4e11a54h12c4f25e54g54h2c4';
  protected $callBackUri;
  protected $code;
  protected $token;

  public function __construct($callbackUri) {
    $this->callBackUri = $callbackUri;
  }


  public function getFacebookAuthUri()
  {
    // Get a Facebook authorization for the publish_stream and
offline_access
    return 'https://graph.facebook.com/OAuth/authorize?client_id='
      .$this->clientId
      .'&redirect_uri='
      .$this->callBackUri
      .'&scope=publish_stream,offline_access';
  }

  public function setCode($code)
  {
    $this->code = $code;
  }

  public function authenticate()
  {
    $code = Result::FAILURE_CREDENTIAL_INVALID;
    $identity = null;
    $messages = array();

    if (null !== $this->code) {
      $accessTokenResult = $this->getAccessToken();
      if (isset($accessTokenResult->error)) {
        $messages[] = $accessTokenResult->error->message;
      } else {
        parse_str($accessTokenResult, $options);
        $this->token = $options['access_token'];
        $userProfileResult = $this->getUserProfile();
        $userProfileResult = $this->decodeResponse($userProfileResult
);
        if (isset($userProfileResult->error)) {
          $messages[] = $userProfileResult->error->message;
        } else {
          $identity = $userProfileResult;
          $code = Result::SUCCESS;
        }
      }
    }
    return new Result($code, $identity, $messages);
  }

  protected function getAccessTokenOptions()
  {
    return array(
      'client_id' => $this->clientId,
      'redirect_uri' => $this->callBackUri,
      'client_secret' => $this->clientSecret,
      'code' => $this->code
    );
  }

  protected function getAccessToken()
  {
    $accessTokenUri = 'https://graph.facebook.com/OAuth/access_token';
    $accessTokenOptions = $this->getAccessTokenOptions();
    return $this->makeFacebookRequest($accessTokenUri,$accessTokenOpt
ions);
  }

  protected function getUserProfile()
  {
    // Use the access_token to retrieve the user's profile
    $graphUri = 'https://graph.facebook.com/me';
    $graphOptions = array('access_token' => $this->token);
    return $this->makeFacebookRequest($graphUri,$graphOptions);
  }

  protected function makeFacebookRequest($uri, $options)
  {
    $client = new \Zend\Http\Client();
    $result  = $client->setUri($uri)
              ->setMethod('GET')
```

```
                ->setParameterGet($options)
                ->request();
      return $result->getBody();
    }

    protected function decodeResponse($result)
    {
      return \Zend\Json\JSON::decode($result);
    }
  }
```

Each adapter should implement the interface `\Zend\Authentication\Adapter` which forces the public method `authenticate()` to be implemented. This FacebookAdapter implements an OAuth system to communicate with Facebook. Zend Framework also has an OAuth component according to the OAuth Core 1.0 Revision A Specification. The above Facebook authenticate implementation does not use this component but is fully customized to match Facebook his understanding of the OAuth protocol. Often providers their OAuth implementation differs from the one that is implemented in `\Zend\Oauth`.

OAuth resolves the need for any username and password sharing between applications and replaces it with a user controlled authorization process.

This authorization process is token based. Your users will authorize whether your application gets access to their private data stored in the other application in this case Facebook. Once your application is authorized it will receive an Access Token associated with the user his account. Using this Access Token, your application can access the private data from the other application without continually requiring the user's credentials. For brevity there is no mapping between a Facebook native object and your own business object identity object in the code above. You should always do so. Relying on third party objects tightly couples your architecture to theirs which isn't optimal. See the code accompanying the book for an example on how to do this.

Once the adapter is implemented you are ready to utilize it in your auth controller.

```
//YourProject/application/modules/application/controllers/
AuthController.php

   use YourProject\Authentication\FacebookAdapter,
         Zend\Authentication\Storage\Session,
         Zend\Authentication\AuthenticationService;


   public function facebookAuthAction()
   {
     $this->broker('redirector')
          ->setUseAbsoluteUri(true)
          ->setGotoSimple('facebook-auth');
```

```
    $facebookAuthAdapter = new FacebookAdapter(
      $this->broker('redirector')->getRedirectUrl()
    );

    if (!$code = $this->getRequest()->getQuery('code', false)) {
      $this->_redirect(
        $facebookAuthAdapter->getFacebookAuthUri()
      );
    }

    $facebookAuthAdapter->setCode(
      $this->getRequest()->getQuery('code')
    );

    $storage = new Session(
      Session::NAMESPACE_DEFAULT,
      'facebook'
    );
    $auth = new AuthenticationService($storage);

    $result = $auth->authenticate($facebookAuthAdapter);
    if ($result->isValid()) {
      // do some action
    }
  }
```

Because the callback URI is used for the initial call and later to retrieve an access token it should always be set in the adapter. That is why it is a constructor parameter that is build with the help of the redirector action helper. This action helper is able to build an absolute URI for the module-controller-action of choice. In the example above I took the opportunity to show you a way to customize the storage adapter for the authentication. This way the Facebook authentication will not conflict with existing valid authentications. When you map the Facebook identity to your own business object this storage differentiation will most likely not be needed anymore.

To auth with Facebook all the user has to do is click on a link that goes to this action.

Display the link in some view script.

```
<a href="<?php
    echo $this->broker('url')
              ->direct(
                  array(
                      'action' => 'facebook-auth',
                      'controller' => 'auth'
                  )
              )
?>">authenticate with Facebook</a>
```

Checking if the user is authenticated and retrieving his identity

`\Zend\Authentication` makes the authentication persistent. The default storage adapter is the current session. Different storage systems are available. Even a stateless storage – `\Zend\Authentication\Storage\NonPersistent` - for when you want to re-auth the user on every request. Think of a system where you have a token stored in a cookie that together with the user his IP address needs to be re-authed against a service each time. If the token is not there or the IP changes the user is not authed.

But regardless of the storage, you can always check if the user is authenticated through the same interface.

```
$auth = new \Zend\Authentication\AuthenticationService();
if ($auth->hasIdentity()) {
    $identity = $auth->getIdentity();
}
```

The Authentication service is checked if it has an identity stored and if so reference retrieve it with the `getIdentity()` method.

Do not worry, you do not have to check on each protected page whether auth has an identity. If you were thinking about implementing that approach you are mixing responsibilities. We let the ACL handle authorization. If for some reason you do want to make this check often than you can create action and view helpers for `\Zend\Authentication` handling.

Logout or clearing the auth storage

When a user is authed, he needs a way to logout again. Because the identity is stored in the storage it is only a matter of clearing that storage.

```
$auth = new \Zend\Authentication\AuthenticationService();
if ($auth->hasIdentity()) {
  $auth->clearIdentity();
}
```

or

```
$auth = new \Zend\Authentication\AuthenticationService();
if ($auth->hasIdentity()) {
  $auth->getStorage()->clear();
}
```

Authorization with \Zend\Acl

`\Zend\Acl` provides an easy to use Access Control List implementation for permissions management. What this means is the list will decide whether or not the user is allowed access to a resource. And what kind of permission or privilege the user has on those resources.

Whenever the user wishes to do something the list will be checked to see if it is permitted. You can set access control on almost everything imaginable with the Zend Framework implementation. The first thing shown is how to control access for controller actions.

How to do it...

When a user is not logged in he can only see the home page, error page and the login screen. All other web pages should be restricted and should forward him to a login screen or display an error message telling him he doesn't have the right to view that page. We will implement this functionality with the means of our controller actions.

`//YourProject/library/YourProject/Acl/Navigation.php`

```
<?php
namespace YourProject\Acl;
use Zend\Acl\Role\GenericRole,
      Zend\Acl\Resource\GenericResource,
      Zend\Acl\Acl;

class Navigation extends Acl
{
  public function __construct()
  {
    $this->addRole(new GenericRole('guest'));
    $this->addRole(new GenericRole('user'), 'guest');
    $this->addRole('admin', 'user');

    $this->addResource(new GenericResource('application'));

    $this->addResource(
        new GenericResource('application.index'),
        'application'
    );
    $this->addResource(
        new GenericResource('application.index.index'),
        'application'
    );
```

```
        $this->addResource(
            new GenericResource('application.auth'),
            'application'
        );
        $this->addResource('application.error', 'application');

        $this->addResource(
            new GenericResource('application.profile'),
            'application'
        );
        $this->addResource(
            new GenericResource('application.profile.index'),
            'application.profile'
        );

        $this->allow('guest', 'application.index.index');
        $this->allow('guest', 'application.error');
        $this->allow('guest', 'application.auth');
        $this->allow('user', 'application.index');
        $this->allow('user', 'application.profile');
        $this->allow('admin');
      }
    }
```

//YourProject/library/YourProject/Controller/Plugin/Acl.php

```
<?php
namespace YourProject\Controller\Plugin;
use Zend\Controller\Plugin\AbstractPlugin,
  Zend\Controller\Request\AbstractRequest,
  Zend\Authentication\AuthenticationService,
  Zend\Controller\Action\HelperBroker;

class Acl extends AbstractPlugin
{
  protected $acl = null;

  public function __construct(\Zend\Acl\Acl $acl)
  {
    $this->acl = $acl;
  }

  public function preDispatch(AbstractRequest $request)
  {
    $role = 'guest';
    $auth = new AuthenticationService();

    if ($auth->hasIdentity()) {
      $user = $auth->getIdentity();
      $role = $user->role;
    }

    $resourceModule = $request->getModuleName();

    $resourceModuleController = $resourceModule
                  . '.'
                  . $request->getControllerName();

    $resourceModuleControllerAction = $resourceModuleController
                  . '.'
                  . $request->getActionName();
    $allowed = false;

    if (
      $this->acl->hasResource($resourceModuleControllerAction) &&
      $this->acl->isAllowed($role, $resourceModuleControllerAction)
    ) {
      $allowed = true;
    } else if (
      $this->acl->hasResource($resourceModuleController) &&
      $this->acl->isAllowed($role, $resourceModuleController)
    ) {
      $allowed = true;
    } else if (
      $this->acl->hasResource($resourceModule) &&
      $this->acl->isAllowed($role, $resourceModule)
    ) {
      $allowed = true;
    }
    if (!$allowed) {
      $loader = new HelperBroker();
      $redirector = $loader->load('redirector');
      if ($auth->hasIdentity()) {
        $redirector->setExit(false)
               ->setGotoSimple(
                 'acl',
                 'error',
                 'application'
               );
      } else {
        if ($request->getRequestUri() !== '/') {
          $redirector->setExit(false)
                 ->setGotoSimple(
                   'index',
                   'auth',
                   'application'
                 );
        }
      }
    }
  }
}
```

```
//YourProject/application/Bootstrap.php
```

```
    protected function _initPlugins()
    {
      $broker = new \Zend\Application\ResourceBroker();
      $broker->load('frontcontroller')
             ->getFrontController()
             ->registerPlugin(
               new Plugin\Acl(new \YourProject\Acl\Navigation())
             );
    }
```

How it works...

`//YourProject/library/YourProject/Acl/Navigation.php` is the actual list.

Before implementing an access control list you should first define your requirements.

Our application should be aware of 4 roles:

guest: this will identify all non authed users

user: a guest who is authed and thus an identified user

admin: authed super user, has total access.

The list will work with these roles. Once a role has been created it needs to be added to the list. To create a role you instantiate a `\Zend\Acl\Role\GenericRole` object. The `GenericRole` constructor accepts a role id. This id can be any string or integer as long as it make sense to you and your application. The role will be the identifier for the user his privileges. If you do not want to bother instantiating the `GenericRole` yourself, you can also let the `addRole()` method instantiate it for you. This approach has been taken for the admin role.

```
\Zend\Acl\Acl::addRole($role, $parents = null)
```

`$role` can be a string (not an integer) or can be an object that implements the `\Zend\Acl\Role Interface`.

Each role can extend another role throught the `$parents` option. So whatever access rules the parent role has the child role will inherit. Thus guest will be the base role with user having the same rules as guest unless overwritten. And admin on its turn extends user. Each overwritten rule will take precedence because Zend searches the rules from child to parent structure.

After defining the roles it is time to think which resource should be implemented. Resources are the things you want to access control and you can handle this with the use of a blacklist or a whitelist approach.

Both have advantages and disadvantages:

resources whitelist: with every new resource you will have to allow the user to this resource. The list can grow very quickly and requires high maintenance. The advantage is that whenever you add a new resource it is denied for everybody (except for the admin), this way making sure no inappropriate access is ever granted.

Resources blacklist: everybody has access to everything until you explicitly start denying access to specific resources. This list can also grow but is less likely to explode. The benefit is less maintenance because with every undefined resource the roles have access. This is however a potential security breach, because you might add a resource which should not be accessible for everybody but you forget to put the deny rule in your ACL.

A whitelist is implemented, so for every controller an extra resource has to be created. The list has to know about module-controller-action resources.

The format used is a string composed out of `module.controller.action` - for which only the module is mandatory and will give the user total access to the module unless refined.

For your convenience a `\Zend\Acl\Resource\GenericResource` is readily available with the `\Zend\Acl\Resource` interface implemented. A `GenericResource` can also be instantiated for you behind the scenes.

```
\Zend\Acl\Acl::addResource($resource, $parent = null);
```

`$resource` can be a string which will instantiate a `GenericResource` or a `\Zend\Acl\Resource` Interface implementation.

Now that you have a resource and roles it is time to setup allow and deny access rules.

Guests should have access to only the home page, the error controller and the auth controller.

Authed users can have access to the full index controller and profile controller. The admin will have full access.

```
\Zend\Acl\Acl::allow($roles = null, $resources = null, $privileges =
null, Assertion $assert = null)
\Zend\Acl\Acl::deny($roles = null, $resources = null, $privileges =
null, Assertion $assert = null)
```

When no roles or resources are passed access is allowed for all roles or for all resources. If needed multiple roles or resources are passed as an array.

With the list implemented it is time to use it for access control. For a use case for granting access to module, controller, actions the best place to do this is in a Front Controller Plugin. `//YourProject/library/YourProject/Controller/Plugin/Acl.php`

This plugin is designed specifically to be used with our ACL. The ACL needs to be injected during instantiation and will be checked on each `preDispatch`.

The application will check whether we have an authed user, if so it will take his role otherwise it will fall back to a guest role. Next thing to do is to dynamically create resource ids.

`$resourceModule`: Resource id such as application.

`$resourceModuleController` Resource id such as application.index

`$resourceModuleControllerAction` Resource id such as application.index.index

Next the code checks whether the resource ids exists and the role is allowed on those available. This is done using the most narrow resource id (the module-controller-action combination) to the broadest one (the module). Because we are using a whitelist approach the resources should exist and the role should have access to it. By default all roles are denied access unless the ACL indicates that the role is allowed access on the questioned resource. If access is denied an authed user is redirected to an access refused error page. A not authed user will be redirected to the authentication page.

To use this plugin we register it to the front controller in our bootstrap.

There's more...

Using privileges

```
\Zend\Acl\Acl::allow($roles = null, $resources = null, $privileges =
null, Assertion $assert = null)
```

For each resource privileges can be added. This is a refinement tool which could give you extra control. Imagine a controller action (a resource) for which you only want to allow GET access. You can specify this in the privileges.

Other privileges are often used when you use an ACL list to control user actions and not pages. Such actions can be posting and deleting comments.

In an ACL list:

```
//YourProject/library/YourProject/Acl/Actions.php
```

```
$this->addResource(new GenericResource('comment'));
$this->allow('user', 'comment', 'post');
$this->allow('admin', 'comment', 'delete');
```

In the controller action which processes a comment post:

```
$identity = $auth->getIdentity();
$acl = new \YourProject\Acl\Actions();
if (
  $acl->hasResource('comment') &&
  $acl->isAllowed($identity->getRole(), 'comment', 'post')
) {
  // allow the user to comment
}
```

Mixing allow and deny rules

Depending on the ratio between allow and deny rules for a specific controller it might be easier to start mixing.

Imagine a user controller with the actions index, edit, delete and update. All users have access to all controller actions except delete.

Instead of specifying allow rules for every allowed action:

```
$this->allow('guest', 'application.user.index');
$this->allow('guest', 'application.user.edit');
$this->allow('guest', 'application.user.update');
```

it might be easier to allow access to the entire controller and refuse the delete action.

```
$this->allow('guest', 'application.user');
$this->deny('guest', 'application.user.delete');
```

Using assertions to refine Allow or Deny Rules

```
\Zend\Acl\Acl::allow($roles = null, $resources = null, $privileges =
null, Assertion $assert = null)
```

Assertions are very powerful and could be used to refine the permissions for a role or resource. You can make a assertion whether the article is published, if not guests and users will not be granted access until it is published.

Setup the ACL as usual but include an assertion which implements the \Zend\Acl\Assertion interface.

`//YourProject/library/YourProject/Acl/Actions.php`

```
$this->addRole('guest');
$this->addResource('article');
$this->allow(
  'guest',
  'article',
  'view',
  new Assertion\IsArticlePublished()
);
```

The actual assertion only needs to implement one method assert.

`//YourProject/library/YourProject/Acl/Assertion/IsArticlePublished.
php`

```
<?php
namespace YourProject\Acl\Assertion;
use Zend\Acl;

class IsArticlePublished implements Acl\Assertion
{
  public function assert(
    Acl\Acl $acl,
    Acl\Role $role = null,
    Acl\Resource $resource = null,
    $privilege = null
  )
  {
    return $resource->isPublished;
  }
}
```

Our resources are dynamic and thus we want to implement the `\Zend\Acl\Resource` interface method `getResourceId()`. This will indicate the "article" resource which we declared in our ACL.

```
//YourProject/library/YourProject/Entity/Article.php
```

```
<?php
namespace YourProject\Entity;
use Nbe\Entity\AbstractEntity,
    Zend\Acl\Resource;

class Article extends AbstractEntity
              implements Resource
{
  protected $id;
  protected $text = 'an attractive article';
  protected $isPublished = true;

  public function getResourceId()
  {
    return 'article';
  }
}
```

Our Roles are also dynamic so you can implement the `\Zend\Acl\Role` interface to dynamically give back the role for the user. The method `getRoleId()` will be used by the ACL to check which role is asserted.

```
//YourProject/library/YourProject/Entity/User.php
```

```
<?php
namespace YourProject\Entity;
use Nbe\Entity\AbstractEntity,
    Zend\Acl\Role;

class User extends AbstractEntity
              implements Role
{
  protected $id;
  protected $role = 'guest';

  public function getRoleId()
  {
    return $this->role;
  }
}
```

and finally the actual check.

```
$actionsAcl = new \YourProject\Acl\Actions();
$role = new \YourProject\Entity\User();
$article = new \YourProject\Entity\Article();
if ($actionsAcl->isAllowed($role, $article, 'view')) {
    echo $article->text;
}
```

The above article is published so the text will be outputted. Changing the `isPublished` property to `false` will stop the article to be outputted for guests and users. If you are an admin, the article will always be outputted.

Summary

`\Zend\Authentication` and `\Zend\Acl` are two components that work perfectly together. They supplement each other very well and will probably be used in all applications. These components are easy to understand and integrate into various setups. `\Zend\Acl` is also integrated in `\Zend\View\Helper\Navigation` and only requires you to set the resource parameter in the Navigation Page Container, inject the ACL into the view helper and the ACL will make certain only allowed links are assembled and displayed. I know you will have fun with this.

11
Zend Form made easy

In this chapter, we will cover:

- Create a form
- Adding protection to your forms
- Styling your forms with Decorators
- About display groups and sub forms

Introduction

Zend Form is probably one of the most controversial components of the framework. The reason is easy, the concept is hard to grasp. But once understood it feels intuitive and flexible enough for all your desires. `\Zend\Form` is build on the decorator design pattern. It is a pattern created to solve the class explosion problem which is created by extending classes to add functionality.

Create a form

To create a form you must first understand the parts which make up Zend Form. A form consists out of elements such as an input field, select box, radio, submit button and so on. Optionally a method, action and other form attributes can be set, but to have a working form you need at least 1 element. Lets get started creating a login form.

How to do it...

```
//YourProject/library/YourProject/Form/Login.php
```

```
<?php
namespace YourProject\Form;
use Zend\Form\Form,
    Zend\Form\Element;

class Login extends Form
{
    public function init()
    {
        $this->addEmail();
        $this->addPassword();
        $this->addSubmit();
        $this->setupForm();
    }

    protected function setupForm()
    {
        $this->setName('login');
    }

    protected function addEmail()
    {
        $element = new Element\Text('email');
        $this->addElement($element);
    }

    protected function addPassword()
    {
        $this->addElement('password', 'password');
    }

    protected function addSubmit()
    {
        $element = new Element\Submit('submit');
        $this->addElement($element);
    }
}
```

```
//YourProject/application/modules/application/controllers/
authController.php

    use YourProject\Form\Login;

    $loginForm = new Login();
    $loginForm->setAction(
        $this->broker('url')->direct('form','auth')
    );

    $loginForm->setMethod('post');
    $this->view->loginForm = $loginForm;
```

In the corresponding view script

```
    $this->loginForm->email->setLabel('your email');
    $this->loginForm->password->setLabel('your password');
    $this->loginForm->submit->setLabel('login');
    echo $this->loginForm;
```

How it works...

Creating a form works with a blank sheet. You start with an empty `\Zend\Form\Form` which you instantiate – or in this case `\YourProject\Form\Login` which extends `Form` – and start adding elements to it with the use of the public method `addElement()`.

`addElement($element, $name = null, $options = null)`

`$element` may be either an element short name from `'\Zend\Form\Element\[SHORT NAME]'` in which case the second parameter `$name` must be provided with a a unique form field name and optionally `$options` for configuring the element. These parameters will be proxied to the element factory method `createElement()`.

The password element is created and added using this approach.

`$element` can also be an object of type `\Zend\Form\Element` in which case `$name` may be optionally provided and any provided `$options` will be ignored. Options can be set to the element during element instantiation or using its appropriate setters.

Our Login form has 3 elements: an email, password field and a submit button. The form will receive an HTML id naming the form thanks to the setName() method. When extending `\Zend\Form\Form` you should not override the constructor but implement the template method `init()`. It will be called automatically on initialization and you do not risk screwing up the constructor.

Once your form is configured in its most basic configuration – this means without adding filtering and validation to the form elements – you can instantiate it in a controller or your service layer as long as it has a corresponding view script for reuse. This form handler will tell the form to post to a specific controller action. You could set this information also in the Login form class itself, but the form should not care to where and how to send his data to. That should be set by your form handler. Setting the action and method attributes is done with `setAction()` and `setMethod()` respectively.

After setting the form action and method attributes the form is passed to the view. There you set the form logic specific to the view. This means labels and how the form should render. Each element can be magically accessed by means of the names you used when adding the element or through a method `getElement($name)`. And for each element a label can be set with the method `setLabel()`. Then it is time to render the form by simply echoing it.

There's more...

Processing the form in your form handlers

In your form handler you set the action and the method but the form should also be processed when it's submitted.

```
$loginForm = new Login();
$loginForm->setAction(
    $this->broker('url')->direct('form','auth')
);
$loginForm->setMethod('post');
$this->view->loginForm = $loginForm;


if ($this->getRequest()->isPost()) {
    if ($loginForm->isValid($this->getRequest()->getPost())) {
        $formValues = $loginForm->getValues();
        //do something with the form data here
    }
}
```

The important stuff for this recipe is highlighted. You know which kind of request will be made and the first thing to do is check whether it is of that type. When it is a post request the post data is given to the form to validate with the method `isValid()`. This will dispatch the data to the corresponding form elements which will do a filtering and eventually validation. When the data given by the request is a valid block of data – all elements got the information expected - the form will go through and let you access the filtered data with the help of the method `getValues()` which returns a filtered array representing the submitted data.

Set and access processed and unprocessed form data

After validating a form you need to access the validated data.
`\Zend\Form\Form::getValues().`

`\Zend\Form\Form::getValue($name)`

Sometimes `$form->getValues()` is too much information at once and you want to access only the filtered value of one element. `$name` is the element name.

`\Zend\Form\Form::getUnfilteredValues()`

When you want to access the raw input set to each element before filtering is applied.

`\Zend\Form\Form::getUnfilteredValue()`

When you want to access the raw input set to a specific element before filtering is applied.

On most elements setting a value is done with the method `setValue()`.

For elements of the type `\Zend\Form\Element\Multi` such as `MultiCheckbox`, `Select` and `Radio` have different value setters.

`setMultiOptions(array $options)`

`addMultiOptions(array $options)`

`addMultiOption($option, $value = '')`

The `$options` array follows the `addMultiOption` parameter list, so option => value pairs.

Adding protection to your forms

Filter input, escape output. Forms are all about user input and any good developer knows you have to distrust user input. `\Zend\Form` allows filtering and validating on each `Element`. Besides security another benefit of filtering user input is usability, allowing the user to input his phone number digits with the separator of his choice.

Adding validators is done at your service layer. Elements accept all the validators and Filters described in the first chapter.

How to do it...

```
//YourProject/library/YourProject/Form/Login.php

    protected function addEmail()
```

```
{
    $element = new Element\Text('email');
    $element->addValidator('EmailAddress');
    $element->setRequired(true);
    $this->addElement($element);
}

protected function addPassword()
{
    $element = new Element\Password('password');
    $element->addValidators(
        array(
            array('StringLength', false, array(4, 20)),
            'Alnum'
        )
    );
    $element->setRequired(true);
    $this->addElement($element);
}
```

How it works...

Two things have happened to the email Text Element.

First the element is now required - set with the method `setRequired($boolean)` - which means that the data array you validate against should have a key value for email.

Secondly because you added a validator EmailAddress – with the method `addValidator($validator, $breakChainOnFailure = false, $options = array())` - that value should be in a valid email format. Submitting anything else will make the Form validation fail. The `addValidator()` method accepts a fully configured validator instance or a short name. The `$breakChainOnFailure` prevents validators to run after the failing validator to be fired.

For the password field three things have happened.

It is now also required and has two validators - with the method

`addValidators(array $validators)` - added. A value in the validators array can be

a validator short name

a fully configured validator instance

an array which reflects the parameter list of the addValidator method.

This method is useful to add multiple validators at once and is equal to

```
$element->addValidator('StringLength', false, array(4, 20));

        ->addValidator('Alnum');
```

A special validator – but not used in the above example - is `NotEmpty`, which will validate when a submitted field is not empty. To check whether the field is effectively submitted - empty or not - you should use the `setRequired` option.

There's more...

Difference between the validator NotEmpty and SetRequired()

There is a difference between the validator `NotEmpty` and the method `setRequired(true)`. When an element is required it will only validate when the field is present in the data you validate against. By default elements are not required and a missing field will not invalidate your form. When you do not want a available field to be empty you need to use the validator `NotEmpty`. The require-check happens before any validator is executed.

By default `setRequired(true)` will also add a `NotEmpty` validator for your element.

Doing `$element->setRequired(true)` is the same as

```
$element->setRequired(true)->addValidator('NotEmpty')
```

This behavior can be turned of with

```
$element->setAutoInsertNotEmptyValidator(false)
```

Adding filters for easy validation and usability

The user input can be filtered and validated by putting filters and validators on the elements. First the filters will be applied and after filtering the data will be validated. All data can for instance be trimmed before being validated.

```
//YourProject/library/YourProject/Form/Login.php

       protected function addEmail()
       {
           $element = new Element\Text('email');
           $element->addFilter('StringTrim');
           $element->addValidator('EmailAddress');
           $element->setRequired(true);
```

```
            $this->addElement($element);
        }
```

`\Zend\Form\Element::addFilter($filter, $options = array())`

`$filter` can be a Filter instance or a short name. When providing a short name you can pass arguments to the `Filter` constructor with the array `$option`.

Adding multiple Filters can be done with

`\Zend\Form\Element::addFilters(array $filters)`

`$filters` is an array for which the values can be a

- filter short name
- filter instance
- array containing filter short name and options – reflecting the parameter list of the `addFilter` method.

Targeting multiple elements at once

`\Zend\Form\Form::getElements()`

Sometimes you want to be able to target all elements from a form at once. This could be to set validators, filters or decorators en masse.

The way to go about this is to request all elements from the form which will return an array and do your operation on each element.

```
protected function setupForm()
{
    $this->setName('login');
    foreach ($this->getElements() as $element) {
        $element->addFilter('StringTrim');
    }
}
```

This piece of code will set a `StringTrim` Filter on all the form elements.

Styling your forms with Decorators

Zend Forms comes with full HTML Markup by default. The only thing you have to do is `echo` the form. Sometimes however the design team has other plans on how a form should look like and it is up to you or your HTML team to generate matching markup.

How to do it...

View script for the login form

```
<?php
$this->loginForm->email->setLabel('your email');
$this->loginForm->password->setDescription('please provide a strong
password between 4 and 20 characters');
$this->loginForm->password->setLabel('your password');
$this->loginForm->submit->setLabel('login');

$this->loginForm->setElementDecorators(
    array(
        'ViewHelper',
        'Errors',
        array(
            'Description', array(
                'tag' => 'p',
                'class' => 'description'
            )
        ),
        array(
            'HtmlTag', array(
                'tag' => 'dd',
                'id'  => array('callback' =>
                         function($decorator) {
                           return $decorator->getElement()
                                    ->getId(). "-element";
                     }
                )
            )
        ),
        array('Label', array('tag' => 'dt')),
    )
);

$this->loginForm->clearDecorators();
```

```
$this->loginForm->addDecorators(
    array(
        'FormElements',
        array(
             'HtmlTag',
             array(
                 'tag' => 'dl',
                 'class' => 'zend_form'
             )
        ),
        'formDecorator'
    )
);

echo $this->loginForm;
```

How it works...

First we need to take a look at the `setElementDecorators()` method.

`setElementDecorators(array $decorators, array $elements = null, $include = true)`

`$elements` let you apply the decorators to only those elements you pick. It is an array containing the element names. We are setting – not adding – the decorators for all elements.

The `$decorators` array can accept:

decorator shortname

alias key with => decorator short name value

an array with first value the 'decorator shortname' and second value the decorator options array.

an array with the explicit key value pairs "decorator => decorator short name" and "options => the decorator options array".

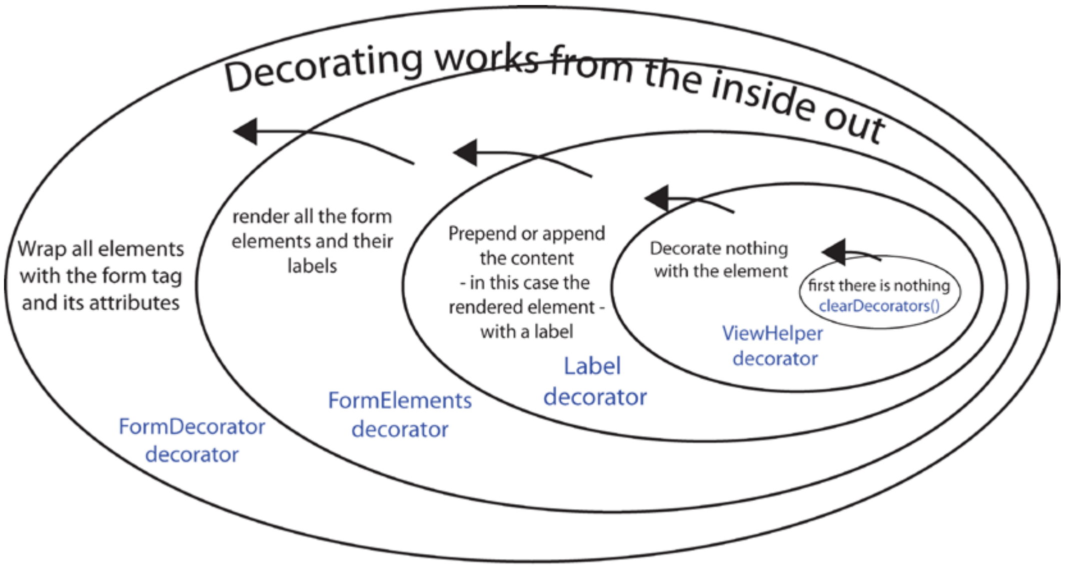
an array with the explicit key value pairs "decorator => an array with an alias => the decorator short name" and "options => the decorator options array".

an array with the first value 'an alias => decorator shortname' and the second value the decorator options array.

If you do not prefer setting all decorators through the use of an array, you can also get all elements - `getElements()` - and on each element use the `addDecorator()` method.

Now that you understand how to add decorators it is time to learn what they are.

To understand how Zend Forms are rendered you first have to understand the decorator pattern. When you are faced with a class or multiple classes that need to have functionality added you can create subclasses and implement the 'added or changed' functionality requests. Yet this often leads to class explosions or classes with too much functionality implemented in them to face the requirements. How this can be solved is by adding functionality on the fly, a bit like cherry-picking. You pick the functionality you want and do not bother with the rest.


Decorating works from the inside out
Wrap all elements with the form tag and its attributes
FormDecorator decorator
render all the form elements and their labels
FormElements decorator
Prepend or append the content - in this case the rendered element - with a label
Label decorator
Decorate nothing with the element
ViewHelper decorator
first there is nothing
clearDecorators()

1. In the next steps Element Decorators are set – this means clearing and then adding them – to all the elements with `setElementDecorators()`.

2. The `ViewHelper` Decorator is added which tells the element how to render itself.

```
<input type="text" name="email" id="email" value="" />
<input type="password" name="password" id="password" value="" />
<input type="submit" name="submit" id="submit" value="login" />
```

You do not see this markup because at this step the elements are not yet added to the form, which is outputted as a whole at the end. Echoing all three elements direclty will result in this markup.
In example `echo $this->loginForm->email->renderViewHelper()`

3. The `Errors` Decorator is added which renders the error messages of the element in an unordered list when it failed to validate. So you will only see this decorator add to the markup when there are error messages to render.

```
<input type="text" name="email" id="email" value="test" />
<ul class="errors">
    <li>
        'test' is no valid email address in the basic format local-
part@hostname
    </li>
</ul>
<input type="password" name="password" id="password" value="" />
<input type="submit" name="submit" id="submit" value="login" />
```

4. To the rendered content the `Description` Decorator is applied together with the `setDescription()` method to display descriptions on the element. Can be useful to explain why the information is needed.
5. You can pass options such as the tag to be used and additional attributes such as a class.

```
<input type="text" name="email" id="email" value="test" />
<ul class="errors">
    <li>
        'test' is no valid email address in the basic format local-
part@hostname
    </li>
</ul>
<input type="password" name="password" id="password" value="" />
<p class="description">
    please provide a strong password between 4 and 20 characters
</p>
<input type="submit" name="submit" id="submit" value="login" />
```

6. Next each element - together with the errors and descriptions - need to be wrapped with an HTML tag. For this the `HtmlTag` decorator is used, which enables you to wrap content with an HTML tag of your choice.
7. This decorator is a bit special because it can accept a callback function for the attribs passed. This is any arbitrary PHP callback and not to be confused with the callback Decorator. The callback function should accept the HtmlTag decorator as a parameter.

```
<dd id="email-element">
    <input type="text" name="email" id="email" value="test" />
    <ul class="errors">
        <li>
            'test' is no valid email address in the basic format
local-part@hostname
        </li>
    </ul>
</dd>
<dd id="password-element">
    <input type="password" name="password" id="password" value="" />
    <p class="description">
        please provide a strong password between 4 and 20 characters
    </p>
</dd>
<dd id="submit-element">
    <input type="submit" name="submit" id="submit" value="login" />
</dd>
```

8. The `Label` decorator will render the labels for the form elements. By default the markup is prepended to the content provided, in this case the rendered elements.
9. When passing a tag option the Label Decorator internally uses a HtmlTag decorator to wrap the label with that tag.

```
<dt id="email-label">
    <label for="email" class="required">your email</label>
</dt>
<dd id="email-element">
    <input type="text" name="email" id="email" value="test" />
    <ul class="errors">
        <li>
            'test' is no valid email address in the basic format
local-part@hostname
        </li>
    </ul>
</dd>
<dt id="password-label">
    <label for="password" class="required">your password</label>
</dt>
<dd id="password-element">
    <input type="password" name="password" id="password" value="" />
    <p class="description">
        please provide a strong password between 4 and 20 characters
    </p>
</dd>
<dt id="submit-label">
```

```
    <label for="submit" class="optional">login</label>
</dt>
<dd id="submit-element">
    <input type="submit" name="submit" id="submit" value="login" />
</dd>
```

10. Now that all **`elements`** are "styled" - with decorators - it is time to start styling the Form. First all default decorators need to be removed with `clearDecorators()`. This will only remove them from the Form and not from DisplayGroup(s), SubForm(s) and Element(s). Next the elements need to be added to the form with the `FormElements` decorator. This adds the rendered elements to the form. Before outputting the form you need to add the elements before you can see them.
11. Up until now if you did an echo on the form you would receive no output. After applying this decorator to the form - not the elements - the elements are added and outputted.

12. Because at step 5 an HtmlTag <Dt> and in step 6 an HtmlTag <Dd> was indirectly used the definition list needs to be finalized the <dl> tag.
13. So we use the HtmlTag again to wrap the content some more.

```
<dl class="zend_form">
    <dt id="email-label">
        <label for="email" class="required">your email</label>
    </dt>
    <dd id="email-element">
        <input type="text" name="email" id="email" value="test" />
        <ul class="errors">
            <li>
                'test' is no valid email address in the basic format 
local-part@hostname
            </li>
        </ul>
    </dd>
    <dt id="password-label">
        <label for="password" class="required">your password</label>
    </dt>
    <dd id="password-element">
        <input type="password" name="password" id="password" value="" 
/>
        <p class="description">
            please provide a strong password between 4 and 20 
characters
        </p>
    </dd>
    <dt id="submit-label">
        <label for="submit" class="optional">login</label>
    </dt>
    <dd id="submit-element">
        <input type="submit" name="submit" id="submit" value="login"
/>
    </dd>
</dl>
```

14. After this the markup is ready to be wrapped with the <form> tag together with its method, action and other attributes using the `FormDecorator` Tag.

```
<form id="login" enctype="application/x-www-form-urlencoded" action="/
auth/form" method="post">
    <dl class="zend_form">
        <dt id="email-label">
            <label for="email" class="required">your
            email</label>
        </dt>
        <dd id="email-element">
            <input type="text" name="email" id="email"
            value="test" />
            <ul class="errors">
                <li>
                    'test' is no valid email address in
                    the basic format local-part@hostname
                </li>
            </ul>
        </dd>
        <dt id="password-label">
            <label for="password" class="required">your
            password</label>
        </dt>
        <dd id="password-element">
            <input type="password" name="password" id="password"
value="" />
            <p class="description">
                please provide a strong password between 4
                and 20 characters
            </p>
        </dd>
        <dt id="submit-label">
            <label for="submit"
            class="optional">login</label>
        </dt>
```

```
            <dd id="submit-element">
                <input type="submit" name="submit" id="submit"
                value="login" />
            </dd>
        </dl>
    </form>
```

output the entire decorated form with `echo $this->loginForm`.

Of course there are other decorators to be explored. An example is `FormErrors`. Instead of displaying the error messages per element as is done with the `Errors` decorator, this decorator will allow to group all form errors to the top (prepend) or bottom (append) of the form markup.

Take a look at the other available decorators and all the options they can handle in the `ZendFrameworkInstallPath/library/Zend/Form/Decorator` folder.

There's more...

Instead of adding decorators use them directly

When you do not like the power decorators give you or you find them hard to handle you can render the elements immediately.

Each decorator can be called by using the render[DECORATORNAME].

```
<form action="<?php echo $this->loginForm->getAction(); ?>"
      method="<?php echo $this->loginForm->getMethod(); ?>" >
  <div>
    <?php
        $this->loginForm->email->getDecorator('Label')
                               ->setTag(null);
        echo $this->loginForm->email->renderLabel();
        echo $this->loginForm->email->renderViewHelper();
        echo $this->loginForm->email->renderErrors();
      ?>
  </div>
  <div>
    <?php
        $this->loginForm->password
               ->getDecorator('Label')->setTag(null);
        echo $this->loginForm->password->renderLabel();
        echo $this->loginForm->password->renderViewHelper();
```

```
            echo $this->loginForm->password->renderDescription();
            echo $this->loginForm->password->renderErrors();
          ?>
      </div>
      <div>
        <?php echo $this->loginForm->submit
                                ->renderViewHelper();
        ?>
      </div>
    </form>
```

Decorators can be easily configured by getting them from the element and using the appropriate setters or `setOption($key, $value)` and `setOptions(array $options)` methods.

Setting and getting attributes

All form components can have attributes which will be rendered in their respective HTML markup. This is done with the method `setAttrib($name, $value)`

```
$element->setAttrib('class', 'mySpecialClass');
echo $element->getAttrib('class');
```

Most of the time these attributes are used by the decorators applied to the element. When you want to set an attribute to the decorator directly because it must not impact the element itself, you can try the `setOption()` or `setOption`s() method.

```
$element->getDecorator('Label')
        ->setOption('class', 'yourCoolClass');
echo $element->renderLabel();
```

Adding more than one of the same decorators to an object

You can add as many or as few decorators as you want to your `Element`, `Form`, `SubForm` or `DisplayGroup`, but you will face a problem when you want to add two or more of the same type to the same object.

Internally decorators are added to the decorator stack with their class names as array keys. This means that the second decorator will overwrite the first one. To get around this you need to specify your own decorator name alias which will then be used as the key.

In **`How It Works`** you saw the array configuration possible for setting decorators. Some of those configurations involved using an alias key.

```
setDecorators(
    array(
            array(
                    array('divDecorator' => 'HtmlTag'),
                    array('tag' => 'div')
            ),
            'array(
                    array('mainDivDecorator' => 'HtmlTag'),
                    array('tag' =>'div', 'class' => 'main')
            ),
    )
)
```

The aliasses 'divDecorator' and 'mainDivDecorator' are used to avoid naming conflicts for two or more of the same decorators used for the same element, group, sub form or form.

About display groups and sub forms

Forms are used to gather user information. The more structured the form is displayed on the screen the more it will appeal to the user. For grouping there are two solutions, display groups and sub forms.

You use display groups to aid yourself in displaying grouped elements.

By grouping you gain some benefits, all decorators added will not be applied to the rendered content of each element separately but all elements in a display group are rendered together, by default the fieldset decorator is used for this. Another benefit is you can select a group – a set of elements you grouped – and also add decorators at element level . This is an alternative approach to the `setDecorators $elements` parameter.

How to do it...

Grouping elements in a display group. This should be done in the view: it is an aid for displaying.

```
$this->loginForm->addDisplayGroup(
    array('email', 'password'),
    'loginGroup'
);
$this->loginForm->loginGroup->removeDecorator('Fieldset');
$this->loginForm->getDisplayGroup('loginGroup')
```

```
        ->addDecorator('HtmlTag', array('tag' => 'div'));
```

How it works...

You add a `DisplayGroup` to a `Form` or `SubForm` by using the method `addDisplayGroup(array $elements, $name, $options = null)` which accepts the array of element names as the first parameter. The second is reserved for the name of the group, which is used to retrieve the group through the magic name or the getter `getDisplayGroup($name)`.

There's more...

Sub forms to request more information on the fly.

`SubForm` is a bit more than a displaying and selection tool. It allows for creating wizard like forms, and only when all steps of the form are completed will the form validate. You will rarely see forms on multiple screens, but you will see subforms getting requested by ajax and injected in the currently displayed form. Actually `Zend\Form\SubForm` extends `Form` and overrides the default decorators to not use the `formDecorator` so it is nothing more than a form without the default <form> tag markup added.

The benefit of using a Form inside a Form with `SubForm` instead of hiding elements with CSS is that you can have `isRequired` and `NotEmpty` configurations on your sub form elements. Something you cannot have when using display groups. When you hide display group to the user he cannot set the required information and the form would invalidate. Removing the `isRequired` or `NotEmpty` validator to make the display group not invalidate the form is not an option. `DisplayGroup` is for displaying only, not for controlling the user input. You need to use `SubForm`.

Two common approaches are as follows:

Initialize a custom `SubForm` and then add it to the main `Form` when needed with `addSubForm(Form $form, $name, $order = null).`
The main form is stored inside a session to maintain state. AJAX is a new request so state between the initial request and the AJAX request needs to stay maintained.

Retrieve the sub form with AJAX, inject it in the current form markup as you would with the previous approach and when the form is submitted the logic checks whether sub form data is submitted with querying the Request object directly. If it is submitted the `SubForm` is added to the `Form` prior to validation. This way no sessions are needed.

Summary

Mastering Zend Form with its Decorators can look overwhelming at first, but it is exiting and fun. You will start to love the flexibility and the speed in which you can create and handle forms.

Zend Form provides an object oriented interface to building forms.

It promotes Do-Not-Repeat-Yourself (DRY) by extending implementations

integrates input validation and filtering

supports `SubForm` which is used to create logical element groups. Creating multi-page forms. For instance Wizards. Only once all added sub forms validate would the form be considered valid. Sub forms should be created and manipulated in the Service Layer.

Supports `DisplayGroup` which is a way to create visual groupings of elements. All elements in a display group are rendered together. The most common use case for this is for grouping elements in fieldsets. Display groups should be created and manipulated in the View.

The foundation of Form is Element which is the smallest part of the form. This represents an object representation of a regular HTML form element. Instances are useful to bind filters (`\Zend\Filter\Filter`) to the input , bind validators (`\Zend\Validator\Validator`) to the input and bind Decorators. Elements should be created and manipulated in the Service Layer, but decorators should be added and used in the view together with the setting of metadata like class, id and so on.

Remember that decorators handle the markup rendering of the form and implements the Decorator Design Pattern, thus the order in which you append them is important. First In First Out (FIFO) is the approach.

`Form`, `SubForm`, `DisplayGroup` and `Element` all rely on Decorators for rendering so decorators can be added, set and removed on all four. At a minimum you will need to use the `ViewHelper` decorator on your elements.

The most important thing that must be remembered is that everything has its place. Adding elements, filters and validators happen in the form class extending `\Zend\Form\Form`, adding an `action` and `method` in the form handler and manipulating the markup happens in the presentation layer, namely inside a view script.