php|architect's

Guide to Programming with ZEND FRAMEWORK

Cal Evans

nb™ php|architect nanobooks

php|architect's Guide to Programming with Zend Framework

by Cal Evans

php|architect's Guide to Programming with Zend Framework

First Edition: January 2008
ISBN: **978-0-9738621-5-7**
Produced in Canada
Printed in the United States

Disclaimer

Although every effort has been made in the preparation of this book to ensure the accuracy of the information contained therein, this book is provided "as-is" and the publisher, the author(s), their distributors and retailers, as well as all affiliated, related or subsidiary parties take no responsibility for any inaccuracy and any and all damages caused, either directly or indirectly, by the use of such information. We have endeavoured to properly provide trademark information on all companies and products mentioned in the book by the appropriate use of capitals. However, we cannot guarantee the accuracy of such information.

Written by	Cal Evans
Published by	Marco Tabini & Associates, Inc. 28 Bombay Ave. Toronto, ON M3H 1B7 Canada (416) 630-6202 / (877) 630-6202 info@phparch.com / www.phparch.com
Publisher	Marco Tabini
Technical Reviewer	Matthew Weier O'Phinney
Layout and Design	Arbi Arzoumani
Managing Editor	Elizabeth Naramore
Finance and Resource Management	Emanuela Corso

Dedications

I would like to dedicate this book to the following people, without whom, it would not have happened:

- *To my mother—for instilling in my my love of writing.*
- *To my wife, the lovely and talented Kathy—who I love dearly because she puts up with me.*
- *To my kids, Becky and J.C.—who I love an adore, even when I'm ignoring them to write.*
- *To Marco, Elizabeth, Paul and Sean—for friendship.*
- *To Mark de Visser—for the greatest job I've ever had.*
- *To Matthew Weier O'Phinney—for being nice when you could have been mean.*
- *To Mr. Jimmy Buffet—You don't know me but I could not have finished this book without your music. My Jimmy Buffet playlist is my bank of bad habits.*

Contents

Foreword

I was delighted to be asked to provide a foreword for Cal's book. He's often quoted me as saying "All frameworks suck". Given that context, you might be surprised that I believe this book should be a good read.

I have long been a critic of PHP frameworks for several reasons, one of which is the plethora of frameworks available. The chief advantage of using a framework is maintainability - frameworks provide a system and method for organizing code, but each framework is different, and if developers have to start from scratch with each new one then the advantage is lost.

Having an official framework in the form of the Zend Framework means there is one framework likely to be well known and understood by a large number of developers. With the support available from Zend, developers ought to be able to get up to speed easily. The engineers who have contributed to this framework have a deep understanding of the nuances of PHP and how to avoid the performance traps inherent in building layers between PHP and your application. In other words, Zend Framework provides a great toolset for developers. As well as the MVC classes, the Zend Framework provides a set of really useful utility classes that can be integrated into any PHP application, regardless of how that application is architected. As such, there's something to keep everyone happy.

Cal has been preaching the Zend Framework at PHP conferences and in the DevZone for some time now, and I am very pleased that his deep knowledge of and enthusiasm for the subject has now been captured in a more formal format.

Laura Thomson
January 2008

Chapter 1

What makes the Frame-work

"All Frameworks suck." - *Laura Thomson, 2007*

Why Use a Framework?

In almost all the non-trivial projects I have built, I have had to make the decision about whether I should use an existing framework or write my own. You may not think that those are the only two options. You may think that your project doesn't need a framework for whatever reason. However, in most non-trivial applications, you will end up building something that resembles a framework. Most of the time it starts out by combining similar code from different areas of the project to simplify maintenance. Before you know it you have a database abstraction layer, base classes, abstract classes, and eventually, you've got yourself a framework. So in reality, it really does boil down to just those two choices. When you look at it in that light and assuming your project is non-trivial, the "Why" becomes apparent. You use a pre-existing framework to save you the time and hassle of having write one yourself.

Which Framework is Right for Me?

In PHP we have a plethora of frameworks to choose from. I heard it described once as "two frameworks get together within the confines of SourceForge and out comes three more." Selecting the correct framework could take weeks, even months if you

were to carefully analyze each one. Thankfully, if you have purchased this book, I am guessing you already know that you do not want to build your own and that you think Zend Framework is the right one for your project.

To answer the question fully though, we really have to examine why we use a framework. Aside from the "two choices" scenario described above, programmers use a framework to simplify application development by providing much of the common code. A good framework will also enforce some structure on the code. However, be wary of any framework that enforces rigid style conventions that you do not already adhere to. Your chosen framework should be flexible enough to adapt to the style and needs of your project.

The ultimate goal when selecting a framework, is to find one that allows you to work in a way that is natural to you, provides you the services that are most common to your application and allows you to concentrate on building the code business logic of your application.

To be fair though, the right framework for you is the one that lets you be productive quickest and leverage your new skills the longest.

Why Zend Framework

The Zend Framework is really a hybrid framework and as such can be used in a much larger range of projects than strict "application frameworks". While many components in Zend Framework can be used stand-alone like a component library; it is, at its core an implementation of the "Model-View-Controller" (MVC) pattern.

MVC in a Nutshell

MVC, like so many great things in computers, came out of Xerox's PARC in 1978-1979. These days MVC is a common pattern for frameworks to implement because it separates the code into three logical groups.

The model can be thought of as the representation of the data that your application will utilize. In simple terms, the model can be thought of as the "nouns" of your project. An "order", a "member", an "article"; these are all examples of potential models in your system.

The view contains all the display logic. In the majority of PHP applications, this means the HTML output of your application. However, as we will discuss later in the book, even in web applications, this can mean a variety of output formats. Whatever the format, the view is responsible for the merging of the data from the model and the actions of the controller and sending it to the proper client. (In most cases with PHP, that's a web browser).

The controller is responsible for the domain logic in your applications. It represents the verbs or events. "Add," "edit" and "submit" are all actions your application can take. The controller embodies these actions for you.

There are many good examples of MVC-implemented frameworks in PHP. If that were its only selling point then Zend Framework would be just another framework in an ever-growing list. Zend Framework however, separates itself by allowing you to pull pieces of it out and use them independently. The Zend Framework teams calls this "use at will" architecture. Most of the components that are not part of the MVC core can be pulled out and used as "standalone" components in your application. Examples of the "use at will" components are

- Zend_Cache
- Zend Rest_Client
- Zend_Feed
- Zend_Log

Each of these can be used independently of the framework itself and that means their functionality can be easily incorporated into existing applications. Simply put, if all you need is a single component, you can use just that component. However when the job requires a full framework, you have that option also. This goes back to answering the "Which" question previously mentioned above. The more projects your Zend Framework skills are useful in, the more they are worth to you.

It should be noted here though that MVC is not something you can retrofit into an existing application. If you are maintaining existing code, the component library aspect of Zend Framework will be of much more interest to you because you can easily integrate the pieces you need without disturbing your existing legacy code. However, if you are building in "green fields" then the MVC aspect of Zend Framework will be of more interest because you have the luxury of building from scratch.

Introducing Zend Framework

Zend Framework was designed and built to improve developer productivity. Unlike other frameworks that require large configuration files to work, most aspects of a Zend Framework application can be defined at runtime using simple PHP commands. This saves developers time because instead of complex configuration files controlling every aspect of the application, you only configure the parts that deviate from the norm.

Zend Framework was written entirely in PHP 5. It will not run on any server that does not have a minimum of PHP 5.1.4 installed. The current version has been thoroughly tested and over 80% of the code is covered by test cases using PHPUnit.

Zend Framework was built on several key concepts:

- Best Practices
- Community Driven
- Extensionability
- Extreme Simplicity
- Liberal BSD License

Unlike many other frameworks available for PHP, Zend Framework chose not to implement the ActiveRecord pattern and not to ship with an Object-Relation Mapper (ORM). Contrary to popular opinion, this was not an oversight but a conscious decision by the framework team.

A Brief History of Zend Framework

Coding on Zend Framework officially started in July of 2005. It was announced to the general public in the same year at the first ZendCon as one part of Zend's PHP Collaboration Project. (The other two parts of the initiative are Zend's Developer Zone and an Eclipse based IDE for PHP.) The first public release was on March 4, 2006, version 0.1.2. More than a year later, the first 1.0 version was released on July 2, 2007.

The Zend Framework Community

Possibly the greatest asset that Zend Framework has is its community. The community around Zend Framework is growing daily. While several of the developers working on Zend Framework actually work for Zend, the majority of them do not. The process of proposing and reviewing new components is open and community driven. Because the community is comprised of both beginner and advanced programmers, there is never any shortage of help for new adopters. Whether you prefer mailing lists, forums or chat, Zend Framework community is always there and willing to help. The community realized early on that with any framework, getting started is the hardest part. Therefore, there are numerous tutorials and quick-start guides to help both novice and advanced users get up and running. The project Web site (`http://framework.zend.com`) houses not only the documentation and downloads for the project but a full bug-tracking/ticketing system that allows anyone to register and submit bugs. This level of openness helps them meet the first goal of the project, "Community Driven".

The Zend Framework License and Intellectual Property Concerns

All contributors to Zend Framework sign a "Contributors License Agreement" stating that the code they are contributing is IP clean. The practice and agreement is similar in nature to that required by the Apache group. This was done not as an exclusionary practice but to give peace of mind to companies considering adopting Zend Framework to build commercial applications. To facilitate its adoption by both open source projects as well as commercial entities, Zend Framework was released under a BSD style license. This allows for the framework to be used in the widest possible range of projects and puts the fewest restrictions on adopters. A complete copy of the BSD License can be found on Zend Framework Web site (`http://framework.zend.com/license`).

What You Need To Go From Here

Learning any framework is a daunting task. In this book, my goal is to get you up and over the learning curve so you can be productive faster. There are a few things

you need to make things easier. First, you need a good grasp of Object Oriented Programming in PHP 5. Zend Framework is all about Object Oriented Programming. If you don't understand PHP 5's object model, I recommend you stop reading, visit the PHP manual and check out the Object Oriented section. You can get a lot of what you need there (`http://php.net/manual/en/language.oop5.php`) but there are other sections you need to read as well. You will have to know the difference between a public and a protected property as well as what a static class/method is and when you should use it. In the rest of this book I'll assume a working knowledge of Object Oriented Programming. If you don't understand it, you will be lost. Second, you will need a working development environment. If you don't already have a development environment, you need to stop now and go download one of the following two packages:

- Zend Core (`http://www.zend.com/products/zend_core`). Zend Core is a free product that comes pre-compiled for your OS. It is an all-in-one installer for PHP, Apache and MySQL. If you are installing Zend Core for Windows, it will work with IIS if you tell it to do so or optionally install Apache for you.
- XAMPP (`http://www.apachefriends.org/en/xampp.html`). XAMPP by ApacheFriends is a great all-in-one package. With a single installer you get the latest versions of Apache, PHP, Perl, MySQL and several support libraries and tools. In may cases, this is going to be overkill but you do have the option of turning services off (like the FTP Service) when you don't need them.

Both of the packages described above will give you a solid working environment for you to customize to your liking. Before continuing with the book however, you need to get everything setup and running just the way you like it. Take a day or so if necessary but don't skimp on this section. It is important that your environment work for you and that you know how to modify it if necessary. When you can write PHP code and have it execute properly, then come back and we will continue. I promise I will wait.

The examples in this book will assume a few things. You will want to check your `php.ini` file to make sure this setting is set properly or the examples may not work.

```
allow_url_fopen = true
```

Lastly, you need an *Integrated Development Environment* (IDE) that you are already familiar with. It does not matter if you use Zend Studio or Notepad, you need to have an editor that you are comfortable working with. We are going to be covering a lot of new ground for most readers and the last thing you want to do is learn new tools and new techniques at the same time.

Summary

In this chapter we've covered some key points about Zend Framework:

- You use a framework to reduce the time and effort it takes to get a project completed.
- You use Zend Framework because you think you can use it in the widest range of projects you are building or are going to build.
- Zend Framework has a supportive community to help you when you get stuck.
- You need to have a firm grasp of object oriented programming in PHP 5 before you try and start working with Zend Framework.
- Zend Framework is not a miracle tool, it is a power tool. Like any tool, understanding when not to use it is as important as understanding when and how to use it.

Chapter 2

Getting Started

Building Your First App

Enough words, let's actually do something. Here are the steps necessary to build a "HelloWorld." At this point, I'm going to assume that you heeded my warning about a working development environment. If you didn't, run (don't walk) back to Chapter 1 and re-read that section again.

- Download a copy of Zend Framework
- Create your directory structure
- Create your bootstrap file
- Create your `.htaccess` file (Apache)
- Create your controller
- Create your view script helper. (optional)
- Fire up a browser and revel in your handiwork

That's all there is to it. Since some of those steps may need a little explanation, I'll give you a few more details.

Step 1: Download a copy of Zend Framework

There are 3 ways you can do this; the easy way, the hard way and then the easiest way. The easy way is to go to `http://framework.zend.com/download` and grab the latest `zip` or `tar`. Quick, simple, easy and only for those who do not ride roller coasters. If, however, you live on the edge, grab the latest version out of svn (svn checkout `http://framework.zend.com/svn/framework/trunk`). Finally, the easiest way is to unpack `example1.zip`; I've included a full copy of Zend Framework in each of the example files. However you get it, you end up with a lot of files in a directory structure that looks something like Figure 2.1.

Figure 2.1

Zend Framework is broken up into 2 major parts, the *library* and the *incubator*. The `library` directory contains the officially released code that has been vetted and blessed. The `incubator` is for code that has passed the Proposals procedure but is not yet mature enough to be included in the main `library`. The `library` directory contains the main Zend Framework. If you do not see the `incubator` directory, have no fear, it means you took the easy way out and downloaded the `zip` file. We won't be using any code in the incubator in this book.

In production environments, it is important that `library` be in your `include_path`. If that sentence didn't make sense to you, don't worry for now. For now, just put it in your web server's root.

Step 2: Create Your Directory Structure

Zend Framework can be configured to meet just about any directory configuration. The default configuration that I am describing here is the recommended setup. Do it this way and you will have fewer problems.

A note here about directories: I develop on a Windows based laptop while staging and production are on Linux. Later on, we will discuss ways to minimize the pain of moving between environments. For most of the examples in this book however, I will assume a Windows environment. You Linux and OSX users out there are probably savvy enough to figure out that when I say `\` I mean `/` and so forth.

My development environment is contained entirely in `c:\web`. This is considered my *web server root*. Beneath it I have a directory for each project I am working on, as well as directory for common code like {{include PEAR}}PEAR and Zend Framework. So, for this project my `c:\web` looks like this:

```
C:\web\         <-- Web Server Root
    htdocs\     <-- web application root
      www\      <-- web site root
```

Now, below the `htdocs` directory we need to add a space for our application. Zend Framework is designed so that only the bare minimum goes into your `www` directory. This is a safety feature since the web server can only see and serve files in `www`. Anything outside of that, it can't serve, therefore it is more difficult to leak sensitive information if it is stored outside of `www`.

To flesh out our directory structure, let's add the needed directories for Zend Framework as well as one for housekeeping.

```
C:\web\             <-- Web Server Root
    htdocs\          <-- web application root
```

```
lib\           <-- All 3rd party libraries
  Zend\        <-- Where the framework lives
application\   <-- Zend Framework based application
  controllers\
  models\
  views\
    scripts\
      index\   <--One dir for each controller
www\           <-- web root
   images\     <--the rounded corners for our apps.
```

Now our directory structure has been fleshed out.

Step 3: Create Your Bootstrap File

When working with Zend Framework, you will often hear people refer to the "bootstrap" file. This is mainly because programmers like to make things sound more complicated than they actually are. (I am pretty sure that it's a Guild law.) For the purposes of our discussion, "bootstrap file" means `index.php`. It's that simple. It's the file your web server calls when a request is made to your application. Your bootstrap file is always located in the webroot of your application.

Your bootstrap file can be very simple or range into the complex. For our "HelloWorld" we just need a simple one. Let's start our `index.php` with the following lines of code:

```
<?php

$lib_paths = array();

$lib_paths[] = "c:/web/htdocs/application";
$lib_paths[] = "c:/web/htdocs/lib";

$inc_path = implode(PATH_SEPARATOR, $lib_paths);

set_include_path($inc_path);
```

This first part may seem strange to beginners and even stranger to advanced programmers. Here we are manually setting our "include path". Before you scream "scalability!" or "portability!", keep in mind that this is merely sample code. It's also

okay to do this in a development environment where you may not have total control over your server. Do not, under penalty of extreme slowness, do this on a production server. Your production environment should be setup by your admin and you should already have your `include_path` setup properly.

Let's look at what that code is actually doing. Our "HelloWorld" app needs 3 directories in the `include_path` so that PHP can find everything.

- `c:/web/htdocs/application`. This is where the application actually lives. It is very important that PHP can find all the pieces of the app. Mess this one up and it's a short trip to the fatal error.

- `c:/web/application/lib`. This is the home of all libraries not actually part of the application. Among other things, this is where the framework itself actually resides.

Beyond that, we simply take the array we just built, scrunch it up into one long string and shove a `PATH_SEPARATOR` between each one. Then we use this "FrankenString" we've created as PHP's new "include path".

As I said before, this works fine in a development environment. Heck, it works okay in production if you don't have more than 5 people at a time hitting your site. However, if you deploy this code to a busy site, you will feel the pain.

Let's continue our bootstrap file by adding the following code:

```
require_once 'Zend/Loader.php';
require_once 'Zend/Controller/Front.php';
```

Here we pull in the basic classes we know we will need to complete the bootstrap. The bigger your application, the longer this list will be. However, "HelloWorld" only needs the bare minimum.

I know that nobody ever writes code with errors in it. I'm probably the only person who has ever seen a fatal error in a production application. However, Zend Framework is loaded with exceptions that can be thrown to signal a variety of conditions. In development, we want those errors to show up so we can fix them. In production, we want to do something different. Either way, one of the worst things that can happen is for an uncaught exception to show in our user's web browser. So, we wrap

everything form here on out in a `try{}`. If nothing else catches the exception, this will and we can deal with it as we see fit. Add the following code to your bootstrap file:

```
try
{
```

The *front controller* in Zend Framework is what takes care of everything. It sets up the environment, figures out which controller to call and routes it. Finally it handles routing the output when everything is done. The front controller follows the *singleton* design pattern, meaning that there is and can be, only one instance of it. Therefore in our bootstrap, we don't instantiate it like a normal class, we make a call to its (static) `getInstance()` method. This will use the instance of FrontController already in place. Therefore, we add the following lines to our file:

```
Zend_Loader::loadClass('Zend_Controller_Front');
$frontController = Zend_Controller_Front::getInstance();
```

Since Zend Framework works off of configuration instead of convention, here in the bootstrap we do have to set a few parameters for things to work right. The first one tells the FrontController to pass through any uncaught exceptions. This is fine for now, because we are in development but you would almost never want this option set in production. Later on you will see that we parameterize this and put the setting in a config file. This allows us to have different settings for development, staging, and production.

The second parameter tells Zend Framework not to try and find a view script. Our example is a very simple "HelloWord," and all of our code will be contained in the `IndexController.php` file.

The third parameters tell the FrontController not to try and find an ErrorController when it hits a problem. In a larger application, this could be used to log the error, politely notify the user that an error has occurred, and/or perform any number of housekeeping tasks. However, in our "HelloWorld," we don't need any of that so in this case, we just turn it all off.

The fourth parameter tells the front controller where to find the other controllers. Again, in larger applications, this would not be hard-coded but be parameterized for portability.

Add the following code to your file:

```
$frontController->throwExceptions(true);
$frontController->setParam('noViewRendered', true);
$frontController ->setParam('noErrorHandler', true);
$frontController->setControllerDirectory('c:/web/htdocs/app/controllers/');
```

That's it. Now we throw the switch and watch the magic.

```
$frontController->dispatch();
```

Should there be an uncaught exception, the following code will print the stack trace to the output device so we can hopefully find the error and correct it.

```
} catch (Exception $exp) {
  $contentType = 'text/html';

  header("Content-Type: $contentType; charset=utf-8");
  echo 'an unexpected error occurred.';
  echo '<h2>Unexpected Exception: ' . $exp->getMessage() . '</h2><br /><pre>';
  echo $exp->getTraceAsString();
}

?>
```

That's one of the simplest bootstrap files you can have. I know I spent a lot of time on this, but going forward we will use a variation of this in every example in the book. In future projects, I'll only show the pieces that change. You can, however, refer back to this one for the complete version.

Step 4: Create Your .htaccess File

The Zend Framework can work without `mod_rewrite` in Apache but it requires the manual configuration of the routes. For the purposes of this book, we will assume

you are using Apache (see Chapter 1 for two packages that will install it for you if you like).

In the `www` directory, you will need a file named `.htaccess`. In it, place these two lines:

```
RewriteEngine on

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

The first one, of course, tells Apache to turn on `mod_rewrite` for this directory and all of it's subdirectories. The second line tells `mod_rewrite` that unless the requested URI is a resource (image, stylesheet, javascript, etc) to route it `index.php`. The FrontController will take it from there.

Step 5: Create Your Controller

I hear you, FINALLY, some code. Well, yes and no. Zend Framework does such a great job of taking care of things for us that there's really very little left to do at this point. This file we are about to look at is `IndexController.php`. It should be saved in `app/controllers/`.

We can always assume that `Zend_Loader` is available because we included it in the bootstrap. In this case, there is no advantage or disadvantage to using it so we will use it. In some cases, mainly when the file name is a variable, the `Zend_Loader` is much faster at fetching the files.

```
<?php

  Zend_Loader::loadClass('Zend_Controller_Action');
  class IndexController extends Zend_Controller_Action
  {
```

All actions in a controller have to be named `*Action`. If you want your URL to be `http://yourap.example.com/show/users`, `show` is the name of your controller and `usersAction()` would be the method called inside of it. This is not to say that all methods inside a controller have to end with the word "Action". There are many

cases where a controller needs helper functions. In most cases, these are protected methods that are only used within the controller.

One more note on action names: most actions are a single word like in the example above, however, as we will see in later chapters, sometimes it takes multiple words to express the action, as in `processRegistrationAction()`. In Zend Framework, you have to name your methods in camel case, as noted above. However, when calling the action form a URL, you separate words with a dash. (-) So the call to the action above would be `http://www.example.org/member/process-registration`.

In our example, `indexAction` is the method that is being called. As you can see it gives us a big old "Hello-World" smile when things work correctly. Add these lines to your `IndexController.php` file:

```
    public function indexAction()
    {
      echo "<center><h1>HelloWorld</h1></center>";
    }
}

?>
```

In an effort to keep this simple, I am showing you a special controller and action. The `IndexController` and `indexAction()` are called if you do not specify a controller and action on the calling URL. The same results can be seen if you type `http://www.example.com/index/index` into the browser URL.

Step 6: Fire Up a Browser and Revel In Your Handiwork

Go ahead, you've earned it. Fire it up and point it to your development web server. If everything is correct, you will see something that looks like Figure 2.2.

Summary

You've tasted code and I know you are looking for more. In this chapter my goal was to show you that while there are several steps to setting up a Zend Framework app, they are all pretty easy and going from 0 to code can be done in a very short time.

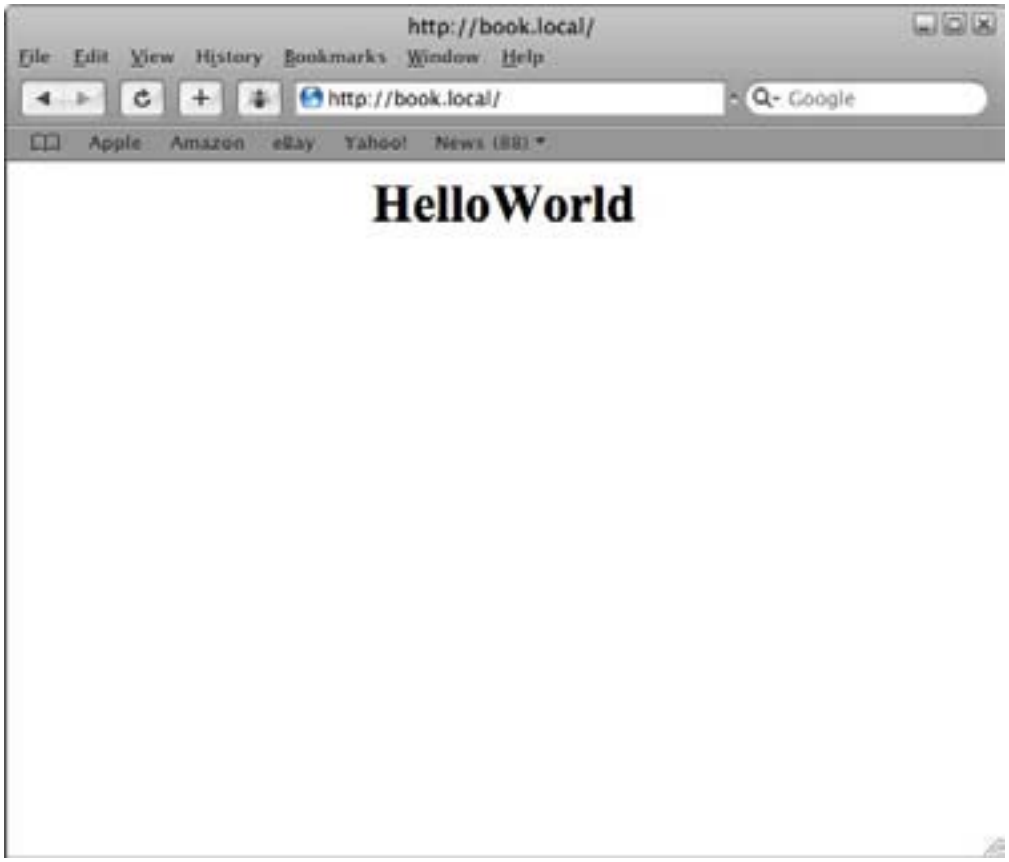

Figure 2.2

By the end of the book, you should not only understand all the steps we've done but you'll be able to do them in your sleep.

Chapter 3

The Controller

As we discussed in *Chapter 1* the purpose of the *controller* is to contain the domain specific code for your application. I gave the example that controllers are the verbs in your application. The verb analogy really applies to the actions that a controller contains. The analogy starts to break down quickly if you over-think it, so don't. It's just an overview to help you understand what code you can expect to see in a controller. As we will discuss in the next chapter, there are applications that are simple enough not to need a model. In these cases, all of the code goes into either the controller or the view. To demonstrate the functionality of a controller, let's build a simple application that only contains a controller and two actions.

Laying the Groundwork for a Sample Application

If you followed the instructions in the last chapter then you already have most of what we need to make this work. However, we do have three changes to make.

First, open your bootstrap file (`index.php`) and remove the line:

```
$frontController->setParam('noViewRendered', true);
```

We will be using *view scripts* to handle the output in this sample.

Second, open up `IndexController.php` and remove the following line:

```
echo "<center><h1>HelloWorld</h1></center>";
```

Save the file. This gives us a clean `IndexController` to work with.

If you don't want to do all of this, you can also download and unpack `example2.zip` which you can get from the *php|architect* Web site (`http://www.phparch.com`).

Finally, we need to make a place for our view scripts. Since this chapter is on the controller, just follow the instructions for now. I'll explain it all in *Chapter 5*.

- Navigate to the directory `app/views`
- Create a directory named `scripts`
- Enter the `scripts` directory
- Create a directory named `index`
- Enter the `index` directory
- Create a file named `index.phtml`
- Create a file named `extract.phtml`

There, now we are all setup and ready to go.

Creating a Sample Application

This example code is purposefully simple for two reasons. First, if your code gets too complex in your controller, then you really need to consider refactoring and moving the bulk of the code into one or more models. A good rule of thumb is that if your controller contains more than seven actions or more than a couple hundred lines of code, it's time to start refactoring.

Controllers should be limited to that code that is necessary to manipulate the business objects and hand-off the results to the view for display processing. Second, it is simple because I want to show you the base necessities for making a controller work without getting bogged down in the details.

In this example we will build a simple front-end for *Yahoo!'s Term Extraction API* (one of my personal favorite APIs). The example will display a page and allow the user to enter a URL. Upon form submission it will do the following:

- Open the page
- Extract the body content
- Hand it off to Yahoo!'s Term Extraction API for analysis
- Hand the results to the view for display

The end result is that you can enter a URL and it tells you what Yahoo! thinks are the keywords for that page.

Before you get started, you may want to check the docs for the API (`http://developer.yahoo.com/search/content/V1/termExtraction.html`) and get an *appid* from Yahoo! (`http://developer.yahoo.com/wsregapp/`) You will, of course, need an account on Yahoo! before proceeding. Both the account and the appid are free but necessary. Note, the Yahoo! information used in the sample does not correspond to a real account. If you try and use it, things will not go pleasantly for you.

First, we need to setup the form that the user uses to submit the URL. As we discussed in the previous chapter, `IndexController` and `IndexAction()` are the default controller and action in any Zend Framework application. So let's use those to display our form.

Open the file `app/view/scripts/index/index.phtml` that we just created. In it place the following:

```
<?xml version="1.0" encoding="UTF-8"?>
 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
     xhtml1/DTD/xhtml1-strict.dtd">
 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
 <head>
   <title>Keyword Content Analyzer</title>
 </head>
 <body onload="document.mainform.url.focus();">
   <form method="POST" action="/index/extract" name="mainform">
     URL To Analyze:<br />
     <input name="url" id="url" value="http://devzone.zend.com" onFocus="this.
         select();"><br />
     <input type="Submit" name="submit" id="submit" value="Analyze" >
   </form>
   <!-- Insert your content here -->
 </body>
```

```
</html>
```

Nothing really complex, a simple form with an input box and a submit button. The one thing to note is the action of the form. By now you should be able to recognize that this will call the `extractAction()` method of `IndexController`.

Now open your `IndexController.php` file . We need to populate `indexAction()` with the code necessary to display that form. Here is the `indexAction()` method you need.

```
public function indexAction()
 {

 }// public function indexAction()
```

That's right, nothing is needed. Zend Framework's implementation of the controller by default will look for a view script and display it if it's available. Above we removed the line setting `noViewRenderer` to `true`. In the example in the last chapter, we took care of the output in the controller. Normally, that is not the way it should be done and we did that there so we could concentrate on other things, like getting the directory structure set up. At this point, however, we want to do it the way it was designed and allow the view to handle the output. Since it does that automatically, we actually do less.

If everything worked properly up to this point, you will see something that looks like Figure 3.1:

As we discussed, when the form is submitted, it submits to `IndexController::extractAction()`. That's really where the bulk of our code for this example is. Let's take a look at that code now; open up your `IndexController.php` file and add the following lines:

```
public function extractAction()
 {
   /*
    * get the URL passed in from the form
    */
   $url = Zend_Filter::get($this->getRequest()->getPost('url'), 'StripTags');
```

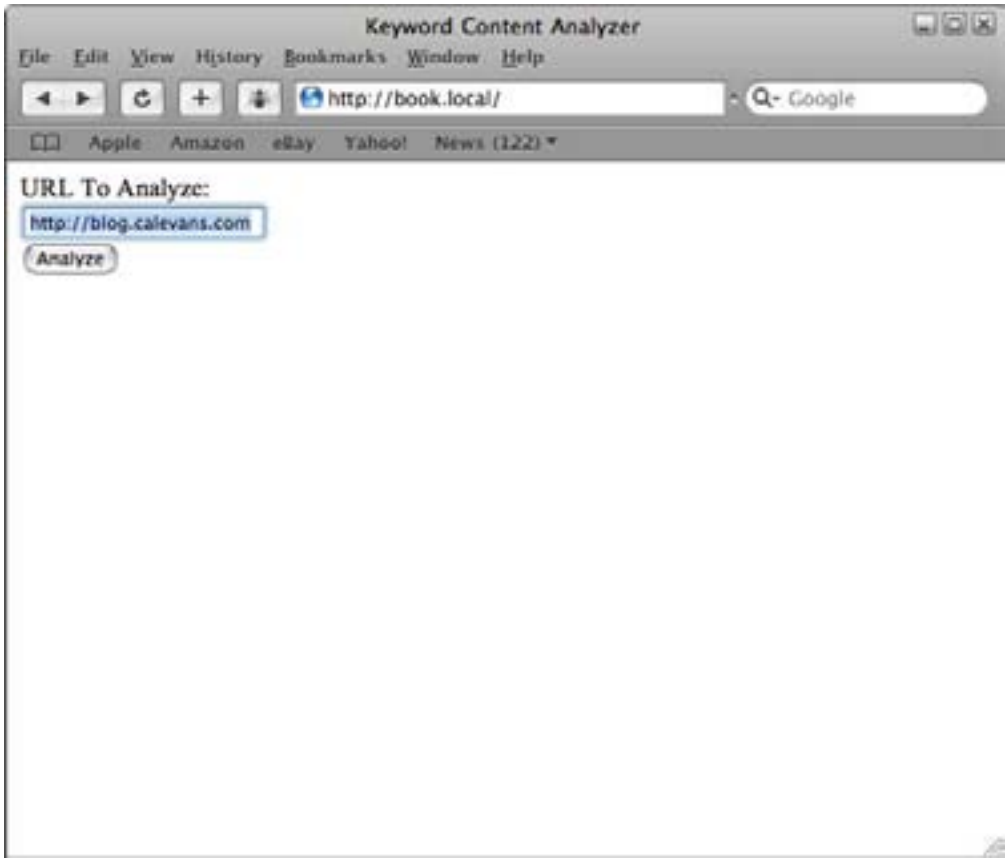

Figure 3.1

First, we get the url that the user wants us to parse. You will note that even in this sample application, we are filtering any and all user input. In this case, a simple `StripTags` keeps most of the script-kiddies out. If this were production code, I would most likely pass it through a more stringent filter chain. The `Zend_Validate` class makes filtering your input easy enough so that not doing so should be considered at the very least a serious social faux pas. In most cases I would consider failing to properly filter all input a career limiting move.

I'll mention this only in passing, you may not be familiar with the style of coding represented in this line:

```
$this->getRequest()->getPost('url')
```

This is called *fluent interfaces*. The idea is that you can reduce the amount of code necessary to work with objects. In the above example, each method returns a reference to the object. Therefore you can "daisychain" the methods together. This is the equivalent of:

```
$obj = $this->getRequest();
$obj->getPost('url');
```

Another tangible benefit for developers of using fluent interfaces is that you can stack method calls like this:

```
$url = Zend_Filter::get($this->getRequest()
               ->getPost('url'), 'StripTags');
```

While this provides no performance gains, it does make your code more readable and easier to maintain.

Zend Framework makes extensive use of fluent interfaces and you will get used to seeing them, using them, and even writing them before we are finished.

Let's add a few more lines:

```
/*
 * read page into memory
 * requires allow_url_fopen to be true
 */
$page = file_get_contents($url);
```

A quick note about this line. There are many ways to fetch the contents of a URL in PHP. This just happens to be one of the shortest ways to do so. You could use `fopen()`, `file()` or `curl` to achieve the same results with more code if you like.

Adding a few more lines:

```
/*
  * strip out everything but the content
  * Many thanks to #phpc members ds3 and SlashLife for the RegEx
  */
 $matches = array();
 preg_match('/<body[^>]*>(.*?)<\/body\s*/isx', $page, $matches);
 $content = $matches[1];
```

There is one part of PHP that I stumble over each time I work with it and that is *regular expressions*. There are many good books written about regex; this is not one of them. Basically this line removed everything that is not between the `body` tags in the page.

Continuing on:

```
/*
  * Filter out the cruft
  */
 $content = preg_replace('/(<style[^>]*>[^>]*<\/style\s*>)/isx', '', $content)
     ;
 $content = preg_replace('/(<script[^>]*>[^>]*<\/script\s*>)/isx', '',
     $content);
 $content = preg_replace('/(&.*?;)/isx', '', $content);

 $content = Zend_Filter::get($content, 'StripTags');
```

Even though what we are left with is the body of the page, there is usually still a lot of junk in there that we don't want to analyze. Yahoo! will not differentiate between tags and words so we need to strip out everything we can find before handing it to them. I don't pretend that this is an exhaustive list of the lines necessary to clean up HTML. These are just the ones I need to clean up my examples. The first two lines remove any embedded style and script tags. The third line removes any embedded HTML special characters. The final line passes the contents through the `Zend_Filter` to remove any other HTML. This should leave us with reasonably clean body content.

Continuing on (remember you'll need to insert your own *appid*):

```
/*
 * send it off to Yahoo for analysis
 */
```

```
$client = new Zend_Rest_Client('http://search.yahooapis.com/
    ContentAnalysisService/V1/termExtraction');
$client->appid('rqP7px7V34L2DVtFJq04ZFTHgiRJQmlvnQze7T313MFdGLAy1.
    lw8PZBWeRqk40R_30')
   ->context($content)
   ->output('xml');
$result = $client->post();
$client = null;
```

Now that we have reasonably clean content, let's hand it off to Yahoo! to let them work their magic. Here we will use another piece of Zend Framework for the first time, `Zend_Rest_Client`. We talked about pieces of the Zend Framework that could be pulled out and used independently of the rest of the framework; `Zend_Rest_Client` and its helpers are a few of those pieces. Here we set the *appid*, the content, and the output type we expect back. Then we make the post and get the results. Note here that we do a `POST` and not a `GET`. This is because the body content we are handing to the API could be large. `GET` has a limit on the size of the URL while `POST` does not.

Continuing on:

```
/*
 * Hand everthing off to the View for output
 */
$this->view->url = $url ;
$this->view->result = $result->Result ;
```

We've done the hard part, now take the relevant data that we have gathered and hand it off to the view so we can use it. Once we leave the controller, nothing we have done here will be accessible to us unless we specifically hand it of to the view. Above we hand off the two important pieces of information we have gathered, an analysis of the URL the user requested and the list of keywords that Yahoo! gave back to us. We assign them to properties of the view so that we can access them during the output.

Add a few more lines:

```
 return;
} // public function extractAction()
```

That's it for the business logic. Now we will look at the very simple output.

As we discussed before, Zend Framework wants to handle the output for you unless you tell it not to. In this case we are letting it handle it so we need a view script for each action method. We created the placeholder file earlier, `extract.phtml`. Now open that up and let's put some code in it.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Keyword Content Analyzer Results Page</title>
</head>
<body >
  <a href="/index/index">Home</a>
<p>
  Page: <a href="<?PHP echo $this->escape($this->url);?>"><?PHP echo $this->
      escape($this->url);?></a><br />
</p>
<hr />
<pre>
<?PHP
  $counter = 1;
    foreach($this->result as $item) {
      echo $counter++.":".$this->escape($item)."<br />";
    }
?>
</pre>
<hr />
<p>The results listed here should not be relied on for any serious commercial
    use. If you do, well, you've been warned.</p>
</body>
</html>
```

The code above isn't complex enough to a require line-by-line discussion. There really shouldn't be anything that is unusual to you. The thing to notice is that the properties we set on the view when the controller was processing are now available to use via the `$this` object. We display the URL using `$this->url` and we parse the array of keywords using the array `$this->result`.

Once you are satisfied that you understand the code above, save it. Now, you should be able to enter the URL of a web page and have it return Yahoo!'s content analysis. If everything worked then you should see something like Figure 3.2.

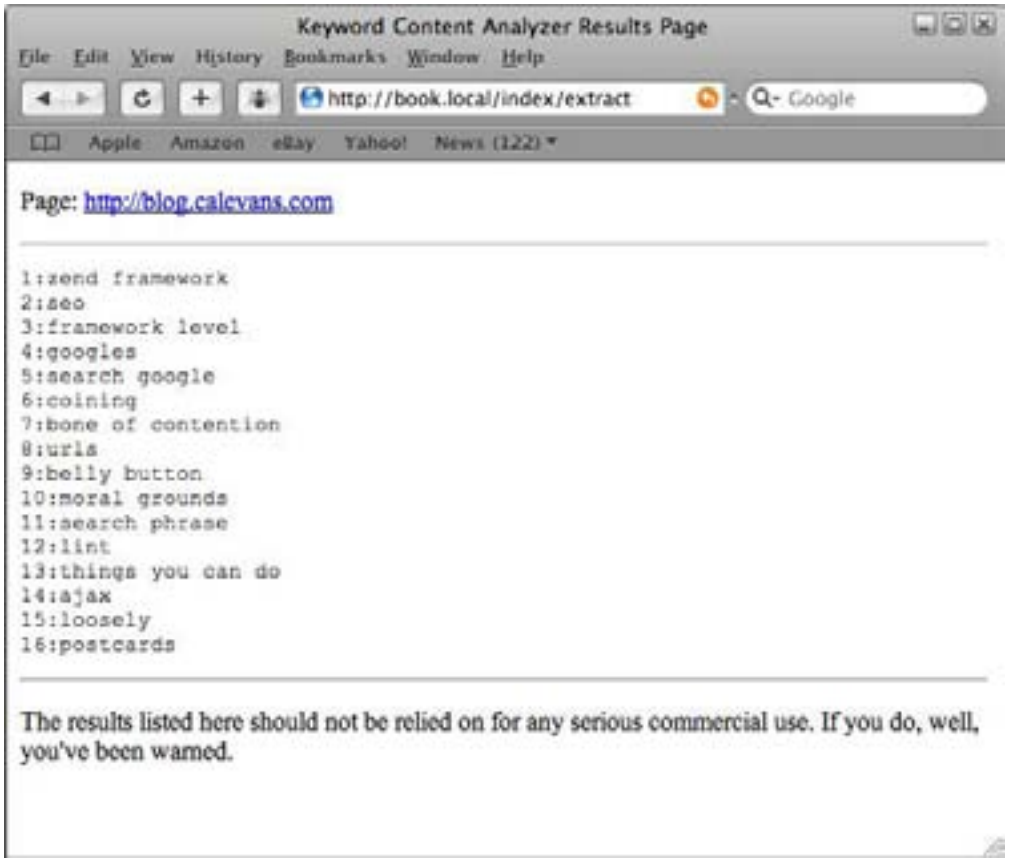

Figure 3.2

BaseController

If everything worked correctly and you have a working application, pat yourself on the back. Let's talk a second about security and then, as a bonus, we will extend this a little bit and introduce a new component.

As it sits right now, anyone with a copy of `curl` or `wget` could hammer your little application until either your server couldn't take it or Yahoo! cut you off; neither of which are good things. To prevent this from happening we need to introduce the

concept of a *token*. If you are familiar with this concept, skip ahead to the section on implementation.

A token is simply a piece of information that the form recognizes and the server recognizes. To be effective, it has to be hard to guess. We generate the token on the server site, store it in the session and in the form as a hidden field. Then when the form comes back, we compare the value to what is stored in the session. If they don't match, we know something is wrong and we abort the form. It sounds simple and truly, it is. Let's look at how to implement this.

In this example, we will build a simple and "reasonably secure" token. It's an md5 hash of the current server time plus a secret that only the application knows. In our example, the secret is the word "book". I don't recommend using something this simple in a commercial application, but as example code, it will work.

Before we dive in, let's talk about how, and more importantly *where* we want to implement this. The obvious answer on where to implement the code to generate and check the token is in the `IndexController`. However, if you have multiple controllers, you will find yourself re-implementing this code multiple times. In the OO world, the term for that is "bad code". A better solution is to create a `BaseController` to house all of these common functions that multiple controllers will need to access. (Actually, the best solution would to be create a `TokenManager` class that all parts of the system could use. But that wouldn't let me show you how to create a `BaseController`, so work with me here.)

In our simple case here, the `BaseController` will only contain the code necessary to deal with tokens.

The `BaseController` we are building here is a forced example. In a production system the two methods we are placing in the `BaseController`, would be better off as action helpers. Our application is simple enough so that a `BaseController` is not strictly necessary. I have found however, that it is easier to create one in the beginning, even if I don't absolutely need it as opposed to finding that I need one well into the coding phase of the project. This is one of those "Do as I say, not as I do" examples; don't try this at home.

Create a file, `controllers/BaseController.php`. If you don't want to take the time to do it yourself, all this code is in `example2.zip`.

```
<?php
```

```
Zend_Loader::loadClass('Zend_Controller_Action');
Zend_Loader::loadClass('Zend_Session_Namespace');
```

First, we move the `loadClass` for the `Zend_Controller_Action` into this since this class now subclasses it and not the `IndexController`. Second, let's load up the `Zend_Session_Namespace`. We will need it for storing the token.

```
class BaseController extends Zend_Controller_Action
{

  protected function generateToken($seed='book')
  {
    $token = md5($seed.mktime());
    $globalSession = new Zend_Session_Namespace('global_data');
    $globalSession->token = $token;
     return $token;
  }
```

This function generates and stores the token. As you can see, you can specify a seed to add to the `mktime()` function; however, if you don't it will use the default, in this case, "book". We take the seed, add the value that `mktime()` returns to us and take an md5 hash of that. While this is by no means foolproof, it's secure enough for most applications.

Once generated, we use the `Zend_Session_Namespace` to create or open a namespace `global_data`. If we were programming straight procedural PHP, the equivalent of this command would be `$_SESSION['global_data']=array();`. It is possible to use both the `Zend_Session_Namespace` and `$_SESSION` to manipulate the `$_SESSION` superglobal directly. However, this is not recommended. With the `Zend_Session_Namespace`, we get a nice OO wrapper for the PHP session. Finally, the function returns the newly generated token to the calling code. As you will see, we take that return value and hand it off to the view for storage in the form itself.

```
  protected function tokenCheck($tokenToCheck='')
  {
    $globalSession = new Zend_Session_Namespace('global_data');
    $returnValue   = (!empty($tokenToCheck) and $tokenToCheck==$globalSession->
        token);
    return $returnValue;
```

```
}
```

This function allows us to check a token against the stored value. Again, we instantiate a `Zend_Session_Namespace` with the namespace `global_data`. We are actually checking for two different conditions here, failing either will cause the function to return *false*. First we want to know that the token passed in is not empty. An empty token is invalid by definition. No token==no form. Second, we check to make sure that the token passed in exactly matches the token we stored in the session. Any discrepancy will again cause the function to return *false*.

Now, let's look at how we use this. You've already seen the `IndexController` above and for the most part, that does not change. Let's just look at the changes.

First, we now have something to put in the `indexAction()`. Modify the following lines in your `IndexController.php` file:

```
public function indexAction()
{
  $this->view->token = $this->generateToken();
  $flash = $this->_helper->getHelper('flashMessenger');
  if ($flash->hasMessages()) {
    $this->view->message = implode("<br />", $flash->getMessages());
  } // if ($flash->hasMessages())
} // public function indexAction()
```

We first ask for a new token and store it in the view. Then we use one of the standard view helpers, the *flash messenger*. We will cover view helpers in the view chapter, but for now just know that there's no need to reinvent a message passing mechanism. Zend Framework comes with one built in. Here you see that we check to see if any messages have been stored in the `flashMessenger`. If so, we load them into the view. Since we don't want to have to deal with the array in the view, let's just build a string and hand that to the view for display.

Now, add this code to the top of `extractAction()`:

```
public function extractAction()
{
  /*
   * get the token
   */
```

```
$token = Zend_Filter::get($this->getRequest()->getPost('token'), 'StripTags'
    );

/*
 * Execute the token check
 */
if (!$this->tokenCheck($token)) {
  $this->_helper->flashMessenger->addMessage("I'm sorry but I think there
      has been an error. Please try again.");
  $this->_redirect('/index/index');
  return false;
} // if (!$this->tokenCheck($token))

$this->generateToken();
```

Since the token is part of the form, we can retrieve it like any other piece of form data. As always, we sanitize the string before use.

Then we hand it off to `tokenCheck()` to evaluate. It should be noted at this point that I am normally a vindictive programmer. In situations like this I don't normally issue any message to the user because if the `tokenCheck()` fails, the chances are good that someone is doing something they shouldn't. Normally, I would simply fail silently and let them try to figure out what is going wrong. However "vindictive" isn't what we normally consider a programming "Best Practice" so I won't teach you that. If the check fails, we add a message into the `flashMessenger` telling the user that something is wrong. I'm intentionally vague because if someone is trying to abuse the system I don't want to give them too many clues as to what they are doing wrong.

The last line should have caught your eye. If it didn't then either you already know why it's there or you are not paying attention. After we check the token and it's good, we ask for another token. This time we don't save off the return value because we don't care. What we do care about is that the token we just validated can't be used again. Calling `generateToken()` here causes a new token to be stored in the session. Subsequent calls to `extractAction()` with the token we already have will immediately fail.

Finally, let's get to the HTML. In `index.phtml`, you need to add this directly below the <`body`> tag:

```
<?PHP
if (!empty($this->message)) {
```

```
    echo "<div id='message'>".$this->message."</div>";
}
?>
```

This will take care of displaying the message if there is one.

Add this line directly below the <form> tag:

```
<input type="hidden" id="token" name="token" value="<?PHP echo $this->token;?>"
    />
```

This actually puts the token into the form.

That's it, you've now thwarted most of the script kiddies and the idly curious.

Helpers and Plugins

An area we've not touched on are *action helpers* and *plugins*. In *Chapter 4* we will get into plugins a bit deeper and actually create one. The `Zend_Controller` supports action helpers and plugins. The two may seem very similar but they serve two different purposes.

Plugins are for when you want to do something on every request, regardless of the controller, without any interaction with the controller. As you will see in *Chapter 4*, plugins require only that you register the plugin with the front controller. Then the code it encapsulates will be executed. All code in plugins is encapsulated in either the `preDispatch()` method or the `postDispatch()` method. Since we give an example of plugins in *Chapter 4*, I won't go into more detail on them here.

Helpers are for when you want shared functionality that you want to use selectively within controllers, and for which you need some introspection or coupling with the controller. Helpers also allow you to insert functionality before or after any action. In many cases this saves you from having to subclass the controller like we did above in creating the `BaseController`.

As with plugins, helpers have `preDispatch()` and `postDispatch()` methods that, if exist, will be called automatically. However, helpers also have a `direct()` method. `direct()` is called if you make a call directly to the helper. In our mythical `MyHelper`, if we defined `MyHelper::direct()` then we could access it's functionality with the following line:

```
$this->_helper->myHelper("Ping");
```

Assuming that `direct()` took a single string parameter, this would cause `MyHelper::direct()` to be executed.

Below is a line from our `extract()` method above:

```
$this->_helper->flashMessenger->addMessage("I'm sorry but I think there has been
     an error. Please try again.");
```

In this case, we did not have to instantiate the helper `flashMessenger`, the `HelperBroker` does that for us automatically if it has not been done yet and then calls the method `addMessage()`. In many cases, especially in cases where helpers are encapsulating common code that will be called directly and not via `preDispatch()` or `postDispatch()`, you never have to worry about initalizing your helper. There are cases however when you will need to manually initialize it.

Placing the Helper

As with everything in the Zend Framework, action helpers have a default place on the file system. Below is our current project as it sites on the file system. You will notice the new section as commented below; this is where we put the helpers. As you will see in future chapters, it's also where we put any of our custom code.

```
C:\web\
    htdocs\
     lib\
       Zend\
       Cal\            <-- Custom code I've written
         Helper\       <-- Where we put our helpers.
     application\
       controllers\
       models\
       views\
         scripts\
           index\
     www\
       images\
```

Again, as with everything you can override the default as well.

Using the Helper

We've discussed the easiest way to use your helper, by allowing the `HelperBroker` instantiate it for you. However, before it can do that, it has to know where to look. `HelperBroker::addPrefix()` is the tool you use to do that.

```
Zend_Controller_Action_HelperBroker::addPrefix('Cal_Helper');
```

This line will tell the `HelperBroker` that we have helpers named `Cal_Helper_*`. Since we did not specify a different path, it will assume a default of `lib\Cal\Helper\` as can be seen above.

There are times when you will want to override the default path structure and store your helpers elsewhere on the file system. To do this we use the `addPath()` method.

```
Zend_Controller_Action_HelperBroker::addPath('./Cal/My/Helper','Cal_Helper');
```

This tells the `HelperBroker` to map `Cal_Helper` to `Cal/My/Helper`, overriding the default `Cal/Helper`.

One final method you have to letting the `HelperBroker` know about your helper is to instantiate it yourself and hand it to the `HelperBroker`.

```
$myHelper = new Cal_Helper_MyHelper();
Zend_Controller_Action_HelperBroker::addHelper($myHelper);
```

In the case where your plugin has `preDispatch()` or `postDispatch()` code, you do have to manually tell the `HelperBroker` to instantiate it. In many cases, your controller's `init()` method is a good place to do this as it happens before the `preDispatch()` hook is called. To do this, you would need to put a line similar to this in your `init()`:

```
Zend_Controller_Action_HelperBroker::getStaticHelper('MyHelper');
```

Since we registered the prefix earlier, we can refer to the helper by its root name (everything between the last _ and the `.php`).

Summary

In this chapter, our goal was to understand a simple controller. The controller we built, along with a supporting cast of characters is complete without a model. More importantly, we've begun to understand that in Zend Framework, even when you are alone, you are in a crowd.

The controller we built does what it is supposed to do; it encapsulates all of the logic necessary to make the application work. It does not, however, encapsulate all of the code. For those of you new to working within a framework instead of just in a simple web page, that's going to be the hardest thing to get used to. Different pieces go in different places. Yes, that means when you are trying to find something, you will end up opening 4 different files just to find the one line of code you are looking to change. However, the ability to separate things into logical components and more importantly, reuse some of the components in future projects makes this worthwhile.

Finally, we introduced several new components so you could see how easy it really is to use pieces of Zend Framework. Honestly, the hardest part about learning any new framework is learning what pieces are there for you to use. Zend Framework makes using things like `Zend_Session_Namespace` very easy, however, if you don't know it's there, you will never use it.

Now, take a break, you've earned it. Grab a cup of joe and relax a bit. If you are just dying to read something, head over to `http://framework.zend.com` and just read over the list of components in the core framework. I think you will be surprised at what's there.

Chapter 4

The Model

In a previous chapter I said that the model can represent the data or the "nouns" in your application. In most cases, those two are synonymous. However, there are different schools of thought in how to build models. In some instances, you will find that a model is not needed for your particular situation.

Types of Model Implementation

There are several different levels of model implementation that can be found in applications.

- No Model
- Light Model
- Heavy Model

No Model

The first example was an application that did not require a model. It was simple and all the business logic was encapsulated within the controller. Cases like this are usually very simplistic situations that do not involve data or heavy business logic. Be careful however, the code in a controller can't easily be shared. If there is a chance

that your business logic will need to be utilized in multiple places, you are better off building a model.

Light Model

In the case of a light model implementation, the model simply is a casing that you stuff the data elements in. The controller and view act upon the data in the model but the model has no or very limited functionality itself. In many cases where a framework ships with a code generator that builds models from the database schema, they are light models containing only the basic functions for Create, Update and Delete (CRUD).

Heavy Model

Heavy models encapsulate not only the data the model represents but also the business logic necessary to act upon it. Many times there will be a base class that the heavy model subclasses so that all models can share the common code they encapsulate.

Zend Framework will allow us the freedom to code in most any style we want. Since I'm a big fan of heavy models, we will talk about them.

Adding Registration & Login

In this example, we are going to take our front end for Yahoo!'s content analyzer and add registration and login to it. The sample code for this chapter can be found on the web site as `example3.zip`.

Creating and Connecting to the Database

To make this example work you will need a access to a MySQL server. In the example code you will see, the database is called `example3`. We only need one table in `example3`, named `member`. Here is the DDL to create `member`.

```
CREATE DATABASE example3;
 USE example3;
```

```
CREATE TABLE member (
      id' int(10) unsigned NOT NULL auto_increment,
      emailAddress varchar(80) default NULL,
      userPassword varchar(50) default NULL,
      firstName varchar(50) default NULL,
      lastName varchar(50) default NULL,
      PRIMARY KEY  ('id')
    );
```

If you don't have a good tool for manipulating a database, I recommend SQLyog (`http://www.webyog.com`). I was an early beta tester for this product and worked with them on features for several years. The community edition is free and should give you most of the tools you will need on a day-to-day basis.

One note before we go any further. In an effort to write example code that is clear and easy to read, I am taking a shortcut that I hope none of you will ever do outside of example code. As you will see when we get into the code, I am storing the passwords in clear text in the database. This is not an acceptable practice and should not be emulated by anyone, ever. Even though they can be cracked, md5 and sha1 are more secure than clear text; use them!

Finally, before we dive into the code, I want to talk a bit about database connections. We will cover database connections and the `Zend_Db` class in more depth later on. For the example in this chapter however, you will need to know the basics. The `Zend_Db` class is a factory class that builds your adapter to your specification and hands it back to you. Here is a list of the basic parameters that you can pass in via the array:

- *host*: a string containing a hostname or IP address of your database server.
- *username*: the user account for authenticating a connection to the server.
- *password*: account password for authenticating a connection to the server.
- *dbname*: your database name.

It's also important to know that just creating the database connection object does not actually connect you to the database. The adapter saves the connection parameters you pass in but does not actually make the connection until it's first needed. This helps keep the creating of the adapter fast.

The Member Class-Registering New Members

The class we are building here is the `member` class. It represents a *member* in your system. In this example, *members* need to do simple things like log in and log out. Behind the scenes, to support these actions, we need to be able to populate a member from the database (load) and commit their data to the database (save). Finally, there are two helper functions that it makes sense to store in the `member` class; `checkEmail()` and `login()`. These are static functions so they can be called without instantiating the class itself. I won't go into a detailed explanation of static methods. If you are not familiar with them you may want to check the PHP manual (`http://devzone.zend.com/manual/language.oop5.static.html`) and read up on them. For those of you familiar with what they are but are still unsure when you should use them, this is a great example. Both of these methods are germane only to a member but are not really business logic that manipulates the data of a member. In the grand scheme of things you waste a couple of processor cycles instantiating a member object just to call the login function. However, it is business logic and therefore does not really belong in the controller itself. So I made them static functions in the `member` class. We can use them without instantiating a member but the code is contained in a logical container.

```
Zend_Loader::loadClass('Zend_Db');

  public function __construct($memberId)
  {

    if ($memberId) {
      $this->load($memberId);
    } else {
      $this->_data = array('id'=>0,
                  'emailAddress' => '',
                  'userPassword' => '',
                  'firstName'  => '',
                  'lastName'   => '');
    } // if ($memberId)

  } // public function __construct($memberId)
```

Like any class in PHP, the `__construct` method is used to set things up. In this case, if you pass in an existing member id then we will fire the `load()` method to populate

the `_data` array. Otherwise we populate it manually. It is not strictly necessary to populate the array manually, we could have simply initialized it. However, I like to keep things from accidentally slipping into the array. As you can see below, the `__set` method checks to make sure the element exists in `_data` before setting it. This means that I can always rely on the contents of the array being elements of my database. It is, however, necessary to make sure that changes to the database structure are reflected here. This is where an ORM would kick in and automatically do that. It is not difficult to build an ORM that handles this functionality for you, it's just not necessary for most applications and I find that the code bloat it adds is not generally worth it.

The `load()` and `save()` methods do exactly what you would expect them to do. `save()` will commit the data to the database and `load()` will retrieve it. While `load()` is straightforward and not worth killing trees to list here, `save()` does have a little bit of code in it for us to discuss.

```
public function save()
  {
    $db = Zend_Db::factory('Pdo_Mysql', array(
              'host'   => '127.0.0.1',
              'username' => 'example',
              'password' => 'example',
              'dbname'   => 'example3'));

    $data = array();
    $data['emailAddress'] = $this->emailAddress;
    $data['userPassword'] = $this->userPassword;
    $data['firstName'] = $this->firstName;
    $data['lastName'] = $this->lastName;
```

Above we create our connection and an array of the data we want to commit to the database. Here however, we need to make a decision. If we already have an id from the database then this is an update not an insert. However, if we do not have an id, we need to create the record. If we are inserting, then of course, we need to capture the id of the newly inserted record.

```
if (!$this->_id) {
  $db->insert('member', $data);
  $this->_id = $db->lastInsertId();
```

```
    } else {
      $db->update('member', $data, 'id = '.(int)$this->_id);
    } // if ($memberId)

  } // public function save()
```

To save trees, I won't reprint the `load()` code here but feel free to print it out and tape it in here if you feel cheated. If you do examine the code, you will see that we get a database adapter from the `Zend_Db::factory()` method and the use it to communicate with the database. Before you go jumping on me with cleats on, yes, I realize I've got duplicated code here. I left it this way to see if you were on your toes. I'm glad you were. In a later chapter I'll show you how to tuck code like this neatly away and never have to deal with it more than once.

```
public static function login($emailAddress=null, $userPassword=null)

{
  $returnValue = false;
  /*
   * Validate the email address
   */
  $emailValidator = new Zend_Validate_EmailAddress();
  /*
   * Email Address is invalid. Notify the user.
   */
  if (!$emailValidator->isValid($emailAddress)) {
    return false;
  } // if (!$validator->isValid($emailAddress))


  $db = Zend_Db::factory('Pdo_Mysql', array(
              'host'   => '127.0.0.1',
              'username' => 'example',
              'password' => 'example',
              'dbname'   => 'example3'));
  $db->getConnection();

  $sql = 'SELECT id FROM member WHERE emailAddress = ? and userPassword=?';
  if ($result= $db->fetchRow($sql,array($emailAddress,$userPassword))) {
    $returnValue = $result['id'];
  } // if ($result = $db->fetchRow($sql, array($emailAddress,$userPassword)))

  return $returnValue;
```

```
}
```

As we talked about above, the `login()` function is static. It takes as parameters the email address and password and returns you either a valid id or zero. Zero, in this case, is an invalid id and indicates that the process failed. The calling method can then decide what to do with the information. In most cases, it will create a new instance of `member`.

Another way to do this would be to have the controller instantiate an instance of `member` and then call `login()` as a regular function. However, if you do this and the login fails, you then have to destroy the instance. Using this method, you don't actually create the member until you know you have a valid login.

```
/*
 * return true if the email address is in the database, false if not or error.
 */
public static function  emailCheck($emailAddress)
{
  /*
   * Validate the email address
   */
  $emailValidator = new Zend_Validate_Email_Address();

  /*
   * Email Address is invalid. Notify the user.
   */
  if (!$emailValidator->isValid($emailAddress)) {
      return false;
  } // if (!$validator->isValid($emailAddress))

  $db = Zend_Db::factory('Pdo_Mysql', array(
              'host'   => '127.0.0.1',
              'username' => 'example',
              'password' => 'example',
              'dbname'   => 'example3'));

  $db->getConnection();

  $sql = 'SELECT id FROM member WHERE emailAddress = ?';
  $result = $db->fetchRow($sql, array($emailAddress));

  if ($result) {
    return true;
```

```
        } else {
          return false;
        } // if ($result = $db->fetchRow($sql, array($emailAddress,$userPassword)))

      } // public static function  emailCheck($emailAddress)
```

Here is the other static method in the `member` class, `emailCheck()`. As its name suggests, this method validates that the email address does not already exist in the database. This is used in the registration process.

The point of this example is to add some new functionality to the application. Now we could simply keep adding actions to the `IndexController`, however, that's not nearly as fun as creating a new controller. (Besides, it's hard to play "find the code" if there's only one controller.) If you need a refresher course in controller, go back and review the previous chapter. I'm going to spare a few trees here and not rehash the things we talked about last chapter and show you the new stuff.

Ok, our new controller is going to be the `MemberController`. As the name suggests, any action that have to do with maintaining the member (`processRegistration()`, `processLogin()`, etc.) all go here. The actions I've just described are different than what we've been working on as they are called primarily to execute code and process input, they do not display pages of their own. In all of my examples for the rest of the book, we will preface an action with the word *process* when it does not have it's own view but redirects to another action after processing.

```
<?php
require_once 'controllers/BaseController.php';
require_once 'models/Member.php';
Zend_Loader::loadClass('Zend_Validate_EmailAddress');
Zend_Loader::loadClass('Zend_Filter');
Zend_Loader::loadClass('Zend_Filter_Htmlentities');

class MemberController extends BaseController
{
```

Ok, the first thing our `MemberController` needs to do is handle logins. The actual login form is displayed by calling `/index/login`, we will look at it a little later. You will recognize a lot of this code from previous actions, we start by filtering out input and checking our token. Finally at the bottom, we make our call to `Member::login()` and

pass in the filtered credentials that were just entered. If `Member::login()` returns a positive number then we know we have a user, we load it into it's own session namespace, set a welcome message for the user and redirect them to the homepage. If this weren't just example code, you would probably want to capture the referring URL, how the user got to the login page and then redirect them to that page once they are logged in. However, since this is example code, we don't need to be that nice.

```
public function processloginAction()
{
  /*
   * get the token
   */
  $token = Zend_Filter::get($this->getRequest()->getPost('token'), 'StripTags'
      );

  /*
   * Execute the token check
   */
  if (!$this->tokenCheck($token)) {
    $this->_helper->flashMessenger->addMessage("I'm sorry but I think there
        has been an error. Please try again.");
    $this->_redirect('/index/login');
    return false;
  } // if (!$this->tokenCheck($token))

  /*
   * Invalidate the token
   */
  $this->generateToken();

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

  /*
   * Get the email address and password.
   */
  $emailAddress = Zend_Filter::get($this->getRequest()->getPost('emailAddress'
      ), 'StripTags');

  $passwordFilter = new Zend_Filter();
  $passwordFilter->addFilter(new Zend_Filter_HtmlEntities())
           ->addFilter(new Zend_Filter_StripTags());
  $userPassword = $passwordFilter->filter($this->getRequest()->getPost('
      userPassword'), 'StripTags');
```

```
    if ($memberId=Member::login($emailAddress,$userPassword)) {
      $memberSession = new Zend_Session_Namespace('member');
      $memberSession->member = new Member($memberId);
      $this->_helper->flashMessenger->addMessage("Welcome Back {$memberSession->
          member->firstName}.");
      $this->_redirect('/index/index');
    } else {
      $this->_helper->flashMessenger->addMessage("I'm sorry but there was a
          problem logging you in. Please try again.");
      $this->_redirect('/index/login');
    } // if ($memberId=Member::login($emailAddress,$userPassword))

  } // public function processloginAction()
```

Similar to `processLoginAction`, we take the input a user gives us, filter it, check our token and then decide if we can allow this person to register. The first thing we have to do is make sure that the `emailAddress` isn't already in the database. We do this with a quick call to `Member::emailCheck()`. `emailCheck()` returns *true* if the email address is already in the database. In that case, we set a message and return the user to the registration page. A kinder developer would set the values for the form fields to what the user submitted. As is obvious, I'm not a kind developer when it comes to sample code.

If the email address is not in the database then we go ahead and create a `member` object, populate it and save it. We do not, however, log the user in. This leaves us room in the process for sending an email validation or to add a system administrator approval to the workflow. For now however, all members can log in once they have completed the registration.

```
  public function processregisterAction()
  {
    /*
     * get the token
     */
    $token = Zend_Filter::get($this->getRequest()->getPost('token'), 'StripTags'
        );

    /*
     * Execute the token check
     */
    if (!$this->tokenCheck($token)) {
```

```
      $this->_helper->flashMessenger->addMessage("I'm sorry but I think there
          has been an error. Please try again.");
      $this->_redirect('/index/login');
      return false;
    } // if (!$this->tokenCheck($token))

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

    /*
     * Get the email address and password.
     */
    $emailAddress = Zend_Filter::get($this->getRequest()->getPost('emailAddress'
        ), 'StripTags');
    $passwordFilter = new Zend_Filter();
    $passwordFilter->addFilter(new Zend_Filter_HtmlEntities())
             ->addFilter(new Zend_Filter_StripTags());
    $userPassword = $passwordFilter->filter($this->getRequest()->getPost('
        userPassword'), 'StripTags');
    $firstName  = Zend_Filter::get($this->getRequest()->getPost('firstName'), '
        StripTags');
    $lastName   = Zend_Filter::get($this->getRequest()->getPost('lastName'), '
        StripTags');

    // if true then error
    if (Member::emailCheck($emailAddress)) {
      $this->_helper->flashMessenger->addMessage("I'm sorry but {$emailAddress}
          is taken already");
      $this->_redirect('/index/register');
    } // if (!Member::emailCheck($emailAddress))

    $member = new Member();
    $member->emailAddress = $emailAddress;
    $member->userPassword = $userPassword;
    $member->firstName  = $firstName;
    $member->lastName   = $lastName;

    $member->save();
    $this->_helper->flashMessenger->addMessage("Welcome to the system. You may
        now log in.");
    $this->_redirect('/index/index');

  } // public function registerAction()

} // class MemberController extends BaseController
```

That's it for the `MemberController.php`. That is not, however the extent of the changes.

Allowing Members to Login

Since we've added features to support a member logging into the system , let's now give them the ability to actually login. (Otherwise, it's really a pointless exercise). To support the two process actions in our `MemberController`, we need corresponding actions to display the forms. These methods could have been placed in the `MemberController` without a problem, however, I felt they fit better in the `IndexController` with the processing code, the code that will actually manipulate the member, in the `MemberController`.

```
public function loginAction()
{
  $this->view->token = $this->generateToken();

  $flash = $this->_helper->getHelper('flashMessenger');
  if ($flash->hasMessages()) {
    $this->view->message = implode("<br />", $flash->getMessages());
  } // if ($flash->hasMessages())

}

public function registerAction()
{
  $this->view->token = $this->generateToken();

  $flash = $this->_helper->getHelper('flashMessenger');
  if ($flash->hasMessages()) {
    $this->view->message = implode("<br />", $flash->getMessages());
  } // if ($flash->hasMessages())

}
```

These two methods are identical to each other. As with the other forms we have built, we generate a token, then check the `flashMessenger` for messages that need to be displayed to the user. Beyond that, they don't do much. All of the HTML is located in the view script.

We need to add one final piece. On any page, we may want to display the user's name or check to see if they are logged in, we need to give the view access to the member object if it's there. Therefore we create an `init()` function on our `BaseController`. Now I know this chapter isn't on the controller so it may not be

the best place to introduce a new controller concept but it just didn't come up in the last chapter. `init()` is a special function. Whenever Zend Framework instantiates a controller, the base Zend Framework class (in our case `Zend_Controller_Action`) has a lot of important code in the `__construct()` method. It is possible to override the `__construct()` in your controller but it is not recommended. Instead, Zend Framework gives us the `init()` method. `init()` is the last thing called in `__construct()` and it is the appropriate place for us to put anything we want executed upon construction.

In our case, we want to check and see if we have a member already instantiated. If we do, let's give the view a reference to it so we can use it's properties in our views. To do this, we add this code to our `BaseController.php` file.

```
public function init()
{
  $memberSession = new Zend_Session_Namespace('member');
  if($memberSession->member) {
    $this->view->member = $memberSession->member;
  }
}
```

Now, on to the view scripts; since both view scripts are almost identical, I'll only show the `register`. You can find the login script code in the `example3.zip` file on this book's Web site.

A close examination of the code is a futile effort since there's not that much of it. We collect the pieces we want to populate the `member` class and send it back up to the server. (If you've made it this far and don't understand this code, you are cheating, go back to *Chapter 1* and start over.)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Register Page</title>
</head>
<body onload="document.mainform.emailAddress.focus();">
  <?PHP
  if (!empty($this->message)) {
    echo "<div id='message'>".$this->message."</div>";
```

```
  }
  ?>
  <p><a href="/index/index">Home</a></p>

  <form method="POST" action="/member/processregister" name="mainform">
    <input type="hidden" id="token" name="token" value="<?PHP echo $this->token
        ;?>" />
    Email Address:
    <input type="input" name="emailAddress" id="emailAddress" value="<?PHP echo
        $this->escape($this->emailAddress);?>" onFocus="this.select();"><br />
    Password:
    <input type="password" name="userPassword" id="userPassword" value="<?PHP
        echo $this->escape($this->userPassword);?>" onFocus="this.select();"><br
         />
    First Name:
    <input type="input" name="firstName" id="firstName" value="<?PHP echo $this
        ->escape($this->firstName);?>" onFocus="this.select();"><br />
    Last Name:
    <input type="input" name="lastName" id="lastName" value="<?PHP echo $this->
        escape($this->lastName);?>" onFocus="this.select();"><br />

    <input type="Submit" name="submit" id="submit" value="Register" >
  </form>

</body>
</html>
```

Finally, since we added that piece of code to our `BaseClass`, let's actually use the `member` class in a view. Here is the index view again, note the short little `if ($this->member)` section. That's all it takes to now add the user's name to the page.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Keyword Content Analyzer</title>
</head>
<body onload="document.mainform.url.focus();">
<?PHP
if ($this->member) {
echo "<div id='welcome'>Logged in as: {$this->member->firstName} {$this->member
    ->lastName}</div>";
}
```

```
?>
  <?PHP
  if (!empty($this->message)) {
    echo "<div id='message'>".$this->message."</div>";
  }
  ?>

  <p><a href="/index/login">Login</a></p>

  <form method="POST" action="/index/extract" name="mainform">
    <input type="hidden" id="token" name="token" value="<?PHP echo $this->token
        ;?>" />
    URL To Analyze:<br />
    <input name="url" id="url" value="http://devzone.zend.com" onFocus="this.
        select();"><br />
    <input type="Submit" name="submit" id="submit" value="Analyze" >
  </form>

</body>
</html>
```

Since the code to populate `$this->member` in the view is in our `BaseController`, we can add that piece of code to any view script. More importantly, we can make decisions based on whether `$this->member` exists or not. For instance, we can decide whether to allow access to a piece of information on a page that only members are allowed to see.

Summary

The model is an important concept to understand. When used correctly, it forms the heart of any application. Views and controllers are the scaffolding that allow you to build your application but the main body of your application will be in the models. The model we have discussed in this chapter is a heavy model as it contains all the data and the code necessary to implement a member. There are other valid schools of thought on how to build models. Lucky for us, Zend Framework allows us to code the way we want and as we showed in the previous chapter, for simple applications, we can work without models if we want.

Also, we learned that instead of overriding the `__construct()` in a controller, place code that you want executed at instantiation in the `init()` function. `__construct()`

will call `init()` as the final step of instantiation. This leaves `__construct()` for those important steps necessary to make the framework work.

Chapter 5

The View

By this point, you have used the view enough so that you should be reasonably familiar with it. However, since it's an integral part of any MVC application, let's burn a few words talking about it.

First off, the point of the view is to render the output. In most cases, this means it will output the HTML. However, there are edge cases where this is not true. The view can be used to render a PDF, in a web service, it can output XML, JSON, YAML, or whatever the application needs. It would be a mistake to always assume that your view is outputting straight HTML.

The view can be thought of as a *templating system*. Like a templating system, it allows you to keep all of your display logic separate from your business logic. We've seen in previous examples how output from code executed in the controller, or output from a module, can be passed to The view for use in output as with most modern templating systems. While it is possible to use a templating system within Zend Framework, it is not really necessary. For our examples here, we won't be using one.

Instantiating the View and Rendering Output

In Zend Framework, The view is represented by `Zend_View`. In most cases (including the examples in this book) the view is instantiated automatically by the controller and we access it with `$this->view`. In our `IndexController`, the lines below create two parameters in the view and store the appropriate values in them. Using the

magic method `__set()`, any parameter assigned a value like this will be created if it does not already exist.

```
$this->view->url = $url ;
$this->view->result = $result->Result ;
```

When the controller is done, `view->render()` is automatically called for us to send the output to the proper output device. It is possible to override this behavior if you need more control over the process.

View Script

The *view script* is what actually contains the display logic necessary to output the data prepared by your controller action and/or your model. The first thing that the view will do upon `render()` is find the proper script. The view looks at the controller name, the action name and merges that with the location of the scripts to create a URI for the view script. For example if you are calling the URL `http://example.com/member/login`, then it will combine the default location of the view scripts `app/view/scripts`, with the controller name, "member" and the action name "login" with the default extension ".phtml" to create the URI `app/view/scripts/member/login.phtml`. If you are paying close attention, you will recognize that URI as one of the files we created in the previous chapter.

If you do not want to use the default view script, you can call `render()` manually at any point and specify the script to call. An example would be:

```
$this->view_render('default.php');
```

The script `default.php` has to be located somewhere in one of your script paths, which brings us to our next point. If you are calling custom scripts, you need to make sure you either put them in the directory where the view is going to look for them, or you need to add your own custom directories into the script path. There are three methods that come into play here, `getScriptPaths()`, `setScriptPath()` and `addScriptPath()`. These three methods allow you to view and set where your view looks for its scripts. `setScriptPath()` sets the entire path; using `setScriptPath(null)`

will clear it completely (and most likely break your app if you don't add something in). `addScriptPath()` on the other hand, pushes a directory onto the stack in LIFO order. (For those of you not familiar with the term, think of a LIFO stack like a PEZ dispenser, the Last candy you put In is the First one you get Out.)

In most cases, when you are using the default `Zend_View` however, all of this is academic as it will take care of all of this for you automatically.

Escaping Output

By this point, everybody should have heard the mantra, "filter input, escape output". You will notice that in the examples of the script helper code we've seen so far, all output is filtered through `$this->escape()`. `escape()` is a method of the `Zend_View` that by default uses `htmlspecialchars()` to escape your output. However, there are times when you will need additional processing. It is possible to tell `Zend_View` to use a different escape method. The `setEscape()` method allows you to tell `Zend_View` exactly what to do when `escape()` is called. The default behavior is the equivalent of calling `$this->view->setEscape('htmlentities')`. You can specify your own function to replace `htmlentities()` if you like, however. By this point, I would hope that you are programming using OO instead of procedural. So, to specify your method, you have to give `setEscape()` an object reference and then a method name, such as `$this->view->setEscape($myobj,'myEscapeMethod')`. The final method of calling `setEscape()` is for static method calls. If you want to use a static method of a class for your escaping, then you use the syntax `$this->view->setEscape('MyClassName','myEscapeMethod')`. In all cases, the method you specify should take, as it's first parameter, the value to be escaped. If the method takes other parameters, they should all be optional. The method's return value should be the escaped output.

View Helpers

In addition to scripts, the view has *helpers*. These helpers assist you in performing complex functions repeatedly. Instead of writing the code over and over, for exam-

ple, properly formatting a date, you can simply use a view helper. Zend Framework comes with several view helpers installed by default:

- `declareVars()`
- `formButton()`
- `formCheckbox()`
- `formFile()`
- `formHidden()`
- `formLabel()`
- `formPassword()`
- `formRadio()`
- `formReset()`
- `formSelect()`
- `formSubmit()`
- `formText()`
- `formTextarea()`
- `url()`
- `htmlList()`

View helpers can really help you reduce the code that is actually in your view script. Let's modify our code a bit to use one as an example. First, we have to change our controller. In our `Index::extractAction()` example, we use SimpleXML to extract the keywords that Yahoo! is returning to us. Currently, we are taking this array of objects and handing it to the view script for processing. That's not really the best way to do things as it means that the view script has to convert each SimpleXMLElement in the array to a string before it can use it. In keeping with the principals of MVC, unless

you have to make a decision based on a piece of data, all processing should be done in the model or the controller. With that in mind, let's move this processing to the controller where it belongs. Our new code in `IndexController.php` looks like this:

```
/*
 * Hand everything off to the view for output
 */
$this->view->url = $url ;
$this->view->result = array();
foreach($result->Result as $item) {
  $this->view->result[] = (string)$item;
} // foreach($result->Result as $item)
```

Now the variable `$this->result` is an indexed array of strings. Now we go to our view script and find the current `forNext` loop we use to output the result array and make a call to the view helper `htmlList()`. Since we are making another couple of minor cosmetic changes to the HTML and it's all pretty small, here is `extract.phtml` in its entirety.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
  <title>Keyword Content Analyzer Results Page</title>
</head>
<body >
  <a href="/index/index">Home</a>
<p>
  Page: <a href="<?PHP echo $this->escape($this->url);?>"><?PHP echo $this->
      escape($this->url);?></a><br />
</p>
<hr />
<?PHP echo $this->htmlList($this->result);?>
<hr />
<p>The results listed here should not be relied on for any serious commercial
    use. If you do, well, you've been warned.</p>
</body>
</html>
```

`htmlList()` actually takes 3 parameters:

- items: Array (Indexed or associative); the array to process and output.
- orderedList: Boolean; If true then the list output is an ordered list, if false, it is an unordered list.
- attribs: Array (Associative); A list of attributes to set in the `ul` or `ol` tag. These can be things like style, class, id, etc.

Let's expand our example a bit to include the other parameters. Modify your code as such:

```
<?PHP echo $this->htmlList($this->result,true,array('style'=>'color: red;'));?>
```

Now run your code and you should see something like Figure 5.1.

In addition to the default helpers that come with Zend Framework, you can create your own. First, let's look at where we put them and what we call them. If we look at our original directory structure from the *Getting Started* chapter, we see this that the logical place for them is under the `views` directory, like this.

```
app\ <-- Zend Framework based application
       controllers\
       models\
       views\
             helpers\
             scripts\
```

By default Zend Framework will look for your custom helpers in `views\helpers`. There are five basic rules you must follow when writing custom view helpers.

- Your class name must begin with the helper prefix.

The default prefix is `Zend_View_Helper`. You can change that if you like, I'll describe how in a second but at the very least, your class needs to be named `Zend_View_Helper_MyHelper`.

- Your class name has to end in a CamelCased version of the helper name.

As shown above, `MyHelper` is the actual helper name so we append it to the prefix.

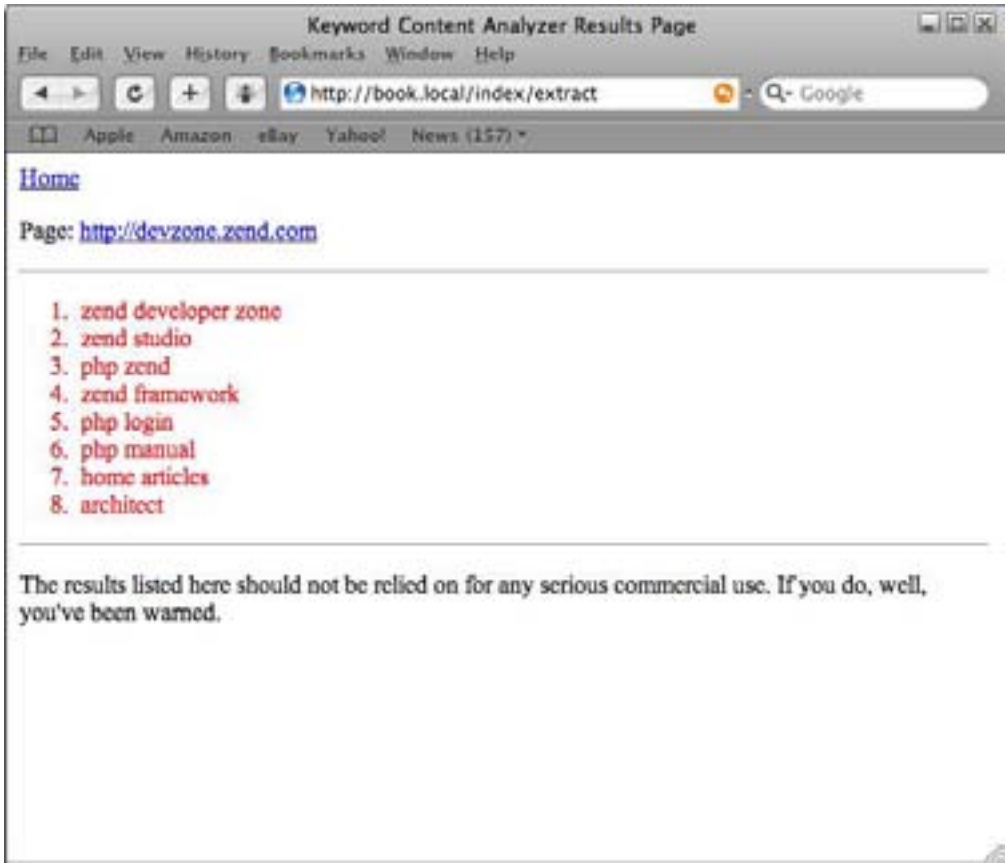

Figure 5.1

- Your helper class needs to have a method named for the class.

In the case of `MyHelper`, you are required to have a method named `myHelper()` inside the class as its main public interface.

- Your helper class, in most cases, should not echo or generate any output.

Your helper is there to help, not output. Let the view handle all of the output, all your helper needs to do is take in the appropriate parameters and return whatever it wants outputted. All return values from a helper should be properly escaped.

- The file that contains your class should be named after the class.

In our example, the class `View_Helper_MyHelper` would be contained in a file named `MyHelper.php`.

If you don't want to keep your helpers in the default location or if you want to have a custom prefix for your view helpers, you can use the methods `setHelperPath()` or `addHelperPath()`. `addHelperPath()` takes as its first parameter a string that is the directory you want to add to the helper path stack. The second optional parameter is the class prefix for helpers stored in that directory.

`setHelperPath()` takes either a string or an array as the first parameter, both of which represent the director(-ies) to search for helpers. The second, optional, parameter is the default prefix for all directories. So, if you just want to add your own directory and prefix to the list, use `addHelperPath()`. If you want to override the default behavior and set everything yourself, use `setHelperPath()`.

There is, of course `getHelperPath()` and `getHelperPaths()`. The former takes the name of a class and it will return to you the path it is located in. The second returns an array of paths that will be searched for helpers.

Let's look at some code. I know that all of that may sound complicated but in practice, it's straightforward. Let's look at a sample helper that does almost nothing but it will show the salient points.

```
<?php
class Zend_View_Helper_MyHelper
{

  public function myHelper()
  {
    return 'This is being output from the custom helper <br />';
  }
}

?>
```

Ok, save this in a file named. . . any one want to venture a guess? If you said `views/helpers/MyHelper.php`, you were right!

Now in our `index.phtml` let's add this line:

```
<?PHP echo $this->myHelper(); ?>
```

Really, you can put this anywhere. In the sample code I put below the form close tag. Now, bring up your browser and go to your home page. If you've done everything correctly, you should see something that looks like the image shown in Figure 5.2.

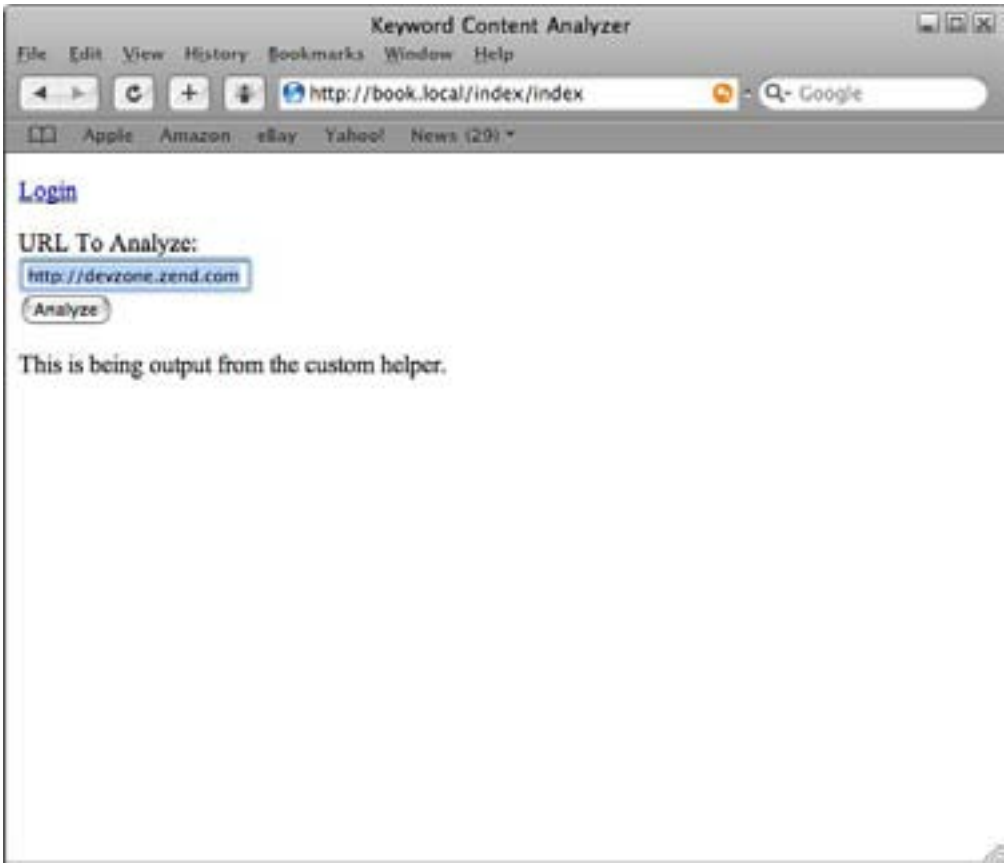

Figure 5.2

Everything looks good, right? No! Even though everything worked, we forgot the last part of rule number four, "All return values from a helper should be properly escaped." We've done no escaping to this output at all. Granted, we would have to

shoot ourselves in the foot on purpose for it to cause a problem in this example, but it's still a good idea to do things right the first time.

Since we know that the escape can be overridden, we can't assume that `htmlspecialchars()` is the method used to escape. We have to use the view's `escape()` method. The problem is that we don't have direct access to the view from the helper. To help us out here, Zend Framework recognizes a special method `setView()` in a view helper. If that method exists, it will call it, passing in a reference to the current view as the only parameter. It makes sense for us to save that off so we can use it in our main method.

Now, with a reference to the view, we have access to the escape method. When you put all of this together, our `MyHelper` class should now look like this.

```
<?php
class Zend_View_Helper_MyHelper
{

  public $view;

  public function setView(Zend_View_Interface $view)
  {
    $this->view = $view;
  }

  public function myHelper()
  {
    return $this->view->escape('This is being output from the custom helper <br
        />');
  }
}

?>
```

Now, save that, and refresh your index page. You should see something that looks like Figure 5.3.

See the `<br />` tag? That tells us our escaping is working. Feel free to remove it at your leisure.

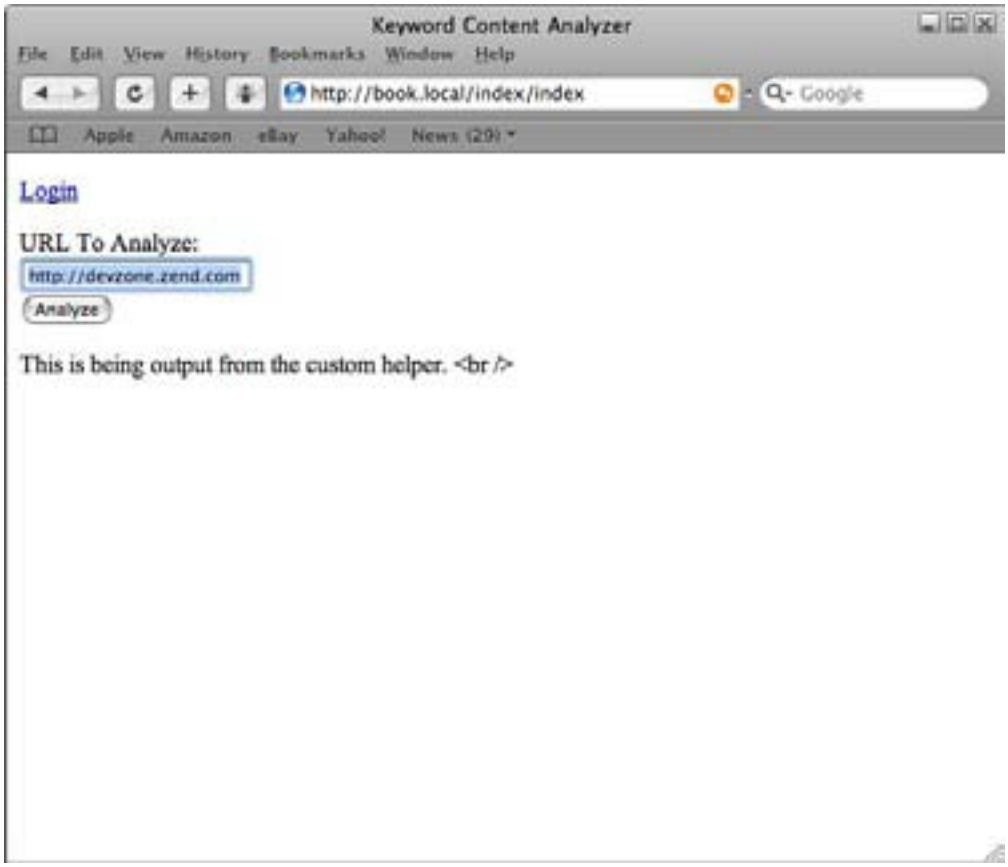

Figure 5.3

Summary

In this chapter, we went over the details of the view as implemented in Zend Framework. We talked about how to use them, how to set some of the optional parameters and how to extend them through the use of helpers; most importantly though we discussed escaping. I say this is the most important because it transcends all frameworks. It is important, whether you are coding using Zend Framework or just straight PHP, that you understand this one point. Escape your output. Always, without ex-

ception, no edge cases, period. As a matter of fact, that is ½ of the timeless and sagely advice "Filter Input, Escape Output". As a developer, it is your responsibility to make sure that your code is safe. So do everyone a favor, head down to the local tattoo parlor and have that tattooed to your forehead. Every morning when you get up, look in the mirror and repeat it 10 times; eventually, you'll get the point. (Either that or it will hit you after someone has compromised your server because you forgot.)

Chapter 6

Data Access

Database access in Zend Framework is distributed among several classes and "the right one to use" greatly depends on what you are trying to do. This is not an exhaustive look at the different methods of data access; we mainly look at `Zend_Db` and the database adapters.

Connecting to the Database

Zend Framework uses the `Zend_Db` class to connect to a database. `Zend_Db` and it's related classes give you a PDO-like interface to talk to your database. There are two ways to get a connection to your database. You can use `Zend_Db` as a factory to create it for you or you can instantiate the adapter you need directly. Our first example will show the factory method since that is the most generic and in most cases the easiest to use.

```
require_once 'Zend/Db.php';

$db = Zend_Db::factory('Pdo_Mysql',
             array('host'   => 'localhost',
               'username' => 'example3',
               'password' => 'example3',
               'dbname'   => 'example3'));
```

This is the method we used in previous chapters to get our db connection. The first parameter the factory method needs is the type of adapter to instantiate. Using the factory, this can easily be parameterized so that your application can support multiple backends. Even when I don't want to support multiple backends, this is still my preferred method of getting my connection because it allows me to change my mind.

The second parameter passed to the factory is an associative array of connection options. These are the same options you would pass in if you instantiate the adapter directly. If you missed it in the previous chapter, here is a list of parameters you can pass in.

- host: a string containing a hostname or IP address of the database server
- username: the username associated with your database account
- password: the password associated with your database account
- dbname: database instance name on the RDBMS server
- port: this is an optional parameter. Some database servers will allow themselves to be run on a user-specified port. If is the case with yours and you or your admin has changed the port number, you can specify it here. If you don't understand any of that then ignore this parameter.
- options: this parameter is an associative array of options that are generic to all `Zend_Db_Adapter` classes. See list below.
- driver_options: this parameter is an associative array of additional options that are specific to a given database extension. One typical use of this parameter is to set attributes of a PDO driver.
- profiler: if *true* then `Zend_Db` will instantiate a *profiler object*

The *options* parameter can contain any of the following:

- `Zend_Db::FETCH_ASSOC` : returns the data in an array of associative arrays
- `Zend_Db::FETCH_NUM` : returns the data in an array of indexed arrays

- `Zend_Db::FETCH_BOTH` : this one is for those of you who just can't figure out what they want. It will return data in an array of arrays. The array keys are both strings as used in the `FETCH_ASSOC` mode, and integers as used in the `FETCH_NUM` mode
- `Zend_Db::FETCH_COLUMN` : return data in an array of values. The value in each array is the value returned by one column of the result set
- `Zend_Db::FETCH_OBJ` : return the data in an array of `stdClass` objects
- `Zend_Db::CASE_FOLDING` : this controls how the array keys are returned. The valid values are `Zend_Db::CASE_NATURAL` (the default), `Zend_Db::CASE_UPPER`, and `Zend_Db::CASE_LOWER`
- `Zend_Db::AUTO_QUOTE_IDENTIFIERS` : if the value is *true* (the default), identifiers like table names, column names, and aliases are delimited in all SQL syntax generated by the Adapter object This allows you to use identifiers that contain SQL keywords, (i.e. a field named "password" or "date") or special characters

The first four options described all related to how the data is returned to you or the `FetchMode`. You can specify the `FetchMode` in the options array or you can specify it with the `setFetchMode()` method. Using `setFetchMode()` allows you to change your mind before you issue the query. It also allows you to change things for a specific query so that you are not stuck with the creation option.

If you are only using a single database server and have no intention of ever changing, you can instantiate the adapter directly, bypassing the factory. Here is an example to chew on.

```
require_once 'Zend/Db/Adapter/Pdo/Mysql.php';

$db = new Zend_Db_Adapter_Pdo_Mysql(array('host'   => 'localhost',
                      'username' => 'example3',
                      'password' => 'example3',
                      'dbname'   => 'example3'));
```

Either way you do it, you end up with a database adapter ready to use, well almost. What you have is an adapter that can make a connection to the database, because

you've not yet done anything that requires access to the database. To speed things along, Zend Framework does not actually make a connection to the database until you do something. This means that getting an adapter is cheap and painless. The overhead of the connection is delayed until you actually need it. The Zend Framework documentation refers to this as "Lazy Connections".

Fetching Data

Ok, so you have your adapter and you need to fetch some data. Let's take a look at the member model we created in the previous chapter. In the method `checkEmail()`, we want to see if the email address the user specified has been used. After getting an instance of the adapter, we make the call.

```
$sql = 'SELECT id FROM member WHERE emailAddress = ?';
$result = $db->fetchRow($sql, array($emailAddress));
```

First you will notice the `?` in the select statement. For those not familiar with binding, the first `?` will be replaced with the first value in the array, the second, with the second value, you see where this is headed. Among other things, this allows us to write a parameterized query once and use it over and over again without having to concatenate the values in.

In the case of the example above, we could have used any one of the `fetch*()` methods of the adapter. Below I'll list all of the `fetch*()` methods and give examples but let's whip up a quick action that we can use to show all of the output. Pass this code into your `IndexController` and save it.

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

  $db = Zend_Db::factory('Pdo_Mysql', array(
              'host'   => '127.0.0.1',
              'username' => 'example',
              'password' => 'example',
              'dbname'   => 'example3'));

  echo '<pre>';
  $sql = "select * from member";
```

```
    echo "<h2>fetchAll()</h2>";
    print_r($db->fetchAll($sql));
    echo "<h2>fetchAssoc()</h2>";
    print_r($db->fetchAssoc($sql));
    echo "<h2>fetchCol()</h2>";
    print_r($db->fetchCol($sql));
    echo "<h2>fetchPairs()</h2>";
    print_r($db->fetchPairs("select firstName, lastName from member"));
    echo "<h2>fetchRow()</h2>";
    print_r($db->fetchRow($sql));
    echo "<h2>fetchOne()</h2>";
  print_r($db->fetchOne("select emailAddress from member where id=4"));
  echo '</pre>';
  }
```

Now, assuming you have created at least one (and optimally more than one) login when working through the previous chapter, you should be able to go to `/index/testDB` and get something that looks like Figure 6.1

This will show you at a glance what the different methods will return. NOTE: Please, please, please, never put code like that in a production system. If I have to explain why, you really need to start over with Chapter 1.

Now, let's look at the options we have for fetching data.

fetchAll()

By far the simplest way to fetch data from your database, `fetchAll()` does as it's name implies and fetches everything. It returns an array of arrays. The index of the outer array is incremented for each row but has no specific meaning to the database. Using `fetchAll()` will return the following:

```
Array
(
  [0] => Array
    (
      [id] => 4
      [emailAddress] => cal@example.com
      [userPassword] => calevans
      [firstName] => Cal
      [lastName] => Evans
    )
```

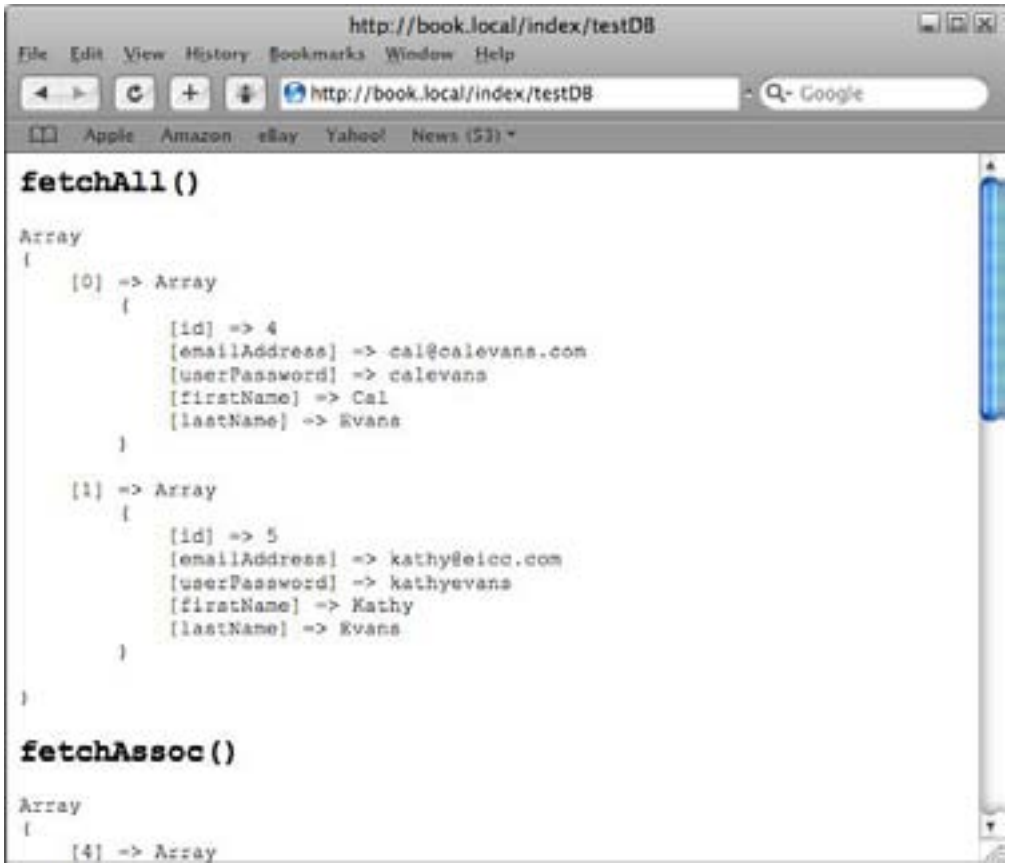

Figure 6.1

```
    [1] => Array
      (
        [id] => 5
        [emailAddress] => kathy@example.com
        [userPassword] => kathyevans
        [firstName] => Kathy
        [lastName] => Evans
      )

  )
```

fetchAssoc()

`fetchAssoc()` returns a result set similar to `fetchAll()` but the index of the outer array is the primary key of the table. In our example, it will return this:

```
Array
(
  [4] => Array
    (
      [id] => 4
      [emailAddress] => cal@example.com
      [userPassword] => calevans
      [firstName] => Cal
      [lastName] => Evans
    )

  [5] => Array
    (
      [id] => 5
      [emailAddress] => kathy@example.com
      [userPassword] => kathyevans
      [firstName] => Kathy
      [lastName] => Evans
    )

)
```

fetchCol()

In our sample code, the results produced aren't too helpful. `fetchCol()` is designed to fetch a single column. The outer array's index is simply an incrementing integer and has no special meaning to the database. Since we are attempting to fetch multiple columns, the first column in the database is what is returned. In our case, it's the primary key, `id`.

```
Array
(
  [0] => 4
  [1] => 5
)
```

fetchPairs()

`fetchPairs()` assumes that the SQL statement you are executing returns exactly 2 columns. It returns an array where the index is the first column and the value is the second. While this is very handy, make sure that the first column is unique.

```
Array
(
  [4] => Cal
  [5] => Kathy
)
```

Since, in my sample data, `lastName` is not unique, if I use the SQL statement:

```
select lastName, firstName from member
```

I get :

```
Array
(
  [Evans] => Kathy
)
```

Not exactly the result set I was looking for.

fetchRow()

As the name implies it fetches a single row. If your query returns multiple rows, `fetchRow()` will only return the first one in the result set. The resulting associative array is indexed on the column name and the value is the column value.

```
Array
(
  [id] => 4
  [emailAddress] => cal@example.com
  [userPassword] => calevans
  [firstName] => Cal
  [lastName] => Evans
)
```

fetchOne()

Finally, the most restrictive and the only `fetch*()` that does not return an array. `fetchOne()` assumes that your SQL query only returns a single column from a single row. In this case, the value returned is scalar and is the value of the column specified.

In our example code above, our SQL statement we request the email address for a specific row:

```
select emailAddress from member where id=4
```

The result is a string containing the value `cal@example.com`.

Since you most likely do not have an `id=4` in your database, the value of `$result` will be *false* for your test.

Given all the options we have, `fetchOne()` would have been a good choice also. However, since everything else in `Member` assumes that `$result` is an array, I used `fetchRow()`.

Profiler

One of the cool features of `Zend_Db` is the *profiler*. We brushed by it earlier when we discussed how to turn it on but let's take a closer look at it now. Anyone who has worked with SQL data knows the pain of figuring out what is causing the query to take so long to return results. In simple applications like our example, profiling and debugging SQL queries is easy because we rarely execute more than one per action. However, in real-world applications with multi-part transactions, it can be difficult to isolate a problem. It can be if you don't have `Zend_Db` and the profiler, that is.

Let's take a look at this secret weapon and how it used. For this example, I will be using the `testDBaction()` action we created above. If you skipped that step, stop, go back a page or so and follow the instructions.

To turn on the profiler, scroll up to where we create the connection and make it look like this:

```
$db = Zend_Db::factory('Pdo_Mysql', array(
            'host'   => '127.0.0.1',
            'username' => 'example',
```

```
            'password' => 'example',
            'dbname'   => 'example3',
            'profiler' => true));
```

The obvious difference is the last line. Now go to the bottom of the method and add these lines in. (Just so you know, if you are too lazy to follow these instructions, you can just download `example3.zip`, it's all in there.)

```
echo "<hr /><table border='1'><tr><th>Query #</th><th>Time</th><th>Query</th></
    tr>";
$profiler   = $db->getProfiler();
$totalTime  = $profiler->getTotalElapsedSecs();
$queryCount   = $profiler->getTotalNumQueries();
$longestTime  = 0;
$longestQuery = null;
$count    = 0;

foreach ($profiler->getQueryProfiles() as $query) {
  echo "<tr>";
  echo "<td>".$count++."</td>";
  echo "<td>".$query->getElapsedSecs()."</td>";
  echo "<td>".$query->getQuery()."</td>";
  if ($query->getElapsedSecs() > $longestTime) {
    $longestTime  = $query->getElapsedSecs();
    $longestQuery = $query->getQuery();
  } // if ($query->getElapsedSecs() > $longestTime)
  echo "</tr>";
} // foreach ($profiler->getQueryProfiles() as $query)
echo "</table>";

echo 'Total Queries Executed : ' . $queryCount . "\n";
echo 'Total Time       : ' . $totalTime . ' seconds' . "\n";
echo 'Average query length   : ' . $totalTime / $queryCount . ' seconds' . "\n";
echo 'Queries per second   : ' . $queryCount / $totalTime . "\n";
echo 'Longest query length   : ' . $longestTime . "\n";
echo "Longest query      : \n" . $longestQuery . "\n";
echo '</pre>';
```

One thing that this book never promised was how to design pretty web pages. Most of you have probably noticed that the output of this method is pretty dang ugly. Really, that's ok, it's just sample code, cut me some slack. If you got beyond the aesthetics of the output, you should see something like what is displayed in Figure 6.2

Figure 6.2

Now in our basic example, the tables have so little data in them that the connection takes the longest time to execute. In a real world example though analyzing the queries in this way would help you identify potential bottlenecks.

The profiler has several public methods that will help you analyze your queries.

- `getTotalNumQueries()`: returns the total number of queries that have been profiled since the profiler was created or last cleared
- `getTotalElapsedSecs()`: returns the total number of seconds elapsed for all profiled queries

- `getQueryProfiles()`: returns an array of all the query profiles expressed as `Zend_Db_Profiler_Query` objects
- `getLastQueryProfile()`: returns the most recent query profile expressed as a `Zend_Db_Profiler_Query`, regardless of whether or not the query has finished. If the query has not yet completed, the end time will be null.
- `clear()`: clears any past query profiles from the stack. Both `getQueryProfiles()` and `getLastQueryProfile()` return instances of `Zend_Db_Profiler_Query`. `Zend_Db_Profiler_Query` has its own methods that can be used to examine the properties of the query.
- `getQuery()`: returns the SQL text of the query. If the query is a prepared statement, `getQuery()` will return the text as of the time it was prepared. This means it will have parameter placeholders, not the values used in the execution of the statement. (For those, see the next method)
- `getQueryParams()`: returns an array of parameter values used when executing a prepared query
- `getElapsedSecs()`: returns the number of seconds it took the query to run

The profiler also allows you to filter the profiles in many ways so that you only analyze the data you are interested in. `setFilterElapsedSecs()` allows you to set a minimum execution time before a query will be profiled. `setFilterQueryType()` allows you to specify which query types you want profiled. `getQueryProfiles()` allows you to retrieve only certain types of queries. Combine all of these together and you've got a powerful tool that will help you in your coding.

Summary

`Zend_Db` is a powerful tool in the right hands. It is not, however, the only way to access data via Zend Framework. `Zend_Db` has some advanced tools that you will want to investigate as you begin to run into the limits of using `Zend_Db` directly.

- `Zend_Db_Table`

- `Zend_Db_Select`
- `Zend_Db_Table_Row`
- `Zend_Db_Table_Rowset`
- `Zend_Db_Table_Relationships`

All of these are valuable in their own right and as your needs grow more complex, you will want to investigate them and start integrating them into your applications.

Chapter 7

Authentication

While the information in the last chapter was important, we didn't have a lot of code. This chapter however, we revisit our sample application. In this chapter we are going to add pages to the system that requires the user to be logged in before it can be viewed. Previously, we added the ability for users to register and log in. However, other than acknowledging the user by name on each page, we really didn't do much with the information that he/she is logged in. Now let's do something with that information.

Before we delve into the code though, it's important to make a distinction here. Ours is a simple application thus far, and the pages we are going to protect are protected on the basis of whether or not the user is logged in. The process of verifying logged in status or not is called *authentication* or `auth`. In more complex applications, you have levels of users, and certain classes of users have specific rights that other classes don't have. Determining whether a user has specific rights and acting upon those rights is called *access control*. We are dealing with access control only in the most rudimentary way, by only checking to see if the user is logged in or not. When you start building applications using Zend Framework and you need a finer grain of control, you will want to implement `Zend_Acl`.

About Zend_Auth

Zend_Auth can be a complicated beast, as it allows you to define your own authentication methods using custom adapters. Natively, Zend Framework comes with 3 adapters that you can use, Zend_Auth_Adapter_DbTable, Zend_Auth_Adapter_Digest, and Zend_Auth_Adapter_Http. As with everything in Zend Framework, you can chose to use one of these as if, if they meet your needs, extend them to do exactly what you need done, or write your own. By extending the Zend_Auth_Adapter_Interface you can build your own custom adapters to authenticate against any backend service you choose. Zend Framework documentation provides excellent examples of how to create your own adapters.

While different adapters may have vastly different options and parameters, they all have one thing in common, they all return a Zend_Auth_Result when authenticate() is called. Zend_Auth_Result is the payload wrapper. With it you can tell if the authentication passed or failed, get any error codes or messages, and get the identity information. We will look at these methods in deeper detail later.

I'd like to say one final word about the Zend_Auth system before we dive into some code details. By default Zend_Auth uses the Zend_Session for persistent storage. You can override this and use your own custom storage solution by implementing Zend_Auth_Storage_Interface in your class. If you chose the default, Zend_Storage will create or use the Zend_Auth namespace in the session. If you are confused about this, skip back a few chapters where we discussed Zend_Session and namespaces in a bit more detail.

I've glossed over a lot of the functionality of Zend_Auth simply because it's not germane to our discussion and example. However, once you get going on your own real-world application, I'm sure you will investigate further. About the third time you think you have hit a show-stopping hurdle, only to find that Zend Framework either already supports the functionality you need, or you can easily extend it with a few lines; you'll really start to appreciate just how flexible Zend Framework is.

We will be implementing a simple auth scheme using Zend_Auth_Adapter_DbTable as our adapter. Digest and HTTP are both good choices but since we are already working with a database, we won't add unnecessary complexity to our simple application.

I should probably mention here that `Zend_Auth` implements a *singleton pattern*. This, of course, means that there can only be one instance of the class. To get a handle to that instance, you use the static method `Zend_Auth::getInstance()`.

Using Zend_Auth

Now, let's get our hands dirty. Let's add a page that stores URLs that have been analyzed and allows us to re-display old results.

Ok, we've got new actions that have to be added to the system. The first question to be answered here is what controller owns our two pages, `listSearches` and `listResults`. It's really a toss up. There are only two controllers in the system right now, `IndexController` and `MemberController`. You can add them to `IndexController` under the theory that all general pages start in `IndexController`; or you can add them to `MemberController` under the theory that anything requiring one to be logged in needs to go in `MemberController`. I'm going to take the third option which is creating a controller, `HistoryController`.

I want to segregate the pages by controller based on function. Since we are creating a new function here, it just makes sense to create a new controller. You have to be careful though because you can end up with hundreds of controllers in a large scale application if you are not careful. My second reason however, is a bit more pragmatic. In any decent sized application, you will have multiple developers. If I were to simply shove this new functionality into `IndexController` or `MemberController`, then the chances of me having to merge my work with another developers work upon check-in is much greater. By creating `HistoryController`, I'm ensuring that I won't step on anyone else's work while adding my own. While this may sound like specious reasoning, do not discount these types of factors in a multi-developer project. It's an important consideration.

Now let's take a look at what we actually want to do. We want to record each unique URL that a person analyzes. We also want to record multiple results for each URL so our users can contrast and compare them. So let's break it down. We now need a few more tables in our database. First, `web_property` is any URL that is analyzed. It holds one record for each unique URL. Next, we need a table to collect a list of analysis performed on a URL, we will call that `web_property_analysis`. Finally, we need a table to hold the results from each analysis, which we will call

`web_property_analysis_results`. So, here is the DDL for our new tables. Add these into the database `example3`.

```
CREATE TABLE ‘web_property‘ (
       ‘id‘ int(11) default NULL,
       ‘url‘ varchar(2048) default NULL,
       UNIQUE KEY ‘url‘ (‘url‘(1000))
       ) ENGINE=MyISAM DEFAULT

CREATE TABLE ‘web_property_analysis‘ (
       ‘id‘ int(11) default NULL,
       ‘searchDate‘ datetime default NULL,
       ‘web_property_id‘ int(11) default NULL
       ) ENGINE=MyISAM

CREATE TABLE ‘web_property_analysis_results‘ (
       ‘id‘ int(11) default NULL,
       ‘web_property_analysis_id‘ int(11) default NULL,
       ‘keyword‘ varchar(150) default NULL,
       ‘position‘ tinyint(4) default NULL
       ) ENGINE=MyISAM
```

Ok, now that we have our database, we will need our models. A `WebProperty` will have a collection of `WebPropertyAnalysis` objects. Each `WebPropertyAnalysis` will have an array of strings that represent the words or phrases returned in the order they are returned.

Ok, ok, I know none of this is really coding yet but we have a few housekeeping issues to deal with before we can actually kick in on the code. Unpack `example4.zip` from the web site. While you do that, here is a list of the main items that changed between `example3` and `example4` that don't directly relate to our example.

Delete `MyHelper.php`. Hey, it was fun to write our own helper but right now it just outputs needless code. So delete it from `scripts/helpers` and edit `scripts/views/index/index.phtml` to remove the reference to it.

Moved Database Connection to `BaseModel`. This isn't its final resting place but I know it was driving you OO and DRY advocates out there up the wall. I actually wrote it that way on purpose because I didn't want to get bogged down in a `BaseModel` when we were building things; however, now is the time. If we leave it in `BaseModel` then only models will have the ability to get a database connection. This is not the best solution. However, we'll leave it there for now. Since we only have one model right

now, it was easy to make all the changes to use the new `BaseModel::getDb()` function. Check out `models/Model.php` to see where things changed.

Finally, yes, I know I keep saying that, it's codin' time! The heart of the concept we are implementing is login. Since we have already isolated the business logic necessary for a user to log in, the first place we start is `Member->login()`.

You will notice as you marvel at the code below, that we use the first part of login as-is and we still need to make sure we have an email address and that it's valid. As with last time, password in my book is optional. If you don't care, why should the system enforce rules on you that you don't need?

```
public static function login($emailAddress=null, $userPassword=null)
{

  $returnValue = false;
  $emailValidator = new Zend_Validate_Email_Address();

  if (!$emailValidator->isValid($emailAddress)) {
    return false;
  } // if (!$validator->isValid($emailAddress))
```

Here is where we start to deviate from the previous code. Since we moved the code to generate a database connection into the `BaseController`, it's much cleaner.

```
$db = BaseModel::getDb();
$db->getConnection();
```

This is where the magic is. You have probably noticed by now that we totally ignore `Zend_Auth`. If we wanted to set a special persistent storage method or some other auth parameter we would have to get the instance of `Zend_Auth` via the `Zend_Auth::getInstance()`. However, since we are using the default parameters, we can bypass `Zend_Auth` and work directly with our adapter. We create our `Zend_Auth_Adapter_DbTable` and hand it the `db` we created earlier so it can talk to the database. A quick side note, all of the parameters we are setting here manually can be handed to the adapter at creation time. However, it's easier to explain them if we don't.

Next, we tell the adapter everything it needs to authenticate. Since this is a database, that involves the table to look at and the fields to match. The *identity*

column is the column in the database used to represent the identity. This is analogous to the user name column but because you may have something better than user name, you get to tell the adapter which column to use. In our case we are using email address however you can use any unique column. Next we set the *credential column*. The credential column is analogous to the password column. In our example, it actually is the `userPassword` column but you can specify whatever column you want. In addition to telling the adapter what table and fields to use, we also set the values for the identity and credential columns. These, you will recall, were passed into our function call as parameters.

```
$authAdapter  = new Zend_Auth_Adapter_DbTable($db);
$authAdapter->setTableName('member')
      ->setIdentityColumn('emailAddress')
      ->setCredentialColumn('userPassword')
      ->setIdentity($emailAddress)
      ->setCredential($userPassword);
```

One note here about passwords. We are writing example code here so we really don't care about the security of passwords, therefore we are storing them in clear text in the database. If you don't understand why that's a bad thing then please return this book to the person you stole it from because obviously you are not a programmer. If we were storing the passwords in the database encrypted, the most common function used is the MySQL `password()` function. However, by default the `Zend_Auth_Adapter_DbTable` doesn't know what RDBMS you are using, so it assumes that the credential column is stored in plain text. You can change this by using the `setCredentialTreatment()` function. `setCredentialTreatment()` takes one parameter, the name of the function to use to prep the password.

There are two things you need to know though. First the function must be one that your RDBMS supports. Whatever you pass to `setCredentialTreatment()` is included in the SQL statement. Second, you need to pass in the function name, as a string, with a `?` as the placeholder for where the credential value goes. For example, if you were using MySQL's default password function, you would use

```
$adapter->setCredentialTreatment('password(?)');
```

That would force the adapter to use the MySQL password function instead of just clear text password. The default for `credentialTreatment` is `?`.

Moving on, now comes the point of all this, the actual authentication. Calling `authenticate()` actually does the deed. In our case, it builds the SQL necessary to check that the email address and password combination is valid and then executes it. Other authenticators will execute the code necessary to validate the identity and credential columns. This will return to us a `Zend_Auth_Result`. With the result, we can glean several different things like error codes and error messages when things went wrong, or the value of the identity column. In our case, we are looking for a simple pass/fail so we are ignoring the rest. In a real application however, you would want to check the errors and throw an exception should things go horribly wrong.

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

The final step in our login process is to actually load the `Member` class if the member validated. Since we are using the `Zend_Auth_Adapter_DbTable` the adapter knows how to retrieve the rest of the record for us using `getResultRow(). getResultRow()` takes two parameters, a list of fields to include or a list of files to omit. To allow you to pass in multiple fields in either the allow or deny parameters, you can pass in simple arrays instead of strings. In our case, we really only need the id so we pass it in as the first parameter as a string. We send back the id as the return parameter or *false* if the credentials did not validate.

```
            if ($result->isValid()) {
                $memberArray = $authAdapter->getResultRowObject('id');
                $returnValue = (int)$memberArray->id;
            } // if ($result->isValid())


            return $returnValue;

    }
```

That's all there is to authenticating, as you can see, it's pretty easy. Those of you paying attention will realize that it's not really much simpler than what we had originally, a SQL statement querying the database; this is true. However, this code is

more portable and flexible. `$result` is automatically stored in the session namespace `Zend_Auth` for us. This means that once authenticated, we can check it at any point. Also, should we desire, we can switch authentication adapters by changing a single line. We could just as easily use `Digest` to `HTTP`, or write our own that does LDAP or something custom.

Now that we are using `Zend_Auth` to actually log the user in, let's do something with it. I won't reproduce the entire `HistoryController` here, you can unpack `example4.zip` and see it. For the most part, it looks like the code we've done in the other controllers. Nice, neatly formatted, well documented, and oh yes, functional. Here's the fun bit. We want to limit the `HistoryController` actions to people who are logged in. We could of course, put a check in each action to make sure, however, it's easier to just put it in the `init()`.

```
public function init()
{
  parent::init();
  if (!$this->view->member) {
    $this->_helper->flashMessenger->addMessage("You must be logged in to access
        that page.");
    $this->_redirect('/index/index');
    return;
  }

  return;
} // public function init()
```

In the above code, we use concepts we've already discussed to simply check to see if there is a member object in the session. If there is then we've got a valid login.

That's it. That's all it takes to implement a flexible and robust authentication system in your application. In most real-world applications, you are going to want to take it a bit further than the simple pass/fail we've done here. So here are some things to know.

While we talked about `Zend_Auth` using `Zend_Session` for persistent storage, we did not actually implement that. Since we were creating our Member object anyhow, it wasn't necessary. However, in your applications, you may want this functionality. To get this, you can't bypass `Zend_Auth`. Our sample code, using persistent storage, would look something like this.

```
$auth = Zend_Auth::getInstance();
$authAdapter  = new Zend_Auth_Adapter_DbTable($db);
$authAdapter->setTableName('member')
      ->setIdentityColumn('emailAddress')
      ->setCredentialColumn('userPassword')
      ->setIdentity($emailAddress)
      ->setCredential($userPassword);
$auth->authenticate($authAdapter);
```

If you are paying attention you'll notice that we used the `Zend_Auth->Authenticate` method instead of the one in the adapter. This will create a `Zend_Auth` namespace in the session and store the `Zend_Auth` object in it. This would allow us to execute code like this to retrieve it.

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

The `Zend_Auth` automatically knows how to access its storage if it exists. Also, if you are using the `Zend_Auth` persistent storage and the identity to determine logged in status then you will want to know about `clearIdentity()`. Yep, it does just what you think it would do. Call this in your `logoutAction()` like this:

```
$auth = Zend_Auth::getInstance();
$auth->clearIdentity();
```

Summary

We talked a lot before we actually wrote any code and honestly, the code we wrote wasn't all that exciting at first glance. However, if you step back and consider the possibilities I think you'll begin to understand why `Zend_Auth` is much better than just hacking together a quick SQL statement.

`Zend_Auth` is flexible and when building applications, that's important. In our very simple application, it was easy for us to use SQL to authenticate; most real-world ap-

plications are not that simple. `Zend_Auth` gives us the option to use database, digests, or roll our own, all with a common interface. It's even possible to use `Zend_Auth` in association with `Zend_Service` to authenticate against a web service on another server or domain.

The automatic persistence is important as well. While we did not use it in our simple application, most of the time in your real world applications you will. You may not choose to persist using a session but the great thing about `Zend_Auth` is you can easily tell it to use your own custom storage mechanism. This opens the door to anything from basic file storage to database persistence to more exotic schemes like in memory or even memcached persistence. Your imagination is really the only limiting factor.

Chapter 8

Super Secret Ninja Class: Globals.php

This is it, this is the chapter of the book you've been waiting for. I'm going to divulge to you a secret concept known only to me, the wise sage who showed me, any anyone else who reads this book.

In MVC, there are times when things just don't quite fit. You can either cut and paste a lot of code and violate the sacred principal of DRY or you can sit back and find a way that makes sense and does not violate any tenets of Zend Framework; I'm going to take the second path. A good example of this is our database connection. Up to this point, I have purposefully put the database connection code close to where we are going to use it. In the first database example, this involved cutting and pasting the connection code every time we needed a new connection. Obviously that's not a good idea.

Next I moved it into the `BaseModel.php`. This worked okay for our models, however we run into the problem that if anything other than a model needs to connect to the database it doesn't have access to the connection. We can either go back to cutting and pasting or we can kludge something together. I chose the second path, the kludge. I made `BaseModel::getDb()` a static method so that anything that included the `BaseModel.php` could get a db connection. This however, is not the purpose of `BaseModel`. In larger applications, this can start to be a problem, especially in

a team environment. We need a more permanent solution. The one that I now use is `Globals.php`.

Setting Up Globals.php

To be able to fetch many common items in the application from a single place, without repeating ourselves, we will need to put it in a common place. Here's where I will deviate a little from the standard Zend Framework directory structure. We need a place where we can place items that are related to the application but not a model, view, controller or module. Since what we are dealing with is mainly configuration options, let's create a directory under `\app` called `config`. Our directory structure should now look like this:

```
C:\web\ <-- Web Server Root
     htdocs\ <-- web application root
     app\ <-- ZF based application
       controllers\
          models\
           views\
           config\ <-- our new dir.
     www\ <-- the web root.
```

Now in our new directory place a file named `Globals.php`. `Globals.php` will become the second most important file in the application, behind only the bootstrap. Here's the code for the first version of it.

`Globals.php` is never instantiated. It is a static class in that it only has static properties and methods. Building it this way has two clear advantages.

- We never have to check to see if it's already been instantiated and in the session somewhere. If it's included (we'll take care of that in a minute) then it's available.
- It allows us to use it as a factory for the various helpers we will need.

Ok, so, let's look at some code. For those of you reading this electronically, get those cut and paste fingers warmed it. For this first iteration we only need the `Zend_Loader`

and the basic `Zend_Db`. Later on we will add additional classes as our little `Globals.php` grows up.

```
require_once 'Zend/Loader.php';
Zend_Loader::loadClass('Zend_Db');

class Globals
{
```

This is a storage property for an instance of the `db`. Since there is very little reason to create a new instance each time we need one, as in any given page we may need to access it in multiple places but never need multiple instances. Therefore, the first time that the system asks for an instance of the database connection we create it and store it here. Subsequent requests return the instance stored here.

```
  private static $_db = null;
```

This is the method that actually creates or returns the instance of the database connection.

```
  static public function getDBConnection()
  {
```

First and foremost we make the check to see if we have already created a connection to the database. If we have, just return it, if not, proceed to step 2.

```
    if (self::$_db != null) {
      return self::$_db;
    } // if (self::$_db != null)
```

Welcome to Step 2. Here we actually create the `db` using the same method we've been using elsewhere. However we create it into our static property for later retrieval.

```
    self::$_db = Zend_Db::factory('Pdo_Mysql',
                  array ('host'   => 'localhost',
                     'username' => 'example',
                     'password' => 'example',
```

```
                    'dbname'   => 'example3'));
        Zend_Db_Table::setDefaultAdapter(self::$_db);
```

Finally, we return the `db` instance we just created and stored.

```
        return self::$_db;
      }

    } // class Globals
```

Now lets look at how to retrofit the current application to use the new `Globals.php` class. On that front, there is good news. There are only five files in the application that instantiate an instance of the database connection. Let's look at one of them and then I'll tell you where the rest are. First though, `Globals.php` needs to be accessible to the entire application. (Couldn't very well call it `Globals.php` if it were not available everywhere, could we?) We could go around adding `require_once` in several key places and hope we got them all, however, the easiest place to add it in is in the bootstrap.

So load up your bootstrap file (`www/index.php` if you are jumping around in the book and not reading it in the order I intended you to). Find the other `require_once` lines and add this one to the list.

```
require_once 'config/Globals.php';
```

There, now you can access `Globals::*` whenever and wherever you want. Now, let's just figure out where we want. The easy thing to do would simply be replace the method in `BaseModel` and be done with it. That actually would work but realistically, that method doesn't need to be there and someday soon it will come back to haunt us because it is. So the first thing we do is remove `BaseModel::getDb()`. Now, let's look at `Member.php`. We have to find the calls to `getDB()`.

About line #26, if you've been cutting and pasting, you will see the line:

```
$db = $this->getDb();
```

Replace that line with:

```
$db = Globals::getDBConnection();
```

I know, I know, you are thinking that it can't be that easy; yes it can be. Now find the rest of the `getDb()` lines and replace them. To help you out, here's a list of the files they are in:

```
controllers/HistoryController.php
models/WebProperty.php
models/WebPropertyAnalysis.php
```

That's it. We have now converted the application to run with our `Globals.php` and moved our database connection code into it's final home. Now let's see what else we can do with this new toy.

Using Globals.php with Zend_Cache

Outside of security, the most important aspect of any web based application is performance. It's one thing to code for a small intranet application that will only be used by 10-50 people. That's not going to be taxing on any server you run it on. However, when your shiny new web 2.0 application goes online and thousands of people realize what a genius you really are, performance is going to be a real issue in a hurry. All of a sudden you are going to be looking for ways to do more at once, do things faster and off-load processing. One common strategy is *caching*. If you've been working in programming more than 10 minutes, in most all programming cases, it means storing the results of some processor intensive processes for reuse. The easiest one that comes to mind is the results of a SQL query. If your home page has a SQL query that takes two seconds to execute, having 2 users and causing that to execute once per minute is not a big deal. However, having 1,000 users and causing that query to execute 500 times per minute is a big deal. Especially if in that minute, the data is not going to change. If the data isn't likely to change then you've needlessly hit your database 499 times. When your numbers start getting up to 10,000 users and higher, all of a sudden, this is a serious problem.

Lucky for us, this is an easy problem to solve using `Zend_Cache`. `Zend_Cache` is a simple frontend for caching. By simple, I mean simple to implement and easy to

use. The magic that happens behind the scenes is anything but simple. To keep this at a level that everybody can use, we will only be using some of the simple options.

One of the biggest benefits of `Zend_Cache`, outside of enhancing the performance of your application is that it can use multiple backend storage devices. To keep things simple, we are going to be using file based storage but it comes with backend drivers for;

- File
- SQLite
- MemCached
- APC
- Zend Platform

Like everything else in Zend Framework, you can always extend it if you need a storage device that is not listed. Since the API is common among all devices, you can easily develop using "File" and then switch to APC or Zend Platform for production.

Before we dive into the code, let's take a look at the different options we have to play with.

`Zend_Cache` is divided into frontends and backends. The methods and properties available on each frontend differ based on the actual frontend you are using. Which frontend you use will depend upon what you are actually trying to cache. Zend Framework ships with the following frontends:

- `Zend_Cache_Core`. `Core` is the most basic caching. This is the one we will be using. You can store any serialized string. This is by far the most common use of `Zend_Cache`.
- `Zend_Cache_Frontend_Output`. `Output` uses PHP's output buffering to capture the output to be cached. Because of this, `Output` is the easiest cache frontend to use. With a few lines of code, you can easily begin caching output in existing sites. Output can be used to cache specific portions of a page.

- `Zend_Cache_Frontend_Page`. Similar to `Output`, `Page` is used to cache the output of a page. Unlike `Output` though, it will cache the entire page instead of selective portions of the page. You don't really want to put too much code above the check for the cache. You really only want the code necessary to instantiate the cache. If the page is in the cache then everything will be returned form there. If not, then you will continue your normal page coding.

- `Zend_Cache_Frontend_Function`. `Function` is used to cache the output of a function. When setup properly, you make the function call to your cache controller instead of your user defined function call. If the output is in the cache and still fresh, the output is returned from the cache, otherwise, all of the parameters are passed to the original function and the results are then cached. It takes a bit more to implement this but for functions that are time intensive, this is the best way to cache their results.

- `Zend_Cache_Frontend_Class`. `Class`, as it's name implies, caches an entire class at a time. It differs from `Function` in that it allows calls to be made to static methods as well as normal ones. Other than that, it acts similar to `Function` in that the cache controller stands in proxy of the actual class. If the results can be returned from cache then they are, if not, the class method is invoked and the results are cached.

- `Zend_Cache_Frontend_File`. `File` uses the modification time on the master file to determine whether the cache is fresh or not. The example that the manual gives is a good one. If you have a configuration object that is populated by parsing an XML file, you could use `File` to store the resulting config object. This would save you re-parsing the XML each time until the config file itself was changed.

While we have fewer default choices on the backend, they provide you with the basics that most any web application will need to cache data.

- `Zend_Cache_Backend_File`. This is the most common and the one we will be using in our examples. You provide `Zend_Cache` with a directory to use to store cache files. Each time you cache something, it is stored in a file in that directory. This is the easiest to understand and use; however is it not the best choice on high traffic sites as your file system may become the bottleneck.

- `Zend_Cache_Backend_Sqlite`. `Sqlite` is an interesting choice for a backend, I'll have to admit. In a lot of cases, what is being cached is database output. Storing the cache in another database, even a lightweight one, doesn't save you that much. However, it is available to you and should you want to use the concepts employed in it, you could easily write a MySQL backend for storing long-term pages, etc.

- `Zend_Cache_Backend_Memcached`. Now we start moving into the really high performance backends. The `Memcached` backend utilizes your existing `memcached` server to store caches. Since `memcached` stores everything in memory, this will give you much better performance than `File` at the cost of complexity.

- `Zend_Cache_Backend_Apc`. Alternative PHP Cache as it's name implies, utilizes the native caching mechanism in APC to store your cache. This is an excellent high-performance choice if you already have APC installed.

- `Zend_Cache_Backend_ZendPlatform`. Since Zend Platform is built by Zend, you would expect tight integration with it. I'm glad that Zend put support for other high performance options like `memcached` and APC into the `Zend_Cache` as well. For production servers, this (of course) is my favorite. It's high-performance, easy to manage and tightly integrated. The only downside is the same one that all of the high-performance ones suffer from, more moving parts.

For our little demo application, the `Core` frontend and the `File` backend make a great pair. You will get a feel for the concepts involved but not get bogged down in the details of the more complicated pieces.

Before we begin though, we need to look at three key concepts that are important.

- Key Identifier. It does you no good to store something in the cache that you can't retrieve. To retrieve it you have to give it a unique identifier. This is used by the backend to identify the cache and make it retrievable. Your key identifier can be anything you want but it has to be unique and it has to be a string.

- Lifetime. Each item stored in the cache has a lifetime. The lifetime of a cache, always expressed in seconds, determines how long the cache is considered fresh. Cache's are not automatically discarded after their lifetime. However,

the next time a request comes in for an expired cache, Zend Framework will refresh the cache and start the clock over again.

- Conditional Execution. The `Zend_Cache_Core::get()` method returns a *false* on a cache miss (no cache or expired cache). This allows you to wrap your code in an `if()` statement. Your code only executes if there was a cache miss. As you will see in our examples, this is an important concept for us as we are caching SQL results.

Ok, enough talk, let's start implementing. The first thing we need to do is add code to `Globals.php` to return our cache controller. Since we don't really know when and where we will use it yet, it's best just to handle it in `Globals.php`. (Besides, it wouldn't make much sense to put it in this chapter if it weren't going in `Globals.php`). So here's what we need to add to `Globals.php`.

Fist, let's include the `Zend_Cache` class.

```
Zend_Loader::loadClass('Zend_Cache');
```

Next, we need a container for the cache controller.

```
private static $_cache     = null;
```

Place that just below the `private static $_db` line.

Now, we need the static function. Place this below the `getDBConnection()` function.

```
static public function getCache()
{
  if (self::$_cache != null) {
    return self::$_cache;
  } // if (self::$_db != null)

  self::$_cache =  Zend_Cache::factory(
    'Core',
    'File',
    array('lifetime'=>600, 'automatic_serialization'=>true),
    array('cache_dir'=>"c:\web\htdocs\cache"));

  return self::$_cache;
}
```

As you can see, we are using the `Core` frontend and the `File` backend. The first array we are passing in is the array of frontend options. We passed in a lifetime of 600 seconds (10 minutes for those of you not yet awake) and the `automatic_serialization` parameter. This way the `Cache` object will handle the serialization for use relieving us of one more task. The second array is the array of backend options. The `File` backend takes several options but the only one we want to change is the `cache_dir`.

A lot of developers have something generic like `tmp` that they stuff everything into. That's a good idea of you get charged by the directory. However, since most hosts don't charge by the directory, it's a good idea to segregate your different working files by directory. In most applications I build I have a directory for sessions, one for temporary work space, one for the cache and if I'm working with a templating engine like Smarty, one for the template cache. Do yourself a favor and segregate your files. It comes in handy when you want to blow away just one type of files. (i.e. in our case, the cache).

Now, we've got the plumbing installed, let's hook it up. For our sample application we are only going to cache one thing, the list of searches we've already done. So, open your `HistoryController` and let's dive in.

```
public function listSearchesAction()
{
  $cache = Globals::getCache();

  if (!$results=$cache->load('searchList')) {
    $db = Globals::getDBConnection();

    $sql = 'select wp.id as wp_id,
              wpa.id as wpa_id,
              wpa.search_date,
              wp.url
            from web_property_analysis wpa,
              web_property wp
           where wpa.web_property_id = wp.id
           order by url,
                search_date;';
    $results = $db->fetchAll($sql);
    $cache->save($results,'searchList');
    $this->view->history = $results;
  } else {
```

```
        $this->view->history = $results;
      } // if($results = $db->fetchAll($sql))
    } // public function listAction()
```

Now as you can see, most of that is exactly as we've already written. However, there are a few subtle changes. The first line for example; in order for us to do anything with the cache, we need an instance of the `cache` controller.

```
$cache = Globals::getCache();
```

Next, we attempt to load the last results from the cache:

```
if (!$results=$cache->load('searchList')) {
```

The parameter for `$cache->load()` is the name of the cache or the *cache key*. In this particular case, the cache key is pretty simple, it's a description of the select statement we are caching. However, you will quickly realize that the cache key can be tricky. The cache key has to be unique throughout the application. Once you get into larger applications, this is going to be more difficult. In many cases, when I'm caching the results of a SQL statement, the easiest thing to do is to take an md5 hash of the SQL statement. This makes for an ugly but unique and easy way to reproduce a cache key.

The `if` statement above will fail for two reasons. First, if the SQL results have not yet been cached, of course it will fail. The second reason it will fail is if the cache is no longer fresh. (I must resist the urge to make a "expired cache's smell bad" joke here). Since we set our lifetime to 600 seconds, if 601 seconds have passed since the cache was built then `load()` will fail.

In both cases, when load fails, `!$results=$cache->load('searchList')` will return *true* (it's a double negative thing. If you still don't get it, read it again you'll have that "aha!" moment). Since the result of that is *true* if `load()` fails, we will execute the code inside of the `if()`. This will execute our SQL and save the cache results using the `save()` method. `$cache->save()` requires two parameters:

- The contents we are wanting to cache. This must be serializable.

- The cache key to use. Remember this has to be easy to recreate as we need to use it on load.

`Save` takes an optional third parameter of an array of tags, we'll get to that in a minute as it's an important concept to understand.

If perchance the cache has a hit instead of a miss, the results are unserialized into `$result`. From there we process it normally. Either way, at the end of the `if()` statement, we have a valid results array stored in `$results`.

Now, if you have been paying attention, you will recognize that the whole `if()` statement thing is an example of the third key concept of caching that we discussed a few pages ago, *conditional execution*. Ok, I know some of you were hoping that it was more difficult than that but it's really not. It means what it says, on cache miss, execute the code to build the cache. On cache hit, skip all that stuff.

Ok, I promised I'd talk about *tagging*. Tagging is an important concept in caching with `Zend_Cache`. The idea is similar to tagging found elsewhere on the web. You give a cache one or more semantic tags that are meaningful to you. Just like you would not give each post in a blog a unique tag, you don't want to give each cache item a unique tag. Tagging is a way of grouping your cache together. Really, there is only one reason you want to do this – deleting! If you tag all of your caches properly and group them logically, you can invalidate specific areas of your cache with a single command without resorting to clearing it all.

Let's take a simple example, not related to our example application but easily understood; a blog that you are caching with `Zend_Cache`. At its most primitive, a blog has posts and comments. Since comments are related to posts, it would make sense to cache them with the post. If our caching is working properly, when a comment is added, it won't show up until the cache has expired. In many cases you will want a long lifetime on blog posts and comments, but need to clear it often. You may need to clear it just after the post is made but less frequently as time goes on and the post gets buried by other posts.

There are two ways we can deal with this. First, we can, upon the post of a comment, clear the entire blog cache and rebuild them all from scratch. On a popular blog, this is as bad as having no cache at all since it's constantly being rebuilt.

The other option is to employ a cache tagging scheme that allows us to identify the portions of the cache that the comment will affect and invalidate just those parts. This is where tagging comes in. In our little blog example, we could use a tag of

post_id_12 for the cache of the post and the cache of its comments. The save command would look like this:

```
$cache->save($post,$cacheKeyPost,array('post_id_12'));
.
.
.
$cache->save($comments,$cacheKeyComments,array('post_id_12'));
```

Now both the post and the comments are tagged with post_id_12. Should either of them change, we can issue a single command and clear the relevant portion of the cache.

```
$cache->remove('post_id_12');
```

If we want to remove multiple tags at once, we can use the clean() method:

```
$cache->clean(
    Zend_Cache::CLEANING_MODE_MATCHING_TAG,
    array('post_id_12', 'post_id_13', 'post_id_14'));
```

There are also two other modes that clean can be used in. The first is to remove everything in the cache. For those times when your user clicks the "Clear the Cache" button (you don't really have one of those, do you?) or you feel that everything needs to be rebuilt, call clean with the Zend_Cache::CLEANING_MODE_ALL parameter and the entire cache will be deleted.

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

Additionally, you can tell Zend_Cache to physically remove all the files in your cache that have expired or been invalidated.

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

A good tagging scheme can make your cache a joy to work with. A bad one will cause you more pain than it's worth. When you begin to work with caching, stop, take some time and think through your tagging scheme. Trust me on this one, it's time well spent.

The one final thing we need to do with our caching system is to invalidate our cache if we add a new search. We will use the `clean()` method described above and the tag that we specified, even though we only have a single cache at the moment. You must resist the urge to clear the entire cache unless you've got a real good reason to. So, the HOW in this case is easy:

```
Globals::getCache()->remove('searchList');
```

That's to the magic of fluent interfaces, we can do it all in one line. The real question is WHERE. `WebProperty->save()` is the obvious answer. Since any change to any web property would mean that our cache is out of date, we need to invalidate it at that point. So we open `WebProperty.php` and place this as the last line of the `save()` method. Now let's go test it. The easiest way to test for a cache hit or miss is to put a simple echo statement in the `IF` statement in the `HistoryController.php`. This however, is so very unprofessional. We could also deploy the `Zend_Log` but that's really overkill for what we need. No, the best thing to do in this particular case is just to watch the cache directory. If you've already tested the caching then the cache file will already exist. If it's been more than 10 minutes since you tested it then you will have an invalid cache file in your cache directory. So, using the magical incantation for whatever OS you are using, navigate to the cache directory.

Now fire up a browser go to the main page and analyze a URL. If you had a cache, this action should have removed it. That's the easiest way to invalidate a cache; totally get rid of it. Now you know that the next time you execute a search, it rebuilds the cache. Since we've mentioned it, let's do that. Log in if you haven't already and then click on the "List Searches" link. If everything is working as perfectly as it should, you should now see a new cache.

When it comes down to it, that's really all there is to caching, each call to the cache is either a hit or a miss and you need to act accordingly; and when the data changes, invalidate your cache.

Storing Global Configuration Values

The final piece we will implement in our super secret Ninja class, `Globals.php` is a way to store global configuration items. Things like directory names, database credentials, or really anything that you don't want to hard-code into your application. Those are the things that you want in your `Globals.php` so you don't have to remember where they are or how to access them.

Another reason you don't want to hard-code these items is that in most large scale applications, we have multiple environments. Development, Staging, QA and production; each of these may have slightly different variations of things like directory names or database credentials. You want to be able to deploy your codebase once and just monkey with the config file.

You could, without any further changes, add all your properties to `Globals.php` as "static public", hard code their values into the class and be done with it. I'll tell you from experience, as your `Globals.php` class grows and your team grows, managing your `Globals.php` will become a pain. Not only do you have people adding code into it as your needs grow but since each environment can be different, (and thus the values can be different), you end up with a mess. The easiest way to deal with a mess like this is to use `Zend_Config` to store all of your values. This allows us to only have code in the `Globals.php` and still use it to manage our global parameters.

Out of the box `Zend_Config` comes with adapters to handle config data stored in INI or XML files via `Zend_Config_Ini` or `Zend_Config_Xml`. Additionally, you can store data in a native PHP file as an array. If you choose to do this then you use the `Zend_Config` class to load it in.

Now I'm partial to INI files so we are going to use the `Zend_Config_Ini` class. However, the concepts we discuss here can easily be translated to use `Zend_Config_Xml`. `Zend_Config_Ini` supports the commonly accepted INI file format:

```
[header]
key1=value1
key2=value2
key3=value3
subhead.key1=value4
subhead.key2=value5
subhead.key3=value6
```

If you load `Zend_Config_Ini` passing in the "header" parameter then you would get `key1=value1`.

```
$config = new Zend_Config_Ini('/path/to/config.ini', 'header');
echo $config->key1; // will print "value1"
echo $config->subhead->key1; // will print "value 4"
```

Additionally, the INI file format supports inheritance.

```
[header]
key1=value1
key2=value2
key3=value3
subhead.key1=value4
subhead.key2=value5
subhead.key3=value6

[header2 : header]
key1=value7
key2=value8
key3=value9

$config = new Zend_Config_Ini('/path/to/config.ini', 'header2');
echo $config->key1; // will print "value7"
```

This is useful when the same config file can be used for say, production and staging. However, if you are deploying the application to three different developers workstations, each of them will have their own ideas about how the directory structure above the application should be. They could also be on different operating systems, this would cause them to be very different. You could, of course set special headers for each and every environment you are deploying to; production, development, dev-matthew, dev-elizabeth, etc. However, as people come and go, you will find that very cumbersome. I've found that the easiest way to keep things clean is to have an overall INI for production and staging and allow each developer to create his or her own INI for their development environment. Doing so solves many problems and keeps decrees from "on high" as to how systems are to be laid out to a minimum.

Let's dive back into the code. The first thing we need to do is modify our `Globals.php` to manage the `Zend_Config` class. Add this line near the top of `Globals.php` with the other loaders.

```
Zend_Loader::loadClass('Zend_Config_Ini');
```

Next, we need a place to put our instance of `Zend_Config_Ini` so we don't have to keep recreating it.

```
  private static $_db     = null;
  private static $_cache  = null;
  private static $_config   = null;
```

Finally, and by now, you should be able to do this one on your own, we need to put the `getConfig()` function in.

```
public static function getConfig()
  {
    if (self::$_config != null) {
      return self::$_config;
    }

    self::$_config = new Zend_Config_Ini(
      dirname(__FILE__) . DIRECTORY_SEPARATOR . 'config.ini',
      null);

    return self::$_config;
  }
```

`Zend_Config_Ini` takes 3 parameters:

- File Name of the INI file
- The of the section to load. In this case, we passed in NULL so that all sections would be loaded. If we had a segmented INI file, we could have passed in a string with the name of the section to load or an array of strings that correspond to multiple sections in the INI file.

- Optionally, we can pass in either *true* or *false* to answer the question, allow the program to modify the values. The default is *false* and since in our example, we don't want to allow the program to make changes, we don't override it.

The file name we pass in is:

```
dirname(__FILE__) . DIRECTORY_SEPARATOR . 'config.ini'
```

If you understand this then skip this little paragraph. If not, here's what that resolves to. At some point we have to make an assumption to get our bootstrap going. Just as we assume the bootstrap file will be named `index.php`, we will assume that the config file is named `config.ini` and it's in the same directory as `Globals.php`. These are safe assumptions for most of us, however if you ever want to change that, you'll have to figure out where to put it on your own and modify this line. So the `dirname(__FILE__)` resolves to the full path and directory name that `Globals.php` is located in. `DIRECTORY_SEPARATOR` is a PHP constant that holds the appropriate value to separate directories on the OS you are working on. (`\` for windows, `/` for Linux and OSX.) Feel free to hard-code this but if you ever deploy on another OS, you may have problems. I always find it's best to stick with the constant.

The rest, you've seen before. So, now that we can use a `config.ini`, we need to actually make one. For our sample, let's use this one:

```
[main]
title = "Keyword Content Analyzer"
db.host = "localhost"
db.username = "example"
db.password = "example"
db.dbname = "example3"
dirs.cache = "c:\web\htdocs\cache"
```

There is no secret naming convention that has to be followed. In this case, I used the names of the parameters I will be filling.

This now gives us a good working base for our example. Feel free to parameterize any other properties in the example you wish. The above is the contents of the `config.ini` in the example code.

Now, let's implement it. Where shall we begin? How about in `Globals.php` itself? That's where most of the values are hard coded anyhow. Let's start with the database.

```
static public function getDBConnection()
{
  if (self::$_db != null) {
    return self::$_db;
  } // if (self::$_db != null)

  self::$_db = Zend_Db::factory('Pdo_Mysql',
     array ('host'   => self::getConfig()->db->host,
        'username' => self::getConfig()->db->username,
        'password' => self::getConfig()->db->password,
        'dbname'   => self::getConfig()->db->dbname));
  Zend_Db_Table::setDefaultAdapter(self::$_db);
  return self::$_db;
}
```

It's just that easy. Since we are storing the instance of `Zend_Config_Ini` the first time we call `getConfig()` there is no performance penalty to making multiple calls.

Now, let's update our `getCache()`:

```
static public function getCache()
{
  if (self::$_cache != null) {
    return self::$_cache;
  } // if (self::$_db != null)

  self::$_cache =  Zend_Cache::factory(
      'Core',
      'File',
    array('lifeTime'=>600,
      'automatic_serialization'=>true),
    array('cache_dir'=>self::getConfig()->dirs->cache));

  return self::$_cache;
}
```

As you can see, we could have easily parameterized everything in the cache, but really, for our example, the only thing that is going to be a pain for you is the `cache`

directory. This is only because, you may not put all your projects under `c:\web` like I do. Now you can move things around to suit your development style.

Beyond the `Globals.php`, there is one other place we can use our new config toy. The first parameter we stored is the title of the application. Right now it appears in all of the template files. Let's change it in `index.phtml`.

Open `view\scripts\index\index.php` and find the `<title>` tag. Change it to this:

```
<title><?PHP echo Globals::getConfig()->title;?></title>
```

Save it and refresh your browser. Didn't change at all did it? Great! Now, go edit your `config.ini`. Put your name in there for all to see. Something like:

```
title = "Cal's Keyword Content Analyzer"
```

Now save the `config.ini` and refresh the browser. You should now see a change. See isn't it fun when things work?

Going back to the line we changed in `index.phtml`, you should have noticed that unlike the changes to `Globals.php`, this time we used `Globals::getConfig()` instead of `self::getConfig()`. That's the only difference. Anytime you want to make a call outside of `Globals.php` itself you have to use the static class name.

Summary

I hope you've picked something up in this chapter. The concepts we've discussed will go a long way to making your project easier to manage and much more flexible. It may not be a super secret ninja trick to use a static class as basically a "global" but it is pretty cool.

Chapter 9

Web Services

No respectable book on developing web applications would be complete without a chapter on web services. Most of the time, web services are handled late in the book and in a lot of cases, I've felt that they were "bolted on". This chapter, while late in the book, should not be considered bolted on, primarily because our sample application is already a web services consumer. As a matter of fact, it's built around consuming a web service. If you didn't figure this out a long time ago, the Yahoo's Term Extraction API is a web service. So instead of just bolting something on here, I'm going to do my favorite thing and hook Yahoo's Term Extraction API up to Flickr (`http://www.flickr.com`) and see what comes out.

Introduction to Flickr's API

Outside of Google Maps, I think Flickr has to be my favorite web service. If you can boil any content down to a word or a phrase, you can usually get a hit on it on Flickr. This adds a new dimension to just about any application. Note, the pictures you get from Flickr are usually not directly related to your application and sometimes, they can be "not safe for work", so be careful when playing around with Flickr.

Before we dive into the code, let's talk a bit about web services and Zend Framework. *Web services* are basically any data or functionality that is exposed via a web interface. If you are new to web services, I recommend you check out `http://www.programmableweb.com` for a taste of what is available. Web services are not

always free like the ones we are using. Some companies charge for access to their data and services, as is their right. Make sure, before you start building the next great mashup, that you know the services you are dealing with.

Web services come in several flavors; XML-RPC, SOAP, REST, and of course, CUSTOM. (Custom is usually those people who didn't feel like adhering to an existing standard so they just hacked something together and insist that it's better than anything else out there.) For our demonstration, we will be using *REST interfaces* because they are by far the simplest. (REST stands for Representational State Transfer and there's a good article on Wikipedia about it if you want more info. `http://en.wikipedia.org/wiki/Representational_State_Transfer`). REST can be, if you want it to, complex. However, for our purposes, we can define REST as "we call a URL, it returns to us XML". You can already see that in the call to the Yahoo! API we are using.

Zend Framework has a growing collection of APIs that are supported with classes in the framework; one of them is Flickr. You have already seen `Zend_Rest_Client` at work when we wrote our `IndexController`. You know how easy it is to call a simple web service and process the results. In many cases though, the APIs are not as simple as Yahoo's Term Extraction API. Many times there is authentication to deal with and a myriad of methods to figure out. As you can see from `http://www.flickr.com/services/api/`, Flickr has a lot of options for us to choose from. Lucky for us though, someone has already done most of the work for us and created `Zend_Service_Flickr`.

We don't really get a feel for it with this example, but one of the things I love about the way that Zend Framework handles APIs is that all properties become properties of the class and any action you can take using the API becomes a method. The simple example we use with Flickr is `tagSearch()`. Normally, to actually do a search using the Flickr API you would build a valid URL appending each property/value pair to build the query before executing it. Zend Framework's `Zend_Rest_Client` handles all of this for us transparently. We can use simple properties to build the query and call a method of the object to actually make the call and fetch the results.

In our case, as you will see, we instantiate an instance of `Zend_Service_Flickr`, passing in our API key as the constructor parameter.

A word here about API keys – get your own. You can undoubtedly use the one that I use in the sample; however, all that will do is get it canceled. In this

case, it's no big deal because I registered it just for the book. However, Flickr hands them out for free and makes them very easy to get, so get your own. Visit `http://flickr.com/services/api/keys/` for more information.

Once created, we can simply call `tagSearch()`, passing in an array of parameters that we want passed on to Flickr and what we get back is a `Zend_Service_Flickr_ResultSet`, a beautifully object oriented representation of the XML that Flickr handed back.

Integrating Yahoo! and Flickr APIs

As I said, any content you can boil down to a term, word or phrase, can be cross-referenced to Flickr and in a lot of cases, you will come up with a picture; sometimes it will even match the subject. Since we've boiled web pages down to some terms, let's take them and ask Flickr for the most interesting picture it has for each of these terms. Then we will display that.

This does pose an interesting problem of where to put the call to Flickr. The immediate response would be to put it in `WebPropertyAnalysis.php`. Under normal circumstances, you would be right. We could add a field to the table to store the URL and persist the data that way. However, Flickr changes often. Given this, I don't really want to persist the picture's URL in the database. That having been said, as much as I like Flickr, it's a relatively slow API. Too many calls to it will drag your application's performance into the tank. So we want to be judicious in our calls. We don't really want to call it each and every time we have a term.

The answer? Our good friend `Zend_Cache`! We will make a call to Flickr and if we get a response back, we will cache it. In this case, we want to cache it for a while but not forever so I arbitrarily chose to cache Flickr URLSs for 3 days before asking for a new one. This seems to be a good tradeoff between our opposing forces. Feel free to adjust it to suit your needs though.

Now that we've decided exactly what we are going to do, the question still remains where we do it. There are actually several places we could make the call, no one more right than the other. I decided to put this in `WebProperty.php`. Since it marshals the terms, it seemed as good a place as any to do it.

Let's look at the code we add to `WebProperty.php`:

```
public function fetchImageTag($word)
{
```

The first thing we do is get an instance of our cache controller. Also, we need to set our cache key. This is how we will identify our cache. Up to this point, we've used simple phrases for the cache. This works in some cases but here we need something that we can build from the term passed in. Even though the variable name is `$word`, it can actually be a phrase. The `Zend_Cache` has a strict policy of alpha-numeric characters and the underscore. In this particular case, we could use a simple regex to strip out all of the unusable characters but it's just easier to take an `md5` hash of it. `md5` only uses characters we can use in our filenames. It does have the downside of creating some ugly file names for file based caches but then again, who really reads those things anyhow? In this case, we also add a qualifier. Since we don't know that other developers might be caching other things using the same basic schema, we can ensure uniqueness (within reason) by adding `keyword_image` to the word or phrase before calling `md5()`.

We build the cache key here for a reason; we can potentially use it twice, loading and/or saving. To make sure that it's the same thing both times, we build it before we use it.

```
$cache = Globals::getCache();
$cacheKey = md5("keyword_image".$word);
if (!$results = $cache->load($cacheKey)) {
```

Now we have to contact Flickr and see if they have a picture for this phrase.

```
$flickr = new Zend_Service_Flickr(Globals::getConfig()->flickerKey);
```

`tagSearch()` takes two parameters. The first is the term we are looking for and second, an array for key=>values to pass to Flickr. In this case, we really only want one picture and we want the one that has been flagged as the most interesting. So we tell that to Flickr. If it has anything, we will get back a `Zend_Service_Flickr_ResultSet`.

```
$flikrResults = $flickr->tagSearch(
   (string)$word,
   array('per_page' => 1,
```

```
        'page'   => 1,
        'sort'   => 'interestingness-desc'));
```

Since it's not really germane to the example, I won't go into all of the methods and parameters that `Zend_Service_Flickr_ResultSet` gives us. The two that we are going to be interested in are `valid()` and `current()`. `valid()` returns a *true* or *false* telling us if we actually got a valid result set back. If we did, `current()` will return to us an instance of the current picture. The current picture gets us access to the URL of the medium version. If you are going to be working with Flickr much after this example, I strongly urge you to read up on `Zend_Service_Flickr_ResultSet`. It's a powerful tool and makes working with Flickr extremely easy.

If we do have a valid result then we build an image tag. This is one of those places where I say "Do as I say, not as I do." Best practices dictate that the class isn't the best place for us to be generating HTML. It would be better in a real application to simply store off the URL and let the view handle generating all the HTML. If we don't have a valid result, we still initialize `$results` but to an empty string. This way the view script can still execute properly and we can use it as a signal not to cache (why cache an empty string? Someone may upload a picture tomorrow that matches your term and you wouldn't know it for three more days!)

```
if ($flikrResults->valid()) {
  $results = '<img src="'.$flikrResults->current()->Medium->uri.'" title="'.
      $word.'"/>';
} else {
  $results = '';
} // if ($flikrResults->valid())
```

Now if we have results, let's cache them. Now here, we introduce a new optional parameter to `cache->save()`. The fourth parameter overrides the cache life with whatever number of seconds you put in there. Normally I would have just put in the number 259200 (do the math, 86400 seconds in a day times 3 days) but to make it easier to understand an manipulate, I just put in the formulae. If I had left this parameter off, the images would have been cached the default value we set when we instantiate `Zend_Cache`, 600 seconds or 10 minutes. That's not really enough of a cache lifetime to be effective in this instance.

As you can see I also tagged the images with the tag "images". This allows me to implement a "clear flickr cache" using the `$cache->remove()` if I want to. (I don't in this case, but it's always nice to have options.)

```
      if (!empty($results)) {
        $cache->save($results, $cacheKey,array('images'),(86400*3));
      } // if (!empty($results))
    } // if (!$results = $cache->load($cacheKey)
    return $results;
  } // public function fetchPhoto($word)
```

The rest is pretty self explanatory. We cache the image tag if we have it and then return it. This method will return one of two things, either a valid image tag or an empty string.

Now that we can call Flickr, let's implement it in our example. We need to make two changes. The first is the display of the extracted terms, the second is when we display the historical results. Let's make some changes to `IndexController.php`.

To hand the new image tags off to the view, we need to make sure that they are in our results array. In `extractAction()` you will see where we build the `$this->view->result` array, replace it with the following code. The only problem here is that we use `$this->view->result` for committing the keywords to the database as well. Since it's now an array of arrays instead of an array of strings, all we will end up with are keywords named "Array". So let's create another array of just the words that we will use to commit to the database. We'll be uncreative here and just call it `$keywords`.

```
$keywords=array();
foreach($result->Result as $item) {
  $image = $wp->fetchImageTag((string)$item);
  $this->view->result[] = array((string)$item,
                  $image);
  $keywords[] = (string)$item;
} // foreach($result->Result as $item)

if ($wp->id<0) {
  $wp->save();
} // if ($wp->id<0)

$wp->addAnalysis(date('Y-m-d h:i:s',mktime()),$keywords);
```

The second line is our call to our new method. I could have put it in the line that creates the array but for clarity, I broke it out into it's own line. Now, instead of `$this->view->result` being a simple array, it's now an array of arrays. Each sub array has a term as the `0` index and either an image tag or an empty string as the `1` index.

Next, we're going to be changing `views/scripts/index/extract.phtml`. The change here is simple. We currently use the `htmlList()` method of the view to output our array in an ordered list. Now, however we rip that out and build our own little ordered list.

```
<ol>
<?PHP
foreach($this->result as $keywordArray) {
  echo "<li>".$keywordArray[0]."<br />".$keywordArray[1]."<br /><br /></li>";
} // foreach($this->result as $keywordArray)
?>
</ol>
```

It's not exactly rocket science so I won't bother recapping it other than to say that if it doesn't make sense to you, go back and read what we did in the `IndexController.php`.

Now we're going to be changing `HistoryController.php`. The changes here are identical to `IndexController.php`. Here's the code; it goes in `listResultsAction()`:

```
$this->view->results = array();
foreach($results as $term) {
  $image = $wp->fetchImageTag((string)$term);
  $this->view->results[] = array($term,$image);
} // foreach($results as $term)
```

Finally, we're changing `views/scripts/history/listResults.phtml`. This code will look familiar if you've been following closely. It's the same code that we used in `extract.phtml`. We could have spruced this up a bit and made it look better but this is a demo and the second rule of demos is, it has to be ugly to be respected. (If you spend too much time making it pretty, nobody will take your code seriously.)

```
<ol>
<?PHP
foreach($this->results as $keywordArray) {
  echo "<li>".$keywordArray[0]."<br />".$keywordArray[1]."<br /><br /></li>";
```

```
} // foreach($this->result as $keywordArray)
?>
</ol>
```

There now, we have successfully implemented a second web API. This now officially qualifies as a "mashup". However, the other side of Web Services is the service itself. It's not enough in this life to simply be a consumer of other's goods and APIs, you need to create, to stand on your own and say, "here world, here is my data, come mash it up".

Creating Your Own Web Service

Normally, at this point, a lesser author would tell you, it's beyond the scope of this book to discuss the creation of web services. However, lesser authors are working with inferior frameworks. Zend Framework actually makes creating a web service very easy. As always though, let's talk about what we are going to build before we actually build it.

If you've been playing with the demo code as we have built the application like I have, you should have a nice collection of words and phrases built up in your database in the table `web_property_analysis_result`. If not, go analyze a few URLs and get that list built up because we are going to use this data for our new Web API.

The one thing we are collecting that may be of remote interest to anyone is a list of the more popular terms on the web. More correctly, the more popular terms on the little corner of the web we have been visiting. Now let's make a way for people to find out two things.

- What are the most popular *n* terms in our index.
- How many times a specific term appears in our index.

Now, not being Google and having the resources to build a proper index, we suffer from problems such as our index consisting of terms, not words. So if you are looking for "PHP", it may be there, then again, "php login" may be there as well. We are going to ignore that problem since I don't really expect anyone to use this code to index anything but a few simple sites.

So we now have two method calls for our API, `fetchTop` and `fetchTerm`. Let's look at what parameters we will accept.

- `fetchTop()`: only has one parameter, the value of *n*. We will refer to it as "count". So if a value of 10 is passed in for "count", we will return the top 10 terms, along with their counts, as our payload.
- `fetchTerm()`: also accepts a single parameter, the "urlencoded" term to search for. I specify "urlencoded" because many of our terms are multi-word and will have spaces or other illegal characters in them. To get them to us in a way that we can use, clients will have to `urlencode` them and we will decode it before searching.

The other decision we have to make is, what protocol are we going to use for our API? Since I have already expressed my displeasure with custom payloads and rolling your own protocols, let's stick with a simple REST API that returns a valid XML payload.

To keep things clean and easy to understand, I'm going to drop back down into "No Model" mode. The controller for our web service will contain all the code. I don't recommend that you do it this way; it works, but in reality, the concepts should be encapsulated in an object.

Let's build us a web service. The first thing we need is a controller. Whenever I'm working on Zend Framework projects that have an API component, I usually just create an `ApiController`. We figured out not only what methods we needed but also what parameters we needed already; the actual code itself is pretty simple.

```
<?php
require_once 'controllers/BaseController.php';
Zend_Loader::loadClass('Zend_Filter');
```

Like our other controllers, `ApiController` will extend `BaseController`. In this case, with the simple code we are using, there's no real reason to do this. However, we have a convention already established and there's no real reason to break it.

```
class ApiController extends BaseController
{
```

Here is something new that we've not discussed yet. Way back when we talked about the controller, we talked about the helpers and that fact that the `viewRenderer` helper will automatically pair our controller action up with the appropriate view script template and render it for us without us having to specifically tell it to. In normal circumstances, this works fine. However, in an API, the output is XML, not HTML. While it is perfectly acceptable to use `scripts/api/fetchTop.phtml` to build our XML, there's no real need to do it. We can contain it all inside our controller and thus make it easier to work on. Also, if we decide that we want JSON instead of XML, we would then have to code the logic twice. To stop the `viewRenderer` from trying to find `scripts/api/fetchTop.phtml` we have to turn off auto rendering. We could do this in each action if there were actions that were still going to use the `viewRenderer`. However, since in this case, the two actions will not be, we turn it off in the `init()`. Now this overrides our `BaseController::init()` that creates our `Member` object, however, since the API will most likely be called from a third party, that's not really an issue.

```
public function init()
{
  $this->_helper->viewRenderer->setNoRender();
} // public function init()

public function fetchTopAction()
{
```

Now, all of this code should look natural to you by now. `fetchTopAction` takes one parameter and the very first thing we do is get it, filter it and cast it. Then we get a connection to the database.

```
  $limit  = (int)Zend_Filter::get($this->getRequest()->get('limit'), '
      StripTags');

  $db = Globals::getDBConnection();
```

Here we have another new thing that we've not yet talked about. Up until now, we've always built our SQL statements by hand and executed them. Zend Framework however, gives you an object oriented interface to build your SQL. Here we use it to build our `select` statement that will fetch us the top *n* keywords from our table. The syntax

for this is pretty straightforward but for those having trouble following it, I'll describe it for you.

`->from()` takes a few parameters. You can use something simple like:

```
$select = $db->select()
       ->from('products');
```

This generates a simple:

```
select * from products
```

In our example, we need a bit more than that so we pass from two arrays. The first is the table list. In our case, we are only using one table but since the table name is pretty long, we want to give it an alias. So the `tables` array can be defined as an associative array with the key being the alias and the value being the table name.

So if we now write this:

```
$select = $db->select()
       ->from(array('wpa'=>'web_property_analysis_result'));
```

The result would be

```
select * from web_property_analysis_result wpa
```

The second array is the list of fields. In our case, one of them comes from the table, the other is derived. Since we will want to reference the derived field, we have to give it an alias. The syntax for this array is similar to that of the list of tables; an associative array where the key is the alias and the value is the field name, or as in our case, the calculation.

Of course, since we now have an aggregate function in our `select` statement, we now have to have a `group by` clause. Adding the `group by` in this simple, single-table example, is easy with the single `group()` method.

Next because we are popping the top *n* off the stack, we need to order them properly. The `order()` method takes an array of fields that will be placed in the `select` statement in the array's order. In our example below, the first field in the `order` state-

ment will be the `keyword_count` (our derived field) and the `DESC` clause will be applied to it. Next the field `keyword` will be used.

Finally, we add the `limit` clause. If you are used to dealing with MySQL's `limit` clause then you need to know that the order of the parameters is backwards. The first parameter in Zend Framework limit method is the number of rows to return. The second parameter is the number of rows to skip from the top. In our example, the finished production looks like this:

```
$select = $db->select()
       ->from(
          array(
              'wpa'=>'web_property_analysis_result'),
          array('wpa.keyword','keyword_count'=>'count(*)'))
       ->group('wpa.keyword')
       ->order(array('keyword_count DESC','keyword'))
       ->limit($limit, 0);
```

It will generate a SQL statement that looks like this:

```
SELECT
     `wpa`.`keyword`,
     count(*) AS `keyword_count`
FROM `web_property_analysis_result` AS `wpa`
GROUP BY
     `wpa`.`keyword`
ORDER BY
     `keyword_count` DESC,
     `keyword` ASC
 LIMIT 10
```

Now we execute the `select` like we would any select statement that we built by hand.

```
$results = $db->fetchAll($select);
```

Finally, we generate our XML and echo it to the browser and return.

```
    echo $this->toXML($results,'fetchTop',$limit);
    return;
} // public function fetchTopAction()
```

This function is almost identical to the one we just looked at except for minor differences in the `select` statement and the fact that the parameter is a string and not an integer. Earlier we discussed the fact that the parameter had to be `urldecoded` to work. It does, however; that is handled for us automatically in the bowels of Zend Framework. The term we get back has already been decoded and is ready for us to filter and then use.

```
public function fetchTermAction()
{
  $term  = Zend_Filter::get($this->getRequest()->get('term'), 'StripTags');

  $db = Globals::getDBConnection();

  $select = $db->select(array('keyword'))
         ->from(
            array('wpa'=>'web_property_analysis_result'),
            array('keyword','keyword_count'=>'count(*)'))
         ->group('wpa.keyword')
         ->where('keyword=?',$term);

  $results = $db->fetchAll($select);
  echo $this->toXML($results,'fetchTerm',$term);
  return;
} // public function fetchTerm()
```

Now a little sleight of hand. There is no `array2XML` function in PHP. (There really should be as it would just make my life so much easier.) This method is a "poor man's" `array2XML`. It will work for our example here but don't plan on using it for much else. It is not recursive so it won't handle arrays of arrays and some of the values are hard coded to work just for us. If you are familiar at all with SimpleXML then this will look pretty straight forward to you. If you aren't familiar with SimpleXML or XML for that matter, it really is beyond the scope of this book to do those topics justice. My advice to you is if you are interested in XML, there are some great tutorials on DevZone that cover it. If you are not, just know that it works and move on.

```
public function toXml($dataArray,$callingMethod,$parameterValue)
{

  $simpleXml = simplexml_load_string("<?xml version='1.0' encoding='utf-8'?><
       data />");
```

```
        $node = $simpleXml->addChild('results');
        $node->addChild('calling_method',$callingMethod);
        $node->addChild('parameter',$parameterValue);
        $node->addChild('row_count',count($dataArray));

        foreach($dataArray as $keywordArray) {
          $node = $simpleXml->addChild('word');
          $node->addChild('keyword',$keywordArray['keyword']);
          $node->addChild('keyword_count',$keywordArray['keyword_count']);
        } // foreach($results as $keywordArray)

        $xml = $simpleXml->asXML();
        $simpleXml = null;

        return $xml;
      }

    } // class ApiController extends BaseController
```

That's all there is to creating a simple REST web service. The hard part is figuring out which of your data or functionality you want to expose to the rest of the world.

Of course, you can also be the consumer of your API. Creating an Ajax based application that is driven by JavaScript making XHR request back to your API is also acceptable. In that case, there may be parts of your API that you don't want to allow others to use. We will discuss those techniques in a future chapter and as you will see, they are very similar to the form marker technique we have already implemented to protect our forms.

Summary

In this chapter we have talked about consuming web services as well as creating them. There are some great books out there that cover the topic in much more detail than we have here. One thing you may have noticed is that we steered clear of `Zend_Rest_Server` even though we used `Zend_Rest_Client`. The reason is simple; `Zend_Rest_Server` didn't provide us with any substantial benefit that our simple controller didn't provide. There are cases, usually when you are creating a much more complex service than we did here, when it may be beneficial. In those cases, I wholeheartedly recommend it.

All in all though, both consuming and creating web services using Zend Framework is painless. I hope it's sparked your imagination as to what's possible.

Chapter 10

Exceptions

Up until this point, we've done what every programmer does at one point in their career; write code and assume that everything will work correctly. After all, if we are finished with our code, there won't be any errors, right? (I mean it's PHP, it didn't report any errors, how could anything go wrong?) However, every programmer eventually learns the truth and the good ones take the lesson to heart and learn to handle the problems programmatically instead of just barfing bits on the screen and assuming the user knows what to do. Zend Framework handles problems via *exceptions*.

Note, if you are already familiar with the concepts encapsulated by exceptions and aren't interested in lame sports analogies, skip this section.

Exceptions: A Primer

In most modern object oriented languages, when something bad happens, or if a logic condition occurs that the programmer has decided can't be handled in the normal logic flow, an exception is thrown. Hopefully something in the code is setup to catch the exception. (Beginning to grasp the lingo here, exceptions are baseballs; we throw them, we catch them but we do not get paid millions of dollars per year for doing so.)

PHP, however, is the "exception" to the normal implementation in programming languages. PHP itself does not throw exceptions when things go wrong internally; it defaults to the old "barf bits" method of error handling. However, it does support the

concept of exceptions. It gives us the ability to throw and catch them but it will have nothing to do with them. So while exceptions aren't the end-all be-all that they are in other languages, at least we aren't left out in the cold.

So if you made it this far, in your head, you now see a baseball with the word "exception" written on it and different pieces of code tossing it around. Good, we are making process. An exception is actually an object though and it has properties and methods. The PHP manual has a great page describing the native `Exception` class located at `http://www.php.net/manual/en/language.exceptions.php`. `Exception` is a powerful class in that it can carry with it, what went wrong, an error code as well as the file and line number where it was thrown. Most importantly for us as we are building applications, it has a complete stack trace available to show us.

Zend Framework uses `Zend_Exception` and its children to handle exceptions. `Zend_Exception` is a subclass of the native `Exception`. This gives it all the power of `Exception` but identifies itself as belonging to Zend Framework. (Seriously, that's all the subclass does. It's important but if you look at the code for Zend_Exception, there isn't any!) As we will see below, we can handle different types of exceptions differently. So identifying an exception as a `Zend_Exception` allows us to handle it separately than a generic `Exception` or a "`MyApplication_Exception`".

So let's look at a simple piece of code that uses exceptions and see why they are so cool.

```
Public function thisBlows()
{
  throw new Exception('This came from thisBlows()');
  echo "This line will never execute";
  return;
}
```

Now in our main body of code:

```
thisBlows();
```

The first thing you will notice is that we have a line of code that will never execute. Once an exception is thrown, execution stops in that code block. If that codeblock is a method then control is automatically returned to the calling code block.

If you actually typed all of that in, saved it and executed it, then you have probably noticed that nothing happened. In PHP, since exceptions are not part of the native error handling mechanism, nothing happens if you just throw an exception. Like any good baseball, you have to catch it for something to happen. So let's look at catching them. The mechanism we use to catch exceptions is the *Try/Catch* construct.

```
try
{
  thisBlows();
  echo "This is another orphaned line as it will never execute.";
} catch (Exception $e) {
  echo "we caught an exception!";
}
```

Now we have successfully thrown and caught an exception! (Pat yourself on the back.) In this situation, all we did was print something to the screen. In some cases, that's all you need to do, however, the real power with exceptions is that you have the power to clean up any mess and exit gracefully when things go wrong. The easy example has to do with databases. If you use transactions when updating your database you know that there are times that you need to rollback instead of committing. In most cases, the reason you want to rollback is programmatic and not because of database failure or anything. With exceptions you can easily begin a transaction and if something goes wrong, throw an exception. Wrap all the code in a "try/catch" and at the end of the `try` block, commit the transaction. In the `catch`, you can rollback the database, notify the user, even log that you didn't update. Yes, that can all be done with a lot of creative work and some `IF` statements, but exceptions make it so much easier.

Let's look at one final example before we move on to using exceptions in our example. We discussed that we can catch different exceptions and process them separately. This would be the case, if for instance we wanted to handle `Zend_Db_Exception` separately from `Zend_Exception`. The code to do that would look something like this:

```
try
{
  thisBlows();
  echo "This is another orphaned line as it will never execute.";
} catch (Zend_Exception $e) {
```

```
    echo "We caught a Zend_Exception.";
} catch (Exception $e) {
    echo "We caught a generic Exception.";
}
```

As you can see, we can chain as many catch blocks as we need to handle all the problems we may encounter.

The native `Exception` class, along with almost all of it's children (unless overridden by a developer) takes two parameters in it's constructor. The first is the message and most of the time, this is really all you need. Something to tell someone what went wrong. The second parameter is an integer error code. If you've created `MyApp_Exception` and use it in three different places, you may want to be able to make a programmatic decision on how to proceed with the catch. In this case, you have two options. You can either parse the message and look for some key word or use the error code. Of the two, using the error code is much easier.

```
throw Exception('This came from thisBlows()',1);
```

Now, in our catch block, we can make a further determination as to how to proceed:

```
try
{
    thisBlows();
    echo "This is another orphaned line as it will never execute.";
} catch (MyApp_Exception $e) {
    if ($e->getCode()==1) {
        echo "We caught a MyApp_Exception and it came from thisBlows().";
    } else {
        echo "We caught a MyApp_Exception.";
    }
}
```

Of course this is a very simple example. As your program grows, so will your efforts to deal properly with situations that crop up that you did not anticipate and that's where exceptions shine.

A word of warning about catching exceptions; once you catch one, it's your problem; you have to deal with it. This is why subclassing `Exception` for your own use is a good idea. By creating `MyApp_Exception` and only throwing it, you can

safely catch `MyApp_Exception` and handle any problems in your own code leaving `Zend_*_Exception` and `Exception` to be handled at a lower level.

If you've been paying attention at all, some of this will be familiar to you already. That's because in our bootstrap we actually have a try/catch construct.

```
try
{
  $frontController = Zend_Controller_Front::getInstance();
  $frontController->throwExceptions(true);
  $frontController->registerPlugin(new Cal_Plugin_MemberInit());
  $frontController->setControllerDirectory('c:/web/htdocs/app/controllers/');
  $frontController ->setParam('noErrorHandler', true);
  $frontController->dispatch();
} catch (Exception $exp) {
  $contentType = 'text/html';
  header("Content-Type: $contentType; charset=utf-8");
  echo 'an unexpected error occurred.';
  echo '<h2>Unexpected Exception: ' . $exp->getMessage() . '</h2><br /><pre>';
  echo $exp->getTraceAsString();
}
```

At this level, we catch anything that is an `Exception` or a subclass of `Exception`; basically, if it hasn't been handled by this point, we deal with it in a very blunt way; we barf bits onto the screen for the user to see. This is not the best thing to do in a production system. A better solution would be to create a log file for the information and save it to your server's disk, or package it up and save it in your database and then gracefully notify the user that an error has occurred. In development, we really want the details of what went wrong but in production we don't want to bother the user with a stack trace or risk exposing sensitive data to the user. Thankfully, Zend Framework is flexible enough to allow us to have our cake and eat it too. By default, `Zend_Controller_Plugin_ErrorHandler` will attempt to handle some of the errors thrown by Zend Framework itself. Specifically, it will handle missing controller/actions as well as exceptions thrown inside an action. It will not catch exceptions thrown by plugins or within the routing.

If you want to jump ahead and just dive into the code, check out `example7.zip`.

First things first, we need an `ErrorController`. It doesn't have to be much of a controller, as a matter of fact, it can be an absolute minimum, defining an empty method called `errorAction()`.

```
<?php

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


} // class ErrorController extends Zend_Controller_Action
```

With this in place we could define our `error/error.phtml`, change a couple of settings in the bootstrap and things would work. However, that really doesn't give us much in the way of information. Sure in the `error.phtml` we can give the user a nice warm fuzzy feeling by telling them that we know there is an error and are working on it, but where the love for the developer? Let's go a bit deeper in our `ErrorController` and you'll see that we can make both camps happy.

```
<?php

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

All exceptions are logged in an object that is registered in the request called `error_handler`. This gives us a handle to the error.

```
        $errors = $this->_getParam('error_handler');
```

If the problem is that we are missing a controller or an action, let's set a "404" header to send back to the browser along with our message.

```
        switch ($errors->type) {
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_CONTROLLER:
            case Zend_Controller_Plugin_ErrorHandler::EXCEPTION_NO_ACTION:
                $this->getResponse()->setRawHeader('HTTP/1.1 404 Not Found');
                break;
```

```
            default:
                break;
        }
```

Next we, hand off the relevant information to the view for display. We do this whether or not we will display it.

```
        $exception = $errors->exception;

        $this->view->message = $exception->getMessage();
        $this->view->trace = $exception->getTraceAsString();
```

Finally, we log the error in an error log so the developers can see it.

```
$log = new Zend_Log(new Zend_Log_Writer_Stream(Globals::getConfig()->dirs->tmp.'
    Error.log'));
        $log->debug($exception->getMessage() . "\n" .  $exception->
            getTraceAsString());
        return;
    }
} // class ErrorController extends Zend_Controller_Action
```

As you can see it's pretty straightforward; nothing that we've not covered before except for the fact that in the case of an error, this controller/action will be called for us automatically if it's set up. There is one new setting that we need to add to the `config.ini`, `dirs.tmp`. This tells the error action where to write the `Error.log` to. We'll cover that in just a bit.

Now that our controller is in place, let's turn our attention to the view script. to keep with the convention, we need to create view/scripts/error/error.phtml. Here's what it should look like.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
    xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head>
    <title><?PHP echo Globals::getConfig()->title;?></title>
</head>
<body>
```

```
<h1>An error occurred</h1>
<hr />
<?php if (Globals::getConfig()->debug) { ?>
<strong><?php echo $this->message;?></strong><br />
<pre>
<?php echo $this->trace; ?>
</pre>
<?php } else { ?>
<p>We are sorry but the system seems to have hit an error.  Have no fear, we
    have notified our engineers and have a highly qualified team rushing to fix
    the error. In the mean time, may we suggest a game of Solitaire?</p>
<?php } ?>
</body>
</html>
```

Notice that after we notify the user that there has been an error, we branch based on an option in our config file. If debug is *true* then we dump the error message and stack trace that we stored earlier. However, if it's *false*, we speak softly and reassuringly to the user.

Next, we add to our `config.ini` file, the two options that we have been using.

```
dirs.tmp   = "c:\web\htdocs\tmp\"
debug = 0
```

Adjust the `tmp` directory to suit your setup. To see the debug messages, set `debug = 1`.

Finally, we need to revisit our bootstrap. There is a lot in there now that we just don't need.

Here is our new bootstrap, now that we have offloaded the error handling to the `ErrorController`. Notice that while we are rearranging things, I took the liberty of getting rid of the variable `$frontController` and use fluent interfaces to make it all one call.

```
<?php
ini_set('max_execution_time',600);
$lib_paths = array();
$lib_paths[] = "c:/web/htdocs/application";
$lib_paths[] = "c:/web/htdocs/lib/";

$inc_path = implode(PATH_SEPARATOR, $lib_paths);
```

```
set_include_path($inc_path);

require_once 'config/Globals.php';
require_once 'Zend/Loader.php';
Zend_Loader::registerAutoload();

Zend_Controller_Front::getInstance()
  ->registerPlugin(new Cal_Plugin_MemberInit())
  ->setControllerDirectory('c:/web/htdocs/application/controllers/')
  ->dispatch();
```

Now, back to our regularly scheduled example. First, while we could define our own custom exception that subclassed `Zend_Exception`, we won't. It's enough to know that we could. For our simple needs, `Zend_Exception` will do just fine.

The first thing we need to do is figure out where we want to throw exceptions and where we want to catch them. In reality, we should touch most areas of code that we've written and either throw exceptions on specific cases or catch and process them. Since this is just an example though, I'm only going to touch a couple of areas. The first will be `IndexController::extractAction()`. This is really the heart of our little application so let's see where exceptions can be used to streamline the processing. Most of this code you have already seen, however since we touch so much of it, I'm going to show you the entire method so you can see where things go and not just what goes there.

```
public function extractAction()
{
```

Since we will be throwing exceptions throughout processing, let's wrap the entire method in a try/catch block. It should be noted here that you can nest try/catch blocks so that if you have code inside a try/catch that you want to handle with its own exceptions, you can. Heed the warning above though, you catch it, you deal with it.

```
  try {
    /*
     * get the token
     */
```

```
$token = Zend_Filter::get($this->getRequest()->getPost('token'), '
    StripTags');

/*
 * Execute the token check
 */
```

Here, we throw our first exception. Before this, we set the `FlashMessage` helper and redirected to the home page. We've moved this code to the catch block since in all cases, that's what we want to do and this way, we only have to code it once. By throwing this exception, we have now stopped execution in the try block and moved to the appropriate catch block.

```
if (!$this->tokenCheck($token)) {
  throw new Zend_Exception("I'm sorry but I think there has been an error.
       Please try again.");
} // if (!$this->tokenCheck($token))

/*
 * Reset the token
 */
$this->generateToken();

/*
 * get the URL passed in from the form
 */
$url = Zend_Filter::get($this->getRequest()->getPost('url'), 'StripTags');
```

Here is another chance to stop the processing and notify the user of an error condition without actually hitting an error. If we can't load the page then Yahoo! can't analyze it. This will either display a fatal notice on the screen or show the user a blank screen, depending on how things are setup in your `php.ini`. Neither situation is good as the user doesn't have a chance to correct the problem.

If you've played with the sample app at all, you know that without a protocol (`http://`) it will just blow chow. So let's check it and notify the user if they forgot to put it in.

```
if (substr(strtolower($url),0,7)!='http://') {
        //throw new Zend_Exception("All URLs must start with http://");
} // if (substr(strtolower($url),0,7)!='http://')
```

```
/*
 * read page into memory
 * requires allow_url_fopen to be true
 */
$page = file_get_contents($url);
```

Again here, we check to make sure that Yahoo! has something to analyze. Because we are lazy and using `file_get_contents()` to fetch the web page, it is entirely possible that the page did not load. There are too many reasons why this would happen to list here. What is important to know though is that if `file_get_contents()` fails, it returns *false*. Therefore, we can test for a value of *false* and if it exists, throw an `Exception`. This protects the app from Yahoo! failing on ii and keeps Yahoo! from getting mad at us because we keep asking them to analyze "false".

```
if (!$page) {
  throw new Zend_Exception("There was an error loading the url. ".$url);
} // if (substr(strtolower($url),0,7)!='http://')

/*
 * strip out everything but the content
 * Many thanks to #phpc members ds3 and SlashLife for the RegEx
 */
$matches = array();
preg_match('/<body[^>]*>(.*?)<\/body\s*/isx', $page, $matches);
$content = $matches[1];

/*
 * Filter out the cruft
 */
$content = preg_replace('/(<style[^>]*>[^>]*<\/style\s*>)/isx', '',
    $content);
$content = preg_replace('/(<script[^>]*>[^>]*<\/script\s*>)/isx', '',
    $content);
$content = preg_replace('/(&.*?;)/isx', '', $content);

$content = Zend_Filter::get($content, 'StripTags');

/*
 * send it off to Yahoo for analysis
 *
 */
```

```
$client = new Zend_Rest_Client('http://search.yahooapis.com/
    ContentAnalysisService/V1/termExtraction');
$client->appid('rqP7px3V34H2DVtFJq04ZFTHfiRJQmlvnQze4T313MFEGLAy1.
    lw8PZBWeRqk40R_30-')
     ->context($content)
     ->output('xml');
$result = $client->post();
$client = null;
```

`Zend_Rest_Client` has an `isError()` method but in this particular situation, it does not report error conditions properly. Therefore, we check to see if Yahoo! sent us a message back in the payload. They only do that if there was a problem processing the request. If they do return us an error, we hand it off to the exception to display to the user. Now, this isn't the brightest thing we can do here. Most users won't know that "invalid content" means that for some reason, we handed it content that it couldn't process and even if they do, most aren't going to know what to do about it. In a production application, you would want to catch this problem and see if there was a way you could deal with it or at least try and give the user a bit more help in solving the problem. However, since this is a demo application and you are the only user I'm designing for, I'll just hand it back to you and let you figure it out on your own.

```
if (!empty($result->Message)) {
  throw new Zend_Exception((string)$result->Message);
} // if ($result->isError())

$wp = new WebProperty($url);

/*
 * Hand everything off to the view for output
 */
$this->view->url = $url ;
$this->view->result = array();

/*
 * Build our keywords list
 */
$keywords=array();
foreach($result->Result as $item) {
  $image = $wp->fetchImageTag((string)$item);
  $this->view->result[] = array((string)$item,
```

```
                    $image);
      $keywords[] = (string)$item;
    } // foreach($result->Result as $item)

    if ($wp->id<0) {
      $wp->save();
    } // if ($wp->id<0)

    $wp->addAnalysis(date('Y-m-d h:i:s',mktime()),$keywords);

    return true;
  } catch (Zend_Exception $e) {
```

Now let's deal with those exceptions we've been throwing. Again, in more complex applications, we may want to clean things up, destroy resources, etc. before handing it back to the user. However, in this case, we are just going to notify the user that things have gone awry and we need a little help from them before trying again. Since the only way to get to this action is to come from the `indexAction()`, after we notify the user, we throw it back to that page so it can be displayed.

```
    /*
     * Deal with exceptions.
     */
    $this->_helper->flashMessenger->addMessage("There was an error processing
        your request.");
    $this->_helper->flashMessenger->addMessage("Exception Type:".get_class($e)
        );
    $this->_helper->flashMessenger->addMessage("Exception Message:".$e->
        getMessage());
    $this->_redirect('/index/index');
    return false;
  }
} // public function extractAction()
```

That's one way to use exceptions, throwing them and catching them in the same code block. It's a valid strategy for handing issue that may arise. However, exceptions are much more powerful than just that. As we saw in our initial discussion of exceptions (those of you who skipped it may want to scan back a few pages and look at the sample code), exceptions are useful for stopping processing in methods. One way we can use this is to prevent the instantiation of classes that we only want used

as static. {{include Globals.php}}`Globals.php` is a good example of that. We **never** want to instantiate a copy of `Globals`. All methods are static, all properties are protected. There's just no reason to do it. However, there's nothing currently preventing someone from executing:

```
$myGlobals = new Globals();
```

One way we can prevent this is to throw an exception. Let's add a constructor to `Globals.php`.

```
public function __construct()
 {
   throw new Zend_Exception('You cannot instantiate Globals.php! It is static
        only.');
 }
```

There, now it cannot be instantiated. If we try, well, it won't be pretty. Here, let's give it a go in our `IndexController::indexAction`:

```
public function indexAction()
{
  $fail = new Globals();
  $this->view->token = $this->generateToken();
  $flash = $this->_helper->getHelper('flashMessenger');
  if ($flash->hasMessages()) {
    $this->view->message = implode("<br />", $flash->getMessages());
  } // if ($flash->hasMessages())

} // public function indexAction()
```

Save that and execute it. You should see something like what Figure 10.1 shows.

Make sure you comment that line out or remove it before you go on. Otherwise, nothing is going to work.

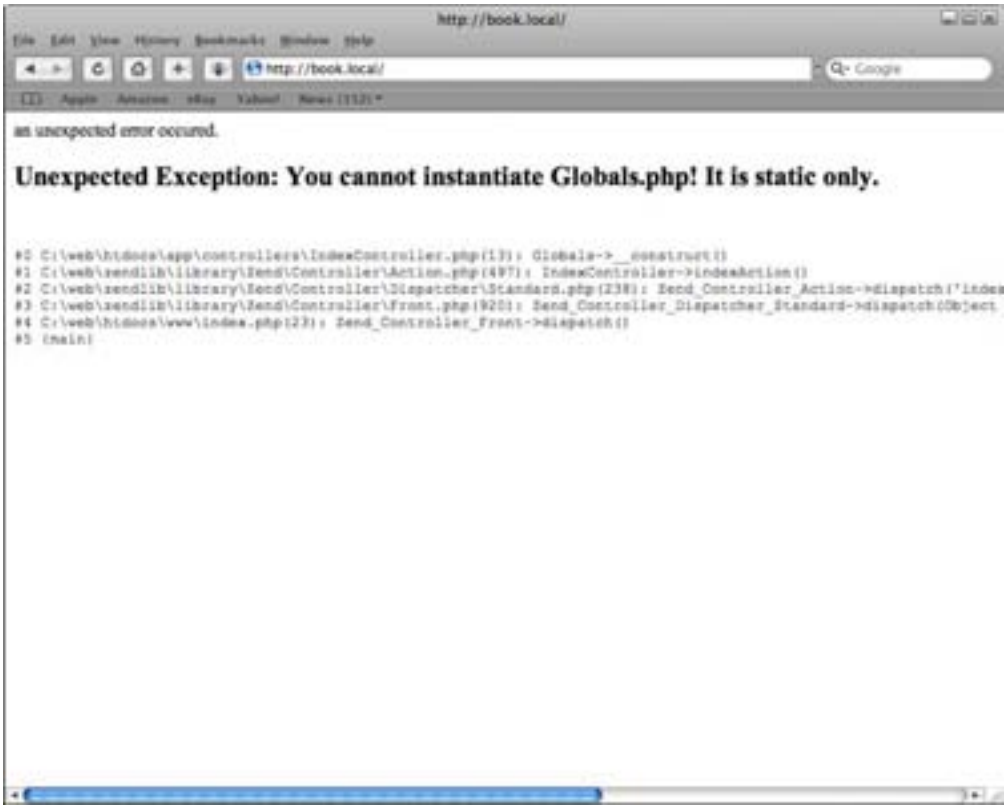

Figure 10.1

Summary

Exceptions are a powerful tool in PHP and in the Zend Framework. More than just an afterthought in building your application, you need to make sure that your program not only doesn't show users ugly errors but that it properly handles all conditions. Very early on in my programming career, one of my mentors, a man named Jim Turner, told me a piece of advice that to this day I've never forgotten.

"There is no such thing as a user error. If your program let the user make a mistake then it's a system error and up to you to fix it."

Exceptions allow us to gracefully handle times when our users do things that they shouldn't, APIs from our partners don't behave as they should or generally, things happen that we didn't plan for.

Chapter 11

Rich Internet Applications

We are winding down now; you know the basics and I'm sure by now you have started trying to apply what you've learned to your own projects. Honestly, if you weren't, I'd be worried since reading this book won't make you a Zend Framework programmer, actually working with it will.

No good web development book these days is complete without a section on *Rich Internet Applications* (RIA). Loosely defined, a RIA is any web based application that has features and functionality similar to traditional desktop applications. My guess is that someone mistakenly assumed that desktop applications were the end-all be-all of application design and that everything should strive to mimic them. More likely though is that developers have noticed that people are more comfortable with desktop applications and so they strive to build web applications that act like them.

There are a lot of good RIAs currently in production. Zimbra's mail client and productivity suite, Gmail, and Google Docs are the ones that come immediately to mind; however there are more out there and new ones being released daily.

RIAs can take many forms. You can build them using Adobe's Flash or Flex, Microsoft's Silverlight, or a host of other technologies including my personal favorite, good old JavaScript. I've used fancier technologies and really love Adobe's Flex but since this book is about Zend Framework, we'll spend our time working the backend and not the frontend.

Making our Sample App Into an RIA

As usual, let's talk about what we want to do before we dive into the code.

The current `index` page does a great job of processing our request. When we give it a URL, it will analyze it and give us back the resulting keywords and, when appropriate, graphics of the keywords from Flickr. Everything is good except that to do all of this, it has to make a complete round trip to the server. If we prettied this page up with a lot of graphics and such, they would potentially get reloaded each time we submitted a new URL. Also, each request carries with it load on the server in the form of returning the page, style sheets, any JavaScript, etc. All of this is overhead that is not necessary thanks to JavaScript and Ajax. However, as both you and I know, the best reason for building out a RIA version of the page is simply because it's cool. So, let's build a RIA version of our main analysis page.

One thing we need to do before we start though is to correct a grievous wrong. Way back near the beginning of the book we talked about models. Models should really contain all of your business logic. However, for the sake of clarity, we put all of our analysis code in the `IndexController::extractAction()`. Even back then we knew that while this was acceptable, it wasn't the right place for it. So let's make some adjustments. As always, you can follow along with me as I describe the changes or you can simply unpack `example7.zip` and skip ahead.

So open `IndexController.php` and find `extractAction()`. As you can see a lot of that code is business logic but not all of it. Some of it is routine setup code and that we need to leave alone. Since it's now really short, I'll pass it in so we can discuss what we left and what we moved.

```
public function extractAction()
{
  try {
```

Ok, all of this is housekeeping. Yes we could move filtering into a model but it really doesn't make sense because it's not business logic. So we grab the URL, sanitize it and do the token check. All of these tasks, while important, do not directly relate to analyzing the contents of a web page, therefore, they are not business logic.

```
    /*
```

```
         * get the token
         */
        $token = Zend_Filter::get($this->getRequest()->getPost('token'), '
            StripTags');

        /*
         * Execute the token check
         */
        if (!$this->tokenCheck($token)) {
          throw new Zend_Exception("I'm sorry but I think there has been an error.
               Please try again.");
        } // if (!$this->tokenCheck($token))

        /*
         * Reset the token
         */
        $this->generateToken();

        /*
         * get the URL passed in from the form
         */
        $url = Zend_Filter::get($this->getRequest()->getPost('url'), 'StripTags');
```

Now we instantiate our copy of `WebProperty`. The business logic of analyzing the contents of the page has been moved into `WebProperty::Analyze()`. Once we have our instance of `WebProperty`, we call `analyze()` to get our keyword list.

```
        $wp  = new WebProperty($url);

        /*
         * Hand everything off to the view for output
         */
        $this->view->url = $url ;
        $this->view->result = $wp->analyze();

        $wp->save();
      } catch (Zend_Exception $e) {
```

`WebProperty::Analyze()` throws exceptions so we still need to catch them. We could catch them in `WebProperty::Analyze()` but in this case, we are using our `catch` to build our notification to the user that there's a problem. This would be a lot more difficult in `WebProperty` so we'll catch them here.

```
        /*
         * Deal with exceptions.
         */
        $this->_helper->flashMessenger->addMessage("There was an error processing
            your request.");
        $this->_helper->flashMessenger->addMessage("Exception Type:".get_class($e)
            );
        $this->_helper->flashMessenger->addMessage("Exception Message:".$e->
            getMessage());
        $this->_redirect('/index/index');
        return false;
      }

      return true;

    } // public function extractAction()
```

Now, let's see what we moved. This code will all look familiar since it's just been moved.

```
    public function analyze()
    {
```

I know you are looking at this next block of code and screaming "filtering"! But no, this is a business logic decision. We made the decision that all URLs have to start with a protocol. Actually, here, instead of throwing an error and handing it back to the user, we just append one and try it. The worst case scenario is that we throw another exception later on.

```
        if (substr(strtolower($this->_url),0,7)!='http://') {
          throw new Zend_Exception("All URLs must start with http://");
        } // if (substr(strtolower($url),0,7)!='http://')

        /*
         * read page into memory
         * requires allow_url_fopen to be true
         */
        $page = file_get_contents($this->_url);

        if (!$page) {
          throw new Zend_Exception("There was an error loading the url. ".$this->
              _url);
```

```
} // if (substr(strtolower($url),0,7)!='http://')

/*
 * strip out everything but the content
 * Many thanks to #phpc members ds3 and SlashLife for the RegEx
 */
$matches = array();
preg_match('/<body[^>]*>(.*?)<\/body\s*/isx', $page, $matches);
$content = $matches[1];

/*
 * Filter out the cruft
 */
$content = preg_replace('/(<style[^>]*>[^>]*<\/style\s*>)/isx', '',
    $content);
$content = preg_replace('/(<script[^>]*>[^>]*<\/script\s*>)/isx', '',
    $content);
$content = preg_replace('/(&.*?;)/isx', '', $content);

$content = Zend_Filter::get($content, 'StripTags');

/*
 * send it off to Yahoo for analysis
 *
 */
```

Here I also decided to parameterize the call to Yahoo!. It was working fine the other way, it's just good form to parameterize anything that could change.

```
$client = new Zend_Rest_Client(Globals::getConfig()->url->yahoo);
$client->appid(Globals::getConfig()->yahooAppId)
     ->context($content)
     ->output('xml');
$result = $client->post();
$client = null;

if (!empty($result->Message)) {
  throw new Zend_Exception((string)$result->Message);
} // if ($result->isError())

$firstResponse = (string)$result->Result[0];
if (empty($firstResponse)) {
  throw new Zend_Exception('Yahoo was not able to find any keywords.');
} // if (!isArray($result))
/*
```

```
         * Build our keywords list
         */
        $keywords  = array();
        $returnValue = array();

        foreach($result->Result as $item) {
          $thisItem = (string)$item;
          if (empty($thisItem)) {
            continue;
          }
          $image = $this->fetchImageTag($thisItem);
          $returnValue[] = array($thisItem,
                         $image);
          $keywords[] = $thisItem;
        } // foreach($result->Result as $item)
        $this->addAnalysis(date('Y-m-d h:i:s',mktime()),$keywords);

        return $returnValue;
    } // public function analyze()
```

Ok, now that we've done that, we've not only corrected something that's been bothering me for a while now, we have also set things up for the next step. As it was, the code to do the analysis was all tied up in the `IndexController`. This means if we wanted to build another action that did something similar we had to either do some real ugly things or re-implement it; and re-implementing it is a no-no as it violates the DRY principal. Now we have it abstracted to a model and can use it in an infinite number of places.

Let's move on to some new code. We need to build two actions now. First, in our `IndexController`, we need an action that will display the new page. For simplicity's sake, we will call it `riaAction()`. Like our `indexAction()`, the sole purpose is to display the view script, `ria.phtml`.

```
public function riaAction()
{
  $this->view->token = $this->generateToken();
  $flash = $this->_helper->getHelper('flashMessenger');
  if ($flash->hasMessages()) {
    $this->view->message = implode("<br />", $flash->getMessages());
  } // if ($flash->hasMessages())

} // public function indexAction()
```

This is the same code as in `indexAction()` and if we really wanted to take DRY to heart, we could remove both of these and use a `__call()` function to decide what to do on actions that don't have `*Action()` functions defined. However, in this case, since it serves the purpose of making things easier to understand, we'll clone `indexAction()` and call it `riaAction()`.

I'm not going to put `ria.phtml` in here, if you want to see it, grab it from the zip file. It's too long and most of it isn't really relevant. I'll say this here before we start dissecting the code. Other than where it's necessary to understand what is going on, I'm not going to delve into the JavaScript code. I'm using Prototype.js for the Ajax goodies because it's my favorite. You could use any of the modern JavaScript frameworks to accomplish our simple task here. If you don't understand the JavaScript, either dive in and figure it out or assume its magic and don't worry about it. For our purposes, either will work. It's the PHP code that we are going to talk about. Let's instead look at the API that is being called in this page and understand what is going on.

For our purposes, when submit is clicked on the form, a call is made to the following URL:

```
www.example.com/api/extract/?url=http://devzone.zend.com&token=md5hasgoeshere
```

The first thing you will notice if you are paying attention is for the first time, we are not passing variables in the key/value format. We are actually using good old fashioned "http" variables. This is not merely a cruel trick on my part to keep changing things up in hopes of confusing you. In this case, I've actually got a good reason; using key/value, we can't pass a URL.

```
www.example.com/api/extract/url/http://devzone.zend.com/token/md5hasgoeshere
```

This example, no matter if you urlencode it or not, will blow chow. To pass the URL in that we want to analyze, we have to make it a variable. (Technically, we could remove the protocol portion of the URL and assume "http" but really, where's the fun in that?)

To make a call to `/api/extract`, we have to create an `ApiController::extractAction()`. If you read the housekeeping section of this chapter then you can already see where this is going. `ApiController::extractAction()` is going to be very similar to `IndexController::extractAction()`. There are a few changes and we'll discuss them but overall, you are familiar with the code.

Like our other functions in `ApiController`, this is still a REST interface, however, we are not returning XML, we will be returning JSON. There are two reasons for this. First Prototype.js will automatically evaluate JSON for us and put it back into usable variables. Second, XML processing in JavaScript is a pain, and I don't like pain. However, under the current, very loose, definition of REST, we still qualify.

```
public function extractAction()
{
  $payload = array();
    .
    .
    .
```

Here is where we start to deviate from `IndexController::extractAction()`. Instead of handing everything to the view, we build an array called `payload`. Our frontend expects a few things to be constant.

- The token. Each time we process, we regenerate the token. Since we are not reloading the page, we have to send it back up in the payload for the front end code to store. This keeps people from using our API without using our front end.
- The URL. We pass it back in in case the front end wants to display it or verify it.
- The result array. This array is in the same format as we've been previously working with. An array of arrays, each one containing a keyword and possibly an image tag.
- The message. In this case, we simply pass in the word 'success' as the message. However, we could have passed in any number of lines for the front end to display to the user. In the event of an exception being thrown, you can see that we build a dummy `payload` array, primarily for passing back the error messages.

```
$wp  = new WebProperty($url);

/*
 * prepare the payload
```

```
     */
    $payload['token'] = $newToken;

    $payload['message'] = array('Success');
    $payload['url']   = $url ;
    $payload['result']  = $wp->analyze();

    $wp->save();

  } catch (Zend_Exception $e) {
    /*
     * Deal with exceptions.
     */
    $payload['token'] = $newToken;
    $payload['message'] = array();
    $payload['message'][] = "There was an error processing your request.";
    $payload['message'][] = "Exception Type:".get_class($e);
    $payload['message'][] = "Exception Message:".$e->getMessage();
  }
```

Now here's something different. `Prototype.js` will re-constitute our JSON encoded string if we pass it in as a value to the http header X-JSON. Zend Framework gives us the tools to do this if we want. This construct, handled outside of the try/catch (since either way, try or catch, we now have a populated payload array) sets the headers for the response.

If you haven't yet, you may want to grab the FireBug extension for FireFox (assuming you use FireFox). It makes working with Ajax requests so much easier.

The first line sets the "Content-Type" header to `application/json`. Technically, this isn't really necessary as Prototype will ignore this and try to process the response no matter what the content type. However, if someone accidentally calls your API URL with all the proper fields and values from a browser, this will cause the browser to ask them what they want to do with the response since neither IE not FireFox know how to process JSON.

The second header actually will contain our JSON encoded payload. Normally in a case like this, for clarity's sake, I would JSON encode the payload on a separate line, store it in a variable and then use the variable to set the header. You will notice that

we are using the `Zend_Json` class to encode the payload. `Zend_Json` will use the native JSON encoding methods built into PHP if they are available. If you did not compile them in then it will revert to it's own code. Either way, you get the same thing, a JSON version of the array `$payload`.

The final line, `appendBody()` is actually superfluous but I stuck it in there for clarity. Since Prototype will decode the JSON for us automatically from the header, there's no real reason to put anything in the body of our response. However, just so you know that we didn't forget it, we are adding an empty string to the body.

```
        $this->getResponse()->setHeader('Content-Type','application/json')
                  ->setHeader('X-JSON',Zend_Json::encode($payload))
                  ->appendBody('');
        return ;

    } // public function extractAction()
```

That's it, you now have an API that can be called from your `ria.phtml` to analyze a webpage and display the results. If you run it, the final output should look something like Figure 11.1 (unless you are good at web design. In that case, you've probably made it look a lot prettier).

The table is, of course the list of keywords returned from Yahoo. The highlighted ones are keywords that Flickr returns an image for. Clicking on any of the yellow cells of the table will display an image. (Can you guess which one I clicked on?)

Summary

RIAs are a great thing and I expect their use to grow in coming years. They do, however, force us as developers to rethink how we build our applications. In theory, any action our application can take may need to be built as an API so that a RIA can access it. As web applications move out of the browser and on to mobile devices, gaming consoles DVRs, and other non-traditional devices, the world of possibilities and potential users, opens up before us. It is possible to conceive applications as being written solely as APIs. The models containing the business logic can then be used by controllers that implement a traditional web based interface. However be-

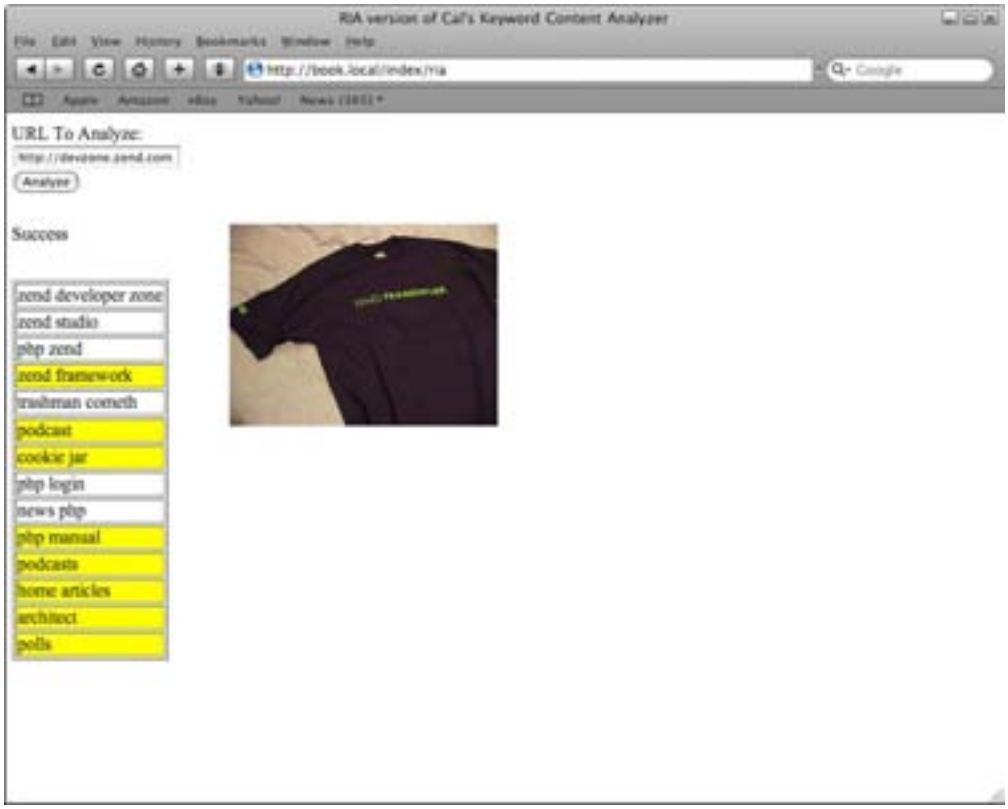

Figure 11.1

cause everything is exposed as an API, any application with the proper permissions can access your data and processes.

Zend Framework allows us to easily expose just about any action or piece of data as an API. However, as you saw in the first part of the chapter, if you don't think about it when you are setting your architecture, you will have to go back and refactor. It's better to do it from the beginning.

Chapter 12

Zend Framework Party Tricks

PHP is a web scripting language. No matter how popular it gets, that's where it started and that's where it's used most of the time. However, there are times when you need to do non-web things. In the past, I've used PHP as a scripting language for writing the DOS equivalent of BATCH files or the Linux equivalent of shell scripts. I've found that most of what I need to do can be accomplished from within PHP. There is even a project to allow you to build desktop applications via PHP, the PHP-GTK project.

Cleaning Your Cache Through CLI

While the PHP-GTK project is great, in our sample application we really don't need a desktop view. We do, however, potentially need to do some housekeeping. If we are not careful, our cache can grow huge as our application gets big. While the cache is self cleaning, on an extremely large cache (thousands of files) we are putting the burden of cleanup on the user because the cache will clean itself as it is called. If a large portion of your cache is expiring all at once the poor user who visits your site just after they expire has to wait as `Zend_Cache` cleans them all up. Is that enough justification for you? I can go on or we can assume that you see the need for a process to run in the background and keep things clean.

What we need is a process to run in the background to keep things clean. It doesn't have to run often to be effective. In our case, we'll design it to run every night at midnight and clean things up.

Different OSes have different ways of accomplishing repetitively scheduled tasks. On Linux, there is usually a cron daemon, on Windows, you can use the Scheduler. Since syntaxes vary, I am not going to actually show how you schedule the job to run on a regular basis. I'll leave that as your final exercise.

This is all find and dandy except that applications built on Zend Framework use URLs for the most part to declare controllers, actions and parameters. When working from the command line, we don't actually have a URL.

Yes, `wget` and `curl` are available for most modern OSes and it is possible to use them to call a URL with the proper controller, action and parameters. However, using `curl` and `wget` defeat the purpose of this chapter so we are going to ignore them. Besides, they require that you process through your web server. To achieve true CLI status, we need to remove that dependency as it's unnecessary. If you still insist on using `wget`, `curl` or some other type of browser-related technology to accomplish your scheduled tasks then you can skip this chapter and pick up your diploma on your way out the room.

Up until now, everything we have done has been browser-based and therefore involved a URL. However, working with Zend Framework from the command line is just as easy with a few extra pieces of code. As always, before we dive into the code, let's talk about what we want our CLI process to accomplish.

As described above, we sometimes need to be able to clear the cache. Most of the time, we just want to clear out the expired caches, however, there are times when we will want to clear out the entire cache. If we are using the `File` backend then that means removing every file in the cache, valid or invalid. If we clear out the old files every evening then the browser processes don't have to do it and can operate at optimum speed.

To accomplish this, we will of course need a controller. We could simply put this code in the API controller, however, this isn't really an API call. Also, as you will see, we have some special needs for CLI calls, so let's create a `ProcessController.php` to hold all of our processes.

Next we need a bootstrap. Up until this point, we've used `index.php` as our bootstrap. However, `index.php` does not, nor should it, know how to process parameters passed in on the command line. Also, as you will see, there are some other issues

that we will need to deal with that are not germane to `index.php`. We will call our new bootstrap, `clearCache.php`. Since `clearCache.php` is a new type of file for our project, we need some place to put it. As you can see if you unpack `example8.zip`, I've added a directory, `processes`, to the root directory of our project. It is important that this directory be at the same level as `www` since in production, the include path will be set relative to the project root. If you move processes, you will need to adjust your `include_path`.

Zend Framework, for all its wonderfulness, is designed to build web based applications; `Zend_Controller_Router` is no exception. To solve this we have to make our own version of it, `Cal_Controller_Router`. This is probably the simplest piece of code we've written in a while. All we really need to do is stop the `route()` method from firing. So our subclass simply overrides the `route()` method. Everything else stays the same.

Finally, to complete the process of "de-webifying" the necessary pieces of Zend Framework, we need a subclass of the `Zend_Controller_Request_Abstract`. `Cal_Controller_Request_Cli` will let us mimic the necessary functionality while adding our own to make life simpler.

You should note, we've not discussed naming conventions and "other libraries" at all so far because everything we've been doing is within the framework or the application. However, in this case, we need to create a class that operates at the framework level but is not actually part of the framework. This piece of code could potentially be used on other projects so we don't want it inside the application directory structure. Thankfully, Zend Framework team has defined us a place to put our own libraries. They suggest a `/lib` directory in the project's root directory. For conformity's sake, I'm going to stick to the naming convention established by Zend Framework. Each piece of the class's name represents a directory except for the last piece, which represents the file name. Therefore, my new class, `Cal_Controller_Request_Cli` is actually `lib/Cal/Controller/Request/Cli.php`.

We understand that we want to manipulate the cache and we know the basic pieces we need to do that.

Cal_Controller_Router. As stated above, this very simple class contains one method, `route()`. In our subclass, `route()` does nothing as we just want to kill the default functionality. To make this clear, I do put a comment in the class, only so no one looks at it in puzzlement.

Cal_Controller_Request_Cli. `Zend_Controller_Request_Http` assumes that we are processing a web request. It assumes that certain properties and methods are available. In a CLI application, those properties and methods are not available to us. However, by creating our own `Request` class, we can provide substitute functionality for those methods that Zend Framework needs. Zend Framework makes this easy for us by providing us with `Zend_Controller_Request_Abstract` on which we can base any new `Request` object we feel we need.

Zend_Console_Getopt. This seems as good a place as any to take a detour and talk about passing options in via the command line. In the Unix world the standard for retrieving and validating command line options has long been the `getopt` library. Since PHP 4.3, PHP developers have had access to `getopt` functionality on Unix based platforms and since PHP 5.0, PHP users have had `getopt` on all platforms, including Windows. `Zend_Console_Getopt` provides users of Zend Framework with a convenient OO interface to `getopt`. Additionally, it makes setting the `getopt` rules easier to use.

`Getopt` works by defining a series of flags or parameters that an application will accept. Parameters can have aliases, for example -v and -verbose can both do the same thing as long as `verbose` is aliased to `v`.

Neither `getopt` or `Zend_Console_Getop` actually define workflow or do any processing based on the parameters defined and passed in. They are simply a way to define and retrieve the parameters and validate their values.

Setting up the Bootstrap

When we discuss our "bootstrap," we will discuss how we define our options and how to tell the application if they are required or optional. For now, all you need to know is that when we create an instance of `Cal_Controller_Request_Cli`, one of the things we hand in is a valid set of `Zend_Console_Getopt` rules. If we do, it will create an instance of `Zend_Console_Getopt` and gather the appropriate parameters.

There are three magic parameters that we define and if we pass into `Cal_Controller_Request_Cli`, it will set automatically. These are normally the first

pieces of your URL but since we don't have a URL, we have to pass them in for Zend Framework to be able to function.

- `-m` : module. If your Zend Framework application is divided up into modules, you can pass in the module name to use and it will be set automatically. This parameter is optional.
- `-c` : controller. This is the controller that the action will be found in. The naming standard for controllers is `*Controller.php`, however, just like when we are calling a controller from a URL, we only have to specify the controller name (i.e. `ProcessController` would simply be `Process`). Since, to find the action, you have to know the controller, this parameter is not optional.
- `-a` : action. The action is, of course, the action to execute within the controller specified. Like the controller, the suffix `Action` can be omitted and just the action portion of the name specified.

In most cases, just like we pass additional parameters when calling actions from a URL, we will want to pass parameters into a command line. However, unlike passing them in via a URL, we have to define exactly which parameters we will allow in for `Zend_Console_Getopt` to properly process them. `Cal_Controller_Request_Cli`, however, does not need to know which parameters are defined. Since `Cal_Controller_Request_Cli` holds an instance of `Zend_Console_Getopt`, we can always get to the parameters passed in.

```
<?php

require_once 'Zend/Controller/Request/Abstract.php';
require_once 'Zend/Console/Getopt.php';

class Zend_Controller_Request_Cli extends Zend_Controller_Request_Abstract
{
  protected $getopt;
  protected $_pathInfo;
```

As stated earlier, we can pass in the `Zend_Console_Getopt` rules. If we do, then the controller will gather the options for us and populate the appropriate properties in the object.

```
public function __construct($rules=array())
{
  if (count($rules)>0) {
    $this->getopt = new Zend_Console_Getopt($rules);

    $this->getopt->parse();

    if (!is_null($this->getOption($this->_moduleKey))) {
      $this->_module = $this->get($this->_moduleKey);
    }

    if (!is_null($this->getOption($this->_controllerKey))) {
      $this->setControllerName($this->getOption($this->_controllerKey));
    }

    if (!is_null($this->getOption($this->_actionKey))) {
      $this->_action = $this->getOption($this->_actionKey);
    }
  }
} // public function __construct($rules=array())
```

Once we have the options that were passed in, we need a way to access them. These two functions simply mirror existing functionality in `Zend_Console_Getopt` for the users of this response object.

```
public function getOption($key)
{
  return $this->getopt->getOption($key);
} // public function getOption($key)

public function getOptions()
{
  $keyList = $this->getopt->getOptions();
  $returnValue = array();
  foreach($keyList as $key) {
    $returnValue[$key] = $this->getopt->getOption($key);
  }
  return $returnValue;
} // public function getOptions()

public function setPathInfo($pathInfo = null)
{
  $this->_pathInfo = $pathinfo;
} // public function setPathInfo($pathInfo = null)
```

```
    public function getPathInfo()
    {
      return $this->_pathInfo;
    } // public function getPathInfo()

} // class Zend_Controller_Request_Cli extends Zend_Controller_Request_Abstract
```

As you can see, `Cal_Controller_Request_Cli` isn't a difficult piece of code to write. In most of these cases, we left the existing functionality in place and only added or overrode if it was necessary. Now let's see how we use this.

Creating a New Bootstrap

`clearCache.php` is our bootstrap file. It would be nice if we could make it totally generic but we can't. Options have to be explicitly defined. Therefore, we are really left with two options. You can either define one CLI bootstrap file and define every option that can be passed in to any `Controller::action` you want to process or you can define one CLI bootstrap for each process you want to call and customize the options in each bootstrap for that process.

I've chosen the second option even though it will cause us to violate the DRY principal. We will keep the repeated code to a minimum.

Most of this you should recognize from `index.php`. In this case, setting the include path is a good idea inside of the code. In the bootstrap file, we can rely on the `php.ini` or our web server's configuration file to define a constant set of include paths that every web request will follow. However, that doesn't apply to the command line version of PHP. We could use the default `php.ini` to define them but then we are not playing nice with other applications that may be running on the machine. It's easier just to define them ourselves. You'll notice that they are exactly the same directories that we defined in our bootstrap.

```
<?php
$lib_paths = array();
$lib_paths[] = "c:/web/htdocs/app";
$lib_paths[] = "c:/web/zendlib/library";
$lib_paths[] = "c:/web/htdocs/lib/";
```

```
$inc_path  = implode(PATH_SEPARATOR, $lib_paths);
set_include_path($inc_path);

require_once 'Zend/Loader.php';
Zend_Loader::registerAutoload();
require_once 'Cal/Controller/Request/Cli.php;
require_once 'Zend/Controller/Response/Http.php';
require_once 'Zend/Controller/Front.php';
require_once 'Cal/Controller/Router.php';

require_once 'config/Globals.php';
$config = Globals::getConfig();
```

As discussed earlier, we have to define the rule for `Zend_Console_Getopt` to hand to `Cal_Controller_Request_Cli`. This defines both the standard three magic options plus any custom ones. In this case, I want two custom rules, "verbose" and "all". Note that since we are already using the `-a`, "all" does not have a single letter alias. If you want to specify "all" you have to use `-all`.

```
$rules = array();
$rules['m|module-s']   = 'The optional module to use.';
$rules['c|controller=s'] = 'The controller to use.';
$rules['a|action=s']   = 'The action to use.';

// custom rules
$rules['all']      = 'If set, the entire cache will be destroyed.';
$rules['v|verbose']    = 'If set, exceptions and other debugging information
    will be displayed.';

ini_set('session.save_path',Globals::getConfig()->dirs->tmpDir);

try {
```

Now this is a bit different. We grab an instance of the front controller like normal but then we set a lot more parameters manually. Since this is command line, there's no need to ever render views or let the system handle errors. We instantiate a copy of both `Cal_Controller_Request_Cli` and `Cal_Controller_Router` and hand them to the front controller to override the default ones.

```
  $request = new Cal_Controller_Request_Cli($rules);
  $front = Zend_Controller_Front::getInstance();
```

```
    $front->setParam('noViewRenderer', true);
    $front->setParam('noErrorHandler', true);
    $front->setRouter(new Cal_Controller_Router());
    $front->setRequest($request);
    $front->setResponse(new Zend_Controller_Response_Cli());
    $front->setControllerDirectory('controllers/');
```

I don't actually recommend this for production systems as your script may be part of a larger scripting system and therefore exceptions and error code may be necessary. However, here is a simple example of how we can pass options in on the command line and use them in our code. In this case, if the user specified the `-v` option then our application will throw exceptions and they will be caught below. If `-v` is not passed in, our application will swallow the exception and die silently.

```
    $front->throwExceptions($request->getOption('verbose'));
    $front->dispatch();
} catch (Exception $exp) {
    echo 'An unexpected error occurred.';
    echo 'Unexpected Exception: ' . $exp->getMessage() . "\n";
    echo $exp->getTraceAsString();
```

Because our CLI script may be part of a larger scripting application, we need to make sure we use return codes when appropriate. Here we could return a simple '1' to indicate an error condition. However, since our calling script may want more information than that, we die with the error code from the exception. The error code is defined when the exception is instantiated and therefore is programmer controllable. None of this matters at all though if the `-v` option is not passed in.

```
    die($exp->getCode()); } // try
```

Finally, if everything worked correctly, die with a return code of `0` to notify any calling script that everything completed successfully.

```
die(0);
```

As you can see, with very few exceptions, the concepts are exactly like the bootstrap we use for the web.

ProcessController.php

Finally, now that we have setup our infrastructure (which may seem overly complex but in reality, it's not a lot of code) let's get down to the business of clearing our cache.

```
<?php
require_once 'controllers/BaseController.php';

class ProcessController extends BaseController
{
```

First, let's override the `init()` in `BaseController`. We don't actually have a `Member` class at this point but also, we need some custom functionality. We never really want anything in this controller being called from a browser. Therefore, we check. If our request doesn't have `getOptions()` then we won't be able to do much and it's most likely because it's being called from a browser. This is not foolproof and the argument could be made that it would be better to use `instanceof` and a CLI interface. That would be the proper OO way of doing this. However, that adds a level of complexity (and another file) to the sample application. So we will chalk this one up to "Do as I say, not as I do" and keep moving.

```
public function init()
{
  if (!method_exists($this->getRequest(),'getOptions')) {
    throw new Zend_Exception('This action cannot be called from the browser.'
        ,1);
  }

}
```

This is just a simple method to test and make sure things are working without messing with your cache. It does not require any options however; it will echo back to you any that you pass in.

```
public function testAction()
{
  echo "This is the Process test action.  \nIt simply returns a list of the
      parameters passed in.\n-----\n";
```

```
    foreach($this->getRequest()->getOptions() as $key=>$value) {
      echo $key.":".$value."\n";
    } // foreach($optionsArray as $key=>$value)

    return;
  }
```

Now to the heart of the controller, the `cacheCleanAction()`. `cacheCleanAction()` tests to see if you passed in the `-all` option and calls the appropriate method of `Zend_Cache`.

```
  public function cacheCleanAction()
  {
    if ($this->getRequest()->getOption('all')) {
      Globals::getCache()->clean(Zend_Cache::CLEANING_MODE_ALL);
    } else {
      Globals::getCache()->clean(Zend_Cache::CLEANING_MODE_OLD);
    } // if ($this->getRequest()->getOption('all'))
  } // public function cacheCleanAction()

}
```

Now, if you've done everything correctly, on a Windows based computer, you should be able to open a command window, drill down to the directory that `clearCache.php` is in and type:

```
php clearCache.php -c Process -a test -v -all
```

This should return something like:

```
This is the Process test action.

It simply returns a list of the parameters passed in.


-----
c:Process
a:test
all:1
v:1
```

To see what an error looks like on the command line try:

```
php clearCache.php -c Process -a test1 -v -all
```

This should barf bits all over the place in an arrangement that looks something like this:

```
C:\web\htdocs\processes>php clearCache.php -c Process -a test1 --all -v
An unexpected error occured.Unexpected Exception: ProcessController::test1Action
    () does not exist and was not trapped in __call()
#0 [internal function]: Zend_Controller_Action->__call('test1Action', Array)
#1 C:\web\zendlib\library\Zend\Controller\Action.php(497): ProcessController->
    test1Action()
#2 C:\web\zendlib\library\Zend\Controller\Dispatcher\Standard.php(242):
    Zend_Controller_Action->dispatch('test1Action')
#3 C:\web\zendlib\library\Zend\Controller\Front.php(920):
    Zend_Controller_Dispatcher_Standard->dispatch(Object(
    Cal_Controller_Request_Cli), Object(Zend_Controll
er_Response_Cli))
#4 C:\web\htdocs\processes\clearCache.php(91): Zend_Controller_Front->dispatch()
#5 {main}
```

To actually process your cache use the commands:

```
php clearCache.php -c Process -a cacheClean -v
```

or

```
php clearCache.php -c Process -a cacheClean -v -all
```

The first command will clear out all the expired cache files. This is the normal mode of operation. The second version, with the `-all` will remove all cache filed regardless of their state.

Summary

You won't use the command line interface for Zend Framework often. However, there are tasks which are easier done via the command line or via a scheduler. In these instances, it's nice to know that Zend Framework is flexible enough to get the job done. Also, you've not got a great party trick to pull out at your next party or SciFiCon, an actual PHP script that runs from the command line.

Appendix A

Appendix A - Zend_Layout and doing the Two-Step

If you've been paying attention, up to this point, you will have probably noticed that our final version of the application adheres to the "Don't Repeat Yourself" principal in all areas except for one. The view scripts repeat a lot of code. Every page has the same DOCTYPE, header scripts, etc. If we actually worked on the look and feel of the application, we would be repeating a lot more. This is an unfortunate side effect of not using the `Zend_Layout`. There are ways around this but usually they involve writing a lot of extra code or they involve dragging in a templating engine like Smarty. I've got nothing against Smarty or any other templating package but I've just never seen the need for them.

There is however, a solution in Zend Framework that solves this problem, `Zend_Layout`. The reason this is an Appendix as opposed to a mainline chapter is that as I write this, `Zend_Layout` has only been in the [{index Zend_Core}}`Zend_Core` for about a week. It's brand new and if you are buying an early copy of this book, you will have to checkout Zend Framework from the subversion repository to get access to it. It will however, be included in Zend Framework 1.5.

`Zend_Layout` is an attempt to solve the problem most commonly known as *two-step views*. Two-step views help developers maintain a consistent look and feel throughout their applications while maintaining the DRY principal. It allows you to define default layouts that are made up of the common pieces that every page has. Once

your default layouts are created, you can simply work on the 'exceptions', the display code necessary for the problem you are trying to solve. The nice thing about `Zend_Layout` is that it accomplishes this task without changing the underlying codebase. It works using the existing plugin structure. By calling the `startMVC()` functions we see below, `Zend_Layout` registered a plugin with the controller to actually implement a two-step view (2SV).

I could talk more about what `Zend_Layout` can do and how it does it but honestly, it's just more fun to dive into some code. So, let's take a look at what it takes to implement `Zend_Layout` and what we can learn in the process.

`Zend_Layout` is broken up into the main layout controller as well as a lot of little helpers. We fire up the main class, `Zend_Layout`, in our bootstrap and then we leave it alone. In our main bootstrap file (`index.php`) we need to add a single line to get things going:

```
try
{
  Zend_Loader::loadClass('Zend_Controller_Front');
  Zend_Layout::startMvc(array('layoutPath' => Globals::getConfig()->dirs->
       layouts,'layout'=>'main')); // this gets Zend_Layout going
  $frontController = Zend_Controller_Front::getInstance();
  $frontController->throwExceptions(true);
  $frontController->setControllerDirectory('c:/web/htdocs/app/controllers/');
  $frontController ->setParam('noErrorHandler', true);
  $frontController->dispatch();
} catch (Exception $exp) {
```

The highlighted line is what kick starts `Zend_Layout`. As you can see, it takes an array of parameters to configure itself. You can also pass in a `Zend_Config` object, internally, it converts it to an array and then applies the options. The parameters can always be set at a latter time using the syntax:

```
$this->_helper->layout->setLayout('alternate');
```

This line, when used from a controller action, would override either the default template or the one set in `startMVC()` with the template `alternate.phtml`.

The way `Zend_Layout` is designed, any property with a `set*` function can be passed into `startMvc()` via the `options` array. Practically though, there are only a few options

that you will be interested in. Here is a list of the options you can currently pass in or configure from the `startMVC()`:

- `layout` : This is the name of the default layout to use. As with all the options passed into `startMvc()`, this can be overridden in the controller action. As you can see in our example, we set it to `main`. This will cause `Zend_Layout` to look for `main.phtml` in the directory we set for our templates. If left unset, the default for this is `layout`.
- `LayoutPath` : This is the base path of the layouts to use. This is where most of your default templates, and their pieces, will be stored. There are exceptions as we will see in the case of `footer.phtml` below.
- `contentKey`: By default, `the Zend_Layout_Controller_Plugin_Layout` expects that the main content for the page is called `default`.

If you leave `contentKey` as `default` then in your template, you can refer to it as:

```
$this->layout()->content
```

If however, you have a need to change that, you can use `setContentKey()` and `getContentKey()` to change it to whatever you wish. If, for instance, you changed it to `myContent` in your controller with the statement:

```
$this->_helper->layout->setContentKey('myContent');
```

Then in your template, you would use this line to place the content in your template.

```
$this->layout()->myContent
```

This won't be one of the more common parameters you set but should you need it, it is available to you.

- `viewSuffix` : If you don't {{index views!changing suffix}}want to use the default `.phtml` for your views, you can change it by passing in `viewSuffix`. You do not need to pass in the leading '.', just the suffix you want to use. For instance,

```
$this->_helper->layout->setViewSuffix('myview');
```

This would cause your application to look for templates like `header.myview` instead of `header.phtml`.

Now that we have told our application to use layouts, we need to change a few things. First, we need to set `dir.layouts` in our `config.ini` file. Mine looks like this, you can adjust it to suit your setup.

```
dirs.layouts = "c:\web\htdocs\app\views\layouts"
```

Put this anywhere in the `[main]` section of your `config.ini` and save it.

Next we need to layout our base template. Since the point of this book is not design but programming, I'm going to stick with our retro 1996 look that we have going and not bother with anything pretty like style sheets, graphics, or even colors. I will however, show you where these pieces go so you can play with them. Here is our `main.phtml`:

```
<?PHP echo $this->doctype('XHTML1_TRANSITIONAL') ?>
```

First and foremost, let's set our {index DOCTYPE}}DOCTYPE. The new layout helper predefines eight total DOCTYPES for us, six of which are predefined in the `Doctype.php` helper:

- XHTML1_STRICT
- XHTML1_TRANSITIONAL
- XHTML1_FRAMESET
- HTML4_STRICT
- HTML4_LOOSE
- HTML4_FRAMESET

These are pre-defined in the `Doctype.php` helper. Additionally, it will take 2 other DOCTYPEs, CUSTOM_HTML and CUSTOM. These last two allow you to define

DOCTYPEs that are not covered in the list above, as long as they are properly formed. Note that DOCTYPE is used by some of the helpers - `headStylesheet` and `headScript`, in particular - to determine escaping.

```
<html>

  <head>
    <?PHP echo $this->headTitle()->setIndent(4); ?>
    <?PHP echo $this->headMeta()->setIndent(4);?>
    <?PHP echo $this->headLink()->setIndent(4);?>
    <?PHP echo $this->headStyle()->setIndent(4);?>
    <?PHP echo $this->headScript()->setIndent(4);?>
```

Here we get a feel for the type of functions we are going to use in our layout. The `head*()` functions return exactly what you think they do and the `setIndent()` allows you to control the indenting so that your final source code is nicely formatted. When we get into the controllers we will discuss how to set the content that is returned when these functions are called.

```
  </head>

  <body <?PHP echo $this->layout()->body_onload;?>>
```

Here we deviate from the norm a bit. Some of our actions require that there be an `onBody` method; others do not. To accommodate this, we will create a property of the layout called `body_onload`. When it is necessary to specify an `onLoad`, we simply populate the property with the complete `onLoad` parameter. Here we echo it out. If it's not been set then the layout object will return an empty string.

```
    <?PHP echo $this->partial('header.phtml') ?>
```

Here, we want to render the standard header of our pages. In our case, as you will see next, `header.phtml` displays our user greeting, navigation elements and any messages the system needs to communicate to the user. Be careful though, paths to partials can be tricky. We will discuss where `header.phtml` is located and why in the next section.

```
<div id="content">

  <?PHP echo $this->layout()->content ?>
```

This is where we output the content of the actual action/viewscript being called. If, for instance, we had called `www.example.com/index/index` then `IndexController::indexAction()` would have been called. This is where the contents of `scripts/index/index.phtml` will be output in your template.

```
</div>

<?PHP echo $this->footer;?>
```

Unlike the call to `<?PHP echo $this->partial('header.phtml') ?>` the footer partial is actually rendered elsewhere and stored in the property `footer`. This serves two purposes. First, it shows another example of storing content in properties of the layout. Like `body_onload`, footer doesn't actually exist but is created dynamically, as we need it. Also, there may be times that we don't want to display the footer, or another piece of static content for that matter. We could put the logic in `main.phtml` to make the decision but in many cases that would lead to an overly complex template filled with switches and edge cases. In the case of our footer content, as we will see, there's a very specific reason we don't want to display it and it is only blank on a single page. So instead of making the decision in the view or layout, it makes more sense to make that decision in the controller. We will discuss it as we come to it.

```
<?PHP echo $this->inlineScript() ?>
```

Those of you who know me or who have read my blog know what I think of SEO and SEO experts. All of that aside, it is a known fact that the search engines have a limited attention span. They will only consider a specific number of characters from the top of your document as content. Therefore, in most cases, it is considered good 'SEO' to put any raw JavaScript at the bottom. So I've put my call to `inlineScript()` at the bottom of my layout.

```
</body>
```

```
</html>
```

That's the layout. Now all of our pages will look exactly the same. Let's take a look at the two partials we use, `header.phtml` and `footer.phtml`. `Header.phtml` first.

```
<h2><?PHP echo $this->layout()->pageTitle;?></h2>
```

Ok, here's something new, `layout()`. Let's talk about passing things from the controller to the layout or view script. `Header.phtml` is not a view script. You know this because there is no `headerAction()` in any of the controllers. If it's not a view script then it has to be a *layout*. Layouts like this act just like view scripts except that `$this` does not refer to the current view, it is the layout object. This means that anything you stored in `$this->view` inside of your controller, won't be accessible. To solve this, the Zend Framework creates a view helper called `layout`. `Layout` can be thought of as a registry of the values you need to pass to layouts. Actually, the layout object can be accessed in controllers, views and layouts so it's a good place to store things that you may need everywhere. In this case, we've created a new property, `pageTitle` and we populate it in each of our actions. You will see `layout()` used more in this layout so it's important to understand what to store there and when to use it.

```
<div class="header">

<?PHP

// Display Member info
if ($this->layout()->member) {
  echo "<div id='welcome'>Logged in as: {$this->layout()->member->firstName} {
       $this->layout()->member->lastName}</div>";
} // if ($this->view->member)

?>
```

Again, using the `layout` object, we now have access to our member object in the layout. This allows us to make display decisions based on whether the member is logged in. We also change the navigation elements below based on this. This is one of the changes that you will see when we discuss the `BaseController`, previous exam-

ples stored the member object in the view but as we've discussed, that would have made it inaccessible to any layouts.

```
<?PHP

// now show any messages the system has left us.

if (!empty($this->layout()->message)) {
  echo "<div id='message'>".$this->layout()->message."</div>";
}

?>

</div>

<div id="navigation">
<p><a href="/index/index">Home</a>

<?PHP if ($this->layout()->member) { ?>

   <a href="/history/listSearches">List Searches</a></p>

<?PHP } else { ?>

   <a href="/index/login">Login</a>
   <a href="/index/register">Register</a></p>

<?PHP } ?>

</div>
```

That's the header and probably the most complex layout we have in the system (not that it's all that complex). It demonstrates how and when we need to store things in the placeholder as opposed to just the view. In most cases, you can continue to store things as properties of the view if they are not important on a page-wide basis. Our `footer.phtml` is much simpler.

```
<br /><br />
<div class="footer">

<?PHP echo $this->copyright;?><br />
```

```
</div>
```

Actually, it's overly complex even at that. Since the copyright will really never change, the entire thing could be just a static piece of HTML. However, it's fun to overcomplicate things, especially when showing examples; anyone can do it the simple way.

Finally, the last piece of the layout is `index.phtml`. Now that we've handled all the common HTML, our `index.phtml` can be pared down to just that code necessary to display the contents that is unique to `index.phtml`.

```
<!-- Layout for the index::index action -->

<form method="POST" action="/index/extract" name="mainform">
  <?PHP echo $this->formHidden('token',$this->token); ?>
  <?PHP echo $this->formLabel('url','URL To Analyze'); ?>:<br />
  <?PHP echo $this->formText('url','http://devzone.zend.com',array('onFocus'=>'
      this.select();')); ?><br />
  <?PHP echo $this->formSubmit('submit','Analyze'); ?>
</form>
```

Very simply, this is just our form to submit a URL for analysis. You probably noticed by now that I've replaced all the form elements with calls to view helpers. The `Zend_Layout` adds about 15 new view helpers. We've used the layout helper already but here we use the `formHidden`, `formLabel` and `formSubmit` helpers. These helpers allow me to build my form using PHP instead of writing HTML. My HTML is always formatted exactly the same and most importantly, it's syntactically correct. All of the properties that I would normally set by hand can be passed in via the third parameter which is an array of options.

One disadvantage of using the `Zend_Layout` is that, like OOP, your code is spread out among a lot of small files, it can sometimes be difficult to find the right place to make a modification. The upside however, is that your view scripts now only contain the logic necessary for the view script itself. `Zend_Layout` will stitch everything together for us.

I won't go into the changes necessary for the other view scripts that we created, if you are interested, grab the `example9.zip` and peel it open. You will see that all of the existing view scripts have been similarly stripped of the common code and implement only the code necessary for their particular action.

Ok, through this point, we have seen how to set things up and display them. However, without a few minor changes to our controllers, nothing is going to work right. Each of the controllers has minor changes but to conserve space, I'm only going to show snippets that show the changes.

First, let's look at `BaseController`. In `BaseController`, the `init()` changed so much that I'll just show you the whole thing.

```
public function init()
{
  $memberSession = new Zend_Session_Namespace('member');
  if($memberSession->member) {
    $this->view->layout()->member = $memberSession->member;
  }
```

When we were discussing `header.phtml`, we discussed the fact that we have to store the member object in the placeholder to make it available to both the layout and the view. This used to be stored in `$this->view->member`.

```
  $this->view->headMeta('my, list, of, cool, keywords,
                         goes, here','keywords');
```

Back when we were discussing the layout file, `main.phtml`, we talked about the `head*()` methods. One of the methods was `$this->headMeta()->setIndent(4)`. This line shows how to add things to the `headMeta` property. If `headMeta` is called without any input parameters then it outputs all of the head meta tags that have been defined. However, if it's called with two parameters then it will add them to the head registry for later output. In the case of `headMeta`, the first parameter is the content and the second is the keyword. This line will eventually output the following in the final source:

```
<meta name="keywords" content="my, list, of, cool, keywords, goes, here" />
```

Next we'll take a look at this:

```
  $this->view->headTitle(Globals::getConfig()->title);
```

Similar to `headMeta()`, `headTitle()`, if passed a parameter will set the title of the page. If called without a parameter, it will output the stored value for the title of the page.

```
$this->view->layout()->footer =

$this->view->partial('footer.phtml',
          null,
          array('copyright'=>"(c) 2008 Cal Evans, All Rights Reserved")) ;
```

Here is the call to the footer partial. We discussed earlier that displaying `footer` is conditional. Here in `BaseController::init()` we execute the partial and store the results in the view property `footer`. Then, in the actual action we can decide if we want to leave it or clear it. As we will see in `IndexController`, there is actually one page I don't want to display it on.

This call to `partial()` illustrates something important though, the ability to pass in parameters and values to the `partial`. In `footer.phtml`, we use the variable `$this->copyright`, since we are executing this as a `partial`, it doesn't have the view as a context, therefore, anything we want it to use, we have to pass into the call. The third parameter of `partial()` is an associative array of view parameters. (The second one is an array of module parameters, if you are interested. I've managed to stay away from modules this far and I have no intention of breaking that rule now) Any key you pass in on this parameter will be accessible as `$this->key` with its value properly populated.

Now, let's take a look at `IndexController` and the changes that were necessary on the controller level to implement `Zend_Layout`.

The first method that changes is, the `indexAction()` method. Here's a snippet:

```
public function indexAction()
{
  $this->view->token = $this->generateToken();
  $this->_helper->layout()->body_onload='onload="document.mainform.url.focus()
       ;"';
```

The last line is where we set the `body_onload` property we defined back in the `main.phtml` layout.

```
.
.
.
    $this->view->layout()->pageTitle = "Home";
```

At the very end of each action, we now set the property `pageTitle`. We place it in the placeholder because `header.phtml` is going to use this in the defined `H2` tag.

So from a controller's action point of view, the changes are minimal. Our code would have basically worked just as it had been written before but we would have missed the `onLoad` and we would have had an empty `H2` tag set.

In `riaAction()`, we see one new concept, the use of `headScript()` to pass in JavaScript files to be linked in from our `main.phtml`.

```
$this->view->headScript('file',"/js/prototype.js");
$this->view->headScript('file',"/js/scriptaculous.js");
```

Since this is the only action that needs any external JavaScript, we only use the `headScript()` methods here. `headScript()` takes five parameters. The first tells it what type of action it will be taking. Passing in `FILE` tells it that this will be a linked file and generate a simple `<script>` tag pair with a `src` option. Passing in `SCRIPT` tells it that the next parameter (the second parameter) is the contents of a script and should be echoed straight into the file.

The third parameter is `placement`. It determines whether the tag set or script is appended or prepended to the list that will be output. The default for placement is `append`, similar to adding things to the bottom of an array. In our case, we leave it off because we are adding the scripts in the order we want the output.

The fourth parameter is an array of attributes. You can specify three different properties in this array.

- `charset`
- `defer`
- `language`

If you specify one of these three, it will be included in the `script` tag.

The fifth parameter of `headScript` is `type`. The default for this is `text/JavaScript` as any other script type is valid here. This is not validated so it is entirely possible to specify a type that a browser cannot understand. Be careful when setting your script type.

Beyond the use of `headScript()`, `riaAction()` has only one other point of interest, this is the action where we do not show the footer. The reason the footer is not displayed on the RIA page is simple, I needed an example to show this concept. The second to last line in the `riaAction()` method is:

```
$this->view->layout()->footer  = '';
```

Since we put this in the `BaseController::init()`, clearing it here, before we build our template, effectively destroys that content and removes it from the page.

You can examine the code for the other controllers to see where things have been added or changed but as you will see, most of them are just variations on what we've discussed.

I'd like to say one final word about `Zend_Layout` and layout files before moving on to the final section of this Appendix. You have to be careful where you put your `.phtml` files and where you look for them. In most cases, it acts just like you would expect. However, in the case of `header.phtml`, we have a problem. This partial is being called from within a controller. Since it is not in the controller action, but in the `BaseController::init()` `Zend_Layout` cannot determine where the file should be. Therefore it looks for it in the root `scripts` directory, not the root `layout` directory. `Zend_Layout` considers the `partial()` a view script, not a layout script and paths it accordingly. The `header.phtml` however, because it is being called from the layout and not the view, is treated as a layout. If you look at the `views/` directory you will see that `header.phtml` is in `layouts` while `footer.phtml` is in `scripts`.

In the final section of this Appendix I want to talk about how `Zend_Layout` affects our API controller. At first blush, the answer is that it doesn't. Since the API returns JSON or XML, we don't really need layouts. This, for the most part, is correct. That is why, in `ApiController::init()` we've added a line.

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

```
    $this->_helper->layout->disableLayout();
  } // public function init()
```

The second line turns off layouts for all actions in this controller. If you don't add that then it will take your API payload and wrap it in the `main.phtml`. I think we can all agree that this is not the desired behavior. So for all but our `fetch*Action()` methods, we simply turn it off. However, our `fetch*Action()` methods all return an XML payload. To do this, they all pass their array into our `toXml()` method that we've previously described. However, there is a way to use the new `Zend_Layout` to do this differently. I'm discussing this because it's an option, not necessarily a better option. `toXml()` uses SimpleXML to build the payload and it does a great job. However, the XML we are generating is brain-dead simple and even if it wasn't, we don't need to parse it, just build it. I've created a new method called `ApiController::complexToXML()`. It takes the same parameters as `ApiController::toXml()` and outputs the same XML, only the method has changed. None of the techniques we will show in this piece of code are new, we've talked about them previously. However, this is a unique application of the `Zend_Layout`. Remember, we turned off the default behaviors for layout. So here, we make calls to `partial()` to build our XML in a string.

Here is the new method, `complexToXml()`, along with a discussion of what is going on.

```
public function complexToXml($dataArray,$callingMethod,$parameterValue)
{
  $words = '';
  $xml   = '';
  foreach($dataArray as $keywordArray) {
    $words .= $this->view->partial('api/word.phtml',
                          array('keyword'       => $keywordArray['keyword'],
                                'keyword_count' => $keywordArray['keyword_count
                                    ']));
  } // foreach($results as $keywordArray)
```

This `for/next` calls `word.phtml`. `word.phtml`, as you will see below is an XML snippet that builds the XML for one word. Here we spin through our array of words and concatenate a string.

```
    $xml = $this->view->partial('api/complex-to-xml.phtml',
                  array('words'          => $words,
                        'callingMethod'  => $callingMethod,
                        'parameterValue' => $parameterValue,
                        'rowCount'       => count($dataArray)));
```

Now, we take the `$words` string that we built previously and pass it to `complex-to-xmp.phtml` along with the other parameters necessary to complete the XML payload.

```
    return $xml;
  } // public function complexToXml($dataArray,$callingMethod,$parameterValue)
```

Now let's look at the two partial layouts we are using. First `word.phtml`:

```
<word>
  <keyword><?PHP echo $this->keyword;?></keyword>
  <keyword_count><?PHP echo $this->keyword_count;?></keyword_count>
</word>
```

As we discussed, this contains all the XML necessary to deliver a single term. Now the main layout, `complex-to-xml.phtml`:

```
<?xml version='1.0' encoding='utf-8'?>
<data>
  <results>
    <calling_method><?PHP echo $this->callingMethod;?></calling_method>
    <parameter><?PHP echo $this->parameterValue;?></row_count>
    <row_count><?PHP echo $this->rowCount;?></row_count>
  </results>
  <words>
  <?PHP echo $this->words; ?>
  </words>
</data>
```

This is the entire template with placeholders for each of the parameters and one for the snippet containing the words.

That's it, we are simply using the `Zend_Layout` to layout our XML instead of our HTML. This concept, using `Zend_Layout` for non-traditional payload creation, can be

used in just about any situation. Whether you need JSON, CSV, YAML or a custom payload, `Zend_Layout` can be used for more than just layouts.

Summary:

- Watch where you store things. Using `Zend_Layout` introduces an extra level of complexity. Sometimes your normal storage schemes will work and sometimes they won't. Make sure you know when to use each storage container when passing variables to the view.
- `Zend_Layout` is great for implementing layouts in your design. You can have different layouts for different areas of your site, or, as we did, one ugly layout for the entire site. `Zend_Layout` can, however, be used for more than that. Anytime you need formatted output to send as a payload, `Zend_Layout` can help you organize it and reduce or eliminate repeated lines.

We showed the source code for modifying the `IndexController.php` and two of its view scripts. However, to properly implement `Zend_Layout` we had to touch each view script. Since in almost every case, we did the same thing over and over, it didn't seem necessary to print each modified view script. If you are curious, you can unpack `example9.zip` and see what changes were made.

Index

E