

ZEND Framework in Action

Rob Allen
Nick Lo

MEAP

Unedited Draft

 MANNING





**MEAP Edition
Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=329>

Licensed to Menshu You <dollequatki@gmail.com>

Table of Contents

Part 1: The essentials

1. Introducing the Zend Framework
2. Hello Zend Framework!

Part 2: A core application

3. Building a web site with the Zend Framework
4. Ajax
5. Managing the database
6. User authentication and authorisation
7. Forms
8. Searching
9. Email
10. Deployment

Part 3: More power to your application

11. Talking with other applications
12. Mash ups with public web services
13. Caching: making it faster
14. Internationalization and localization
15. Creating PDFs
16. Integrating with other PHP libraries

Appendix A. Stuff you (should) already know

Appendix B. System-specific gotchas

Appendix C. Zend Framework Core Components reference

Introducing the Zend Framework

PHP has been used to develop dynamic websites for over 10 years. Initially all PHP websites were written as PHP code interspersed within HTML on the same page. This works very well initially as there is immediate feedback and for simple scripts this appears to be all that is needed. PHP grew in popularity through versions 3 and 4, and so it was inevitable that larger and larger applications would be written in PHP. It became obvious very quickly that intermixing PHP code and HTML was not a long term solution for large websites.

The problems are obvious in hindsight: maintainability and extensibility. Whilst PHP intermixed with HTML allows for extremely rapid results, in the longer term it is hard to continue to update the website. One of the really cool features of publishing on the web is that it is dynamic with content and site layouts changing. Large websites change all the time. The look and feel of the site is updated regularly. New content is added and content is regularly re-categorized as the needs of the users (and advertisers!) change. Something had to be done!

The Zend Framework was created to help ensure that the production of PHP based websites is easier and maintainable in the long term. It contains a rich set of reusable components containing everything from a set of Model-View-Controller application components to PDF generation. Over the course of this book, we will look at how to use all the components within the context of a real website.

1.1 Introducing structure to PHP websites

The solution to this tangled mess of PHP code and HTML on a website is structure. The most obvious introduction to structured applications within PHP sites is applying the concept of “separation of concerns”. This means that the code that does the display should not be in the same file as the code that connects to the database and collects the data as shown in Figure 1.1.

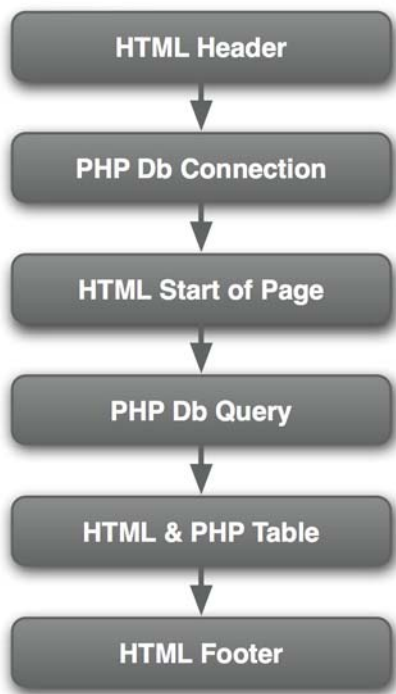


Figure 1.1: The organization of a typical PHP file created by a novice interleaves HTML and PHP code in a linear fashion as the file is created.

The first stages of introducing structure to a website's code happen by default for most developers; the concept of reusability dawns. Generally, this means that the code that connects to the database is separated into a file called something like db.inc.php. Then it seems logical to separate out the code that displays the common header and footer elements on every page. Functions are introduced to help solve the problem of global variables affecting one another

As the website grows, common functionality is grouped together into libraries. Before you know it, the application is much easier to maintain and adding new features becomes possible again. This stage lasts for a little while and the website continues to expand until it gets to the point where the supporting code is so large that you can't hold a picture of how it all works in your head.

PHP coders are used to standing on the shoulders of giants as our language provides easy access to libraries such as the GD image library, the many database client access libraries and even system specific libraries such as COM on Windows. It was inevitable that Object-Oriented Programming would enter the PHP landscape. Whilst classes in PHP4 provided little more than glorified arrays, PHP5 provides excellent support for all the things you'd expect in an object oriented language. Hence there are visibility specifiers for class members (public, private and protected) along with interfaces, abstract classes and support for exceptions.

The improved object-oriented support allows for more complicated libraries (known as frameworks) to evolve, such as the Zend Framework which supports a way of organizing web application files known as the MVC design pattern. This is shown in Figure 1.2.

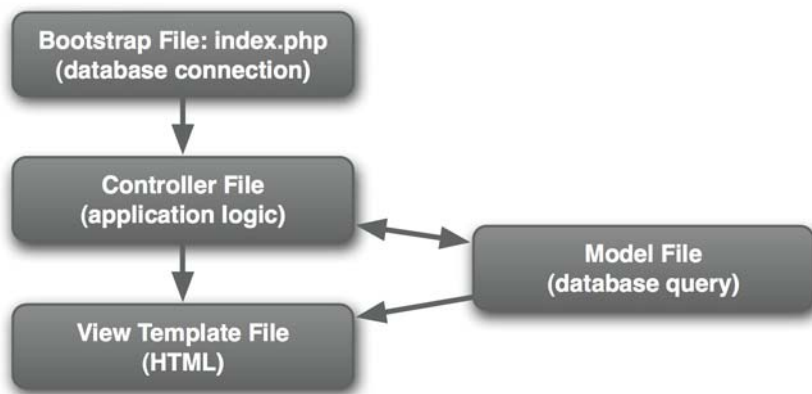


Figure 1.2: The organization of a typical MVC application

An application designed using MVC principles results in more files. Each file is specialized in what it does which makes maintenance much easier. For example, all the code that makes database queries is stored in classes known as Models. The actual HTML code is known as the View (which may also contain simple PHP logic) and the Controller files handle the connection of the correct models to the correct views to display the desired page.

The Zend Framework isn't the only option for organizing a website based on MVC principles; there are many others in the PHP world. Let's look at what the Zend Framework contains and why it should be considered.

1.2 Why use the Zend Framework?

As you have this book in your hands, you probably want to know why you'd be interested in the Zend Framework over all the other PHP frameworks out there. In a nutshell, the Zend Framework introduces a standardized set of components that allow for easy development of web applications. These applications can be easily developed, maintained and enhanced.

The key features of the Zend Framework are:

- Everything in the box
- Modern design
- Easy to learn
- Full documentation
- Simpler Development
- Rapid development

1.2.1 Everything in the box

The Zend Framework is a comprehensive full stack framework that contains everything you need to develop your application. This includes a robust MVC component to ensure that your website is structured according to best practices. Accompanying the MVC component, there are components for authentication, searching, localization, PDF creation, email and connecting to web services, along with a few other more esoteric items as shown in Figure 1.2.



Figure 1.3: The Zend Framework can be divided into ten main modules

That's not to say that the Zend Framework doesn't "play nice" with other libraries; it does that too. A core feature of the design of the framework is that it is easy to use just those bits you want to use with the rest of your application or with other libraries such as PEAR, the Doctrine ORM or the Smarty template library.

1.2.2 Modern design

The Zend Framework is written in object-oriented PHP5 using the modern design techniques, known as design patterns. Software design patterns are recognized high level solutions to design problems and, as such, are not a specific implementation of the solution. The actual implementation depends on the nature of the rest of the design. The Zend Framework makes use of many design patterns and its implementation has been carefully designed to allow the maximum flexibility for application developers without making them do too much work!

The framework recognizes the PHP way and doesn't force you into using all the components, so you are free to pick and choose between them. This is especially important as it allows you to introduce specific components into an existing site. The key is that each component within the Framework has very few dependencies on other components.

1.2.3 Easy to learn

If you are anything like me, learning how a vast body of code works is hard! Fortunately the Zend Framework is modular and has a design goal of simplicity which makes it easy to learn, one step at a time. Each component doesn't depend on lots of other components and so is easy to study. The design of each component is such that you do not need to understand how it works in its entirety before you can use it and benefit from it. Once you have some experience of using the component, building up to use the more advanced features is straight-forward as it can be done in steps. This is key to reducing the barrier to entry for most users.

For example, the configuration component `Zend_Config` is used to provide an object oriented interface to a configuration file. It supports two advanced features: section overloading and nested keys, but neither of these features need to be understood in order to use the component. Once the user has a working

implementation of Zend_Config in their code, confidence increases and so using the advanced features is a small step.

1.2.4 Full documentation

No matter how good the code is, lack of documentation can kill a project through lack of adoption. The Zend Framework is aimed at developers who do not want to have to dig through all the source code to get their job done and so the Zend Framework puts documentation on an equal footing with the code. This means that the core team will not allow new code into the framework unless it has accompanying documentation.

There are two types of documentation supplied with the framework: API and end-user. The API documentation is created using PHPDocumenter and is automatically generated using special “docblock” comments in the source code. These comments are typically found just above every class, function and member variable declaration. The key advantage of using docblocks is that IDEs such as PHPIDE in Eclipse or Zend’s Studio are able to supply auto-completion tool tips whilst coding and so improve developer productivity.

The Zend Framework also supplies a full manual as part of the download and also available online at <http://framework.zend.com/manual>. The manual provides details on all components of the framework and shows what functionality is available. Examples are provided to help you get started in using the component in an application. More importantly, in the case of the more complicated components (such as Zend_Controller), the theory of operation is also covered, so that you can understand why the component works the way it does.

The documentation provided with the framework does not explain how to fit all the components together to make a complete application. As a result, a number of tutorials have sprung up on the web by community members to help developers get started on the framework. These have been collated on a web page on the framework’s wiki at <http://framework.zend.com/wiki/x/q>. The tutorials, whilst a useful starting point, do not tend to go depth with each component or show how it works within a non-trivial application, which is why this book exists.

1.2.5 Simpler development

As we have noted, one of PHP’s strengths is that developing simple dynamic web pages is very easy. This has enabled millions of people to have fantastic websites who may not have had them otherwise. The ability of PHP programmers range from people who are beginners to programming through to enterprise developers needing to meet their deadlines. The Zend Framework is designed to make development simpler for every type of developer.

So how does it make development simpler? The key feature that the framework brings to the table is tested, reliable code that does the “grunt” work of an application. This means that the code you write is the code you need for your application. The code that does the “boring” bits is taken care of for you and is not cluttering up your code.

1.2.6 Rapid Development

The Zend Framework makes it easy to get going on your web application or add new functionality to a current website. As the framework provides many of the underlying components of an application, you are free to concentrate on the core parts of your application, rather than on the underlying foundation. Hence, it is easy to get started quickly on a given piece of functionality and immediately see the results.

Another way the framework speeds up development is that the default use of most components is the common case. In other words, you don’t have to worry having to set lots of configuration settings for each

component just so that you can get started using it. For example, the most simplistic use of the whole MVC is bootstrapped with just:

```
require_once('Zend/Loader.php');
Zend_Loader::registerAutoload();
Zend_Controller_Front::run('/path/to/controllers');
```

Once up and running, adding a new page to your application can be as easy as adding a new function to a class, along with a new template file in the correct directory. Similarly, Zend_Session provides a multitude of options that can be set so that you can manage your session exactly how you want to, however none need to be set in order to use the component for most use-cases.

1.2.7 Structured code is easy to maintain

As we have seen earlier, separating out different responsibilities makes for a structured application. It also means finding what you are looking for is easier whilst bug fixing. Similarly, when you need to add a new feature to the display code, the only files you need to look at are related to the display logic. This avoids bugs created inadvertently by breaking something else. The framework also encourages you to write object oriented code, which helps to ensure that maintenance of your application is simpler.

1.3 What is the Zend Framework?

The Zend Framework is a PHP library for building PHP web application. It provides a set of components to enable you to build PHP applications more easily which will be easier to maintain and extend over the lifetime of the application. That rather simple description doesn't tell the whole story though, so we will look at where this framework came from and then have a brief look at what it actually contains

1.3.1 Where did it come from?

Frameworks have been around for years. The very first web framework I used in a real project was Fusebox which was originally written for ColdFusion. Many other frameworks have come along since then, with the next major highlight being Struts, written in Java. A number of PHP clones of Struts were written, but didn't translate well to PHP. The biggest difference being that Java web applications run in a virtual machine that runs continuously, so the startup time of the application is not a factor for every web request. PHP initializes each request from a clean slate and so the large initiation required for Struts clones made them relatively slow as a result. Recently, a new framework entered the world based on a relatively unknown language called Ruby. Rails (or Ruby on Rails as it is also known) promoted the concept of convention over configuration and has taken the web development world by storm. Shortly after Rails came along, a number of direct PHP clones have appeared, along with a number of frameworks that are inspired by Rails, rather than direct copies.

In late 2005, Zend Technologies, a company that specializes in PHP, started their PHP Collaboration project to advance the use of PHP. There are three strands to the project: an eclipse IDE plugin called PDT, the Zend Framework and the Zend Developer Zone website. The Zend Framework is an open source project that provides a web framework for PHP and is intended to become one of the standard frameworks that PHP applications of the future will be based on.

1.3.2 What's In it?

The Zend Framework is composed of many distinct components grouped into a set of top level modules. As a complete framework, you have everything you need to build enterprise ready web applications. However, the system is very flexible and has been designed so that you can pick and choose to use those bits of the framework that are applicable to your situation. Following on from the high level overview in Figure 1.3 shown earlier, Figure 1.4 gives a good overview of all the components within the framework.

<div>Core:</div> <div>Zend_Controller</div> <div>Zend_View</div> <div>Zend_Db</div> <div>Zend_Config</div> <div>Zend_Filter & Zend_Validate</div> <div>Zend_Registry</div> <div>Authentication and Access:</div> <div>Zend_Acl</div> <div>Zend_Auth</div> <div>Zend_Session</div> <div>Internationalization:</div> <div>Zend_Date</div> <div>Zend_Locale</div> <div>Zend_Measure</div> <div>Http:</div> <div>Zend_Http_Client</div> <div>Zend_Http_Server</div> <div>Zend_Uri</div>	<div>Inter-application communication:</div> <div>Zend_Json</div> <div>Zend_XmlRpc</div> <div>Zend_Soap</div> <div>Zend_Rest</div> <div>Web Services:</div> <div>Zend_Feed</div> <div>Zend_Gdata</div> <div>Zend_Service_Amazon</div> <div>Zend_Service_Flickr</div> <div>Zend_Service_Yahoo</div> <div>Advanced:</div> <div>Zend_Cache</div> <div>Zend_Search</div> <div>Zend_Pdf</div> <div>Zend_Mail/Zend_Mime</div> <div>Misc!</div> <div>Zend_Measure</div>
---	---

Figure 1.4: The Zend Framework contains lots of components that cover everything required to build an enterprise application.

Each section of the framework consists of a number of components, which is usually the name of the main class too. For example, Zend_View is the concrete view class used by applications. Each component also contains a number of other classes too that are not listed in Figure 1.4. The classes that are actually used within your application are discussed as we go through the book and learn about each component.

The Core Components

The core components provide a full-features Model-View-Controller (MVC) system for building applications that separate out the view templates from the business logic and controller files. There are three families of classes that make up the MVC system: Zend_Controller (Controller), Zend_View (View) and Zend_Db (Model). Figure 1.5 shows the basics of the Zend Framework's MVC system.

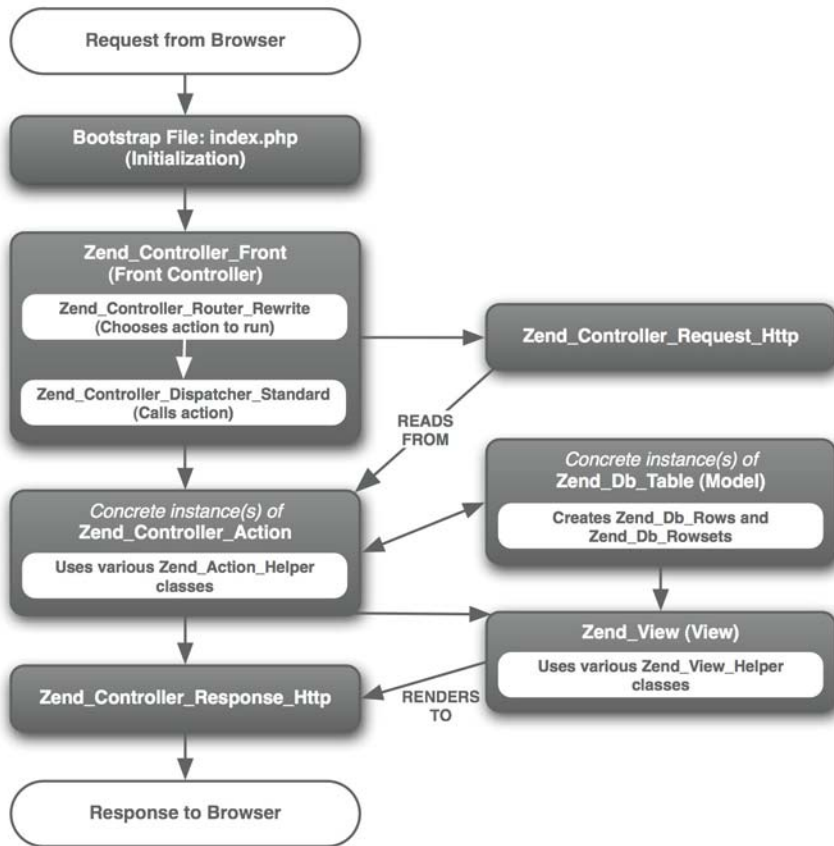


Figure 1.5: MVC: the Zend Framework way

The `Zend_Controller` family of classes provides a front controller design which dispatches requests to controller actions (also known as commands) so that all processing is centralized. As you'd expect from a fully featured system, the controller supports plug-ins at all levels of the process and has built in flex-points to enable you to change specific parts of the behavior without having to do too much work.

The view template system is called `Zend_View` which provides a PHP based template system. This means that, unlike Smarty or PHPTAL, all the view templates are written in PHP. `Zend_View` provides a helper plug-in system to allow for creation of reusable display code. It is designed to allow for overriding for specific requirements, or even replacing entirely with another template system such as Smarty.

`Zend_Db_Table` implements a table row gateway pattern to form the basis of the model within the MVC system. The model provides the business logic for the application which is usually database-based in a web application. Supporting `Zend_Db_Table` is `Zend_Db` which provides object oriented, database independent access to a variety of different databases, such as MySQL, Postgres, SQL Server, Oracle and SQLite.

The most simplistic setup of the MVC components can be done with the very simple code:

```
require_once 'Zend/Controller/Front.php';
Zend_Controller_Front::run('/path/to/your/controllers');
```

It is more likely, however, that a more complicated bootstrap file will be required for a non-trivial application as we will explore in chapter 2 when we build a complete Hello World application in Zend Framework.

Working with the MVC classes are a couple of separate classes that are used to create the core of a complete application. The framework encourages convention over configuration, however some configuration is invariably required (such as database login details). `Zend_Config` allows for reading configuration data in

either INI or XML formats and includes a useful inheritance system for supporting different configuration settings on different servers, such as production, staging and test.

Security is very much on the minds of every PHP developer worth his salt. Input data validation and filtering is the key to a secure application. `Zend_Filter` and `Zend_Validate` are provided to help the developer ensure that input data is safe for use in the application.

The `Zend_Filter` class provides a set of filters that typically remove or transform unwanted data from the input as it passes through the filter. For example, an numeric filter would remove any characters that were not numbers from the input and an HTML entities filter would convert the “<” character to the sequence “<”. Appropriate filters can then be set up to ensure that the data is valid for the context it will be used in.

`Zend_Validate` provides a very similar function to `Zend_Filter`, except that it provides a yes/no answer to the question “is this data what I expect?”. Validation is generally used to ensure that the data is correctly formed, such as the data provided as an email address is actually an email address. In the case of failure, `Zend_Validate` also provides a message indicating why the input failed validation so that appropriate error messages can be provided back the end user.

Authentication and Access Components

Not every application needs to identify their users, but it is a surprisingly common requirement. Authorization is the process of providing access to a given resource, such as a web page, to an authenticated user. That is, authentication is the process of identifying an entity, usually via a token such as a username/password pair, but could equally be via a fingerprint. Authorization is the process of deciding if the authenticated entity is allowed to have access to, or perform operations on, a given resource, such as a record from a database.

As there are two separate processes required, the Zend Framework provides two separate components: `Zend_Acl` and `Zend_Auth`. `Zend_Auth` is used to identify the user and is typically used in conjunction with `Zend_Session` to hold that information across multiple page requests (known as token persistence). `Zend_Acl` is then used to use the authentication token to provide access to private information using the Role Based Access Control List system.

As is becoming a watchword around here, flexibility is a key design decision within the `Zend_Auth` component. There are so many ways to authenticate a user that the `Zend_Auth` system is built with the intention that the user will provide their own. The most common scenario of HTTP digest authentication is provided out of the box, but for any other method, you must create a class that extends `Zend_Auth_Adapter`. Fortunately, this is not difficult as we will see in chapter 6.

As `Zend_Acl` is an implementation of the Role Based Access Control List system, the manual talks in lots of abstract terms. This is because RBACL is a generic system that can provide access to anything by anyone and so specific terms are discouraged. Hence we talk about Roles requesting access to Resources. A Role is anything that may want to access something that is under the protection of the `Zend_Acl` system. Generally, for a web application, this means that a Role is a user that has been identified using `Zend_Auth`. A Resource is anything that is to be protected. This is generally a record in a database, but could equally be an image file stored on disk. As there is such a wide type of resources, the `Zend_Acl` system provides for us to create our own very simply by implementing `Zend_Acl_Role_Interface` within our class.

Internationalization Components

As we live in a multi-cultural world with multiple languages, the framework provides a rich set of functionality to allow for localizing your application to match your target users. This covers minor issues like ensuring that the correct currency symbol is used through to full support for changing all the text on the page to the correct

language. Date and time routines are also provided with a simple object oriented interface to the multitude of ways that we humans display the calendar.

Http Components

The Zend Framework provides a component to read data from other websites. `Zend_Http_Client` makes it easy to collect data from other web sites and services and then present it on your site. A server component is also provided to allow for PHP based serving for web pages. Obviously this component is intended for development and other specialized requirements rather than general web page serving, as proper web-servers like Apache are orders of magnitude faster!

Inter-application Communication Components

When you need to communicate with another application over HTTP, the most common transfer format is one of two flavors of XML: XML-RPC and SOAP. As you would expect from an enterprise-class framework, the Zend Framework provides components to allow for easy processing of both XML-RPC and SOAP protocols. More recently, the lightweight JSON protocol is gaining favor, mainly due to how easy it is to process within the JavaScript of an Ajax application. `Zend_Json` provides a nice solution to both creating and reading JSON data.

Web Services Components

The Zend Framework provides a rich set of functionality to allow access to services provided by other suppliers. These components cover generic RSS feeds along with specific components for working with the public APIs from Google, Yahoo! and Amazon. RSS has come a long way from its niche amongst the more technologically minded bloggers and is now used by the majority of news sites. `Zend_Feed` provides a consistent interface to reading feeds in the various RSS and atom versions that are available without having to worry about the details.

Google, Yahoo! and Amazon have provided public APIs to their online services in order to encourage developers to create extended applications around the core service. For Amazon, the API revolves around providing access to the data on amazon.com in the hope that the new application will encourage more sales! Similarly, Yahoo! provides API access to their Flickr photo data in order to allow additional services for Flickr users, such as the print service provided by moo.com. The traditional Yahoo! properties such as search, news and images are also available via `Zend_Service_Yahoo`.

Google has a number of online applications that allow for API access which are supported by the `Zend_Gdata` component. The `Zend_Gdata` component provides access to Google's Blogger, Calendar, Base and CodeSearch applications. In order to provide consistency, the `Zend_Gdata` component provides the data using `Zend_Feed`, so if you can process an RSS feed, then you can process Google Calendar data too.

Advanced Components

There are a set of other components provided with the Zend Framework that do not fit easily into any category, so I have rather lazily grouped them together into the advanced category. This potpourri of components includes caching, searching, pdf creation, email and the rather esoteric measurement class.

Everyone wants a faster website and caching is one tool that can be used to help speed up your website. Whilst not a sexy component, the `Zend_Cache` component provides a generic and consistent interface to cache any data in a variety of back end systems such as disk, database or even with APC's shared memory. This flexibility ensures that you can start small with `Zend_Cache` and as the load on your site increases, the caching solution can grow up with you to help ensure you get the most out of the server hardware.

Every modern website provides a search facility. Most provide a terrible search facility; so terrible that the site's users would rather search Google than use the site's own system. Zend_Search is based on the Apache Lucene search engine for Java and provides an industrial strength text search system that will allow your users to find what they are looking for. As required by a good search system, it supports ranked searching so that the best results are at the top, along with a powerful query system.

Another component that I've lumped into the advanced category is Zend_Pdf which covers the creation of PDF files programmatically. PDF is a very portable format for creating documents intended for printing. This is because you can control the position of everything on the page with pixel-perfect precision without having to worry about differences in the way web browsers render the page. Zend_Pdf is written entirely in PHP and can create new PDF documents or load existing one for editing.

Email Components

The Zend Framework provides a strong email component to allow for sending emails in plain text or HTML. As with all Zend Framework components, emphasis has been placed on flexibility combined with sensible defaults. Within the world of email, this means that the component allows for sending email using SMTP or via the standard PHP mail() command. Additional transports can be easily slotted into the system by writing a new class that implements Zend_Email_Transport_Interface. When sending email, a simple object-oriented interface is used:

```
$mail = new Zend_Mail();
$mail->setBodyText('My first email!')
    ->setBodyHtml('My <b>first</b> email!')
    ->setFrom('rob@akrabat.com', 'Rob Allen')
    ->addTo('somebody@example.com', 'Some Recipient')
    ->setSubject('Hello from Zend Framework in Action!')
    ->send();
```

This snippet also shows a fluent interface where each member function returns an object to itself so that they can be chained together to make the code more readable.

1.4 Zend Framework design philosophy

The Zend Framework has a number of published goals that make up the design philosophy of the framework. If these goals do not mesh with what your view on developing PHP applications then the Zend Framework is unlikely to be a good fit for your way of doing things. Let's look at what makes the Zend Framework unique.

1.4.1 Provide high quality components

All code within the Zend Framework library will be of high quality. This means that it will be written using PHP5's features and will not generate any messages from the PHP parser (i.e. it is E_STRICT compliant). This is good as it means that any PHP parser messages in your logs come from your code, not the framework; this will help debugging considerably! Zend also defines high quality to include documentation, so the manual for a given component is as important as the code.

It is intended that it will be possible to develop entire applications with no external library dependencies (unless you want them). This means that the Zend Framework is intended to be a "full stack" framework (like Ruby on Rails or the Django Python framework) rather than a set of components. This will ensure that there will be consistency in the way you use all the components: how they are named, how they work and how the files are laid out in sub directories. Having said that, it is important to Zend that the Zend Framework is modular with few dependencies between modules. This ensures that it plays well with other frameworks and

libraries and you can therefore use as much or as little as you want. For example, if you just want PDF generation, you don't have to use the MVC system.

1.4.2 Simple as possible

Another design goal for the framework is the mantra "Don't change PHP". The PHP way is simple, pragmatic solutions and so the Zend Framework is intended to reflect that simplicity in order to provide a simple solution for mainstream developers. It is also powerful enough to allow for specialized usage via extension. The core developers have done a great job in covering the common scenarios and providing "flex-points" to allow for easy changing of the default behavior by those who want something more powerful or specialized.

1.4.3 Clean IP

All contributors to the Zend Framework have signed a Contributor License Agreement. This is an agreement with Zend which defines intellectual property status of the contribution. That is, the contributor warrants that (to the best of her knowledge), she is entitled to make the contribution and that no one else's intellectual property rights are being infringed. This is intended to help protect all users of the framework from potential legal issues related to IP and copyright. The risk is minimal, but with relatively recent actions by SCO against AutoZone shows that a litigator going after the user of the allegedly copyright infringing code is a possibility. As with everything, it is better to be prepared.

1.4.4 Supported by Zend Technologies

An obvious but important consideration is that the Zend Framework is supported by the company Zend Technologies. This means that the framework is unlikely to "die" due to inactivity of the core developers or by lack of updating to the latest and greatest version of PHP. Zend Technologies also have the resources to provide for the "boring" bits of the framework, such as documentation, which (as we all know) few developers really *like* doing!

1.5 Alternative PHP frameworks

As usage of PHP is so broad, no one framework is going to suit everyone. In the PHP world there are many other frameworks vying for your attention and all have their strengths and weaknesses. I have rather arbitrarily picked four other frameworks which all have some traction in the community but these are by no means the only choices. I have listed what I see as their strengths and weaknesses in Table 1.1.

Table 1.1 Key features matrix: Zend Framework, Cake, Code Igniter, Solar and Symfony

	Zend Framework	Cake	Code Igniter	Solar	Symfony
Uses PHP5 to full advantage	Yes	No	No	Yes	Yes
Prescribed directory structure	No (Optional recommended structure)	Yes	Yes	No	Yes
Official internationalization support	Yes	In progress for version 1.2	No	Yes	Yes
Command line scripts required to setup framework	No	No	No	No	Yes
Requires configuration	Yes (minor amount)	No	No	Yes	Yes

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=329>

Comprehensive ORM provided	No	Yes	No	No	Yes (Propel)
Good documentation and tutorials	Yes	Yes	Yes	Yes	Yes
Unit tests for source available	Yes	No	No	Yes	Yes
Community support	Yes	Yes	Yes	Yes (some)	Yes
License	New BSD	MIT	BSD-style	New BSD	MIT

Whilst this book is all about the Zend Framework, the other frameworks are worth investigating to see if they match better with your requirements. If you still need to support PHP 4, then you will need to use either Cake or Code Igniter as the rest do not support PHP 4. In this day and age, however, it's time to leave PHP 4 behind us!

1.6 Summary

In this chapter we have looked at what the Zend Framework is and why it is useful for writing web applications. It enables rapid development of enterprise applications by providing a full stack framework using best practices in object oriented design. The framework contains many components from an MVC controller through PDF generation to providing a powerful search tool.

This book is about providing real world examples and so will have Ajax technology built in wherever it is appropriate. Now, let's move onto the core topic of this book and look at a how to build a simple, but complete Zend Framework application in Chapter 2.

Hello Zend Framework!

In chapter 2, we are going to look at a simple Zend Framework application that will display “Hello World!”. For a standard PHP application, the code to do this constitutes one line in one file:

```
<?php echo 'Hello World';
```

As the Zend Framework requires many more files in order to create the foundation from which a full website can be created. As a result, the code for our Hello World application may appear unnecessarily verbose as we set the stage for the full blown website that will follow in the remainder of the book. We will also consider how to organize the website’s files on disk to make sure we can find what we are looking for and look at the Zend Framework files required to create an application that uses the Model-View-Controller (MVC) design pattern. Let’s dive right in and look at what the Model-View-Controller design pattern is all about first.

2.1 The Model-View-Controller Design Pattern

In order to make sense of a Zend Framework application we need to cover a little bit of theory. There are many framework classes involved along with a few files that we need to create ourselves. Therefore we should cover the basics of the controller system used by the Zend Framework first.

The Zend Framework controller system is an implementation of the Model-View-Controller software design pattern as shown in Figure 2.1. A software design pattern is a standard general solution to a common problem. This means that whilst the exact implementation will differ, the concepts used to solve a problem using a given pattern will be the same. The MVC pattern describes a way to separate out the key parts of an application into three main sections.

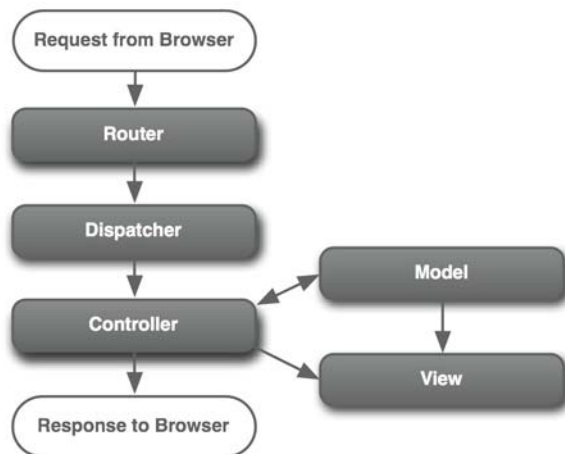


Figure 2.1: MVC pattern diagram showing the three main sections of a web application along with the dispatcher that find the correct controller to be executed in response to a request.

2.1.1 The Model

The model part of the MVC pattern is all the code that works behind the scenes related to how this particular application works. This is known as business logic. This is the code that decides how to apply the shipping cost to an e-commerce order or the code that knows that a user has a first name and a surname. It follows therefore that retrieving and storing data to a database is within the model layer. In terms of the code, the Zend Framework provides the `Zend_Db_Table` class which provides table level access to the database and allows for easily manipulating the data used by the application.

2.1.2 The View

The view is the display logic of the application. For a web application, this is usually the HTML code that makes up the web pages, but can include, say, XML that is used for an RSS feed. Also, if the website allows for export in CSV format, the generation of the CSV would be part of the view. The view files themselves are known as templates as they usually have some code that allows for the displaying of data created by the model. It is also usual to move the more complex template related code into functions known as View Helpers, View Helpers improve the re-usability of the view code. By default the Zend Framework's view class (`Zend_View`) uses PHP within the template files, but another template engine such as Smarty or PHPTAL may be substituted.

2.1.3 The Controller

The controller is the rest of the code that makes up the application. For web applications, the controller code is the code that works out what to actually run in response to the web request. For Zend Framework applications, the controller system is based on the design pattern known as Front Controller which uses a handler (`Zend_Controller_Front`) and action commands (`Zend_Controller_Action`) which work together in tandem. The front controller handler accepts all server requests and runs the correct action function within the action command. This process is known as routing and dispatching. The action class is responsible for a group of related action functions which perform the “real” work required from the request. Within the Controller of the Zend Framework, it is possible to have a single request result in the dispatch of multiple actions.

2.2 The Anatomy of a Zend Framework Application

A typical Zend Framework application has many directories in it. This helps to ensure that the different parts of the application are separated. The top level directory structure is shown in Figure 2.2.



Figure 2.2: Directory layout for a standard Zend Framework application

There are four top level directories within the application's folder:

1. application
2. library
3. tests
4. web_root

2.2.1 The application directory

The application directory contains all the code required to run the application and is not directly accessed by the web server. In order to emphasize the separation between display, business and control logic, there are three separate directories within application to contain the model, view and controller files. Other directories may be created as required, for example for configuration files.

2.2.2 The library directory

All applications use library code as *everyone* reuses previously written code! In a Zend Framework application, the framework itself is obviously stored here. However other libraries such as a custom super-set of the framework, a database ORM library such as Propel, or a template engine such as Smarty may also be used.

Libraries can be stored anywhere that the application can find them - either in a global directory or a local one. A global include directory is one that is accessible to all PHP applications on the server, such as `/usr/php_include` (or `c:\code\php_include` for Windows) and is set using the `include_path` setting within the `php.ini` configuration file. Alternatively, each application can store its libraries locally within the application's directory. In this case we use a directory called `library`, though it is common to see this directory called `lib`, `include` or `inc`.

2.2.3 The tests directory

The tests directory is used to store all unit tests that are written. Unit tests are used to help ensure that the code continues to work as it grows and changes throughout the lifetime of the application. As the application is

developed, code that was written previously often needs to be changed (known as refactored) ready for the addition of new functionality or as a result of other code added to the application. Whilst, within the PHP world, test code is rarely considered important, you will thank yourself over and over again if you have unit tests for your code.

2.2.4 The `web_root` directory

To improve the security of a web application, the web server should only have direct access to the files that it needs to serve. As the Zend Framework uses the front controller pattern, all web requests are channeled through a single file, usually called `index.php`. This file is the only PHP file that needs to be accessible by the web server and so is stored in the `web_root/` directory. Other common files that are accessed directly are images, CSS and JavaScript files, so each has their own sub-directory within the `web_root` directory.

Now that we have an overview of the directory system used by a Zend Framework website, we can proceed to add the files required to create a very simple application that displays some text on the page.

2.5 Hello World: File by File

To create a simple Hello World application we need to create four files within our directory structure: a bootstrap file, an Apache control file (`.htaccess`), a controller file and a view template. A copy of the Zend Framework itself needs to be added to the library directory. The final program will look as shown in Figure 2.3.

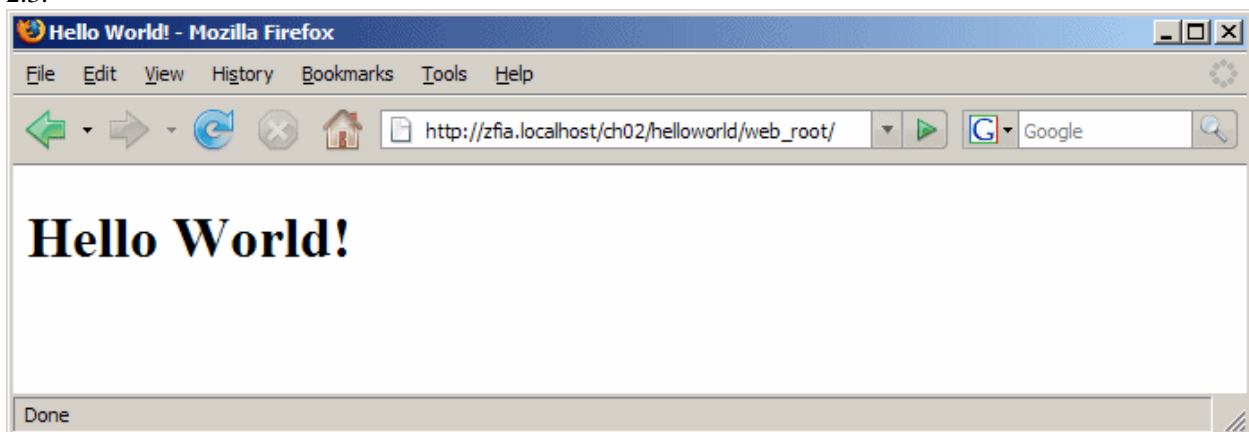


Figure 2.3: A minimal Zend Framework application requires a bootstrap file, a controller and a view working together to produce the text “Hello World!”.

2.5.1 Bootstrapping

Bootstrapping is the term used to describe starting the application up. With the Front Controller pattern, this file is the only file needed in the web root directory and so is usually called `index.php`. As this file is used for all page requests, it is used for setting up the application’s environment, setting up the Zend Framework’s controller system and then running the application itself as shown in listing 2.1.

Listing 2.1: `web_root/index.php`

```
<?php
error_reporting(E_ALL|E_STRICT);                | #1
ini_set('display_errors', true);                |
date_default_timezone_set('Europe/London');     |

$rootDir = dirname(dirname(__FILE__));
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=329>

```

set_include_path$rootDir . '/library'                                |#2
    . PATH_SEPARATOR . get_include_path());                          |

require_once 'Zend/Loader.php';
Zend_Loader::loadClass('Zend_Debug');
Zend_Loader::loadClass('Zend_Controller_Front');

// setup controller
$frontController = Zend_Controller_Front::getInstance();            #3
$frontController->throwExceptions(true);                             #4
$frontController->setControllerDirectory('../application/controllers');

// run!
$frontController->dispatch();

(annotation) <#1: Setup environment>
(annotation) <#2: Set the path>
(annotation) <#3: Zend_Controller_Front is a Singleton>
(annotation) <#4: Throw exceptions. Don't do this in production!>

```

Let's look at this file in more detail. Most of the work done in the bootstrap is initialization of one form or another. Initially, the environment is set up correctly (#1) to ensure that all errors or notices are displayed. PHP 5.1 introduced new time and date functionality that needs to know where in the world we are. There are multiple ways to set this, but the easiest user-land method is `date_default_timezone_set()`.

The Zend Framework is written with the assumption that the library directory is available on the `php_include` path. There are multiple ways of doing this and the fastest for a global library is to alter the `include_path` setting directly in `php.ini`. A more portable method, especially if you use multiple versions of the framework on one server, is to set the include path within the bootstrap as we do here (#2).

The Zend Framework applications does not depend on any particular file, however it is useful to have a couple of helper classes loaded early. `Zend_Loader::loadClass()` is used "include" the correct file for the supplied class name. The function converts the underscores in the class's name to directory separators and then, after error checking, includes the file. Hence the code line `Zend_Loader::loadClass('Zend_Controller_Front');` and `include_once 'Zend/Controller/Front.php';` have the same end result. `Zend_Debug::dump()` is used to output debugging information about a variable by providing a formatted `var_dump()` output.

The final section of the bootstrap sets up the front controller and then runs it. The front controller class, `Zend_Controller_Front` implements the Singleton design pattern (#3). This means that the class definition itself ensures that there can only be one instance of the object allowed. A Singleton design is appropriate for a front controller as it ensures that there is only ever one class that is processing the request. One of the consequences of the Singleton design is that you cannot use the new operator to instantiate it and must, instead, use the `getInstance()` static member function. The front controller has a feature that captures all exceptions thrown by default and stores them into the Response object that it creates. This Response object holds all information about the response to the requested URL and for HTML applications this is the HTTP headers, the page content and any exceptions that were thrown. The front controller automatically sends the headers and displays the page content when it finishes processing the request.

For our Hello World application, I've decided to instruct the front controller to throw all exceptions that occur (#4). The default behaviour to store exceptions within the response object can be quite confusing for people new to the Zend Framework, so let's turn it off and force the error to be displayed. Of course, on a production server, you shouldn't be displaying errors to the user anyway and so you should either let the controller catch exceptions or wrap the `index.php` code with a `try/catch` block yourself.

To actually run the application we call the front controller's `dispatch()` method. This function will automatically create a request and response object for us to encapsulate the input and output of the application. It will then create a router to work out which controller and action the user has asked for. A dispatcher object is then created to load the correct controller class and call the action member function that does the "real" work.

Finally, as we've noted above, the front controller outputs the data within the response object and so a web page is displayed to the user.

2.5.2 Apache .htaccess

To ensure that all web requests that are not for images, scripts or style sheets are directed to the bootstrap file, Apache's `mod_rewrite` module is used. This can be configured directly in Apache's `httpd.conf` file or in a local Apache configuration file named `.htaccess` that is placed in the `web_root/` directory. Listing 2.2 shows the `.htaccess` file required for the Zend Framework:

Listing 2.2: web_root/.htaccess

```
# Rewrite rules for Zend Framework
RewriteEngine on
RewriteCond %{REQUEST_FILENAME} !-f      #1
RewriteRule .* index.php                 #2
```

(annotation) <#1 Only continue if requested URL is not a file on disk.>

(annotation) <#2 Redirect request to index.php.>

Fortunately, this is not the most complicated set of Apache `mod_rewrite` rules and so can be easily explained. The `RewriteCond` statement and the `RewriteRule` command between them instruct Apache to route all requests to `index.php` unless the request maps exactly to a file that exists within the `web_root/` directory tree. This will allow us to serve any static resources placed in the `web_root` directory, such as JavaScript, CSS and image files, whilst directing any other requests to our bootstrap file allowing the front controller to work out what to display to the user.

2.5.3 Index Controller

The front controller pattern maps the URL requested by the user to a particular member function (the action) within a specific controller class. This process is known as routing and dispatching. The controller classes have a strict naming convention requirement in order for the dispatcher to find the correct function. The router expects to call a function named `{actionName}Action()` within the `{ControllerName}Controller` class. This class must be within a file called `{ControllerName}.php`. If either the controller or the action are not provided, then the default used is `index`. Therefore, a call to `http://zfia.example.com/` will result in the "index" action of the Index controller running. Similarly, a call to `http://zfia.example.com/test` will result in the `index` action of the test controller running. As we will discover later, this mapping is very flexible, however the default covers most scenarios out of the box.

Within the front controller system, the dispatcher expects to find a file called `IndexController.php` within the `application/controllers` directory. This file must contain a class called `IndexController` and, as a minimum, this class must contain a function called `indexAction()`. For our Hello World application, Listing 2.3 shows the `IndexController.php` required.

Listing 2.3: The index controller: application/controllers/IndexController.php

```
<?php
Zend::LoadClass('Zend_View');
```

```

class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $ this->view->assign('title', 'Hello World!'); #1
    }
}

```

(annotation) <#1 Assign the title to the view property.>

As you can see, IndexController is a child class of Zend_Controller_Action which contains the request and response objects for access within the action functions, along with a few useful helper functions to control the program flow. For Hello World, our indexAction() function just needs to assign a variable to the view property which is provided for us by an Action Helper called ViewRenderer.

NOTE

An Action Helper is a class that plugs into the controller to provide services specific to actions.

The ViewRenderer action helper performs two useful functions for us. Firstly, before our action is called, it creates a Zend_View object and sets it to the action's \$view property allowing us to assign data to the view within the action. Secondly, after our action finishes it automatically renders the correct view template into the response object after the controller action has completed. This ensures that our controller's action functions can concentrate on the real work and not on the framework "plumbing".

What is the "correct view template" though? The ViewRenderer looks for a template file named after the action with a phtml extension within a folder named after the controller and it looks in the view/scripts directory for this. This means that for the index action within the index controller, it will look for the view template file view/scripts/index/index.phtml.

As we noted previously, the response's body is automatically printed by the front controller, so anything we assign to the body will be displayed in the browser and hence we do not need to echo ourselves.

Zend_View is the View component of the MVC troika and is a fairly simple PHP based template system. As we have seen, the assign() function is used to pass variables from the main code body to the template which can then be used within the view template file.

View Template

Finally, we need to provide a view template for our application. This file, index.phtml, is stored within the views/scripts/index subdirectory. A useful convention that ViewRenderer supplies is to name all view files with an extension of .phtml as a visual indication that they are for display only. Of course it's easily changed by setting the \$_viewSuffix property of ViewRenderer. Even though this is a simple application, we have a separate directory for each controllers view templates as this will make it much easier to manage as the application grows. Listing 2.4 shows the view template.

Listing 2.4: The view template: views/scripts/index/index.phtml

```

<!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" lang="en" xml:lang="en">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title><?php echo $this->escape($this->title);?></title> #1
</head>
<body>
    <h1><?php echo $this->escape($this->title);?></h1>
</body>

```

</html>

(annotation) <#1 convert special characters to their HTML entity representations.>

As, Zend_View is a PHP based template engine, we use PHP within the file to display data from the model and controller. The template file, index.phtml in this case, is executed within a member function of Zend_View and so \$this is available within the template file which is the gateway to Zend_View's functionality. All variables that have been assigned to the view from within the controller are available directly as properties of \$this, as can be seen by the use of \$this->title within index.phtml. Also, a number of helper functions, are provided for use by templates to make them easier to write.

The most commonly used helper function is `escape()`. This function is used to ensure that the output is HTML-safe and helps to secure your site from Cross-Site Scripting (XSS) attacks. All variables that are not expected to contain displayable HTML should be displayed via the `escape()` function. Zend_View's architecture is designed so that creating new helper functions is encouraged. For maximum flexibility, the convention is that view helper functions return their data and then the template file echoes it to the browser.

With these four files in place, we have created a minimal Zend Framework application with all the pieces in place ready for building a full scale website and you should now have a fundamental understanding of how the pieces fit together. We will now look at what is happening within the Zend Framework's code which is providing the MVC foundation that our code has been built upon.

2.6 How MVC Applies to the Zend Framework

Whilst there appears to be many different ways of routing web requests to code within web applications, they can all be grouped into two camps: page controllers and front controllers. A page controller uses separate files for every page (or group of pages) that make up the website and is traditionally how most PHP websites have been built. This means that the control of the application is decentralized across lots of different files which can result in repeated code, or worse, repeated and slightly altered code leading to issues such as lost sessions when one of the files doesn't do a `session_start()`.

A front controller, on the other hand, centralizes all web requests into a single file, typically called `index.php`, which lives in the root directory of the website. There are numerous advantages to this system; the most obvious are that there is less duplicated code and that it is easier to separate the URLs that a website has from the actual code that is used to generate the pages. Usually, the pages are displayed using two additional GET parameters passed to the `index.php` file to create URLs such as `index.php?controller=news&action=list` to display a list page.

If we recap from chapter 1, Figure 2.4 shows the Zend Framework classes that implement the MVC design pattern. There are three separate components used and within each component, more than one class is required for the application.

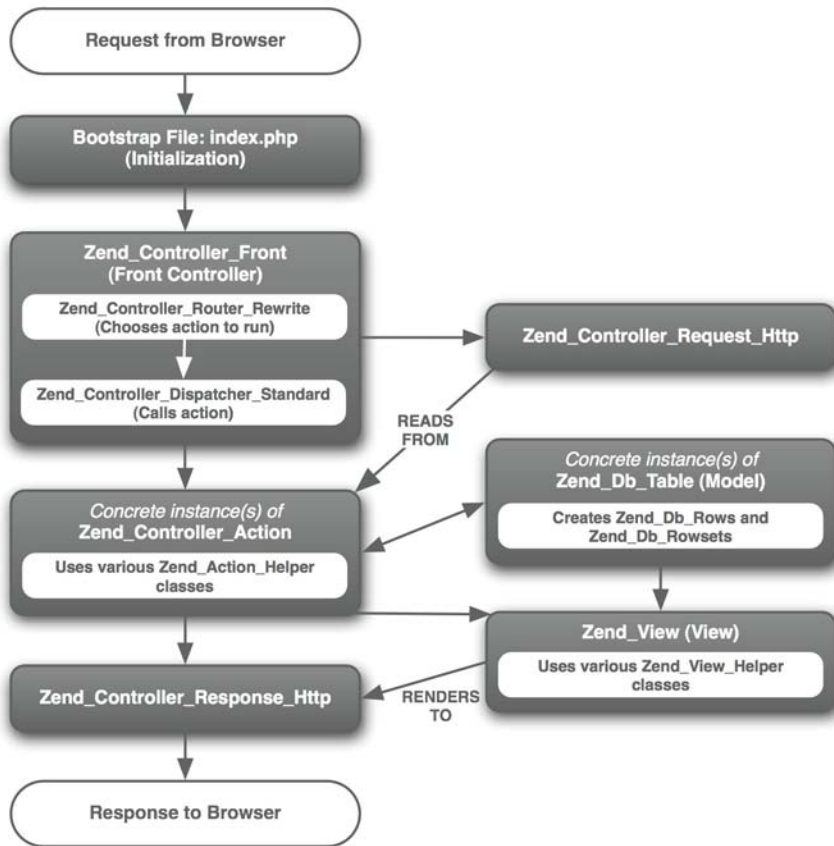


Figure 2.4: The interaction of the various Zend Framework classes used to create an MVC application

One important aspect of all modern web applications is that the URL looks “good” so that it is more memorable for users and is also easier for search engines like Yahoo! or Google to index the pages of the website. An example friendly URL would be `www.example.com/news/list` and it should come as no surprise that the Zend Framework’s front controller uses a sub-component, known as a router, support friendly URLs by default.

2.6.1 The Zend Framework’s Controller

The Zend Framework’s front controller code is spread over a number of classes that work together to provide a very flexible solution to the problem of routing a web request to the correct place to do the work.

`Zend_Controller_Front` is the foundation and it processes all requests received by the application and delegates that actual work to action controllers.

Request

The request is encapsulated within an instance of `Zend_Controller_Request_Http` which provides access to the entire HTTP request environment. What is a request environment though? It is all the variables received by the application from outside the application along with relevant controller parameters such as the controller and action router variables

The HTTP request environment contains all the super globals (`$_GET`, `$_POST`, `$_COOKIE`, `$_SERVER` and `$_ENV`) along with the base path to the application. The router also places the module, controller and

action names into the request object once it has worked them out. `Zend_Controller_Request_Http` provides the function `getParam()` to allow the application to collect the request variables and so the rest of the application is protected from a change in environment. For example, a command line request environment wouldn't contain the HTTP specific items, but would include the command line arguments passed to the script. Thus the code:

```
$items = $request->getParam('items');
```

will work unchanged when run as a web request or as a command line script.

In general, The request object should be treated as read only to the application as, implicitly, the values set by the user shouldn't be changed. Having said that, `Zend_Controller_Request_Http` also contains parameters which can be set in the start up phase of the application and then retrieved by the action functions as required. This can be used for passing additional information from the front controller to the action functions if required.

Routing

Routing is the process of determining which controller's action needs to be run in order to satisfy the request. This is performed by a class that implements `Zend_Controller_Router_Interface` and the framework supplies `Zend_Controller_Router_Rewrite` which will handle most routing requirements. Routing works by taking the part of the URI after the base URL (known as the URI endpoint) and decomposing it into separate parameters. For a standard URL such as `http://example.com/index.php?controller=news&action=list` the decomposition is done by simply reading the `$_GET` array and looking for the 'controller' and 'action' elements. As a modern framework, it is expected that most applications built using the Zend Framework will use pretty URLs of the form `http://example.com/news/list`. In this case, the router will use the `$_SERVER['REQUEST_URI']` variable to determine the which controller and action has been requested.

Dispatching

Dispatching is the process of actually calling the correct function in the correct class. As with everything in the Zend Framework, the standard dispatcher provides enough functionality for nearly every situation, but if you need something special, it is easy to write your own and fit it into the front controller. The key things that the dispatcher controls are formatting of the controller class name, formatting of the action function name and calling the action function itself.

`Zend_Controller_Dispatcher_Standard` is where the rules concerning case are enforced, such that the name format of the controller is always TitleCase and only contains alphabetic characters. The dispatcher's `dispatch()` method is responsible for loading the controller class file, instantiating the class and then calling the action function within that class. Hence, if you decided that you wanted to reorganize the structure so that each action lived in its own class within a directory named after the controller, you would supply your own dispatcher.

The Action

`Zend_Controller_Action` is an abstract class that all action controllers are derived from. The dispatcher enforces that your action controllers derive from this class to ensure that it can expect certain methods to be available. The action contains an instance of the request for reading parameters from and an instance of the response for writing to. The rest of the class concentrates on ensuring that writing actions and managing changes from one action to another one are easy to do; There are accessor functions to get and set parameters, and redirection functions to redirect to another action or another URL entirely.

Assuming that the standard dispatcher is used, the action functions are all named after the action's name with the word "Action" appended. You can therefore expect a controller action class to contain functions such as `indexAction()`, `viewAction()`, `editAction()`, `deleteAction()` etc. Each of these are discrete functions that are

run in response to a specific URL. There are, however, a number of tasks that you will want to do regardless of which action is run. `Zend_Controller_Action` provides two levels of functionality to accommodate this requirement: `init()` and `pre/postdispatch()`. The `init()` function is called whenever the controller class is constructed. This makes it very similar to the standard constructor, except that it does not take any parameters and does not require the parent function to be called.

`preDispatch()` and `postDispatch()` are a complementary pair of functions that are run before and after each action function is called. For an application where only one action is run in response to a request, there is no difference between `init()` and `preDispatch()` as each are only called once. If, however, the first action function uses the `_forward()` function to pass control to another action function, then `preDispatch()` will be run again, but `init()` will not be. To illustrate this point, we could use `init()` to ensure that only administrators are allowed access to any action function in the controller and `preDispatch()` to set the correct view template file that will be used by the action.

The Response

The final link in the front controller chain is the response. For a web application `Zend_Controller_Response_Http` is provided, but if you are writing a command line application, then `Zend_Controller_Response_Cli` would be more appropriate. The response object is very simple and is essentially a bucket to hold all the output until the end of the controller processing. This can be very useful when using front controller plugins as they could alter the output of the action before it is sent back to the client.

`Zend_Controller_Response_Http` contains three types of information: header, body and exception. In the context of the response, the headers are HTTP headers, not HTML headers. Each header is an array containing a name along with its value and it is possible to have two header with the same name but different values within the response's container. The response also holds the HTTP response code (as defined in RFC 2616) which is sent to the client at the end of processing. By default, this is set to 200 which means OK. Other common response codes are 404 (Not Found) and 302 (Found) which is used when redirecting to a new URL. As we will see later, the use of status code 304 (Not Modified) can be very useful when responding to requests for RSS feeds as it can save considerable bandwidth.

The body container within the response is used to contain everything else that needs to be sent back to the client. For a web application this means everything you see when you view source on a web page. If you are sending a file to a client, then the body would contain the contents of the file. For example, to send a pdf file to the client, the following code would be used:

```
$filename = 'example.pdf';
$response = new Zend_Controller_Response_Http();

// set the HTTP headers
$response->setHeader('Content-Type', 'application/pdf');
$response->setHeader('Content-Disposition',
    'attachment; filename="'.$filename.'"');
$response->setHeader('Accept-Ranges', 'bytes');
$response->setHeader('Content-Length', filesize($filename));

// load the file to send into the body
$response->setBody(file_get_contents($filename));

echo $response;
```

The final container within the response object houses the exceptions. This is an array that can be added to by calling `$response->setException()` and is used by `Zend_Controller_Front` to ensure that errors within your

code are not sent to the client, possibly exposing private information that may be used to compromise your application. Of course, during development, you would want to see the errors, so the response has a setting, `renderExceptions`, that you can set to true so that the exception text is displayed.

Front Controller Plug-ins

The front controller's architecture contains a plug-in system to allow user code to be executed automatically at certain points in the routing and dispatching process. All plug-ins are derived from `Zend_Controller_Plugin_Abstract` and there are six event methods that can be overridden:

1. `routeStartup()` is called just before the router is executed.
2. `dispatchLoopStartup()` is called just before the dispatcher starts executing.
3. `preDispatch()` is called before each action is executed.
4. `postDispatch()` is called after each action is executed.
5. `dispatchLoopShutdown()` is called after all actions have been dispatched.
6. `routeShutdown()` is called after the router has finished.

As you can see, there are three pairs of hooks into the process at three different points which allow for increasingly finer control of the process.

One problem with the current router is that if you specify a controller that does not exist, then an exception is thrown. A front controller plug-in is a good way to inject a solution into the routing process and redirect to a more useful page. The Zend Framework supplies the `ErrorHandler` plug-in for this purpose and its use is very well explained in the manual.

Now that we have looked in detail at the controller part of MVC, it's time to look at the View part as provided for by the `Zend_View` component.

2.6.2 Understanding Zend_View

`Zend_View` is a class for keeping the view portion of an MVC application separated from the rest of the application. It is a PHP template library which means that the code in the templates is in PHP rather than another pseudo-language like Smarty for instance. However, it is easy to extend `Zend_View` to support any other template system.

Assigning data to the view

In order for the view to display data from the model, it is necessary to assign it. `Zend_View`'s `assign()` method allows for assigning simple variables using `$view->assign('title', 'Hello World!');` or you can assign multiple variables simultaneously using an associative array:

```
$music = array('title'=>'Abbey Road', 'artist'=>'The Beatles');  
$music = array('title'=>'The Wall', 'artist'=>'Pink Floyd');  
$view->assign($music);
```

As we are using PHP5, we can also take advantage of the `__set()` magic method to write `$view->title = 'Hello World!';` and it will work exactly the same way and the data from the model or controller is now available to the view template.

The view template

A view template is just like any other regular PHP file, except that its scope is contained within an instance of a `Zend_View` object. This means that it has access to all the methods and data of `Zend_View` as if it was a function within the class. The data that we assigned to the view are public properties of the view class and so

are directly accessible. Also, helper functions are provided by the view to make writing view templates easier. A typical view script might look like:

```
<h1>Glossary</h1>
<?php if($this->glossary) :?>
<dl>
<?php foreach ($this->glossary as $item) : ?>
<dt><?php echo $this->escape($item['term']);?></dt>
<dd><?php echo $this->escape($item['description']);?></dd>
<?php endforeach; ?>
</dl>
<?php endif; ?>
```

As you can see, this script is a PHP script with an HTML bias as the PHP commands are always contained within their own `<?php` and `?>` tags. Also, we have used the alternate convention for control loops so that we don't have braces within separate PHP tags as matching braces can be quite tricky when using lots of separate PHP tags.

Note that we do not trust the glossary data that has been assigned to the script. It could have come from anywhere! In the code accompanying this book, the data is created using an array, but it could equally have come from the users of a website. As we do not want any cross site scripting security vulnerabilities in our website, we use the helper function `escape()` to ensure the term and description do not have any embedded HTML.

View Helper Functions

Zend_View contains a number of helpful functions to make writing templates easier and allows for you to write your own. These are functions are known as View Helpers and exist in their own directory. As we have already seen, the most common view helper is the `escape()` function which is built into the Zend_View class itself. Every other helper exists in its own class and is automatically loaded by Zend_View. Let's create a simple formatting helper for displaying a cash amount. Consider that we need to display a monetary value that may be negative. In the UK, for a value of 10, the display would be £10.00 and for a value of -10, then the display would be -£10.00.

We would use this helper in our templates like this:

```
<p>He gave me <?php echo $this->formatCurrency(10);?>.</p>
```

Which outputs the correctly formatted amount as shown in Figure 2.5.

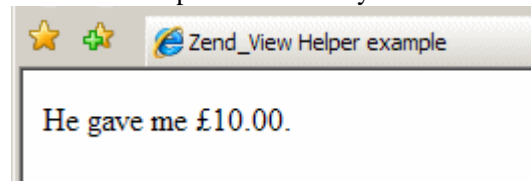


Figure 2.5: Front controller plug-in correctly routing to the IndexController's noRoute action.

To separate our helpers from the default ones, we will use the class prefix `ZFiA_View_Helper` and so our helper class is called `ZFiA_View_Helper_FormatCurrency` as shown in listing 2.6.

Listing 2.6: ZFiA_Helper_FormatCurrency view helper

```
class ZFiA_View_Helper_FormatCurrency {

    public function formatCurrency($value, $symbol='&pound;') #1
    {
        $output = $value;
        $value = trim($value);
        if (is_numeric($value)) { #2
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=329>

```

        if ($value >= 0) {
            $output = $symbol . number_format($value, 2);
        } else {
            $output = '-' . $symbol . number_format(abs($value), 2);
        }
    }
    return $output;
}
}

```

(annotation) <#1 Helper function is named using camelCase.>

(annotation) <#2 If \$value is not a number, then ignore it.>

#2 shows a security consideration. If we don't know that \$value is a number, then we do not return it as part of the output. This helps to ensure that we do not inadvertently introduce an XSS vulnerability.

The name of the function within the helper class is the same as the function that is called within the template, formatCurrency() in our case. Internally, Zend_View has an implementation of the __call() magic function to find our helper class and execute the formatCurrency() function. In order to find it though, we need to register the directory and class prefix with Zend_View using the setHelperPath member function:

```
$view->setHelperPath('./Helper', 'ZFIA_View_Helper');
```

This allows us to have many helper functions within the same directory. The file containing ZFIA_View_Helper_FormatCurrency must be called FormatCurrency.php and must be in the Helper directory. Note that, in a break from the usual convention within the framework, there is no actual requirement that FormatCurrency.php must exist in the directory ZFIA/View/Helper; it is placed in the directory that was registered with setHelperPath(). It is wise to follow the convention though as it makes finding files easier for the developer!

View helpers are the key to extracting common code from your view templates and ensuring that they are easy to maintain and should be used whenever possible to simplify the view files.

Security considerations

When writing the view code, the most important security issue to be aware of is Cross Site Scripting (also known as XSS). Cross site scripting vulnerabilities occur when unexpected HTML, CSS or Javascript is displayed by your website. Generally, this happens when a website displays data created by a user without checking that it is safe for display. As an example, this could happen when the text from a comment form contains HTML and is displayed on a guestbook page “as is”.

One of the more famous examples of an XSS exploit is the MySpace worm known as Samy. This used specially crafted JavaScript in the profile that was displayed when you made Samy your friend. The JavaScript would run automatically whenever anyone viewed the page and if you were logged into MySpace, then it made Samy your “friend”. Thus whenever anyone looked at your page, they were also made “friends” of Samy’s. This resulted in an exponential increase in friends for Samy. Fortunately, the code wasn’t too malicious and didn’t steal each user’s passwords along the way as over 1 million MySpace profiles were infected within 20 hours.

The easiest way to preventing XSS vulnerabilities is to encode the characters that have special meaning in HTML. That is, we need to change all instances of < to <, & to & and > to > so that the browser treats them as literals rather than HTML. Within the Zend Framework, we use the helper function escape() to do this. Every time that you display a PHP variable within a template file, you should use escape() unless you need it to contain HTML in which case, you should write a sanitizing function to allow only HTML codes that you trust.

2.6.3 The Model in M-V-C

We have spent a lot of time in the chapter talking about controller and the view as these are the minimum required for a hello world application. In a real application though, the model side of MVC will take on more importance as this is where the business logic of the application resides. In most cases, the model is linked in some way to a database which will hold data to be manipulated and displayed by the application.

Database abstraction with Zend_Db

Zend_Db is the Zend Framework's database abstraction library which provides a suite of functions to insulate your code from the underlying database engine. This is most useful for those cases when you need to scale your application from using, say SQLite to MySQL or Oracle. Zend_Db uses the factory design pattern to provide the correct database-specific class based on the parameters passed into the factory() function. For example, to create a Zend_Db object for MySQL you would use:

```
$params = array ('host'      => '127.0.0.1',
                'username' => 'rob',
                'password' => '*****',
                'dbname'   => 'zfia');
$db = Zend_Db::factory('PDO_MYSQL', $params);
```

The Zend_Db abstraction is mostly built upon PHP's PDO extension which supports a wide range of databases. There is also support for DB2 and Oracle outside of PDO, though as they all extend from Zend_Db_Adapter_Abstract, the interface is essentially the same, regardless of the underlying database.

So, what you do get in Zend_Db that you don't get in PDO itself then? Well, you get lots of helper functions to manipulate the database and also a profiler to work out why your code is so slow! There are all the standard functions for inserting, updating and deleting rows, along with fetching rows. The manual is particularly good at describing all these functions, so let's move on and consider security.

2.6.4 Security Issues with databases

The most common type of database security problems are known as SQL injection security breaches. These occur when your user is able to trick your code into running a database query that you didn't intend to happen. Consider this code:

```
$result = $db->query("SELECT * FROM users
WHERE name='" . $_POST['name'] . "'");
```

This typical code might be used to authorize a user after they have submitted a login form. The coder has ensured that the correct superglobal, \$_POST, is used, but hasn't checked what it contains. Suppose that \$_POST['name'] contains the string "' OR 1 OR name = '" (single quote, followed by "OR 1 OR name=" followed by another single quote). This would result in the perfectly legal SQL statement of:

```
SELECT * from users where name=' ' OR 1 OR name= ' '
```

As you can see, the OR 1 in the SQL statement will result in all the users being returned from the database table. With SQL injection vulnerabilities like this, it can be possible to retrieve username and password information or to maliciously delete database rows causing your application to stop working.

As should be obvious, the way to avoid SQL injection attacks is to ensure that the data that you are putting into the SQL statement has been escaped using the correct functionality for your database. For MySQL, you would use the function mysql_real_escape_string() and for PostgreSQL, you would use pg_escape_string(). As we are using Zend_Db, we can use the member function quote() to take care of this issue. The quote() function will call the correct underlying database specific function and if there isn't one, then it will escape the string using the correct rules for the database involved. Usage is very easy:

```
$value = $db->quote("It's a kind of magic");
```


An alternative solution is to use parameterized queries, where variables are denoted by placeholders and are substituted by the database engine with the correct variable. The Zend_Db provides the quoteInto() function for this. For example:

```
$sql = $db->quoteInto('SELECT * FROM table WHERE id = ?', 1);
$result = $db->query($sql);
```

Higher level interaction with Zend_Db_Table

When considering the model of an MVC application, we don't tend to want to work at the level of database queries if we can help it. The framework provides Zend_Db_Table, a table row gateway pattern that provides a higher level abstraction for thinking about data from the database. Zend_Db_Table uses Zend_Db behind the scenes and provides a static class function, setDefaultAdapter() for setting the database adapter to be used for all instances of Zend_Db_Table.

```
$db = Zend_Db::factory('PDO_MYSQL', $params);
Zend_Db_Table::setDefaultAdapter($db);
```

We don't actually use Zend_Db_Table directly. Instead, we create a child class that represents the database table we wish to work with. For the purposes of this discussion, we will assume that we have a database table called "news" with the columns id, date_created, created_by, title, and body to work with. We now create a class called News:

```
Class News extends Zend_Db_Table
{
    protected $_name = 'news';
}
```

The \$_name property is used to specify the name of the table. If it is not provided then Zend_Db_Table will use the name of the class and is case-sensitive. Zend_Db_Table also expects a primary key called id (which is preferably automatically incremented on an insert). Both these default expectations can be changed by initializing the protected member variables \$_name and \$_primary respectively. For example:

```
class LatestNews extends Zend_Db_Table
{
    protected $_name = 'news';
    protected $_primary = 'article_id';
}
```

The LatestNews class uses a table called "news" which has primary key called "article_id".

As Zend_Db_Table implements the Table Data Gateway design pattern, it provides a number of functions for collecting data including find(), fetchRow() and fetchAll(). The find() function is used to find rows by primary key and the fetch methods are used to find rows using other criteria. The only difference between fetchRow() and fetchAll() is that fetchRow() returns a single rowset, whereas fetchAll() returns an array of rowsets. Zend_Db_Table also has helper functions for inserting, updating, and deleting rows with the functions insert(), update() and delete() respectively.

Whilst Zend_Db_Table is interesting in its own right, its usefulness comes out when we add business logic to it. This is the point when we enter the realm of the Model within MVC. There are lots of things you can do and we'll start with overriding insert and update for our News model.

First of all, let's assume that our news database table has the following definition (in MySQL):

```
CREATE TABLE `news` (
  `id` INT NOT NULL AUTO_INCREMENT PRIMARY KEY ,
  `date_created` DATETIME NOT NULL ,
  `date_updated` DATETIME NULL ,
  `title` VARCHAR(100) NULL ,
  `body` MEDIUMTEXT NOT NULL
)
```

To make our News class, a model, the first business logic that we will implement is to automatically manage the date_created and date_updated fields when inserting and updating records as shown in listing 2.7.

These are “behind the scenes” details that the rest of the system doesn’t need worry about and so ideal for placing in the model.

Listing 2.7: Automatically maintaining date fields in a model

```
class News extends Zend_Db_Table
{
    protected $_name = 'news';

    public function insert($data)
    {
        if (empty($data['date_created'])) {
            $data['date_created'] = date('Y-m-d H:i:s');
        }
        return parent::insert($data);          #1
    }

    public function update($data)
    {
        if (empty($data['date_updated'])) {      |#2
            $data['date_updated'] = date('Y-m-d H:i:s'); |
        }
        return parent::update($data);
    }
}
```

(annotation) <#1 Zend_Db_Table’s insert() function continues to do the real work.>

(annotation) <#2 Only set the date field if it’s not already been seen by the caller.>

This code is self-explanatory. When inserting, if date_created hasn’t already been supplied by the caller, we fill in today’s date and then call Zend_Db_Table’s insert() function. For updating, the story is similar, except that we change the date_updated field instead.

We can also write our own functions for retrieving data according to the business logic required by the application. Let’s assume that for our web site, we want the five most recent news items that have been created within the last three months displayed on the home page. This could be done using \$news->fetchAll() in the home page controller, but it is better to move the logic down into the News model to maintain the correct layering of the application and so that it can be reused by other controllers if required:

```
public function fetchLatest($count = 5)
{
    $cutOff = date('Y-m-', strtotime('-3 months'))
    $where = "date_created > '$cutOff'";
    $order = "date_created DESC";
    return $this->fetchAll($where, $order, $count);
}
```

Again, very simple functionality, that becomes much more powerful when placed in the right layer of the M-V-C triumvirate.

2.7 Summary

We have now written a simple hello world application using the Zend Framework and have explored the way the Model-View-Controller design pattern is applied to our applications. You should now have a good idea about what the Zend Framework gives us for making our applications maintainable and easy to write.

One key ideal that the framework developers try to adhere to is known as the 80/20 rule. Each component is intended to solve 80% of the problem space it addresses, whilst providing flex points to enable those developers needing the other 20%. For example, the front controller system provides a very flexible router

covers nearly all requirements. However if you need a specialized router, then it is very easy to insert your own one into the rest of the front controller setup. Similarly, whilst only a PHP view is provided, `Zend_View_Interface` allows for adding other template engines such as Smarty or PHPTAL.

We will now move on to build an application that will utilize most of the components supplied with the framework in order to build a fully functioning community website.

Building a web site with the Zend Framework

Zend Framework in Action is all about using the Zend Framework in a real world context. Initially I planned to build a corporate extranet application to show off the features of the framework. Then I thought about it again and decided that a community site that might be slightly different from the run of the mill company websites that I develop for a living.

Whilst holidaying in Wales last year, my wife and I realized that we didn't actually know the kid-friendly places to go to whilst we were there. A community website where parents could let each other know about which places were pro-child would be very useful to the parents out there. Let's build one called *Places to take the kids!*

Building such a community website, requires a lot of time and effort and it will be tempting to take the easy road towards intermingled PHP and HTML. We expect our website to be a key resource for many years and following good software engineering principles now will pay off many times over the lifetime of the project. The Zend Framework's MVC system will help us to do things the right way. There is even a side benefit of being quicker as we can take advantage of the simplicity and convention over configuration principles to ensure that our code is easy to write and refactor.

Before we dive right in and build the basic website, we will first focus on what we intend to build and the features it will need. We will then be able to set up an initial database and code the first pages.

3.1 Initial planning of a website

Obviously, we can't build a website without some sort of specification. Well, we could, but we are professionals! Rather than write a whole specification that will take up an entire chapter, we will use Agile web development techniques to design (and refactor) as we go along using user stories to sort out the next bit of our site. We are going to need to define our overarching goal, but we don't need to go into the detail about every part of the journey until we are on the road. After looking at what the site will achieve, we will then look at any issues within the user interface of the website which will lead us into the code.

3.1.1 The site's goals

For any website, there is only one question that needs to be answered from the user's point of view: *What does this site do for me?* If we can identify whom we want to be asking this question and what the answer is, then we have a website that will get visitors. There are some secondary questions to ask too, including working out how to fund the site, but these are relatively minor compared to ensuring that the site gets visitors.

One way to describe features of a website is to use "stories". These are paragraphs of prose that explain how a certain feature works from a very high level perspective. A user-story's main benefit over a "proper specification" is that it is written in a way that normal people (i.e. the client!) can understand. We can use the same mechanism to describe the entire website too and use prose to describe the goals of our website:

Places to take the kids is a site for parents to enable them to fully enjoy a day out or a longer holiday confident that the places they visit will be suitable for their children. They will also be able to plan their journey with relevant stops-overs for food and rest that are child-friendly. The site will grow a community of users who will be able to review and recommend places to visit and also communicate with each other using forums. The site will provide simple mechanisms for finding places to visit via browsing through categories or via searching and the user will be able to mark places as part of a planned trip and then print out the details of their trip in a printer-friendly format.

Our site story is only one paragraph long and conveys all we need to know to create a good website and we are now able to start planning. Clearly, we could write even more on what else the site could do, but that can be left for future phases of development. We will start with a list of the main functionality provided by the site and also a basic understanding of the site map.

Main functionality

Let's start by brainstorming a list of all the things that the website is going to need. A mind map is a good way to do this and my initial thoughts are shown in figure 3.1.

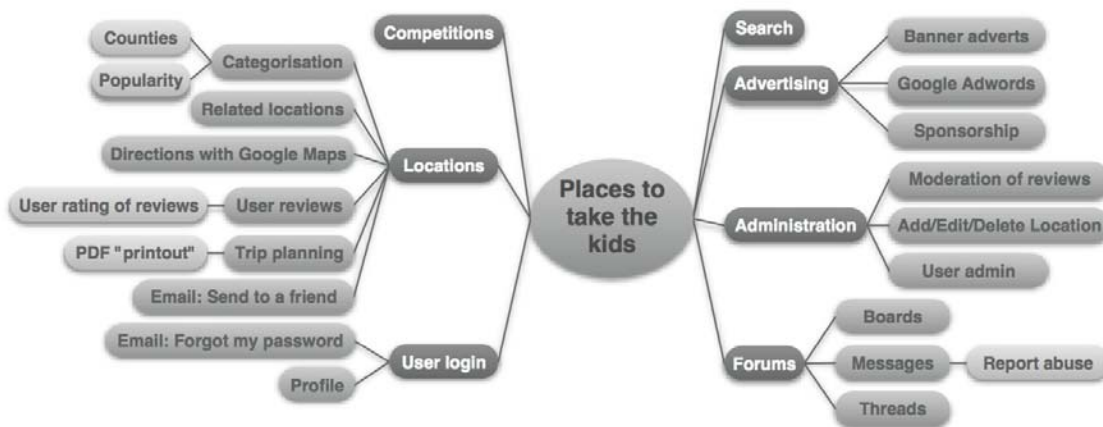


Figure 3.1: Mind maps are a good way to brainstorm features for a new website. For this site, we have found seven main areas of the website that will be required to meet the main goals with the “Locations” section fleshed out the most.

Another nice thing about mind maps is that they can be easily added to on an ad-hoc basis. In this case, the Locations section is key to the site and so has had the most thought about it. I’ve not thought too much about competitions though as they count as a “nice to have” idea rather than a core requirement to meet the site’s goals.

Testing

We will write tests for anything we are unsure about as we go along. This will give us the confidence in our code and allow us to refactor it as we improve the design. Improving the design shouldn’t be hard as we have barely done any! Even if we did lots of up front design, it is a certainty that what we learn whilst building will be of immeasurable value; we want to incorporate what we learn as we go along. Whilst we won’t be discussing all the tests within these pages, the accompanying source code contains all the tests required for us to have confidence in the code.

As testing is a relatively boring exercise, testing for *Places* will be done mainly using unit tests as these can be automated. Testing that requires someone to follow a written procedure will not be done often enough to catch errors as a result of code changes. As the Zend Framework’s MVC system provides a response object,

we can use unit testing to check that the HTML output has the correct data in it, so we can test elements of the page rendering too.

3.1.2 Designing the User Interface

We also need to consider how our new website is going to work in terms of the user interface (UI). I'm a software engineer, not a creative designer, so it's probably best if I don't provide many thoughts on design! A good user interface is much more than just the appearance; you also have to consider how it operates for the user and ensure that they can get to the information they are looking for with the minimum of fuss.

When designing a user interface for the site, we need to think about the features we need, page layout, accessibility, images and menus and navigation. We will now look at the key issues to consider for these elements which form part of the design brief for creation of the actual look and feel of the website.

UI features

The key features we will be considering on *Places* are navigation via a menu and a search system. The navigation will use a drill-down approach with the main menu always visible and the current level menu also shown. This allows for a lot of flexibility whilst not creating a cluttered menu. We could also display a breadcrumb so that the user has an appreciation of where they are in the site and which route they took from the home page to get to the page they are currently viewing.

Page layout

As a community site, we want to make sure that there is plenty of room for content and also room for discretely placed advertisements that will help pay the bandwidth bills. The site's basic look and feel will be long lived as we want to build a brand and also because communities don't tend to like change, so we will need a design that will grow with us as we to improve the site with new features.

Accessibility

As a modern site, we will ensure that we are accessible to all. This means that our site will be standards compliant so that it works in all modern browsers and also that we have given consideration for those users with less than perfect eyes and mouse coordination. The WAI accessibility standard will therefore be considered at all times when building the front end of the website.

Images

They say that an image tells a thousand words, but it also costs a lot of time and bandwidth compared to those words! We will use images to improve how the site looks and to add value for our users. Obviously we will provide images of the locations themselves and images also work well for feature lists and other visual clues to different sections of the pages.

Menus and navigation

For the menus, we will have a main menu across the top of the site and any sub menus in a vertical column as required. This means that we will have a site that is easy to navigate and also has plenty of expansion room as we build up features.

Putting all this together into a design, we have got a site that looks like Figure 3.2.

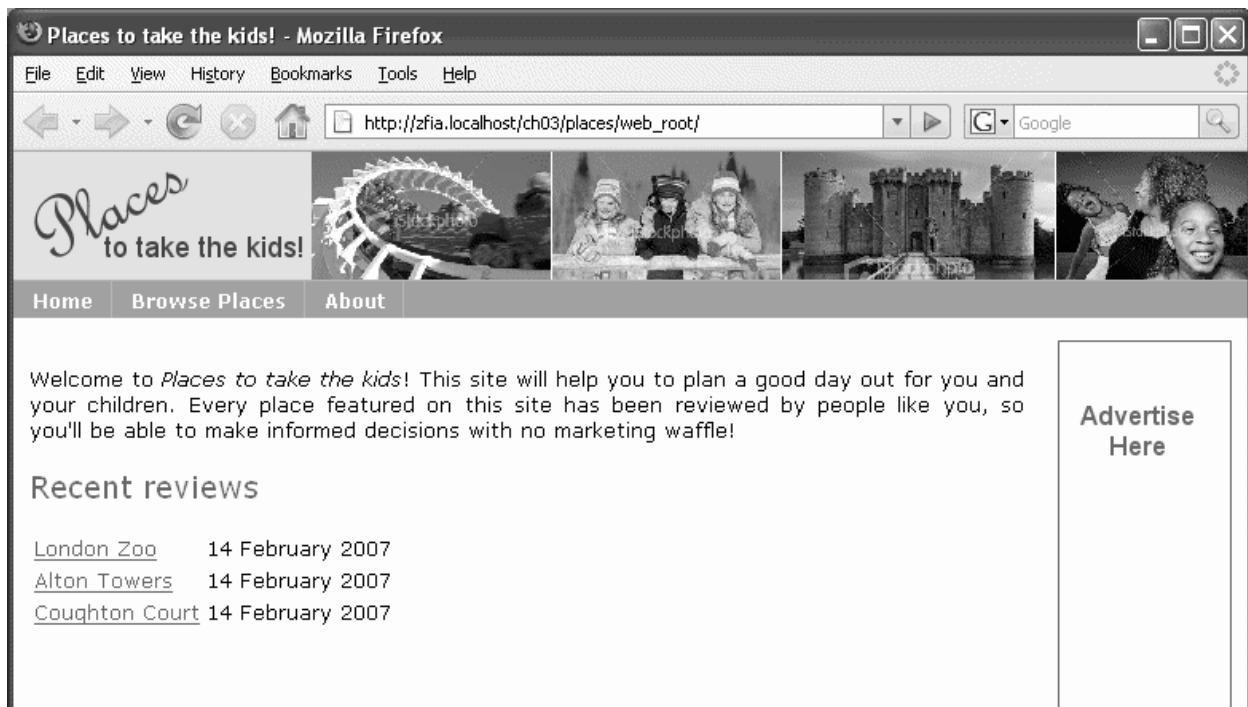


Figure 3.2: The *Places to take the kids* home page maximizes the amount of space for content whilst being easy to use.

There are four sections to this design: header (including menu), main content on left, advert on right and a footer. We are now ready to look at the code required to build the site, starting with the initial set up of the directory structure, bootstrap and loading of configuration information. We will also look at how we build the view templates to ensure that we do not repeat the header and footer code within every action template.

3.1.3 Planning the code

We have looked at the goals of the site and how the user interface will work, so we can now look at how we will organize the PHP code. As with the UI, we need to ensure that our code is not “hemmed in” as the site’s functionality and feature lists grow. We also want the system to do as much of the “plumbing” automatically for us, so for instance, we don’t have to worry about finding the correct class in the file structure, choosing the name of a view template or struggle to relate to database tables. We will also be using Ajax when appropriate, so whilst linking the view to the controllers is important for simplicity, we want to be able to override it.

The Zend Framework is a good choice to meet these requirements. As we discussed in chapter one, the key benefits of the Zend Framework mean that we are basing our website on a flexible, robust and supported platform that will be around as long as our site is.

The Zend Framework MVC system means that we will organize our site into separate controllers, each with a set of view templates. We will also access our data via models that will provide us with a “problem-domain” interface to the database tables that we use. Another feature of the MVC system is modules which allow us to group a set of related controllers, views and models together. We will use this functionality to separate out different logical concerns of the *Places* website.

With the initial planning in place, we can now start creating some code and tests – just as soon as we know where to put it!

3.2 Initial Coding

We can kick off the initial coding to get a skeleton structure from which we can build the features that are required. This means that we will set up the directories that we will need, write the bootstrap file, consider configuration issues and create the database.

3.2.1 The initial directory structure

From our work in chapter two, we already know what directories we need for a Zend Framework application, so we can start from there. *Places* is a bit bigger than Hello World though, so we will need some additional directories to keep the files manageable as shown in Figure 3.3.

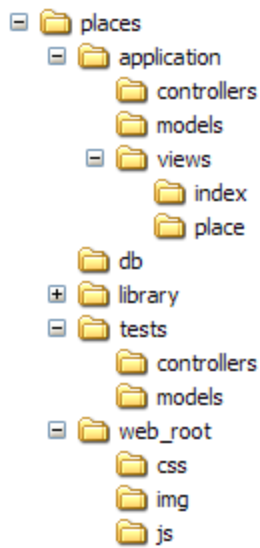


Figure 3.3: Directory structure of the *Places* website.

As with Hello World, we organize our directory structure with the goal of ensuring we can find files again. This means that we separate out all the functionality both logically using modules and also by concern into separate controllers, views and models directories. For security, the `web_root` directory is the only directory that the web server can directly serve files from and so it only contains one file: the bootstrap file which is the next thing we will look at.

3.2.2 Bootstrap

The `web_root/index.php` file from our Hello World application is a good starting point for the bootstrap files required for *Places*. We will however augment it as there are additional things we need to do to make our application easier to develop. These are loading configuration data, initializing the database and automatically loading classes as we use them.

3.2.3 Automatically loading classes

One aspect of the `index.php` in Hello World that is less than ideal is that there is a lot of `Zend_Loader::loadClass()` calls to load up the classes we need before we use them. In a “real” application, there are even more classes in use and so there is going to be a lot of clutter throughout our application just to ensure that we have the right classes included at the right time.

For *Places*, we are going to use PHP's `__autoload()` functionality so that PHP will automatically load our classes for us. PHP5 introduced the `__autoload()` magic function that is called whenever you try to instantiate a class that hasn't yet been defined. The `Zend_Loader` class has a special function `registerAutoload()` specifically for use with `__autoload()`. This method will automatically use PHP5's Standard PHP Library (SPL) function `spl_autoload_register()` so that multiple autoloaders can be used. After `Zend_Loader::registerAutoload()` has been called, whenever a class is created which has not yet been defined, file containing the class is included. This solves the problem of `Zend_Loader::loadClass()` clutter nicely.

3.2.4 Configuration with Zend_Config

There is an implicit requirement that *Places* will store its data to a database, such as MySQL. We therefore need to store the database connection settings and we will use `Zend_Config` for this purpose. `Zend_Config` provides for three types of configuration file format: XML, INI and PHP Arrays. We will use the INI format as it is easy to maintain. `Zend_Config` provides an object oriented interface to the configuration data regardless of which file format has been loaded. Consider an INI file called `config.ini` containing the following:

```
[db]
adapter = PDO_MYSQL
database.host = localhost
database.username = user1234
database.password = 1234
database.name = db_one
```

This file is loaded and data can then be accessed like this:

```
$config = new Zend_Config_Ini('config.ini', 'db');
$adapter = $config->adapter;
$databaseHost = $config->database->host;
```

Note how the “.” within the key name of a setting is automatically turned into a hierarchical separator by `Zend_Config`. This allows us to group our config data easily within the confines of the INI file format. Another extension to the INI format that `Zend_Config` supports is section inheritance using the colon character (“:”) as a separator within the section name. This allows for defining a base set of configuration settings and then changing them based on which section is the master section as shown in Listing 3.1.

Listing 3.1 Example Zend_Config INI file.

```
[general]
db.adapter = PDO_MYSQL                                #1
db.config.host = localhost
db.config.username = user1234
db.config.password = 1234
db.config.dbname = db_one

[live : general]                                       #2
db.config.host = livedb.example.com

[dev : general]
db.config.host = devdb.example.com
```

(annotation) <#1 The “.” in `db.adapter` means that `adapter` is a child of `db`.>

(annotation) <#2 The `live` section inherits from the `general` section.>

We can now load this ini file in two ways. On the live site, the INI file is loaded with the command:

```
$config = new Zend_Config_Ini('config.ini', 'live');
```

which loads the live configuration. To load the dev configuration for use on a developer's local workstation, the load statement would be:

```
$config = new Zend_Config_Ini('config.ini', 'dev');
```


In both cases, Zend_Config will load the “general” section first and then apply the settings in the specific “live” or “dev” section,. This will have the effect of selecting a different database depending on whether the application is running on the live or development server.

The Places configuration file

As we will have only one configuration file, we will call it config.ini as shown in listing 3.2 and place it in the application directory.

Listing 3.2: The initial config.ini for Places contains just database connection information.

```
[general]
db.adapter = PDO_MYSQL
db.config.host = localhost
db.config.username = zfia
db.config.password = 123456
db.config.dbname = places                                #1

[live : general]

[dev : general]

[test : general]
db.config.dbname = places_test                            #2
(annotation) <#1 The default database to connect to is called “places”.>
(annotation) <#2 When running unit tests, we need to control what is in the database, so a different one is used.>
```

The only thing we need to set up at this point is the connection to the database. The test section is for use when using automatic testing of our application. Whilst we are testing, we don’t want to touch our main database, so we use a separate database that we can overwrite at will whilst testing different scenarios. We will load our INI file in the bootstrap and then register it to the Zend Registry class so that the config data can be retrieved wherever we need it:

```
// load configuration
$section = getenv('PLACES_CONFIG') ? getenv('PLACES_CONFIG') : 'live';
$config = new Zend_Config_Ini('./application/config.ini', $section);
Zend_Registry::set('config', $config);
```

In order to ensure that we load the correct section, we use an environment variable “PLACES_CONFIG”. This is set on a per server basis using the Apache configuration command `setenv PLACE_CONFIG {section_name}`. For my development server, I therefore have configured my Apache configuration with `setenv PLACE_CONFIG dev` and on the live site, we would use `setenv PLACE_CONFIG live`.

Zend_Registry

Using global variables is generally considered unwise for large applications as it introduces coupling between modules and it is very difficult to track down where and when a given global variable is modified during the processing of a script. The solution is to pass variables as parameters through functions to where the data is required. However, this causes a problem in that you can end up with lots of “pass-through” parameters which clutter up function signatures for functions that do not use the data, except to pass it on to the next function.

One solution to this is to consolidate the storage of such data into a single object which is easy to find; this is known as the registry. As its name implies, Zend_Registry implements the Registry design pattern and so is a handy place to store objects that are required in different parts of the application.

`Zend_Registry::set()` is used to register to objects with the registry. Internally, the registry is implemented as an associative array and so when you register an object with it, you have to supply the key name that you want to identify it with. To retrieve the object at a different place in the code, `Zend_Registry::get()` is used, where you supply the key name and a reference to the object is returned. There is also a helper function `Zend_Registry::isRegistered()` which enables you to check if a given object key is registered or now.

As a word of warning, the registry is very similar to using a global variable and so unwanted coupling between the registered objects and the rest of the code can occur if we are not careful. `Zend_Registry` should therefore be used with caution. We will use it with confidence for two object: `$config` and `$db`. These two objects are ideal for storing to a registry as they are generally only read from and not written to and so we can be confident that they are unlikely to change during the course of the request. Hence, even though the data is in the registry, we will still pass the relevant configuration data around our application when appropriate if it minimizes coupling or a given section easier to test.

3.2.5 Database Initialization

The configuration file contains all the information we need to initialize the database. As we discussed in Chapter two, we use the `Zend_Db` factory class to create a `Zend_Db_Adapter` specific to our database. In this case, we'll receive an object of type `Zend_Db_Adapter_Pdo_Mysql`. To create the object, we need to pass the adapter name and an array of configuration parameters. `Zend_Config`'s nested parameter system comes into its own here as we can use the `asArray()` function to retrieve a subsection's keys as an associative array ready for passing to `Zend_Db::factory()` as shown:

```
// set up database
$db = Zend_Db::factory($config->db->adapter, $config->db->config->asArray());
Zend_Db_Table::setDefaultAdapter($db);
Zend::register('db', $db);
```

Again, we register the database with the registry and with `Zend_Db_Table`. As our models will be derived from `Zend_Db_Table`, the registry will rarely be used.

3.3 Two-Step View

All but the smallest of websites have common display elements on all pages. Usually this includes the header, footer and the navigation elements, but could also include advertising banners and other elements as required by the site's design. Within the current design of the `ViewRenderer`, the view template associated with the current action is automatically rendered. One solution, therefore is to add two includes to every view template like this:

```
<?php include('header.phtml');?>
<h1>page title</h1>
<p>body copy here</p>
<?php include('footer.phtml');?>
```

The main problem is that we are repeating those two lines of code in every single view template. This clutters up the view template with code that is not relevant to the job in hand and as it happens all the time, is ripe for automation. The easiest way to implement a better solution is to use what is known as the Two-Step View design pattern. This is documented by Martin Fowler like this:

Two Step View deals with this problem by splitting the transformation into two stages. The first transforms the model data into a logical presentation without any specific formatting; the second converts that logical presentation with the actual formatting needed. This way you can make a global change by altering the

second stage, or you can support multiple output looks and feels with one second stage each. (From <http://www.martinfowler.com/eaCatalog/twoStepView.html>)

In terms of building a website with the Zend Framework, this means that the view script that is rendered automatically by the ViewRenderer action helper should only contain the HTML related to that action. The rest of the page is built up independently and the content of the controller action is placed into it. This is shown in Figure 3.4 where we have a master template (site.phtml) providing the overall layout and the action template containing the action specific content.



Figure 3.4: Two view templates make up the complete page.

The master template contains those parts of the page that do not change much, such as the header, footer and menu sections, and is site.phtml. The action template contains the display code specific to the action being dispatched. The current ViewRenderer ensures that all action template files related to one controller will be stored in a sub directory named after the controller. That is, if we have a controller called reviews, then the index action's view template is called index.phtml and resides in the reviews sub-directory. The master layout templates will reside directly in the views/ directory.

There are multiple ways to apply the Two-Step view to the MVC system within the Zend Framework and we are going to use a Front Controller plug in called SiteTemplate. Front Controller plug-ins allow us to inject functionality into the dispatching flow in order to alter the default behaviour. SiteTemplate will collect the view output from any actions that have been run and then place the content into a master template and render it. A plug-in for the front controller can implement a number of functions, known as hooks, that the front controller will call at different stages in the dispatch cycle. The SiteTemplate plug-in uses three hooks: dispatchLoopStartup(), preDispatch() and dispatchLoopShutdown().

Figure 3.5 shows a simplified view of the Zend Framework dispatch loop and where the SiteTemplate plugin's hook functions get called during code the process.

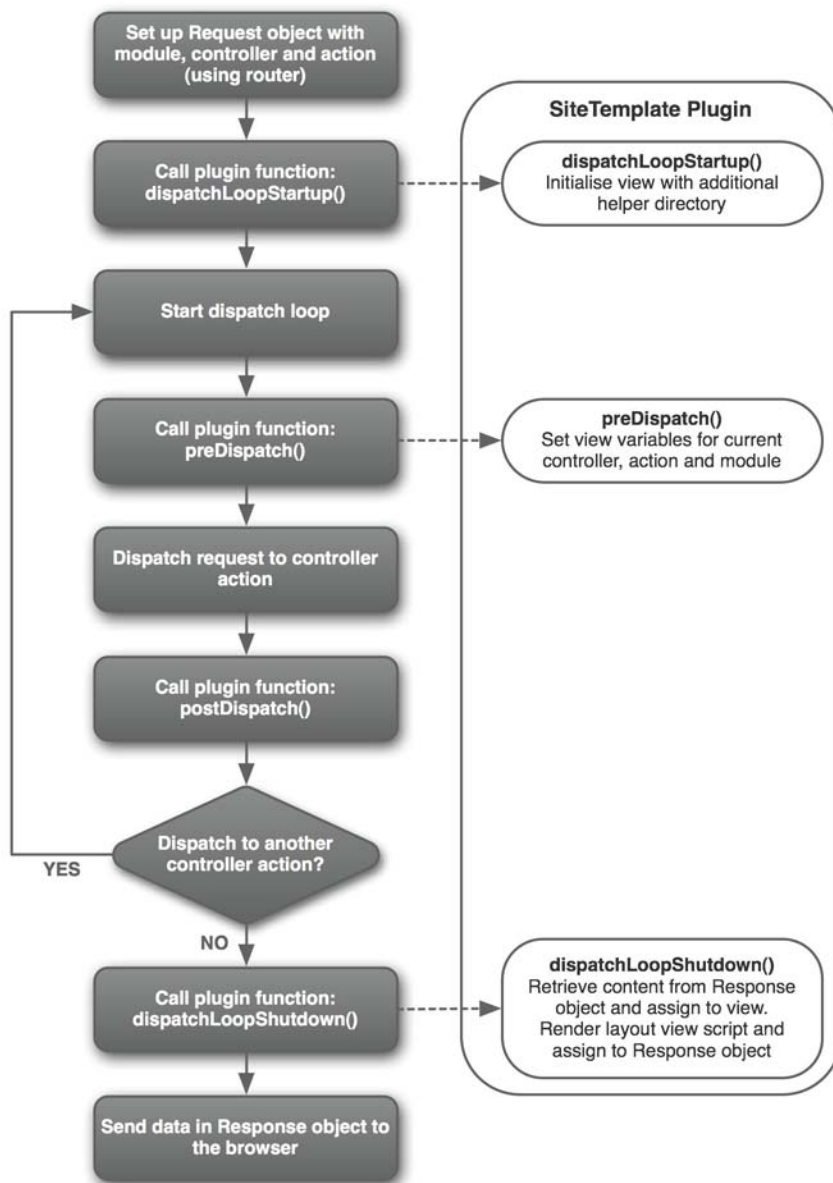


Figure 3.5: Front controller's dispatch loop

The `dispatchLoopStartup()` function is called after all routing is done, but before processing all the controller actions. It is therefore an ideal place to initialize the view. The `preDispatch()` function is called everytime a new controller action function is called. Usually there is only one controller action during a request, however, it is possible to add more using the `_forward()` function or other front controller plug-ins. As `preDispatch()` gets called every time, it is used to ensure that the view object is assigned the current names of the module, controller and action. Finally `dispatchLoopShutdown()` is called after everything is done. It is therefore the perfect time to render the master view template to complete the Two-Step view.

Let's look at the code. Listing 3.3 shows the SiteTemplate's `dispatchLoopStartup()` function.

Listing 3.3: SiteTemplate::dispatchLoopStartup()

```
public function dispatchLoopStartup()
```

```

        Zend_Controller_Request_Abstract $request)
{
    $viewRenderer = Zend_Controller_Action_HelperBroker:: \
        getExistingHelper('viewRenderer');
    if (is_null($viewRenderer->view)) {
        $viewRenderer->init();
    }
    $this->_view = $viewRenderer->view;

    // add View/Helper directory to path
    $prefix = 'Places_View_Helper';
    $dir = dirname(__FILE__) . '/../View/Helper';
    $this->_view->addHelperPath($dir, $prefix);
}

```

(annotation) <#1 Ensure view is created by the view renderer>

(annotation) <#2 Set prefix for *Places* specific view helpers>

(annotation) <#3 Set up path to helper directory>

The `dispatchLoopStartup()` function initializes the view and adds an additional view helper directory to it. By default the view created by the `ViewRenderer` sets up two directories for the view helpers: `Zend/View/Helpers` and the local module's `views/helpers`. Most applications have view helpers that are used across multiple modules and *Places* is no exception. In our case, we will store them in `lib/Places/View/Helper` which mimics the location of the global view helpers provided by the Zend Framework.

After the Front Controller has started this dispatch loop, it calls the `preDispatch()` function of all registered plugins. `SiteTemplate`'s `preDispatch()` function performs the per-action initialization and sets up some useful variables for use in our view templates:

```

public function preDispatch(Zend_Controller_Request_Abstract $request)
{
    // set up common variables for the view
    $this->_view->baseUrl = $request->getBaseUrl();
    $this->_view->module = $request->getModuleName();
    $this->_view->controller = $request->getControllerName();
    $this->_view->action = $request->getActionName();
}

```

As you can see, this is a very simple function that doesn't need further explanation. All it does is ensure that when we are within a view template, we have easy access to the information on the module, controller, action. We also pass through the `baseUrl` which is useful for referencing images, JavaScript and CSS files.

After dispatching has completed, the Front Controller calls the `dispatchLoopShutdown()` function for all registered plugins. This is the plugins' last chance to do anything before the response is sent back to the browser. For `SiteTemplate`, we use this hook to render the master layout with the content from the action views embedded with it. This is shown in Listing 3.4.

Listing 3.4: `SiteTemplate::dispatchLoopShutdown()`

```

public function dispatchLoopShutdown()
{
    if (!$this->_renderLayout) {
        $frontController = Zend_Controller_Front::getInstance();
        $request = $frontController->getRequest();
        $response = $frontController->getResponse();

        $body = $response->getBody();
        $this->_view->actionContent = $body;
    }
}

```

```

        $layoutScript = $this->getLayoutScript();           #3
        $content = $this->_view->render($layoutScript), 'default'; | #4
        $response->setBody($content);
    }
}

```

(annotation) <#1 Flag to turn off rendering of master>
 (annotation) <#2 The rendered action content >
 (annotation) <#3 Name of master template can be overridden>
 (annotation) <#4 Render master template>

dispatchLoopShutdown() does quite a lot for so few lines. Firstly we check that the user actually wants to render the master layout template (#1). There are multiple reasons why she might not, such as using Ajax or sending non html data such as a CSV file. The default is obviously to render the layout and SiteTemplate provides a helper function called setNoRenderLayout() to change this.

To render the layout, we retrieve the current body from the response and assign as a variable in the view (#2). This contains the content from the actions as rendered by the standard ViewRenderer action helper. Finding the master layout script name is delegated to a helper function, getLayoutScript() to allow for overriding the default site.phtml (#3) and then it is rendered and assigned to the response (#4).

To get a reference to the SiteTemplate plug-in, you can use the code:

```

$front = Zend_Controller_Front::getInstance();
$st = $front->getPlugin('SiteTemplate');
This allows access to the two main control functions:
$st->setNoRenderLayout(); and $st->setLayoutScript();

```

We have now implemented the Two-Step view using a Front Controller plug-in which has resulted in a fully automated solution. This means that within our action functions we do not have to think about rendering the master layout as it will all be handled for us. Let's look now at the home page of the *Places* website.

3.4 The Home Page

The home page is the shop front to our application and so we want to ensure that we provide an attractive page with easy to use navigation. We will write unit tests as we go along so that we can be confident that as we refactor to add new functionality, we do not break what we know to be working. The centerpiece of *Places* is reviews of places to take children, so we will start there.

There are four main sections to the home page. At the top, there is a header with logo and main navigation menu. The main content area is in the middle and a right hand column provides space for advertising. The site also contains a footer, which is not shown in Figure 3.6, to provide contact and copyright information on all pages.

3.4.1 The Initial models

As we know we need a list of reviews on the home page, let's start with a database table called reviews where each row represents a new review of a place. We can also assume that any given place will have more than one review, so we will need a list of locations in a table called places. The initial database schema is shown in Figure 3.5 and shows two tables, places and reviews.

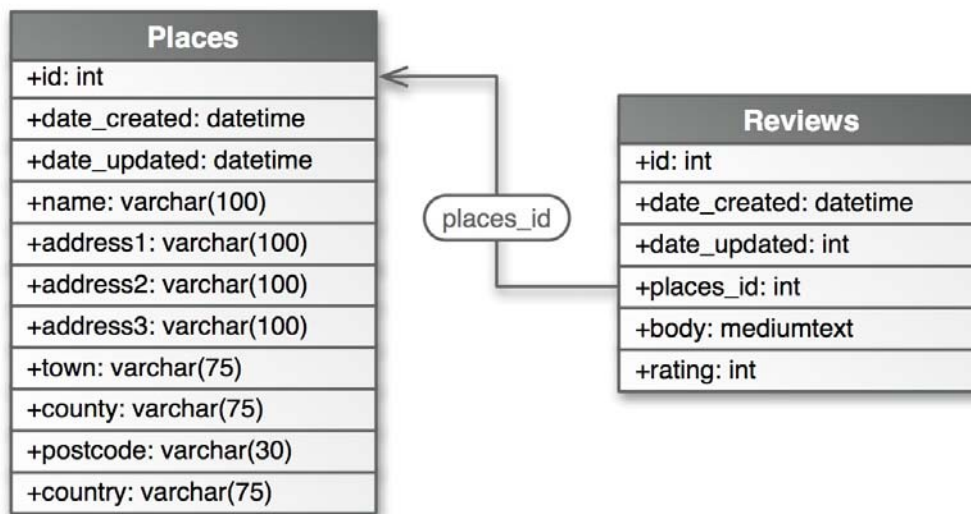


Figure 3.6 In order to build the initial pages, the database consists of two tables with a foreign key from Reviews to Places.

We can now go ahead and create our initial model classes Reviews and Places. These will be created in the application/models directory and initially are very simple:

```

class Reviews extends Zend_Db_Table
{
}

class Places extends Zend_Db_Table
{
}
  
```

As we have already discussed in Chapter 2, Zend_Db_Table gives us all the functionality of a Table Gateway without us having to write any code. As we have no intention of writing code that isn't going to be used, we will not expand our model classes until we need to. For now, we'll just populate the database directly with a few rows so we have something to display on the home page as shown in listing 3.5.

Listing 3.5 MySQL statements to create populate the tables with initial data

```

CREATE TABLE `places` (
  `id` int(11) NOT NULL auto_increment,          #1
  `date_created` datetime NOT NULL,              |#2
  `date_updated` datetime NOT NULL,              |#2
  `name` varchar(100) NOT NULL,
  `address1` varchar(100) default NULL,
  `address2` varchar(100) default NULL,
  `address3` varchar(100) default NULL,
  `town` varchar(75) default NULL,
  `county` varchar(75) default NULL,
  `postcode` varchar(30) default NULL,
  `country` varchar(75) default NULL,
  PRIMARY KEY (`id`)
);

INSERT INTO places (name, address1, town, county, postcode, date_created)
VALUES
('London Zoo', 'Regent\'s Park', 'London', '', 'NW1 4RY', '2007-02-14 00:00:00')
,('Alton Towers', 'Regent\'s Park', 'Alton', 'Staffordshire', 'ST10 4DB', '2007-02-14 00:00:00')
,('Coughton Court', '', 'Alcester', 'Warwickshire', 'B49 5JA', '2007-02-14 00:00:00');
  
```



```

CREATE TABLE `reviews` (
  `id` int(11) NOT NULL auto_increment,
  `date_created` datetime NOT NULL,
  `date_updated` datetime NOT NULL,
  `places_id` int(11) NOT NULL,          #3
  `user_name` varchar(50) NOT NULL,
  `body` mediumtext NOT NULL,
  `rating` int(11) default NULL,
  `helpful_yes` int(11) NOT NULL default '0',
  `helpful_total` int(11) NOT NULL default '0',
  PRIMARY KEY (`id`)
);

INSERT INTO reviews (places_id, body, rating, date_created)
VALUES
(1, 'The facilities here are really good. All the family enjoyed it', 4,
'2007-02-14 00:00:00')
, (1, 'Good day out, but not so many big animals now.', 2,
'2007-02-14 00:00:00')
, (1, 'Excellent food in the cafeteria. Even my 2 year old ate her lunch!', 4,
'2007-02-14 00:00:00');

```

(annotation) <#1 Primary key is called id and automatically increments. >

(annotation) <#2 Date metadata information is useful to store from the beginning. >

(annotation) <#3 Foreign key to another table is named "{table name}_id". >

The SQL in Listing 3.5. needs to be seeded into the database using a MySQL client, such as the command line mysql application or another one like phpMyAdmin.

3.4.2 Testing our Code

Testing code provides confidence. We are going to use unit tests to ensure that the code works the way we intend it to. Unit testing is the process of testing individual functionality separately from the rest of the code. The tests are designed to be run very regularly and so have to be fast as we want to run them whenever we make changes to the code. Whenever you run the tests you can be sure that your code functions correctly. This gives you a safety net to make changes to improve your program because if the tests still work, then the changes were good. If the tests no longer work, then the changes can be backed out.

The process of changing already working code to make it better and easier to understand is known as refactoring. This is the single most important skill required for maintaining an application over a long period of time. Refactoring without tests to check that the changes haven't broken anything is an extremely risky business and so doesn't get done. With tests though, refactoring is easy and possibly even fun; everything changes because of confidence.

As we are going to write tests for the majority of our code, we need a system to run them. PHPUnit (<http://www.phpunit.de>) is a testing framework for PHP that runs each test in isolation and provides functionality to allow us to organize the test code. Listing 3.6 shows the initial testing of our models, although at this stage we do not have much to test!

Listing 3.6: Places model unit test

```

class PlacesTest extends PHPUnit_Framework_TestCase
{
    public function setUp()                |#1
    {                                     |
        // reset database to known state |
        setupDatabase();                  |
    }                                     |

    public function testFetchAll()

```

```

{
    $placesFinder = new Places();
    $places = $placesFinder->fetchAll();

    $this->assertSame(3, $places->count()); #2
}
}

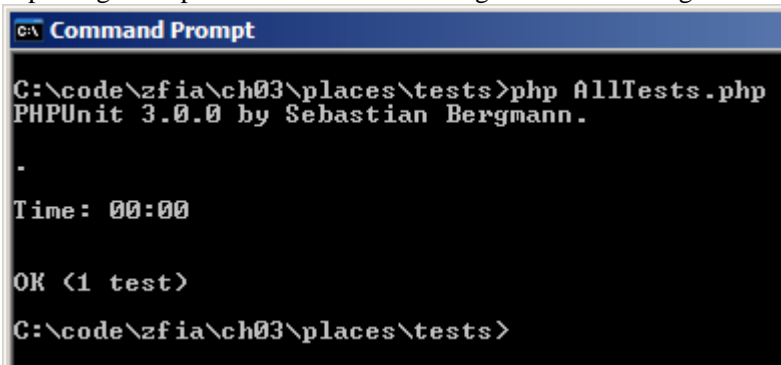
```

(annotation) <#1 setUp() is called before every test to ensure a clean slate.>

(annotation) <#2 Use assertEquals to check we have the correct row count.>

A PHPUnit test case consists of a number of separate functions that contain each test. These functions all start with the word test, so that PHPUnit can identify them. The member function setUp() is run before each test and so can be used to initialize the system for the test. This helps to ensure that each test is not dependant upon any other test and also does not cause problems for subsequent tests. The function tearDown() also exists, which is run after each test and so can be used for any cleanup required.

For testing our models, we want to ensure that the database is in a known state for every test. I have written a separate function, setUpDatabase(), that does this for us. It is not part of this class as it will be used for testing more than just the Places model. Our initial test, testFetchAll(), does a simple sanity check to ensure that everything is working. We collect all the places in our test database and count them. In this case we are expecting three places and that's what we get as shown in Figure 3.7.



```

C:\code\zfia\ch03\places\tests>php AllTests.php
PHPUnit 3.0.0 by Sebastian Bergmann.
.
Time: 00:00

OK <1 test>
C:\code\zfia\ch03\places\tests>

```

Figure 3.7: PHPUnit is run from the command line. This run is successful as all tests passed.

In order for our test case to work, and to support multiple test case classes, we will organize our tests into two suites, one for models and one for controllers. We will group these suites into an encompassing suite, so that we can run all the tests with the simple command `phpunit AllTests.php`. A certain amount of initialization is required, and so we will put this into a separate file called `TestConfiguration.php` as shown in Figure 3.8.

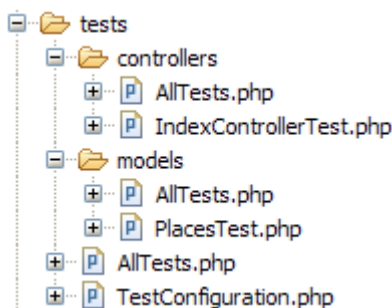


Figure 3.8: The unit tests are divided into controllers and models. The AllTests.php scripts allow each to be run independently if required.

TestConfiguration.php has two functions initialiseTests() and setupDatabase(). The initialiseTests() function sets up environment for the tests and registers the Zend_Db connection. It loads the config.ini file, section called 'test', so that we can specify a different database (places_test) to ensure that we don't accidentally affect our main database. The source code supplied with the book contains the test suite files, and as they are basically cloned from the instructions on the PHPUnit website, I will not go through them here.

3.4.3 The home page controller

The home page controller's action is the default action of the default controller and so the url /index/index will also display the home page. This maps to the indexAction() function within the class IndexController. In this function, all we need to do is collect the data from the Places model and assign to the view.

The Zend_Db_Table provides the fetchAll() function to retrieve multiple rows from the database, so we could just directly call it from the controller. I am, however, going to encapsulate the requirements for the home page's list within the Places class. This is so that we keep the database code where it belongs: in the model. We will add a new function fetchLatest() to the Places class which will automatically order the results by reverse date of update as shown in listing 3.7.

Listing 3.7: The Places model implements fetchLatest() to ensure the business logic is kept in the correct layer.

```
class Places extends Zend_Db_Table
{
    function fetchLatest($count = 10)
    {
        return $this->fetchAll(null,      |#2
            'date_created DESC', $count); |
    }
}
```

(annotation) <#1 The results are ordered by reverse date and are limited to \$count records.>

Zend_Db_Table's fetchAll() function returns a Zend_Db_Table_Rowset which can be iterated over using the foreach() construct in our view template. In listing 3.8, we assign it to the view within indexAction().

Listing 3.8 The IndexController's indexAction displays a list of the places that have most recently been added.

```
class IndexController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $this->view->title = 'Browse places';           #1
        $placesFinder = new Places();                  |#2
        $this->view->places = $places->fetchLatest();    |
    }
}
```

(annotation) <#1 Set the title for the web browser's title bar.>

(annotation) <#2 .Use the model to collect the recently created places and assign to the view.>

The IndexController's indexAction function is quite simple as all it does is assign a couple of member variables to the view. The model has encapsulated all the details of dealing with the database, leaving the controller to work at the "business logic" level. This is the MVC separation at work and so the controller only needs to worry about linking the model data to the view and then asking the view to render itself.

3.4.4 Using the view to display the home page

The first cut of the home page is very simple. We just need to display our list of places with reviews in date reversed order. In order to do this, we need to develop the master site template, an associated CSS file and the actual action content template itself.

3.4.5 Site.phtml: The master layout template

Let's start with the master template, site.phtml and include some basic styling. For *Places*, we are going to use a three column layout to allow for a left hand menu, main content and space for adverts on the right hand side. The master template, shown in Listing 3.9, is a simple XHTML file which uses CSS to style the layout of the site.

Listing 3.9: Site.phtml: the master view template file

```
<!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" lang="en" xml:lang="en">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title><?php echo $this->escape($this->title);?> #1
    - Places to Take the Kids</title>
<link rel="stylesheet" href="<?php echo $this->baseUrl; ?>/css/site.css"
    type="text/css" media="screen" />
</head>
<body>
    <a name="top" id="top"></a>
    <div id="scrn-read" title="links to aid navigation for screen readers">| #2
        [<a href="#center" accesskey="s">Skip to main content</a>] |
    </div>
<div id="header">Places to Take the Kids!</div>
<div id="container">
    <div id="center" class="column">
        <?php echo $this->render($this->actionTemplate); ?> #3
    <div id="right" class="column"></div>
</div>
<div id="footer-wrapper">
<div id="footer">Copyright &copy;2007 Rob Allen & Nick Lo</div>
</div>
</body>
</html>
```

(annotation) <#1 Title was assigned in the action.>

(annotation) <#2 Accessibility feature for screen readers for jumping to main content.>

(annotation) <#3 Render the action template from within the master template.>

One issue when using search engine friendly URLs (that is, without using ?a=b&c=d parameters) is that all references to other files or links within the site have to be fully qualified from the root of the URL. The Request object knows this information and you can access it using its `getBaseUrl()` function. As we need this in the view so that we can reference CSS files, we used the SiteTemplate plugin's `preDispatch()` function to create a new view variable, `baseUrl` which is just a direct assignment of the output of `getBaseUrl()`. This ensures that it is available in the view without having to remember to assign it in every controller action.

The CSS file is similarly simple and the first half controls the layout of the page with the styling of the elements following. It is available as part of the source code that accompanies this book. As it is served directly by the web server, we store it within in the `web_root` directory structure as this is the only area that the web server can access.

3.4.6 Index.phtml: The action view template

The unique content of the home page is contained within index/index.phtml. As we discussed earlier, we have automated the location of the action templates based on the controller and action. For our initial home page, we need to display a title along with the list of places that have recently been updated as shown in listing 3.10.

Listing 3.10: The home page action view: index.phtml

```
<h1><?php echo $this->escape($this->title);?></h1> #1

<?php if(count($this->places) : ?>
<table>
<?php foreach($this->places as $place) : ?>
<tr>
    <td><a href="<?php echo $this->baseUrl;?>/place/<?php
        echo $place->id; ?>"><?php
            echo $this->escape($place->name); ?></a></td> #2
    <td><?php echo $this->displayDate($place->dateUpdated); ?></td> #3
</tr>
<?php endforeach; ?>
</table>
<?php endif; ?>
```

(annotation) <#1 Reuse that title again!>

(annotation) <#2 escape() helper to help guard against XSS attacks.>

(annotation) <#3 DisplayDate() convenience helper .>

Note that we are very careful to escape any string data that we know should not include HTML. This is to ensure that we don't accidentally introduce an XSS vulnerability later down the line when a variable that is currently "known to be safe" ceases to be as the site's functionality changes; it is always better to code defensively at the start when you can.

The DisplayDate View Helper

The power of the view is shown when using view helpers to encapsulate view logic away from the main template files. This allows us to reuse common constructs and also helps to keep the main templates free from "cruft". When displaying the list of places that have been recently updated, we also display the on which date that the last update occurred. The field in the database is in ISO format (YYY-MM-DD hh:mm:ss) which is not very friendly to the user of the website. Whilst we could convert directly in the template, using a view helper, as shown in listing 3.11, ensures that we are consistent in our display of dates and makes it easier to change that display should the client want a different format.

Listing 3.11: View helper to ensure all dates are displayed consistently.

```
class ZFiA_View_Helper_DisplayDate
{
    function displayDate($time, $format='%d %B %Y') #1
    {
        $timestamp = strtotime($time); #2
        return strftime($format, $timestamp);
    }
}
```

(annotation) <#1 Allow for override of the default format for special cases.>

(annotation) <#2 use strtotime() to remove dependency on the format of the \$time variable.>

Although we use `strtotime()` to ensure that the `$time` variable can be in pretty much any format, be aware that it can get confused between UK and US short dates of the format `dd/mm/yyyy` and `mm/dd/yyyy`! We have now completed the first page of the website and *Places* is ready to have bells and whistles added to it.

3.5 Summary

This chapter has introduced a real application, *Places to take the kids*, which will be expanded upon throughout the rest of the book as we discuss all the components of the Zend Framework. In keeping with modern design practices, we haven't tried to design *Places* up-front, but have a story that describes what the site is about and from that we have an initial list of functional ideas. We have then concentrated on getting something going and ensuring that we have some tests in place for refactoring later. Testing is vitally important and we need the freedom to be able to change our code to suit each new piece of added functionality and be sure that it still works!

In order to get it all working, we have set up the directory structure for *Places* and have decided that our view templates should be organized into a master template containing an action content template. As each view template is named after its action, we have created a Front Controller plug-in, `SiteTemplate`, to implement the Two-Step View design pattern to allow for an easily maintainable common look and feel using a master layout view template.

We have also created two models that extend `Zend_Db_Table` to provide database access. The models implement the business logic required to retrieve reviews for a specific place. *Places* is a fully working website ready to add the foundations for enabling Ajax in our new Web 2.0 world.

Ajax

Ajax is a way of using JavaScript to create interactive web pages that send data to and from the server behind the scenes so that the user doesn't see the full page refresh. This means that a web site is more responsive and feels faster to a user. This allows for a web application, such as a web-based email client to behave more like its desktop cousins. As a result, Ajax is gaining in popularity. Although it is server based, the Zend Framework's MVC system that separates different layers of the application helps make it easier to add Ajax functionality to your websites.

In this chapter we will look at what Ajax is and how it used in web applications and examine all the components that make up a simple example in pure JavaScript and also using client libraries. We will also integrate Ajax into a Zend Framework application so that we can investigate how Ajax interacts with the MVC system. Let's consider exactly what Ajax is.

4.1 Introducing Ajax

Ajax enabled applications do not use full page refreshes to provide new information to the user. This can make them more user friendly as they are more responsive. Figure 4.1 shows Google Suggest (<http://labs.google.com/suggest>), which is an Ajax application that shows a dropdown list of sorted search terms as you type into the search box. Google's calendar (<http://calendar.google.com>) also shows good use of Ajax with the ability to drag and drop calendar events in order to change their date and time.

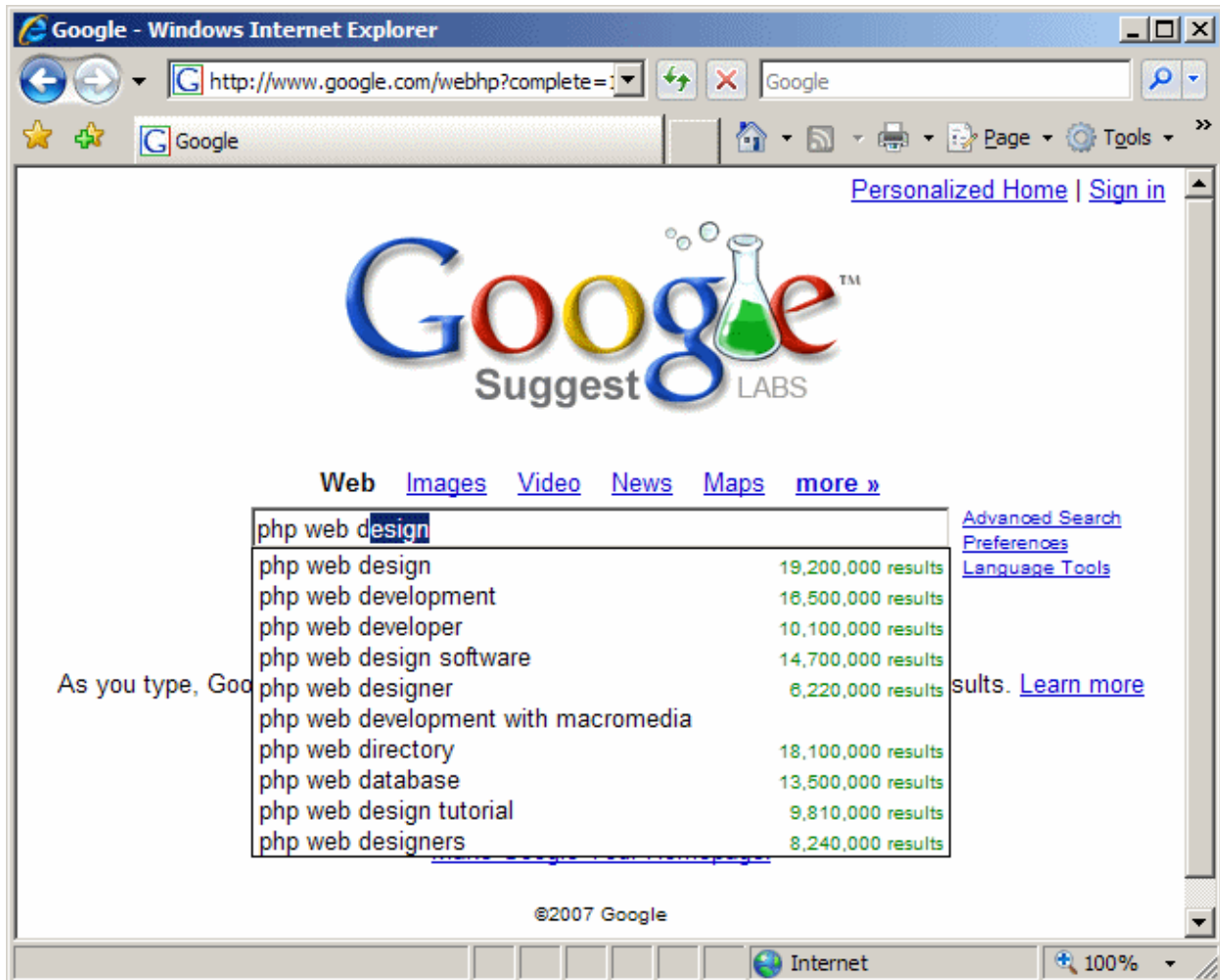


Figure 4.1: Google Suggest uses Ajax to provide contextual auto completion

As with any technology, there are advantages and disadvantages to using it. The main advantages are that the user's workflow is interrupted much less as the application feels more responsive, the user interface is more intuitive and less bandwidth is used as only the required data is sent from the server. The main disadvantages are that Ajax applications tend to break well-known browser features such as the back button and reliable URLs on the address bar for bookmarks or emailing to other people. There are also additional user interface issues for the web designer to be concerned about, such as providing notification of processing as the "spinning globe" no longer performs this task. Websites that use Ajax can also have problems with adhering to the WAI accessibility guidelines, so a fall back system is often required.

4.1.1 Defining Ajax

Ajax came into use in 2005 as a way of describing the suite of technologies used to build dynamic websites and the official expansion is *Asynchronous JavaScript and XML*. So, let's look at each component technology in turn.

Asynchronous

In order for an Ajax application to be an Ajax application, it needs to talk with the server without a page refresh. This is known as asynchronous data exchange and is generally performed using the web browser's

XMLHttpRequest object. A hidden iframe element can also be used. The XMLHttpRequest object is essentially a mini web browser built into the web browser that can be used by JavaScript without interrupting the user. Thus it is ideal for Ajax applications.

JavaScript

The key component in Ajax technologies is JavaScript and the browser's Document Object Model (DOM) to allow for manipulation of the webpage using scripts. JavaScript is a full blown programming language in its own right and has been implemented in mainstream browsers since 1995. The DOM is a standardized way of representing the elements of a web page as a tree of objects that can be manipulated by JavaScript. Within Ajax applications, this is the means by which the application dynamically changes the webpage to show new information without the server's sending new data.

XML

In order to transfer the new data from the server to the web browser within an asynchronous request, XML is used to format the data. In general, the language used to write the server application is different from JavaScript as used in the web browser, so a language neutral data transfer format is used. XML is a well known standard for solving this type of problem, but other formats, notably structured HTML, plain text and JSON (a JavaScript based data transfer format) are also used in Ajax applications.

Note that even if JSON rather than XML is used as the data transfer format, the application is still known as an Ajax application as it sounds better than Ajaj! The term Ajax has taken on meaning beyond its original definition and nowadays it represents a class of technologies that provide dynamic user interaction with the web server.

4.1.2 Using Ajax in enterprise applications

Ajax has many uses in all kinds of web applications. The main uses are:

- Form validation
- Form helpers: Auto-completion and drop down lists
- Dragging and dropping of items on the page
- Let's look at each one in more detail.

Form validation

JavaScript has been used for form validation for a long while. Usually the validation is a piece of ad-hoc code above the form that checks for the obvious errors, but the main validation is left to the server. As your form is not the only way to post data to a given webpage, it is vital that server side validation remains. However, the more validation that can be done before the user has to wait for the full page to refresh the better. There is nothing more frustrating than to click the submit button, wait 10 seconds and then find out that you didn't format your telephone number correctly!

Form Helpers: Autocompletion and drop down lists

Forms are generally complicated and anything that helps the user to complete one is welcome. Form field auto-completion is so useful that all the major web browsers offer this feature to remember what you have typed into a given field. Ajax applications take this idea one step further and enable the user to fill in text fields correctly all the time. Any form drop down list of options that has too many choices in it is a candidate for

replacing with a text field that is auto-complete enabled. Examples would include choosing your country when completing your address.

Form field auto-completion is also useful when the user should either supply a new value or use an existing one. This occurs when (for instance) assigning a task to a category within a project management application. Most of the time, the user would want to select an existing category (and not misspell it!), so the auto-complete dropdown will help her in this task. For those times when she needs to create a new category, then her workflow is not interrupted in the slightest as she can just type the new category straight into the field.

Dragging and Dropping

Dragging and dropping is very common within computer tasks on your desktop. For instance, in a file manager, you can select some files and drag them into another folder. Ajax enables this metaphor to work on the web as well. For instance, you could have a shopping basket in an e-commerce website that allows the user to drag items from the basket to a trash can to remove them from their order. One caveat here is that many web applications don't allow for drag and drop, so users are "trained" to recognize that it is possible on the web. You should therefore always allow for an alternative method of performing the action, or write very clear instructions!

Now that we have looked at what Ajax is and how it is used, we will write a simple application that allows us to investigate how to use JavaScript to send a request back to the server and deal with the response.

4.2 A Simple Ajax Example

We will use form validation and check that a given username is acceptable for use. To create the simplest of simple examples, we need three files for this example: an HTML page for the form, a JavaScript file to perform the XMLHttpRequest and a PHP file to do the server side validation. As the user types each character into the form field, a message appears underneath it showing any errors in their choice of name.

The application in action is shown in Figure 4.2.

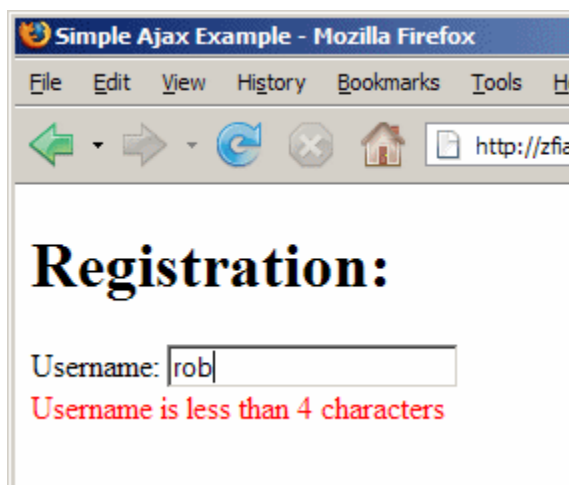


Figure 4.2: Simple Ajax example showing an error message whilst typing into the text field

The flow of information in an Ajax request is a little more complicated than a straight web page request. It all starts with an HTML element containing a JavaScript callback function, such as "onclick" which executes the JavaScript callback code. The JavaScript callback initiates a request to the web server using the XMLHttpRequest system and the server side code such as PHP performs the work. Once the PHP code has

completed, it formats the response in either XML or JSON and sends it back to the browser where it is processed by a JavaScript callback. Finally the JavaScript updates the HTML display to change the text, add new elements via the DOM or change CSS styles. This is shown in Figure 4.3.

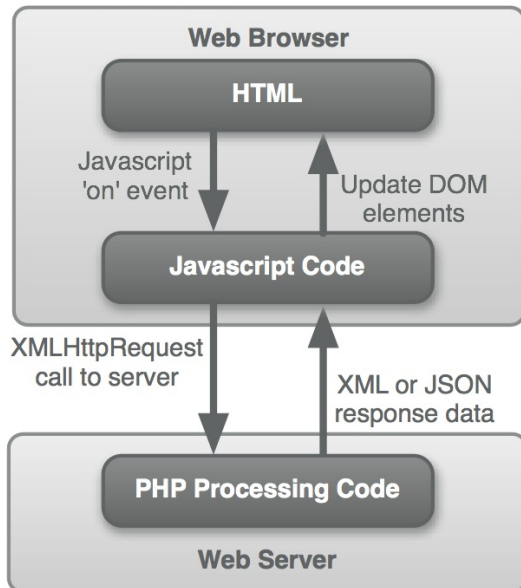


Figure 4.3: Application data flow in an Ajax request

For this example, let us start with the server-side PHP validation code, in listing 4.1, which will check that the submitted username is at least four characters long and doesn't clash with an existing username.

Listing 4.1: ajax.php: A simple server side validation routine in PHP

```

<?php
function checkUsername($username) {
    $existingUsers = array('rasmus', 'zeev', 'andi');
    // empty check
    if ($username == '') {                                     |#1
        return '';                                          |
    } elseif (strlen($username) < 4) {                       #2
        return '<span class="error">
            Username is less than 4 characters
        </span>';
    } elseif (in_array($username, $existingUsers)) {         #3
        return '<span class="error">
            Username already exists
        </span>';
    } else {
        return '<span class="ok">
            Username is acceptable
        </span>';
    }
}

if(!class_exists('PHPUnit_Framework_TestCase')) {           |#4
    $name = isset($_GET['name']) ? $_GET['name'] : '';      |
    echo checkUsername(trim($name));                        |
}

```

(annotation)<#1 No message if there is no username>

(annotation)<#2 Check that the username is long enough>

(annotation)<#3 Check that the username doesn't already exist>

(annotation)<#4 PHPUnit check to avoid echoing result when in test mode>

Note that we put the actual validation code into a function `checkUsername()` and only perform the actual check if we are not using PHPUnit. This makes it possible to test the validation logic easily and the test case is included in the code with this book. Although the list of existing users is an array in this example, it's more likely that the script would execute a database call to check the current list of users.

To access our validation script, we need a form where the user can type in the preferred username. The HTML is shown in listing 4.2

Listing 4.2: HTML file with simple form that requires validation

```
<!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" lang="en" xml:lang="en">
<head>
<title>Simple Ajax Example</title>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<style>
    .ok {color: green;}
    .error {color: red;}
</style>
<script type="text/javascript" src="ajax.js" />
</head>
<body>

<h1>Registration:</h1>

<form action="#">
    <div>
    <label for="name">Username:</label>
    <input id="name" name="name" type="text"
        onkeyup="checkUsername(this.value)" />
    <div id="message"></div>
    </div>
</form>
</body>
</html>
```

(annotation)<#1 Simple styling>

(annotation)<#2 onkeyup event is used so that `checkUsername()` is called on every keystroke>

(annotation)<#3 Placeholder for message returned by server>

This is a simple form that uses the `onkeyup` event on the input field to send the current text that the user has entered to the PHP file on the server. This is done in the JavaScript `check()` function which is stored in a separate JavaScript file called `ajax.js` as shown in Listing 4.3.

Listing 4.3: Hand crafted JavaScript code to call back to the server for validation

```
var request;
if (navigator.appName == "Microsoft Internet Explorer") {
    request = new ActiveXObject("Microsoft.XMLHTTP");
} else {
    request = new XMLHttpRequest();
}

function checkUsername(username)
{
    var success = 4;
    request.abort();
    request.open("GET", "ajax1.php?name=" + username, true);

    request.onreadystatechange=function() {
        if (request.readyState == success) {

```

```

        document.getElementById('message').innerHTML
            = request.responseText;
    }
}
request.send(null);
}

```

(annotation)<#1 IE uses an ActiveX control, everyone else uses a built in object. (IE was the first implementation though)>
(annotation)<#2 Send the request back to the server>
(annotation)<#3 Called by browser when the request object changes state>
(annotation)<#4 If the readyState is "success" (4) then update the place holder div with the returned text>

The magic of updating the HTML page with the new data is done by selecting an element on the page using `document.getElementById()` and then changing its attributes. In this case we change it entirely by replacing its `innerHTML` property with the server's response. We could equally have changed its CSS style or build new child HTML elements such as new `li` elements, based on the data received from the server.

You should now have an appreciation of what Ajax is and how the various components fit together. This example is necessarily simplistic as this book is not about Ajax per-se. To really understand what's going on and the full possibilities available with Ajax, I recommend that you take a look at *Ajax in Action* by Dave Crane and Eric Pascarello.

The JavaScript code I have presented here looks nice and simple and breaks pretty much every rule of defensive programming; there is no error checking at all! Other than the fact that it would clutter up the example, it is hard to get it right for all the browsers out there. We can abstract all the complications away into an client library though, and fortunately, other people have already done this for us, so we will examine them next.

4.3 Ajax Client Libraries

Adding the required JavaScript to create an Ajax application used to be quite painful. Even considering the very simple application above, the `checkUsername()` function mixes code dealing with the request with code that knows about how the HTML document is built. This is not good for long term maintenance, as we have seen in Chapter 2 where we separate concerns using the MVC design pattern for our main application. We should be separating the responsibilities of our client-side Ajax code too and rather than write all the required code ourselves, we can utilize an Ajax library to help us.

A JavaScript library will enable us to build upon the work of others. As PHP application developers, we are often more interested in solving our client's problems rather than worrying about the underlying technology. A JavaScript toolkit library will enable us to work at a higher level with our Ajax in much the same way that the Zend Framework empowers us in our server side application development.

There are many JavaScript toolkit libraries to choose from and it is not obvious how to make the choice. In much the same way as we chose to use the Zend Framework, the key considerations when choosing a JavaScript library are features, performance, documentation and community. We are going to look at two of the more popular libraries: Prototype and its cousin Script.aculo.us, and Yahoo's YUI.

4.3.1 Prototype and Script.aculo.us

The Prototype library provides a set of functionalities to make Ajax calls easy. It has an emphasis on extending the JavaScript DOM to make it easier to work with. Prototype also cross browser compatible. The script.aculo.us library is a higher level library built upon Prototype that concentrates on visual effects such as drag-and-drop and visual effects. Prototype and script.aculo.us development is tried quite closely to Ruby on Rails and you see some Ruby/Rails idioms in the prototype function names.

Let's look at how our simple example changes when using prototype. Other than including the prototype.js file in ajax.html, all the changes are in ajax.js, as shown in listing 4.4.

Listing 4.4: Replacing our JavaScript with prototype for our simple Ajax example

```
var handleSuccess = function(transport) {
    if(transport.responseText !== undefined){
        document.getElementById('message').innerHTML = transport.responseText;
    }
}

var handleFailure = function() {
    document.getElementById('message').innerHTML = "";
}

function check(name)
{
    var sUrl = "ajax.php?name=" + name;
    new Ajax.Request(sUrl,                                #1
        {
            method: 'get',                                | #2
            onSuccess: handleSuccess,                      |
            onFailure: handleFailure                      |
        }
    );
}
```

(annotation)<#1 Prototype supplies the Ajax object which takes two parameters: the url and a configuration array>

Although there is a little more code than with our simple example, this time we have some error checking and also have separated the functionality out. The prototype object that wraps XMLHttpRequest is called Ajax and is created in the check() function. The constructor (#1) takes two arguments: the URL to connect to and a configuration object. The configuration object is defined directly within the function call as an anonymous object using the {} syntax (#2). (In some ways JavaScript takes the concept of dynamic in dynamic languages to a whole new level!)

To configure our Ajax request, we tell prototype that we want to use a GET HTTP call and we register two callback functions for success and failure. The code in handleSuccess() is very simple. The function receives a transport object that encapsulates all that we know about the returned data, so all we have to do is extract the responseText and assign it to the placeholder element.

4.3.2 The YUI library from Yahoo!

The Yahoo! User Interface (YUI) also provides a set of user interface widgets along with some underlying classes to make the creation of Ajax applications easier. The attraction of the YUI is that a big company is supporting it and it has great documentation including examples. As with prototype, there is a good community using the library that has produced extensions that enhance the standard components.

Changing our example JavaScript so that it uses the YUI results in very similar code to the prototype example. We add the required files to the HTML (yahoo.js and connection.js) and then the rest of the work is in ajax.js (see listing 4.5).

Listing 4.5: ajax.js: Integrating YUI into our example Ajax application

```
var handleSuccess = function(o) {
    if(o.responseText !== undefined){
        document.getElementById('message').innerHTML = o.responseText;
    }
}
```



```

var handleFailure = function(o) {
    document.getElementById('message').innerHTML = "";
}

function check(name)
{
    var sUrl = "ajax.php?name=" + name;
    var callback =
    {
        success: handleSuccess,
        failure: handleFailure
    };
    var request = YAHOO.util.Connect.asyncRequest('GET', sUrl, callback); #1
}

```

(annotation)<#1 With YUI we use the Connect object for Ajax connections>

As you can see, the code is pretty much the same. In this case, the class that wraps up XMLHttpRequest is called YAHOO.util.Connect and we call the static method, asyncRequest to initiate the connection to the server. Again a configuration object is used to define which call back functions we want to use, though this time, I have created the object, callback, first and then assigned it to the function.

Using YUI or prototype makes development of Ajax applications much easier and less prone to error than hand-coding from scratch. Obviously there are a lot of other client libraries out there such as Mootools and dojo which are worth looking at before making a decision. For the remainder of this book, we will use the Yahoo! User Interface library for no other reason than I quite like it and Yahoo! hosts the JavaScript files for us, saving us a little bit of bandwidth. Now that we have a library to make development easier, we can now look at how Ajax fits with the Zend Framework.

4.4 Using Ajax with the Zend Framework

The first thing to consider when using Ajax with the Zend Framework is that the Zend Framework does not provide an Ajax components per-se, but the MVC system and other components such as Zend_Json make it easy to add Ajax features to your application. As throughout this book, maintenance of the application is a key consideration and the Zend Framework's separation of the model and controller from the view makes it easy to replace the standard HTML-based view of most pages with another view especially for Ajax requests.

Integration into the Zend Framework takes two forms: handling the Ajax request from the user within the controller and providing the JavaScript elements to handle the response from the server back to the client.

4.4.1 The server-side of an Ajax request

From the Zend Framework's point of view, an Ajax request looks remarkably like any other request into the application. The most significant difference is that the response back to the client is either a snippet of HTML, XML or JSON. To respond to an Ajax request is, therefore, just a case of ensuring that we use a different view and the easiest way to do this is to use a different controller. The advantage of the MVC design pattern comes into play now as the business logic is in the Model and so can be reused without any changes when adding Ajax to an application.

The controller

When providing controller actions that respond to Ajax calls, the view must not send back full HTML pages, but must instead send HTML fragments, JSON or XML data. If we take the simple example application, then the controller will require two actions; one for the displaying the page and one for responding to the Ajax request. Displaying the page, as shown in Listing 4.6, should now look familiar.

Listing 4.6: IndexController action to display the page

```
public function indexAction()
{
    $this->view->baseUrl = $this->_request->getBaseUrl();           |#1
}
```

<#1: store the base url in the view so that we can use it to load the required JavaScript files >

To respond to the Ajax call, we need a separate action, which we will call ajaxAction. We will reuse the same ajax.php file that we have been using for all examples as it has a perfectly good checkUsername() function to do the actual validation. In keeping with the MVC separation, we have put this file in the models directory and then include it for use in our action:

```
public function ajaxAction()
{
    include ('models/ajax.php');
    $name = trim($this->_request->getParam('name'));
    $this->_view->result = checkUsername($name);
}
```

The associated view file, views/scripts/ajax.phtml contains just one line of code to output the result:

```
<?php echo $this->result; ?>
```

This is probably the simplest view file you'll ever see!

4.4.2 The client site of an Ajax request

In the Zend Framework, the JavaScript part of an Ajax request is held within the view section. Usually this means that we use a template.

The view

As we have come to expect, the view (index.phtml) contains the HTML code for the page. This is the same as before except that we have to qualify where to find the JavaScript files using the baseUrl property that we created in indexAction:

```
<script type="text/javascript"
    src="<?php echo $this->baseUrl; ?>/js/yahoo.js"></script>
<script type="text/javascript"
    src="<?php echo $this->baseUrl; ?>/js/connection.js"></script>
<script type="text/javascript"
    src="<?php echo $this->baseUrl; ?>/js/ajax.js"></script>
```

In this case, I've chosen to use the YUI for the Ajax request, but Prototype would have worked just as well. The check() function also needs changing as we need a fully qualified URL for the request back to the server. We therefore pass the base URL property to the check() and rework the JavaScript sUrl variable to be in the Zend Framework format of controller/action/param_name/param_value as shown in Listing 4.7.

Listing 4.7: The URL in the Ajax request back to our server needs to be in the correct format

```
function check(baseUrl, name)
{
    var sUrl = baseUrl + "/index/ajax/name/" + name;
    var callback =
    {
        success: handleSuccess,
        failure: handleFailure,
    };
};
```

```
var request = YAHOO.util.Connect.asyncRequest('GET', sUrl, callback);
}
```

We now have a working Ajax example that uses the Zend Framework's MVC structure. The only changes made to the basic example application were to ensure that the code was correctly separated according to its role. We can now do something more interesting and look at how to fully integrate Ajax into a typical Zend Framework application, such as the *Places* example application..

4.5 Integrating into a Zend Framework Application

When considering the use of Ajax in a Zend Framework application, it's not just the client side JavaScript that needs to be written. We also have to ensure that the View doesn't try to display a full HTML page in response to an Ajax request and that our controller doesn't do work that our model should be doing. To look at these issues in context, we will write a feedback system into the *Places* example application.

Consider this use-case:

We are hoping that *Places* becomes very popular and so we will get lots of reviews on each location. We would like to make it easy for users to let us know whether they thought any given review was helpful for them. We can then display the "helpfulness" of each listed review to users which will guide them whilst reading the site. Also, as the number of reviews gets larger, we can display only the most helpful on the location's main page, and relegate the less helpful reviews to a secondary page.

This is an ideal task for Ajax as we do not want to intrude upon users with a page refresh and hence an in-place update will allow them to continue looking at the other reviews once they have indicated that a given review is helpful or not.

To provide feedback on a review we need to ask the question "was this review helpful to you?" with the possible answers of yes or no. We keep a count of the number of people who have answered yes and the total number of respondents. This will then allow us to inform the user that "N of M people found this review helpful". At a later stage, as we get more and more reviews, we could even consider ordering the list by "helpfulness" of the review. The user interface to our review feedback system is very simple as shown in Figure 4.4.

Reviews

John on 14 February 2007

The facilities here are really good. All the family enjoyed it

26 of 31 people found this review helpful. Was this review helpful to you?

Figure 4.4: Review feedback system has two buttons to provide feedback.

The review feedback system is a single line of text containing the number of people who found the review helpful and the number of people who have provided feedback. The user is left to determine if this ratio means that this review is "helpful" or not. We also have two buttons to allow the user to provide feedback. We will build this feature by starting with the View templates and add HTML and then the JavaScript. We also need two database fields to hold the two count numbers. I'm going to use `helpful_yes` for the first one and `helpful_total` for the second.

4.5.1 Adding HTML to the View Templates

The view template that displays the reviews about a place is `views/place/index.phtml`. We already have a list containing each review, so we just need to add the count information and the Yes and No buttons. We must also make sure that we can identify each HTML element that we want to change dynamically by giving it a unique id. The easiest way to get a unique id is to use a string followed by the review id. The entire code within the list is shown in listing 4.8.

Listing 4.8: The HTML for the review feedback system

```
<p class="helpful">
  <span id="yes-<?php echo $review->id?>">                                #1
    <?php echo $review->helpful_yes;?></span>
  of <span id="total-<?php echo $review->id?>">
    <?php echo $review->helpful_total; ?></span>
  people found this review helpful. Was this review helpful to you?
  <span id="yesno-<?php echo $review->id?>">
    <a class="reviewLink" href="#"
      onclick="new RF(1, <?php echo $review->id;?>,                | #2
        '<?php echo $this->baseUrl; ?>'); return false;">Yes</a>    |
    <a class="reviewLink" href="#"
      onclick="new RF(0, <?php echo $review->id;?>,
        '<?php echo $this->baseUrl; ?>'); return false;">No</a>
  </span>
  <span id="spinner-<?php echo $review->id?>"> </span>
  <div id="message-<?php echo $review->id?>"></div>
</p>
<#1: Set the id of each element that we want to address using the format {label}-{review_id} >
<#2: Create a new instance of a request feedback object, RF. This object will initiate the Ajax connection in its constructor >
```

As you can see in listing 4.8, the HTML for the review feedback system consists of a message informing the user of the helpfulness of the review, two links for providing feedback (Yes or No) and two place holders for providing interactive responses after clicking a link. To make this HTML code useful for an Ajax application we have to make each part easily accessible from Javascript. We do this by setting the id attribute to a unique value using the review's database id. Now that we have a the HTML, we can add some JavaScript to do the work.

4.5.2 Adding JavaScript to the View Templates

As the review feedback system uses Ajax, we will use the YUI framework for the behind the scenes connection to the server. We will place the required JavaScript file, `reviewFeedback.js`, in the "js" directory under `web_root`. We also need some YUI files, but Yahoo! will serve those for us.

We can break down the client side work into a number of distinct tasks:

- Initiate request to server and start a spinner animation
- On success:
 - Update the count
 - Stop the spinner animation
 - Say thank you to the user
- On failure:
 - Inform the user
 - Stop the spinner animation

In order to prevent pollution of the global namespace, we put all the JavaScript code for the review feedback module into a class called RF (for Review Feedback). The constructor initializes everything and connects to the server as shown in Listing 4.9.

Listing 4.9: The JavaScript class constructor

```
function RF(response, reviewId, baseUrl) {
    this.id = reviewId;
    this.baseUrl = baseUrl;

    // turn on spinner and empty the information message
    this.startSpinner();
    this.message("", "");

    // ensure that we don't have to encode parameters
    var response = parseInt(response);           | #1
    var reviewId = parseInt(reviewId);           |

    var sUrl = baseUrl + "/review/feedback/id/"
        + reviewId + "/helpful/" + response;

    // perform the request.
    YAHOO.util.Connect.asyncRequest('GET', sUrl, this); #2
}
```

<#1: Using parseInt() we can avoid having to encode the parameters.>

<#2: The constructor initiates the Ajax call back to the server.>

JavaScript's object model is a little different from PHPs and to create a class, we don't need a "class" keyword, we just need a function which will act as the constructor. The rest of the class's functions are then added to the "prototype" property of the function which will make them available to all instances of the class. Appendix B of *Ajax in Action* has more details if you need to catch up on your JavaScript object model.

As you can see in Listing 4.9, when we initiate the Ajax call with `asyncRequest`, we pass in the "this" as the callback object. This means that we must define the member functions `success()` and `failure()` so that the YUI system can call them when appropriate. The advantage of using an instance is that we can store the `reviewId` and the `baseUrl` into the instance and they will be available in the member functions when we need them. We can also provide helper functions within the class to keep the main call back functions easier to read.

We need to provide a message on success and on failure, so it makes sense to write a function called `message()`:

```
RF.prototype.message = function(class, text) {
    document.getElementById('message-'+this.id).className = class;
    document.getElementById('message-'+this.id).innerHTML = text;
}
```

The thank you message is displayed in the div with an id of `message-{review_id}` and the message function just hides the verbosity of setting the CSS class and the text of the message that we need to display. Two other helper functions, `startSpinner()` and `stopSpinner()` are needed also. To create a spinner, we use an animated GIF image which is added to the HTML in `startSpinner()` and removed in `stopSpinner()`:

```
RF.prototype.startSpinner = function() {
    var spinner = document.getElementById('spinner-'+this.id);
    var url = this.baseUrl + '/img/spinner.gif';
    spinner.innerHTML = '';
}

RF.prototype.stopSpinner = function() {
    document.getElementById('spinner-'+this.id).innerHTML = "";
}
```

We can now define the `success()` call back function that does the real work, as shown in listing 4.10.

Listing 4.10: The success call back function

```

RF.prototype.success = function(o) {
  if(o.responseText !== undefined) {
    var json = eval("(" + o.responseText + ")") ;
    if(json.result && json.id == this.id) {                                     #1
      var id = json.id;

      // update the information text to include the new counts
      document.getElementById('yes-'+ id).innerHTML = json.helpful_yes;
      document.getElementById('total-'+id).innerHTML = json.helpful_total;

      // say thank you and stop the spinner
      this.message("success", 'Thankyou for your feedback.');
```

(annotation) <#1: Sanity checks: Check that we got back the same review Id as we sent.>

```

      this.stopSpinner();

      // remove yes/no buttons as they aren't needed after feedback
      document.getElementById('yesno-'+ id).innerHTML = "";
    } else {
      this.failure(o);                                                         #2
    }
  }
}

```

(annotation) <#2: If the sanity checks fail, then treat this as any other failure.>

As we noted earlier, the success function has to do three different things: update the review count text, display a thank you message and stop the spinner. We also remove the Yes and No buttons as they are no longer needed. To update the count, we deliberately placed a `` with an id of `helpful-{review_id}` around the two numbers that we needed to update. We then set the `innerHTML` property of the two spans to update the new number of “yes” and “total” counts. To update the message and stop the spinner, we use the helper functions that we previously created, which ensures that this function is easier to maintain.

The `failure()` call back function is very similar, except that we only need to display a message and stop the spinner:

```

RF.prototype.failure = function(o) {
  // inform the user of failure and stop the spinner
  var text = "Sorry, our feedback system hasn't worked. Please try later.";
  this.message("failed", text);
  this.stopSpinner();
}

```

This is all the JavaScript code required for our review feedback system. The server side code is similarly simple.

4.5.3 The server code

The client side JavaScript calls into the feedback action of the Review controller. This is a standard action like any other in the system and so is a class function called `ReviewController::feedbackAction()` that is stored in `ReviewController.php`. The most significant difference between an action that responds to an Ajax request and an action that displays HTML directly is the view code; we do not need any HTML.

The output of the action is an array of JSON encoded data and so we do not need a view template file at all and we can assign the data directly to the response. We therefore turn off the view renderer action helper to ensure that it doesn't try and render a complete page. The feedback action function is shown in listing 4.11.

Listing 4.11: ReviewController::feedbackAction() contains the server side Ajax response.

```

class ReviewController extends Places_Controller_Action
{

```

```

public function feedbackAction()
{
    $id = (int)$this->_request->getParam('id');           #1
    if ($id == 0) {
        $return = Zend_Json::encode(array('result'=>false));
        $this->_response->appendBody($return);           #2
        return;
    }

    $helpful = (int)$this->_request->getParam('helpful');
    $helpful = $helpful == 0 ? 0 : 1; //ensure is only 0 or 1

    $reviewsFinder = new Reviews();
    $review = $reviewsFinder->fetchRow('id='.$id);
    if ($review->id != $id) {
        $return = Zend_Json::encode(array('result'=>false));
        $this->_response->appendBody($return);
        return;
    }

    if ($helpful) {
        $sql = "Update reviews SET helpful_yes = (helpful_yes+1),
                helpful_total = (helpful_total+1)
                WHERE id=$id";
    } else {
        $sql = "Update reviews SET helpful_total = (helpful_total+1)
                WHERE id=$id";
    }
    $reviewsFinder->getAdapter()->query($sql);

    $review = $reviewsFinder->fetchRow('id='.$id);

    $return = array('result'=>true, 'id'=>$id,
        'helpful_yes'=>$review->helpful_yes,
        'helpful_total'=>$review->helpful_total);
    $this->_response->appendBody(Zend_Json::encode($return));
    $this->_helper->viewRenderer->setNoRender();           #3
}
}

```

(annotation) <#1 Casting to an integer ensures that the id is “safe” to use in SQL statements.>

(annotation) <#2 The result key in the returned array is used for error reporting.>

(annotation) <#3 Turn off the view renderer.>

Encoding to JSON is taken care of by the Zend_Json component. There are two static functions, encode() and decode() available that will use the PHP JSON extension if available, otherwise will use a PHP implementation. Obviously, the PHP JSON extension is faster, but it is useful to know that the class will work even if it isn’t available.

The rest of the code in feedbackAction() is standard Zend_Framework code that updates the counts of the database fields helpful_yes and helpful_total within the reviews table for the particular view in question. In this case, we construct the SQL directly and execute it using the query() function of the Zend_Db object. Note that as we use the review id from the user within the SQL directly, the cast to an integer is vital to ensure that we don’t accidentally introduce an SQL injection vulnerability.

4.6 Summary

Within this chapter we have looked at what Ajax is and where it fits within a Zend_Framework application. While it is possible to hand craft all the Ajax code required, it is easier to use one of the many libraries

available to simplify development and make the code more robust. We had a look at both prototype and the Yahoo! YUI libraries and saw how easy they are to use. To place it all in context, we have also integrated a mechanism to allow users to feeding back the helpfulness of reviews in *Places*.

We have now covered all the basics of building a Zend Framework application and fitting out the front end with the latest technology; it is now time to look in detail at the database functionality provided with the Zend Framework and how `Zend_Db_Table` can help us build models that are powerful to use, yet easy to maintain.

Managing the Database

For most web sites, a database is a vital part of the application. The Zend Framework acknowledges this with a comprehensive set of database related components that provide varying levels of abstraction. There are two important levels of abstraction provided by the Zend Framework's database classes: database and table. The database abstraction provides for independence of your PHP code from the underlying database server used. This means that it is easier for your application to support multiple database servers. Table abstraction is a way to represent the database tables and rows as PHP objects. This allows for the rest of your application to interact with PHP and never need to know that there is an underlying database.

5.1 Database abstraction with Zend_Db_Adapter

The subject of database abstraction appears to be a religious one with many developers claiming that such layers are a performance drain and serve no useful purpose. This belief comes from the fact that to get the best from a database engine, you have to understand how the engine works and use its specific interpretation of SQL to get the best from it. On the other hand, other developers wouldn't dream of not using a database abstraction layer and argue that it allows them to migrate easily from one database to another and their applications to be distributed more widely.

As always in such debates, it all depends on what the task is. If you are developing an application that needs to extract the maximum performance out of the database, such as say Google, then coding for one database is clearly the way forward. If you are writing an application that you intend to sell to customers, then supporting multiple databases makes your product more desirable as your customer will want to standardize the database servers used wherever possible. There is no right answer. In my current job, we have sold to customers who will only use SQL Server and will not countenance the use of MySQL or PostgreSQL, so we have found database abstraction layers very useful for making sales.

The Zend Framework's interface to the database is via an abstraction layer. The standardized interface to the database that this gives ensures that the higher level components will work with any database. It also means that the support for a given database is encapsulated at a single point and so supporting new database engines is easier. We will now look at using the Zend Framework's Zend_Db_Adapter to connect to a database and then interact with the data within it.

5.1.1 Creating an Zend_Db_Adapter

The heart of the database abstraction components within the Zend Framework is Zend_Db_Adapter_Abstract. Each supported database engine has a specific adapter class that inherits from this class. For example, DB2's adapter is named Zend_Db_Adapter_Db2. The PDO library is also used to for some adapters, such as Zend_Db_Adapter_Pdo_Pgsql for PostgreSQL.

The Factory design pattern is used to create a database adapter using the Zend_Db::factory() function as shown in listing 5.1:

Listing 5.1: Creating a Zend_Db_Adapter instance using the factory function Zend_Db::factory()

```
$params = array ('host' => 'localhost', |#1
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=329>

```
'username' => 'rob',      |
'password' => 'password',  |
'dbname'   => 'places');  |
```

```
$db = Zend_Db::factory('PDO_MYSQL', $params); #2
```

(annotation) <#1 Database engine specific configuration parameters within an associative array.>

(annotation) <#2 The PDO_MYSQL constant specifies the database engine.>

As every adapter extends `Zend_Db_Adapter_Abstract`, only functions within the abstract class can be used if you want database independence. The list of supported databases is quite extensive and includes DB2, MySQL, Oracle, SQL Server and PostgreSQL. I would expect other databases to be supported over time too, especially PDO-based ones.

The database adapters use the lazy loading technique to avoid connecting to the database on creation of the object instance. This means that the even though the adapter object is created using `Zend_Db::factory()`, connection to the database doesn't actually happen until you do something with that object that requires a connection. Calling the `getConnection()` function will also create a connection if there isn't one already.

5.1.2 Querying the database

Running an SQL query directly against a database is very simple:

```
$date = $db->quote('1980-01-01')
$sql = 'SELECT * FROM users WHERE date_of_birth > ' . $date;
$result = $db->query($sql);
```

The `$result` variable contains a standard PHP `PDOStatement` object and so you can use the standard calls such as `fetch()` or `fetchAll()` and to retrieve the data. In this case I used a standard SQL string for specifying my query.

The `query()` function also supports parameter binding to save having to `quote()` all strings. This system allows us to put placeholders into the SQL statement where we want our variables to be and then the adapter (or underlying core PHP) will ensure that our query is valid SQL and doesn't contain a string that isn't quoted properly. The query above could therefore be written as:

```
$sql = 'SELECT * FROM users WHERE date_of_birth > ?';
$result = $db->query($sql, array('1980-01-01'));
```

As a rule of thumb, using parameter based queries is a good habit to get into as it removes the opportunity for accidentally forgetting to `quote()` some data from the user and thereby possibly creating an SQL injection vulnerability in your application. It also has the side benefit that for some databases it is quicker to use bound data parameters.

Not everyone is comfortable creating complex SQL queries though. The Zend Framework's `Zend_Db_Select` classes provides a PHP-based object-oriented interface to the database data.

Zend_Db_Select

`Zend_Db_Select` allows for using PHP to build database query statements in the comfortable language of PHP rather than SQL.

The `Zend_Db_Select` provides a number of advantages. The most important ones are:

- Automatic quoting of meta-data (table and field names)
- Object oriented interface can provide for easier maintenance
- Helps to promote database independent queries
- Quoting of values to help reduce SQL injection vulnerabilities

A simple query using Zend_Db_Select is shown in Listing 5.2:

Listing 5.2: Creating a Zend_Db_Adapter instance using the factory function Zend_Db::factory()

```
$select = new Zend_Db_Select($db);           #1
$select->from('users');
$select->where('date_of_birth > ?', '1980-01-01');
$result = $select->query();                  #2
```

(annotation) <#1 Attach the Zend_Db_Adapter to the select object>

(annotation) <#2 execute the query>

One very useful feature of Zend_Db_Select is that you can build the query in any order. This differs from standard SQL in that you must have each section of your SQL string in the correct place. For a complex query, the maintenance required is simplified using Zend_Db_Select.

5.1.3 Inserting, Updating and Deleting

In order to simplify inserting, updating and deleting database rows, Zend_Db_Adapter provides the functions insert(), update() and delete() respectively.

Inserting and updating use the same basic premise in that you provide an associative array of data and it does the rest. Listing 5.3 shows how to add a user to a table called 'users'.

Listing 5.3: Inserting a row using Zend_Db_Adapter

```
// insert a user
$data = array(                                | #1
    'date_created' => date('Y-m-d'),          |
    'date_updated' => date('Y-m-d'),          |
    'first_name' => 'Ben',                    |
    'surname' => 'Ramsey'                     |
);

$table = 'users';
$rows_affected = $db->insert($table, $data);
$last_insert_id = $db->lastInsertId();        #2
```

(annotation) <#1 Associative array of data.>

(annotation) <#2 The id of the row just inserted.>

Note that Zend_Db_Adapter's insert() method will automatically ensure that your data is quoted correctly as it creates an insert statement that uses parameter binding in the same manner as we discussed in section 5.2.2

Updating a record using the update() method works pretty much the same way except that you need to pass in a SQL condition to limit the update to the rows that you are interested in. Typically, you would know the id of the row you want to update, however update() is flexible enough to allow for updating multiple rows as shown in Listing 5.4.

Listing 5.4: Updating multiple rows using Zend_Db_Adapter

```
// update a user
$data = array(                                | #1
    'country' => 'United Kingdom'             |
);

$table = 'users';
$where = $db->quoteInto('country = ?', 'UK'); #2
$db->update($table, $data, $where);
```

(annotation) <#1 Fields to be updated only.>

(annotation) <#2 Quote strings in where clause.>

Again, update() automatically quotes the data in the \$data array, but does not automatically quote the data in the \$where clause, so we use quoteInto() to ensure that the data is safe for use with the database.

Deleting from the database works in exactly the same manner, only we just need the table and the where clause, so for deleting a particular user the code would be:

```
$table = 'users';  
$where = 'id = 1';  
$rows_affected = $db->delete($table, $where);
```

Obviously, we need to quote strings in the \$where clause again. In all cases insert(), update() and delete() return the number of rows affected by the operation. We can now look at how to handle database specific differences.

5.1.4 Handling database specific differences

All databases are not equal and this is especially true of their interpretation of SQL and the additional functions they provide to allow access to the more advanced features. To call the specific SQL functions of your database server, Zend_Db_Select class has a helper class called Zend_Db_Expr. Zend_Db_Expr is used for calling SQL functions or creating other expressions for use in SQL. Let's consider an example where we want to extract the names of our users in lower case. The code is in Listing 5.5.

Listing 5.5: Using functions in SQL statements

```
$select = $db->select();  
$columns = array(id, "CONCAT(first_name, ' ', last_name" as n); #1  
$select->from('users', $columns);  
$stmt = $db->query($select);  
$result = $stmt->fetchAll();
```

(annotation) <#1 CONCAT() is a MySQL specific function.>

The from() function realizes that a bracket has been used in the columns parameter and so converts it to a Zend_Db_Expr automatically for us. We can however always use Zend_Db_Expr ourselves by setting the columns statement explicitly:

```
$columns = array(id, "n"=> new  
Zend_Db_Expr("CONCAT(first_name, ' ', last_name)));
```

Now that we have considered how to abstract the differences between database engines away by using adapters created by the factory method of Zend_Db, we can now turn our attention to how we use a database within an application. Novice web programmers tend to put the database calls required directly where they need them which leads to a maintenance nightmare with SQL statements spread all over the application. We will now consider ways to consolidate our SQL and see how the Zend Framework's Zend_Db_Table component helps to improve the architecture of our applications.

5.2 Table Abstraction: Zend_Db_Table

When dealing with a database, it is useful to be able to abstract your thinking above the nitty-gritty of the actual SQL statements and consider the system at the domain level. The domain level is where you are thinking about the problem being solved in the language of the problem. The easiest way to do this is to create classes that know how to load and save themselves to the database. A class that represents one row in a

database table is known as the Row Gateway pattern (or Active Record pattern depending on what else the class does).

One area where Row Gateway doesn't work so well is when dealing with lists such as retrieving a list of products in an e-commerce application. This is because it works at the row-level and lists are generally dealt with at the table level. The Zend Framework provides the Zend_Db_Table component to support database manipulation at the table level and so we will look at what it provides for us and how table level support differs from what we have already seen with Zend_Db_Adapter.

5.2.1 What Is the Table Data Gateway Pattern?

Zend_Db_Table has three main components: Zend_Db_Table_Abstract, Zend_Db_Table_Rowset and Zend_Db_Table_Row. As indicated by its name, Zend_Db_Table_Abstract is an abstract class which has to be extended for each table that it acts as a gate way to. When selecting multiple records from the table, an instance of Zend_Db_Table_Rowset is returned to you which can then be iterated over to access each individual row. Each row is an instance of Zend_Db_Table_Row which is itself an implementation of the Row Gateway pattern. This is shown in Figure 5.1.

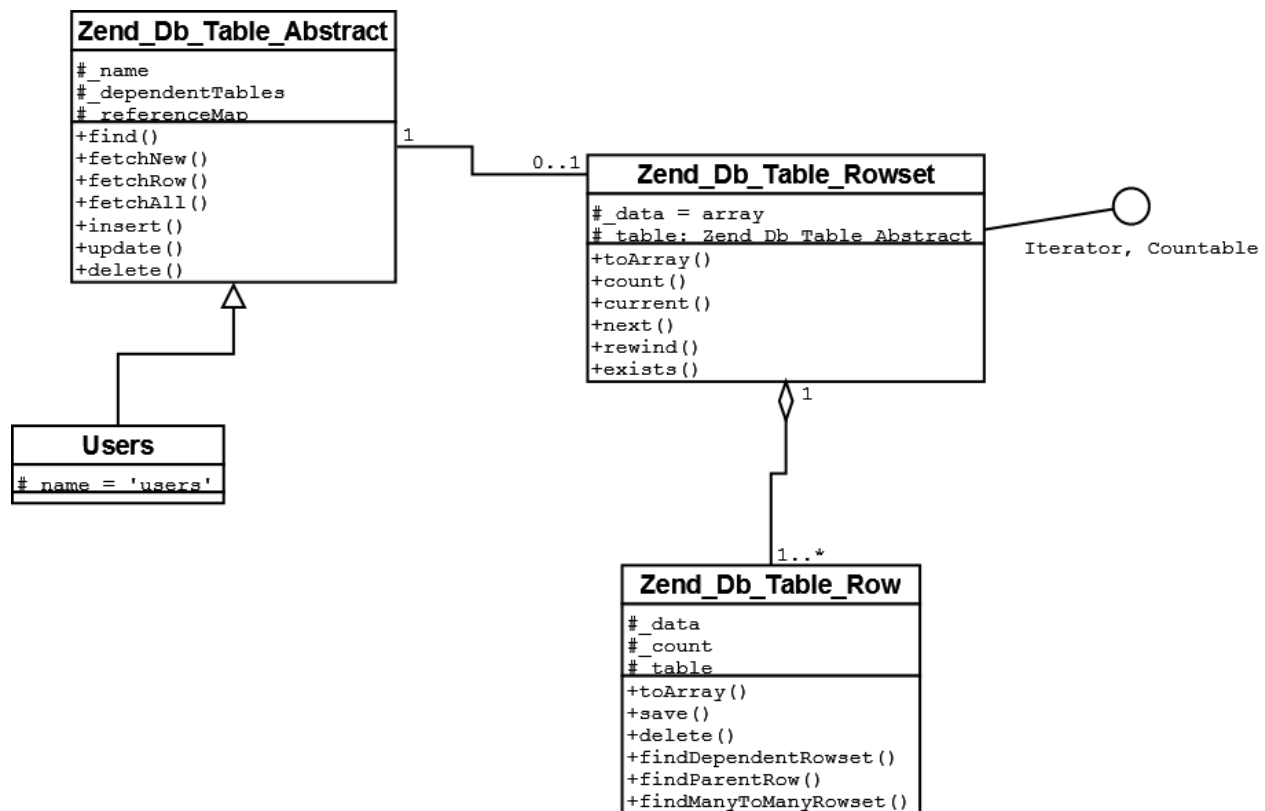


Figure 5.1 The Zend_Db_Table components fit together to produce a clean method of managing a database table and its associated rows.

Whilst, Zend_Db_Table_Abstract is always extended, Zend_Db_Table_Rowset is much less likely to be as it provides pretty much all you need for handling a set of database rows. In more complicated systems, extending Zend_Db_Table_Row into an Active Record by adding domain logic to it can help avoid duplicating of the same code in multiple places, as we'll see when we look at how to use the Zend_Db_Table.

5.2.2 Using Zend_Db_Table

In order to actually use the Zend_Db_Table components, you must create a class to represent your table as shown in Listing 5.6. This is usually named along the same lines as the database table that it accesses, but there is no requirement for it to do so.

Listing 5.6: Declaring a table class for use with Zend_Db_Table

```
class Users extends Zend_Db_Table_Abstract
{
    protected $_name = 'users';    #1
}
(annotation) <#1. The name of the database table is explicitly set>
```

As you can see, in listing 5.1, we have explicitly set the name of the underlying table to “users”. In reality, if the \$_name property is not set, the name of the class (preserving the case) will be used instead. I prefer to name my classes starting with an upper case letter and name my database tables entirely in lower case, hence it is wise for me to explicitly declare the database table name.

Now that we have a Users class, we can fetch data from the table in exactly the same way as if we were collecting data from an instance of Zend_Db_Adapter:

```
$users = new Users();
$rows = $users->fetchAll();
```

Whilst it performs the same functionality, the Zend_Db_Table_Abstract::fetchAll() is used completely differently from Zend_Db_Adapter_Abstract::fetchAll(). As we are working at a higher abstraction level with the table, fetchAll() also works at the higher level as shown by its signature:

```
public function fetchAll($where = null, $order = null,
    $count = null, $offset = null)
```

The function will handle creation of the SQL statement using Zend_Db_Select for us; we just need to specify the parts we are interested in and it will handle the rest. For example, to select all female users born from 1980:

```
$users = new Users();

$where = array('sex = ?' => 'F',
    'date_of_birth >= ?'=>'1980-01-01');
$rows = $users->fetchAll($where);
```

5.2.3 Inserting and updating with Zend_Db_Table

Inserting and updating with Zend_Db_Table is very similar to the usage with Zend_Db_Adapter except that this time we know the table already. Table 5.1 shows exactly how similar.

Table 5.1: Inserting with Zend_Db_Table and Zend_Db_Adapter compared

Zend_Db_Adapter	Zend_Db_Table
<pre>// insert a user \$table = 'users'; \$data = array('date_created' => date('Y-m-d'), 'date_updated' => date('Y-m-d'), 'first_name' => 'Ben', 'surname' => 'Ramsey'); \$db->insert(\$table, \$data); \$id = \$db->lastInsertId();</pre>	<pre>// insert a user \$users = new Users(); \$data = array('date_created' => date('Y-m-d'), 'date_updated' => date('Y-m-d'), 'first_name' => 'Ben', 'surname' => 'Ramsey'); \$id = \$users->insert(\$data); 1.1</pre>

--	--

As you see from Table 5.1, exactly the same process is used, except that with `Zend_Db_Table`, we already know the table's name and we get the id of the newly inserted record back directly. Updating a database row follows the same concept as shown in Table 5.2.

Table 5.2: Updating with `Zend_Db_Table` and `Zend_Db_Adapter` compared

Zend_Db_Adapter	Zend_Db_Table
<pre>// update a user \$table = 'users'; \$data = array('date_updated' => date('Y-m-d'), 'first_name' => 'Ben', 'surname' => 'Ramsey'); \$where = 'id=2'; \$db->update(\$table, \$data, \$where);</pre>	<pre>// update a user \$users = new Users(); \$data = array('date_updated' => date('Y-m-d'), 'first_name' => 'Ben', 'surname' => 'Ramsey'); \$where = 'id=2'; \$users->update(\$data, \$where); 1.2</pre>

Note that you need to quote any values and identifiers in the SQL expression used for the where clause. This is done using the database adapter's `quote()`, `quoteInto()` and `quoteIdentifier()` methods.

Similarities between inserting and updating

The similarities between inserting and updating lead to the conclusion that somehow it might be worth having a single “save” function in `Zend_Db_Table`. This would be incorrect as at the table level, you never know which record (or records!) need updating. At the `Zend_Db_Table_Row` level however, it makes perfect sense. Looking at `Zend_Db_Table_Row`, we discover that this functionality is already written for us as there is a function called `save()`. It is used as shown in Table 5.3.

Table 5.3: Saving with `Zend_Db_Table_Row`

Inserting a new record	Updating a record
<pre>\$users = new Users(); \$row = \$users->fetchNew(); \$row->first_name = 'Ben'; \$row->surname = 'Ramsey'; \$row->save();</pre>	<pre>\$users = new Users(); \$row = \$users->fetchRow('id=2'); \$row->first_name = 'Ben'; \$row->surname = 'Ramsey'; \$row->save(); 1.3</pre>

In this case, the entirety of the differences between inserting a new record and updating a current one are hidden from us. This is because the `Zend_Db_Table_Row` object works at a higher level abstraction.

5.2.4 Deleting records with `Zend_Db_Table`

As you would expect, deleting works exactly the same as inserting and updating only we use the `delete()` function. The `delete()` function is available in both `Zend_Db` and `Zend_Db_Table` and as with insert and update, the usage is very similar between the two classes as shown in Table 5.4.

Table 5.4: Deleting rows with Zend_Db_Table and Zend_Db_Adapter compared

Zend_Db_Adapter	Zend_Db_Table
<pre>// delete a user \$table = 'users'; \$where = 'id = 2'; \$db->delete(\$table, \$where);</pre>	<pre>// delete a user \$users = new Users(); \$where = 'id = 2'; \$users->delete(\$where); 1.4</pre>

The \$where clause in delete() for both Zend_Db_Table and Zend_Db_Adapter does not quote the values for us, so if we were to delete based on anything other than an integer, then we would have to use quote(), quoteInto(). For example, to delete all users who live in London:

```
$users = new Users();
$db = $users->getAdapter();
$where = $db->quoteInto('town = ?', 'London');
$users->delete($where);
```

We have now looked at all the major database operations that can be done using the Zend_Db_Adapter and Zend_Db_Table components. When fitting in database operations into a Model-View-Controller application, they usually fit as part of the Model. This is because the Model is where the business logic of your application is and the data stored in a database is very much related to the business of the application. We will now look using Zend_Db_Table based objects within a Model and, possibly more importantly, how to test it.

5.3 Using Zend_Db_Table as a Model

Zend_Db_Table is usually the point where we integrate the Zend_Db component into a Zend Framework MVC application. The model is usually a set of classes that build off of Zend_Db_Table_Abstract and Zend_Db_Table_Row_Abstract. Within the *Places* application we use a model to represent the registered users of the website. The database table required is quite simple and is shown in Figure 5.2.

Users
<pre>+id: int +date_created: datetime +date_updated: datetime +username: varchar(100) +password: varchar(40) +first_name: varchar(100) +last_name: varchar(100) +date_of_birth: date +sex: char(1) +postcode: varchar(20)</pre>

Figure 5.2. The Users table for Places contains login information along with demographic information for advertisers.

As we want to make money from Places, we expect that advertisers will want to know the demographics of the membership, so we would like our members to tell us their age, sex and location. Of course, this would be optional, whereas the username and password are mandatory.

We will need to access our database tables at two different levels: table and row. The table level is used when we need to display lists and the row level is used when dealing with individual records. In order to deal with this in the code, we need two classes, Users and User which are extensions of Zend_Db_Table_Abstract and Zend_Db_Table_Rowset respectively. The class definitions are shown in Listing 5.7. Note that in usually

you would use two files: User.php and Users.php for the two classes as it makes the application easier to maintain.

Listing 5.7: Creating a Model using Zend_Db_Table components

```
class Users extends Zend_Db_Table_Abstract
{
    protected $_name = 'users';
    protected $_rowClass = 'User';          #1
}

class User extends Zend_Db_Table_Row_Abstract
{
}
```

(annotation) <#1. Link the User class as the class to use when creating Rows for this table.>

Linking the User class to the Users class is done through the property `$_rowClass` in the Users table. This means that the `Zend_Db_Table_Abstract` class will create objects of type `User` whenever it would normally have created an object of type `Zend_Db_Table_Row`. The implementation in listing 5.6 is not very useful over and over the standard `Zend_Db_Table_Row` though as no additional functionality is provided. To make our `User` class useful, we are going to add a new property called “name”. This will be used throughout the application to display the user’s name. It will initially display the combination of first name and surname, but if they are not set, then it will use the user’s username.

Our initial implementation of the function `User::name()` is therefore:

```
Class User extends Zend_Db_Table_Row_Abstract
{
    public function name()
    {
        $name = trim($this->first_name . ' ' . $this->last_name);
        if (empty($name)) {
            $name = $this->username;
        }
        return $name;
    }
}
```

We can use `name()` directly, for example:

```
$rob = $users->find(1)->current();
echo $rob->name();
```

It would however be “nicer” to be able to treat the name as a read-only property of the record so that it is used in exactly the same way as, say `town`. This will increase the predictability of the class as the programmer will not have to think about whether a given property is a field in the database or a function within the class. The easiest and most efficient way to do this is to override `__get()` as shown in Listing 5.8.

Listing 5.8: Overriding `__get()` to provide custom properties

```
class User extends Zend_Db_Table_Row_Abstract
{
    //...
    function __get($key)
    {
        if(method_exists($this, $key)) #1
        {
            return $this->$key();        #2
        }
        return parent::__get($key);    #2
    }
    //...
}
```

(annotation) <#1. Check that there’s a class method names the same as `$key`.>

(annotation) <#2. If so, call the function and return.>

(annotation) <#3. If not, call through to the original implementation of __get().>

We have to call our parent's __get() function last as it will throw an exception if \$key does not exist in the database as a field. Note also that if one of the functions in the class is named the same as a field in the database, then the function will be called rather than the value of the field returned. This is handy if you want to do some processing on a given field before passing it through to the rest of the application, though such a use-case is rare.

5.3.1 Testing the Model

As we have mentioned before, it is good practice to test your code and it is even better practice to test it more than once! To test the users model, I have used PHPUnit and written some tests in the tests/models directory of the places application.

Setting up and tearing down

One key thing about running tests automatically is that each test must not interfere with another test. Also, a given test must not depend upon a previous test having run. To achieve this separation, PHPUnit provides the functions setUp() and tearDown() which are run just before (and after in the case of tearDown(), just after) each test.

In order to separate each database test, we use the setUp() function to recreate the database tables and populate them to a known state. This obviously means that we do not use the master database for testing! I have chosen to use a different database which is configured in the main config.ini for the application. We use the [test] section to specify our test database as shown in Listing 5.9.

Listing 5.9: The [test] section overrides [general] in application/config.ini.

```
[general] | #1
db.adapter = PDO_MYSQL |
db.config.host = localhost |
db.config.username = zfia |
db.config.password = 123456 |
db.config.dbname = places |

[test : general] #2
db.config.dbname = places_test #3
(annotation) <#1. [general] contains the main database connection information.>
(annotation) <#2. [test] overrides [general].>
(annotation) <#3. Override the database name when testing.>
```

In our case, we only need to change the database name from places to places_test; we can rely on the other settings in the general section to provide the other connection information. We are now in a position to create the unit test class skeleton as shown in Listing 5.10.

Listing 5.10: tests/models/UsersTests.php

```
<?php
require_once 'PHPUnit/Framework.php';

if(!defined('ROOT_DIR')) {
    define('ROOT_DIR', dirname(dirname(dirname(__FILE__)))); | #1

    set_include_path('.' |
        . PATH_SEPARATOR . ROOT_DIR . '/library' |
        . PATH_SEPARATOR . ROOT_DIR . '/../..../lib/zf/library' |
        . PATH_SEPARATOR . ROOT_DIR . '/application/models' |
        . PATH_SEPARATOR . get_include_path(); |
    }
}
```

```

require_once 'Zend/Registry.php';
require_once 'Zend/Db/Table.php';
require_once 'Zend/Config/Ini.php';
require_once 'Users.php';      // Users: Table gateway
require_once 'User.php';       // User: Row gateway

class UsersTest extends PHPUnit_Framework_TestCase                #2
{
    public function __construct($name = NULL)
    {
        parent::__construct($name);

        if(Zend_Registry::isRegistered('db')) {                  | #3
            $this->db = Zend_Registry::get('db');                |
        } else {
            // load configuration information from [test] in config.ini | #4
            $configFile = ROOT_DIR . '/application/config.ini';   |
            $config = new Zend_Config_Ini($configFile, 'test');   |
            Zend_Registry::set('config', $config);               |

            // set up database
            $dbConfig = $config->db->config->asArray();              | #5
            $db = Zend_Db::factory($config->db->adapter, $dbConfig); |
            Zend_Db_Table::setDefaultAdapter($db);               |
            Zend_Registry::set('db', $db);                       |
            $this->db = $db;                                       |
        }
    }
}

```

(annotation) <#1. Setup path.>

(annotation) <#2. Class name is the same as the filename. >

(annotation) <#3. Collect database adapter from registry if it has been stored already. >

(annotation) <#4. Load test specific configuration information. >

(annotation) <#5. Connect to database and store adapter for later use. >

This section of the unit test file contains only the initialization code that is run once. As you can see at the top (#1), we have to ensure that we have set up our paths correctly so that the files can be found and we also have to include all the files we are going to need. There is a check for the definition of ROOT_DIR to allow for expansion and placing this unit test file in to a suite of files that can be tested all at the same time; when we are testing as part of a suite, we will not want to alter the path. We also include the files from the Framework that we need for this test.

As the constructor is only called once, it is the ideal place to load the config.ini file and connect to the database. Again, if this class is part of a test suite, then the database connection will already have been made, in which case we just collect it from the Registry. Otherwise, once we have connected, we set the database adapter as the default adapter for Zend_Db_Table and store to the registry. We also assign it as a class member variable as it is going to be used in the setUp() function to set our database to a known state.

Initializing our database is just a case of using the database adapter to create the table and insert some rows. This is done in function called _setUpDatabase() which is called by setUp() as shown in Listing 5.11.

Listing 5.11: Initializing the database in setUp()

```

public function setUp()                                         #1
{
    // reset database to known state
    $this->_setUpDatabase();
}

protected function _setUpDatabase()
{
    $this->db->query('DROP TABLE IF EXISTS users;');           #2
}

```

```

$this->db->query(<<<EOT                                     |#3
    CREATE TABLE users (                                   |
        id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,      |
        date_created DATETIME,                             |
        date_updated DATETIME,                             |
        username VARCHAR(100) NOT NULL,                   |
        password VARCHAR(40) NOT NULL,                     |
        first_name VARCHAR(100),                           |
        last_name VARCHAR(100),                             |
        email VARCHAR(150) NOT NULL,                       |
        town VARCHAR(100),                                  |
        country VARCHAR(100),                              |
        date_of_birth datetime,                             |
        sex char(1),                                       |
        postcode VARCHAR(30)                               |
    )                                                       |
EOT                                                         |

    );

    // insert row #1                                         #4
    $row = array (
        'first_name' => 'Rob',
        'last_name'  => 'Allen',
        'username'   => 'rob',
        'password'    => 'rob',
        'email'       => 'rob@akrabat.com',
        'town'        => 'London',
        'country'     => 'UK',
        'date_of_birth' => '1970-01-04',
        'sex'         => 'M',
        'date_created' => '2007-02-14 00:00:00',
    );
    $this->db->insert('users', $row);

    // insert row #2
    $row = array (
        'first_name' => 'Julie',
        'last_name'  => 'Smith',
        'username'   => 'julie',
        'password'    => 'julie',
        'email'       => 'julie@example.com',
        'town'        => 'London',
        'country'     => 'UK',
        'date_of_birth' => '1981-03-15',
        'sex'         => 'F',
        'date_created' => '2007-02-14 00:00:00',
        'date_updated' => '2007-02-14 00:00:00'
    );
    $this->db->insert('users', $row);
}

```

(annotation) <#1. setUp() is called before every unit test is executed.>

(annotation) <#2. Drop the table if it exists.>

(annotation) <#3. Create the table.>

(annotation) <#4. Insert some known data.>

This is very simple code that just ensures that our database is set up correctly for every test. Note that there is a MySQL-ism in the Drop Table command which would need to be changed for other database engines. To insert the database rows, we take advantage of the database adapter's insert() method which we discussed earlier.

We are now in a position to actually write some tests to ensure that our model works as expected!

Testing the Users table gateway

The Users class overrides the insert() and update() functions to ensure that the date_created and date_updated fields are filled in without the rest of the code having to worry about it. The unit tests to ensure that this code works is shown in listing 5.12. I have separated the tests into two separate functions: one to test inserting and one to test updating.

Listing 5.12: Initializing the database in setUp()

```
public function testInsert()
{
    $users = new Users();
    $newUser = $users->fetchNew();                                     #1

    $newUser->first_name = 'Nick';                                   | #2
    $newUser->last_name = 'Lo';                                     |
    $newUser->username = 'nick';                                   |
    $newUser->password = 'nick';                                   |
    $newUser->email = 'nick@example.com';                           |

    $id = $newUser->save();                                           #3

    $nick = $users->find($id)->current();                             | #4
    $this->assertSame(3, $nick->id);                                   |

    // check that the date_created has been filled in                #5
    $this->assertNotNull($nick->date_created);

    // check that the date_updated has been filled in                #6
    $this->assertSame($nick->date_updated, $nick->date_created);

}

public function testUpdate()
{
    $users = new Users();
    $rob = $users->find(1)->current();                                | #7
    $rob->town = 'Worcester';                                         |
    $rob->save();                                                      |

    $rob2 = $users->find(1)->current();
    $this->assertTrue(($rob2->date_updated > $rob2->date_created));    #8
}
```

(annotation) <#1. Create a new empty row object.>

(annotation) <#2. Fill in some data.>

(annotation) <#3. Save the new object which will insert the data into the database. >

(annotation) <#4. Retrieve the newly inserted row and check it has the right id>

(annotation) <#5. Ensure the date_created field has been filled in>

(annotation) <#6. Ensure the date_update is the same as the date_created>

(annotation) <#7. Edit a row to create a change>

(annotation) <#8. Check that the date_updated is more recent than the date_created>

Within testInsert() there is a reasonable amount of set up code. (#5) and (#6) are the actual tests that we want to conduct, but to do them, we need to have inserted a new record into the database. As you can see, Zend_Db_Table's fetchNew() function returns an empty object, of type User that represents a row in the database. We can then set the fields that we need and then insert it into the database using the save() method. The row's save() method uses that table gateway's insert() or update() as appropriate which then uses our overridden functions to ensure that the date_created and date_updated fields are correctly filled in.

The testUpdate() function does the same basic test, but here we just edit one of the rows in the database that was created by the setUp() functions. It is important to get the row from the database again though to

ensure that the `date_created` field has been changed in the database itself. Now that we have explored the `Zend_Db_Table` class and its associated rowset and row classes, we can look at joining tables together.

5.3.2 Table Relationships with `Zend_Db_Table`

Table relationships are where `Zend_Db_Table` starts getting very interesting. This is the realm of linking tables together using SQL JOIN statements and is one area where it is nice to get the framework doing the leg work for us. We will look at how `Zend_Db_Table` can help us with one-to-many and many-to-many relationships.

One-to-Many relationships

We have already come across one to many relationships in Chapter 3 when we created the initial schema for the *Places* application. Each location can have many reviews. This is shown in figure 5.3.

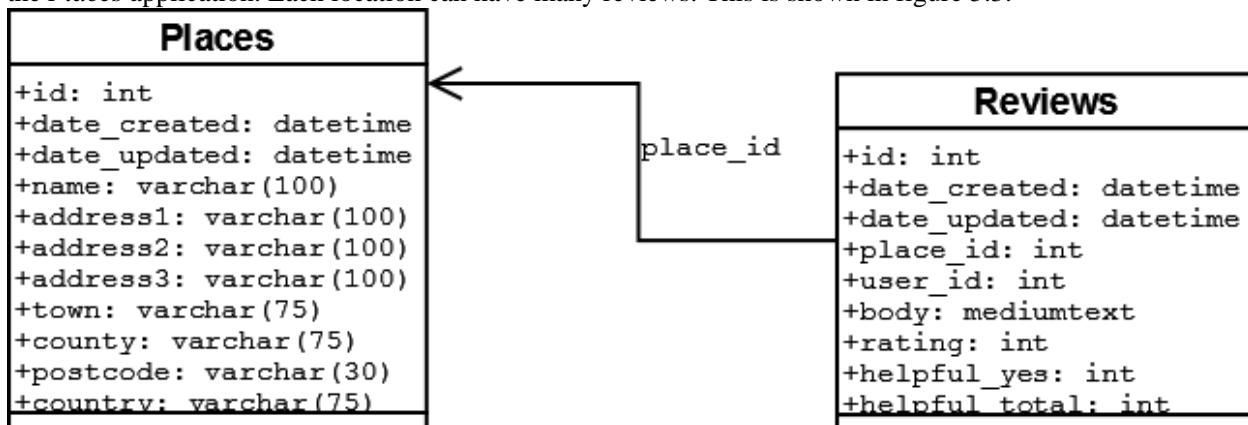


Figure 5.3: A One-to-Many relationship is created via a foreign key (`places_id`) in the `Reviews` table.

To represent this using `Zend_Db_Table` we create two new classes that extend from `Zend_Db_Table_Abstract` and then use the `$_dependantTables` and `$_referenceMap` properties to link them together. This is show in Listing 5.13.

Listing 5.13: One-to-many relationship using `Zend_Db_Table`

```

class Places extends Zend_Db_Table_Abstract
{
    protected $_name = 'places'; #1
    protected $_dependentTables = array('Reviews'); #2
}

class Reviews extends Zend_Db_Table_Abstract
{
    protected $_name = 'reviews';
    protected $_referenceMap = array( #3
        'Place' => array( #4
            'columns' => array('place_id'), |
            'refTableClass' => 'Places', |
            'refColumns' => array('id') |
        )
    );
}
  
```

(annotation) <#1 name of the database table>

(annotation) <#2 classname of dependency>

(annotation) <#3 list of relationships for this table>

(annotation) <#4 one-to-many relationship definition>

To define a one-to-many relationship, the `$_referenceMap` is used. As shown in #4, each relationship has its own sub-array and to ensure maximum flexibility, you inform the class exactly which columns are used in each table. In our case, the reviews table has a column called `places_id` which is linked to the `id` column in the places table.

NOTE

Both columns entries in the reference map are defined as arrays. This is to allow for composite keys containing more than one table field.

To actually retrieve the reviews for a given “place” record the `findDependentRowset()` function is used as shown:

```
$londonZoo = $places->fetchRow('id = 1');
$reviews = $londonZoo->findDependentRowset('Reviews');
```

`findDependentRowset()` takes two parameters: the name of the table class that you wish to receive the data from and an optional rule name. The rule name must be a key in the `$_referenceMap` previously set up, but if you don't supply one, it will look for a rule with a `refTableClass` that is the same as the class name of the table from which the row was created (“Places” in our case).

Many-to-Many Relationships

By definition, each Place has many reviews which were written by different users. Therefore for any given user, we can obtain the list of places that they have reviewed and conversely, for any given place, we can list the users who have reviewed it. This is known as a many-to-many relationship between places and users. To map this relationship within a database, a third table, known as a link table, is required. In our case, the link table is the Reviews table as shown in Figure 5.4.

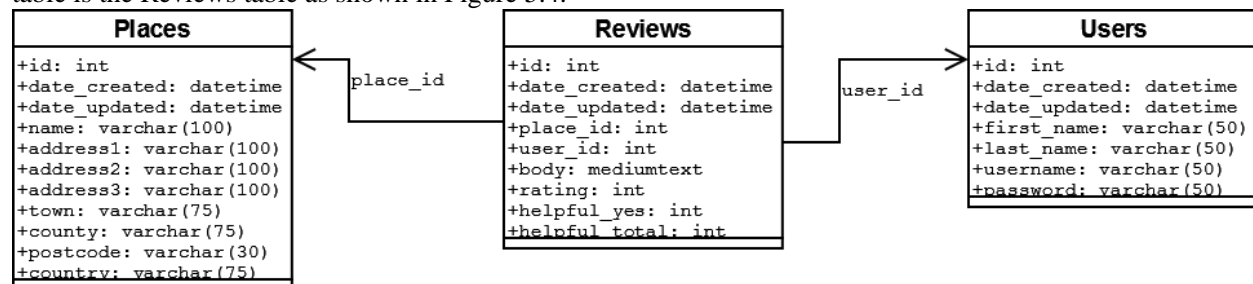


Figure 5.4: A Many-to-Many relationship between Places and Users is created via the Reviews table acting as a link.

The Reviews table has two foreign keys, one to Places and one to Users and so there are two one-to-many relationships also. There are three classes to define as shown in Listing 5.14.

Listing 5.14: Many-to-many relationship using Zend_Db_Table

```
class Places extends Zend_Db_Table_Abstract
{
    protected $_name = 'places';
    protected $_dependentTables = array('Reviews');
}

class Reviews extends Zend_Db_Table_Abstract
{
    protected $_name = 'reviews';
    protected $_referenceMap = array(                                #1
        'Place' => array(
            'columns' => array('place_id'),

```

```

        'refTableClass'    => 'Places',
        'refColumns'       => array('id')
    ),
    'User' => array(
        'columns'          => array('user_id'),
        'refTableClass'    => 'Users',
        'refColumns'       => array('id')
    )
);
}

class Users extends Zend_Db_Table_Abstract
{
    protected $_name = 'users';
    protected $_dependentTables = array('Reviews');    #2
}
(annotation) <#1 Two rules in the $_referenceMap>
(annotation) <#2 Directly dependant classes only>

```

As you can see in Listing 5.14, the Users class follows exactly the same pattern as that Places class and only defines the table classes that are directly dependant on. This means that we don't explicitly state a many-to-many relationship in the code and we let the Zend_Db_Table work out that there is one. To create a list of the places that a given user has reviewed, the following code is used:

```

$users = new Users();
$robAllen = $users->fetchRow('id = 1');
$places = $robAllen->findManyToManyRowset('Places', 'Reviews');

```

As you can see, the function that does the work for us is called findManyToManyRowset(). The first parameter is the destination table and the second is the intersection table class containing the rules that will link the initial table class to the intersection table class. In both cases, the parameters can either be strings or instances of Zend_Db_Table, so the snippet above could be written as:

```

$users = new Users();
$places = new Places();
$reviews = new Reviews();

$robAllen = $users->fetchRow('id = 1');
$places = $robAllen->findManyToManyRowset($places, $reviews);

```

This would produce exactly the same result. Note that if there is more than one rule in the \$_referenceMap of the intersection table that that would link the tables, then you can specify which rule to use as additional parameters to findManyToManyRowset(). Consider the situation where each review has to be approved by a moderator. The new table diagram would look like Figure 5.5.

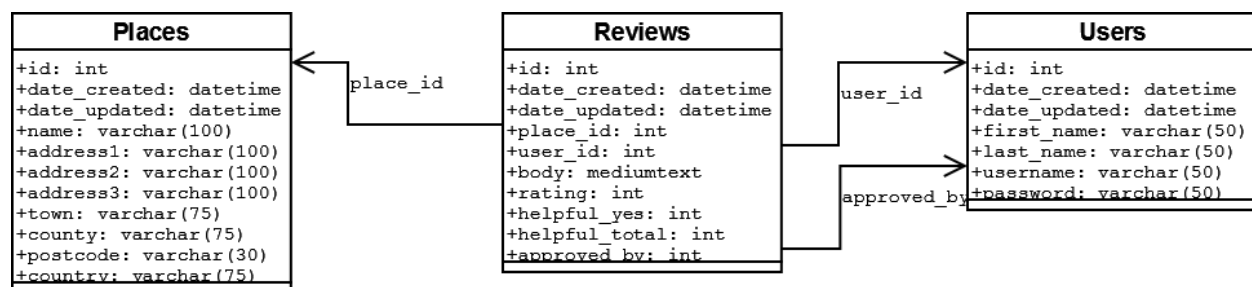


Figure 5.5: Multiple many-to-many relationship between Places and Users is created via two keys in the Reviews table.

In this case, we now have two foreign keys in the Reviews table that link to the Users table. To implement the second link, a new rule is required in the `$_referenceMap` for the Reviews table as shown in listing 5.15.

Listing 5.15: Reviews class with updated `$_referenceMap`

```
class Reviews extends Zend_Db_Table_Abstract
{
    protected $_name = 'reviews';
    protected $_referenceMap = array(
        'Place' => array(
            'columns'          => array('place_id'),
            'refTableClass'    => 'Places',
            'refColumns'       => array('id')
        ),
        'User' => array(
            'columns'          => array('user_id'),
            'refTableClass'    => 'Users',
            'refColumns'       => array('id')
        ),
        'ApprovedBy' => array(
            'columns'          => array('approved_by'), #1
            'refTableClass'    => 'Users', #2
            'refColumns'       => array('id')
        )
    );
}
```

(annotation) <#1 Second rule to link to Users table .>

(annotation) <#2 Field name in reviews table.>

As you can see, we can have multiple rules in the reference map that refer to the same table as long as they have different names (User and ApprovedBy in this case). We then use the columns key to in each map element to tell `Zend_Db_Table` the field name of the correct foreign key in the reviews table.

Selecting the list of places that have been reviewed by a given user is done using `findManyToManyRowset()` again, but this time we pass in the name of the rule we want to use:

```
// find all places that John Smith has approved a review of.
$johnSmith = $users->fetchRow('id = 4');
$places = $johnSmith->findManyToManyRowset('Places', 'Reviews', 'ApprovedBy');
```

The `Zend_Db_Table`'s handling of one-to-many and many-to-many relationships is surprisingly powerful, although requiring manual setting up of the relationships in the intersection class. We will now look at how we insert and update records using a `Zend_Db_Table`.

5.4 Summary

Throughout this chapter we have looked at the support for databases provided by the Zend Framework. As nearly all web applications use databases to store their data, it comes as no surprise to find that rich database support is provided. The foundation is the database abstraction layer, `Zend_Db_Abstract`, that takes care of the nitty-gritty of dealing with the differences between specific database engines and so provides a standardized interface for the rest of the components in the Framework. `Zend_Db_Table` builds upon the foundation to provide a table based interface to the data in the table and so makes working with the database from within the Model of an MVC application much easier. For those times when it is important to work with a specific row of data, `Zend_Db_Table_Row` provides simple access.

We have now covered the all the bases required for the underlying structure of a Zend Framework application and so will move on in our journey to look authentication and authorization which will allow us to restrict certain parts of our application to specific users and to ensure that those users can only perform actions appropriate to the trust that the application has in them.

User Authentication and Authorisation

Most websites nowadays restrict different areas to different people. For example, most e-commerce sites require that you are logged in before you can checkout your order. Other sites have “members-only” pages that can only be accessed after log-in. In *Places*, only registered users are able to write or rate a review. This functionality is known as authentication and authorisation and in this chapter we will look at the support provided by the Zend Framework.

6.1 Introducing Authentication and Authorisation

There are two different processes involved when it comes to allowing a user access to specific pages on a website. Authentication is the process of identifying an individual based on their credentials (usually username and password) and authorisation is the process of deciding if the user is allowed to do something. The Zend Framework’s two components, `Zend_Auth` and `Zend_Acl`, provide comprehensive support for all aspects of authentication and authorisation for websites.

As you must know who the user is, it follows that the authentication process must occur before authorisation, and so we will look at authentication and `Zend_Auth` first, before looking at `Zend_Acl`.

6.1.1 What is authentication?

The goal of the authentication is the process of deciding if someone is whom they say they are. There are three ways to recognise a user which are known as “factors”:

Something they know: password, pin, etc.

- Something they have: Driving licence, credit card, etc.
- Something they are: fingerprints, typing patterns, etc.

When you buy something from a shop using a credit card, two factors are used: “have” (the credit card in your pocket) and “know” (the pin number). For pretty much every website out there (including online banks) the “know” factor is the only mechanism used to identify a user. Generally this means a username and password, although banks in particular tend to ask for multiple pieces of information, such as a memorable date or place, in addition to a password.

Even though we have discovered that accepting a username/password for authorization is the standard for websites, the choice of where to store the information still has to be made. For standalone websites, it is common to use a database table containing the list of usernames and passwords, but there are other options. For sites that are part of a group, such as Yahoo!, then a separate system to handle the authentication is necessary. One common system is LDAP, the Lightweight Directory Access Protocol, which stores the information about the users in a separate service which can then be queried by other applications as required. OpenID and Six Apart’s Typekey are other systems that allow for authorization to be performed by another service.

6.1.2 What is authorisation?

Authorisation is the process of deciding whether or not to allow a user access to a resource or action. In web terms, this usually means we are deciding if someone is allowed to view a certain page or perform an action such as add a comment. One standard mechanism for doing this is to use an Access Control List (ACL) which is a list of permissions that are attached to a resource. The list specifies who is allowed access to the resource and what can be done with it. This means that the list will tell the system if a given user is allowed to view a database record, or execute a controller action.

Whenever a user wishes to do something, the list is checked to see if they are allowed to do the desired action with the desired data item. For example, a user may be allowed to view a news article, but be denied permission to edit it.

6.2 Implementing authentication

Now that we know what authentication and authorisation are, we can look at how they are implemented with a Zend Framework application. We will first look at how to implement authorisation with Zend_Auth using HTTP authentication and then look at how authentication is implemented within a “real-world” application using a database to hold the user information and sessions to store the information across multiple page views.

6.2.1 Introducing Zend_Auth

The Zend_Auth component is the part of the framework that deals with authentication and is separated into the core component and a set of authorisation adapters. The adapters contain the actual mechanisms for authorising users, such as using HTTP with a file or authorising against a database table. The authentication results are known as the identity. The fields stored in the identity depend upon the adapter. For example, HTTP authentication will place only the username into the identity, but database authentication might also include full name and email address. As it is common to display the name of the logged in user, this feature is very useful. To get at the identity information with the application, Zend Auth uses the Singleton pattern, so that the identity results can be retrieved wherever they are required.

The authorisation adapters that are provided “out of the box” for Zend_Auth are: Zend_Auth_Adapter_Http, Zend_Auth_Adapter_Digest and Zend_Auth_Adapter_DbTable. The Http and Digest adapters authorise against a file on disk using the standard HTTP login mechanism built into all browsers. The DbTable adapter is used for authorising against a list of users that are stored in a database table.

6.2.2 Logging In using HTTP authentication

There must be very few people who have not seen the standard HTTP login box as provided by the web browser (Figure 6.1).



Figure 6.1: The standard Http login box provided by a web browser.

Using this login system within your web application has the benefit of familiarity for the user at the expense of there being no easy way to log out. Let's look at how we implement HTTP authentication using Zend_Auth.

To authenticate someone, the process is to create an instance of an Auth Adapter and then authenticate using Zend_Auth's authenticate() function.

```
$authAdapter = new Zend_Auth_Adapter_Http();  
// set up $authAdapter so that it knows what to do  
$auth = Zend_Auth::getInstance();  
$result = $auth->authenticate($authAdapter);
```

The HTTP authentication protocol assumes that the pages that you want to protect are grouped into a realm, which is displayed to the user. For example, in Figure 6.1, the realm is "My Protected Area". The name of the realm must be provided to the Zend_Auth_Adapter_Http adapter and we must also create a resolver class to provide the password for a given username. The resolver class decouples the mechanism for authentication from the mechanics of retrieving the username and password from the user and generally would read the password from a database or file. A flowchart that describes the process is shown in Figure 6.2

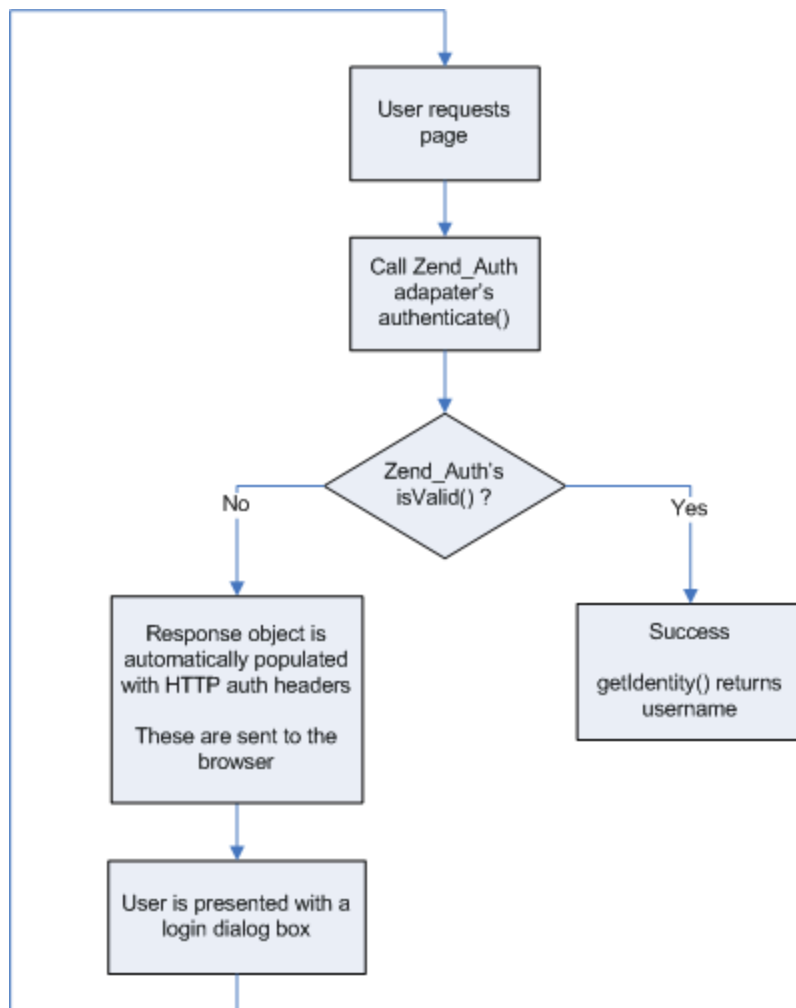


Figure 6.2: HTTP Authentication showing Zend_Auth methods.

To implement this, firstly, we need to configure the Zend_Auth_Adapter_Http as shown in listing 6.1.

Listing 6.1 Configuration of Zend_Auth_Adapter_Http

```
// create a Zend_Auth_Adapter_Http instance
$config['accept_schemes'] = 'basic';
$config['realm'] = 'ZFIA Chapter 06';
$authAdapter = new Zend_Auth_Adapter_Http($config); #1

$resolver = new Zend_Auth_Adapter_Http_Resolver_File('passwords.txt');
$authAdapter->setBasicResolver($resolver);
$authAdapter->setRequest($request); #2
$authAdapter->setResponse($response);
```

(annotation) <#1 Create the adapter>

(annotation) <#2 Access to http headers>

As you can see, configuration of a Zend_Auth_Adapter_Http object is two stage as some settings are configured within the \$config array that is used on construction of the object (#1), and the setting of the resolver, request and response objects is done after creation. The request object is used to retrieve the username and password and the response object is used to set the correct HTTP headers to indicate if authentication was successful (or not).

The `authenticate()` function is used to perform the actual authentication as shown in listing 6.2.

Listing 6.3 Authentication with `Zend_Auth_Adapter_Http`

```
$result = $authAdapter->authenticate(); #1
if (!$result->isValid()) {
    // Failed to validate. The response contains the correct HTTP
    // headers for the browser to ask for a username/password.
    $response->appendBody('Sorry, you are not authorised');
} else {
    // Successfully validated.
    $identity = $result->getIdentity(); #2
    $username = $identity['username'];
    $response->appendBody('Welcome, '.$username);
}
```

(annotation) <#1. Authenticate supplied username and password.>

(annotation) <#2. Retrieve the username and realm from the result.>

When `authenticate()` is called, it looks for the username and password HTTP headers. If they are not there, then the validation has failed, and it will set the “WWW-Authenticate” header in the response object. This will result in the browser displaying a dialog box requesting a username and password. Upon successful validation, the username of the person who has just logged in can be retrieved using the `getIdentity()` function which returns a (#2).

As you can see, HTTP authentication is easy with `Zend_Auth_Adapter_Http`, however most websites do not use it. Every public facing, commercial website provides their own login form to identify the user, usually asking for a username and password. There are a number of reasons for this, the main ones being:

- No way to log out other than exiting the browser completely.
- It’s not optional, so you cannot display the page regardless with different content for logged in users.
- You cannot change the look and feel or provide additional information within the login dialog box. For example, you may want to use email address rather than username.
- It’s not clear what to do if you have forgotten your password or do not have an account

Some of these can be worked around using cookies and JavaScript, but the user experience is still not good enough which is why everyone uses forms and sessions (or cookies). We will now integrate authentication using a login form and `Zend_Auth` into the *Places* application.

6.3 Using `Zend_Auth` within a real application

As *Places* is a community web site, the ability for members to be able to be identified is crucial and we have a users table ready to use. In order to implement authentication, we need a controller action to handle display and processing of a form to allow the user to enter their username and password. We also need to ensure that we know the identity of the current user throughout the application, so we take advantage of the Zend Framework’s Front Controller plug-in system.

6.3.1 Logging In

To log in and out of our application a separate controller, called `AuthController` is required. We use an action for display of the form (`auth/login`) and a separate action (`auth/identify`) to perform the actual identification. The class skeleton therefore looks like:

```
class AuthController extends Zend_Controller_Action
{
    public function indexAction()
```

Please post comments or corrections to the Author Online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=329>

```

    {
        $this->_forward('login');
    }

    public function loginAction()
    {
    }

    public function identifyAction()
    {
    }
}

```

Note that we need `indexAction` as it is marked abstract in the `Zend_Controller_Action` parent class. We simply redirect to `auth/login` if someone goes to `auth/index`. The HTML for a log in form is very simple and goes in the view script for the login action (`scripts/auth/login.tpl.php`) as show in Listing 6.4.

Listing 6.4 Log in Form: `auth/login.tpl.php`

```

<h1>Log in</h1>
<p>Please log in here</p>

<form method="POST"
    action="<?php echo $this->LinkTo('auth/identify'); ?>" #1
    <div>
        <label>Username</label>
        <input type="text" name="username" value="" />
    </div>
    <div>
        <label>Password</label>
        <input type="password" name="password" value="" />
    </div>
    <div>
        <input type="submit" name="login" value="Login" />
    </div>
</form>
(annotation) <#1. View Helper sets base URL.>

```

The HTML in listing 6.4 is very simple as we just need the user to supply two fields to us: username and password. As you can see in (#1), we use a view helper, `LinkTo()`, to create the correct URL to the `auth/identify` controller action. This View Helper is shown in listing 6.5 and exists to encapsulate the retrieval of the base URL from the request. The view script itself is not interested in such details which is why a view helper is the ideal solution.

Listing 6.5 `LinkTo()` View Helper

```

class Zend_View_Helper_LinkTo
{
    protected static $baseUrl = null; #1

    public function linkTo($path)
    {
        if (self::$baseUrl === null) {
            $request = Zend_Controller_Front::getInstance()->getRequest();
            $root = '/' . trim($request->getBaseUrl(), '/');
            if ($root == '/') {
                $root = '';
            }
            self::$baseUrl = $root . '/';
        }
    }
}

```

```

        return self::$baseUrl . ltrim($path, '/');
    }
}

```

(annotation) <#1. Cache the baseUrl.>

The LinkTo() function uses a static variable (#1) to cache the creation of the base URL. This minor optimization is to save the time taken to call through the multiple functions to get at the value from the front controller's request property. To actually log in, we direct the form to the action identify within the auth controller. The identify action's job is to use Zend_Auth to check that the supplied username and password is valid and is shown in listing 6.6.

Listing 6.6 The auth/identify Action

```

public function identifyAction()
{
    $success = false;
    $message = '';
    if ($this->_request->isPost()) { #1
        // collect the data from the user
        $formData = $this->_getFormData(); #2

        if (empty($formData['username'])
            || empty($formData['password'])) {
            $message = 'Please provide a username and password.';
        } else {
            // do the authentication
            $authAdapter = $this->_getAuthAdapter(); #3
            $auth = Zend_Auth::getInstance();
            $result = $auth->authenticate($authAdapter); #4
            if ($result->isValid()) {
                // success: store database row to auth's storage
                // (Not the password though!)
                $data = $authAdapter->getResultRowObject(null,
                    'password');
                $auth->getStorage()->write($data); #5

                $success = true;
                $redirectUrl = $this->_redirectUrl;
            } else {
                $message = 'Login failed';
            }
        }
    }

    if (!$success) {
        $flashMessenger = $this->_helper->FlashMessenger; #6
        $flashMessenger->setNamespace('actionErrors');
        $flashMessenger->addMessage($message);
        $redirectUrl = '/auth/login';
    }
    $this->_redirect($redirectUrl);
}

```

(annotation) <#1. Check for POST.>

(annotation) <#2. Retrieve form data.>

(annotation) <#3. Setup database adapter.>

(annotation) <#4. Do the work.>

(annotation) <#5. Store in the session.>

(annotation) <#6. Send message to next action.>

(#1) We are only interested in this request if it is a POST. This is a minor security improvement, but prevents us using GET where the username and password would be displayed on the address bar and hence may be bookmarked.

(#2) We use a helper function within the class to collect the data from the request and put into an array. The helper function filters the data to ensure that it is okay for use.

(#3) Setting up the Zend_Auth_Adapter_DbTable instance is complicated enough that we factor it out to its own function.

(#4) The authenticate() message does the authentication and returns a result object. The result's isValid() member function is used to test for successful log in.

(#5) If we have successfully authenticated, then we store the user's database record (except the password field) to the session.

(#6) Finally, if authentication has failed, we set up the Flash Messenger action helper to pass through the error message to the next request which is the log in form. The Flash Messenger is a "one time" message and automatically deletes itself when it is read. This makes it ideal for sending validation messages from one screen to the next. In our case it is read by the login controller action and the message is assigned to the view using this code:

```
$flashMessenger = $this->_helper->FlashMessenger;
$flashMessenger->setNamespace('actionErrors');
$this->view->actionErrors = $flashMessenger->getMessages();
```

Listing 6.6 shows the _getAuthAdapter() function which is the last part authentication puzzle. It creates an instance of Zend_Auth_Adapter_DbTable and assigns the supplied username and password to it ready for authentication by Zend_Auth.

Listing 6.6 The auth/identify Action

```
protected function _getAuthAdapter($formData)
{
    $dbAdapter = Zend_Registry::get('db');                                     #1

    $authAdapter = new Zend_Auth_Adapter_DbTable($dbAdapter);
    $authAdapter->setTableName('users')                                     | #2
    ->setIdentityColumn('username')                                       |
    ->setCredentialColumn('password');                                     |

    // get "salt" for better security                                     | #3
    $config = Zend_Registry::get('config');                               |
    $salt = $config->auth->salt;                                           |
    $password = sha1($salt.$formData['password']);                       |

    $authAdapter->setIdentity($formData['username']);                     | #4
    $authAdapter->setCredential($password);                               |

    return $authAdapter;
}
```

(annotation) <#1. Retrieve from registry.>

(annotation) <#2. Set up database specific information.>

(annotation) <#3. Secure the password better..>

(annotation) <#4. Set the authentication data.>

For a short function, quite a lot happens! The Zend_Auth_Adapter_DbTable object requires a connection to the database; fortunately, we stored one in the registry (#1) during the bootstrap startup phase ready for this sort of situation. After creation, we need to tell the adapter the name of the database table to use and which

fields within that table contain the identity and credentials (#2). In our case, we need the username and password fields from the users table.

While you can store the password in the database in plain text, it is more secure to store a hash of the password. A hash can be thought of as a one-way encryption in that it is unique for a given source string, but if you know the hash, you cannot determine the original string. As this is a common method of storing password data, websites have sprung up containing thousands of hashes for the two common hash algorithms (MD5 and SHA1). In order to help prevent reverse engineering should our data fall into the wrong hands, we further protect our users' passwords with a "salt". The salt is private string known only to the application that we prepend to the user's password to create a word that will not be in an online dictionary. This makes the hash very, very difficult, if not impossible to reverse engineer.

At this point we have completed logging in for our application. It would however be helpful if we provided links for logging in and out for the user. A view helper is one way to do this and encapsulates the logic required away from the view templates themselves.

6.3.3 A view helper welcome message

To provide a welcome message within a view helper we can take advantage of the fact that Zend_Auth implements the Singleton pattern and so we do not need to pass around an instance of it. Listing 6.7 shows the LoggedInUser view helper.

Listing 6.6 The auth/identify Action

```
class Zend_View_Helper_LoggedInUser
{
    protected $_view;

    function setView($view)                                | #1
    {
        $this->_view = $view;                                |
    }                                                        |

    function loggedInUser()
    {
        $auth = Zend_Auth::getInstance();
        if($auth->hasIdentity())                             #2
        {
            $logoutUrl = $this->_view->linkTo('auth/logout');
            $user = $auth->getIdentity();                     #3
            $username = $this->_view->escape(ucfirst($user->username));

            $string = 'Logged in as ' . $username . ' | <a href="' .
                $logoutUrl . '">Log out</a>';

        } else {
            $loginUrl = $this->_view->linkTo('auth/identify'); #4
            $string = '<a href="' . $loginUrl . '">Log in</a>'; #5
        }
        return $string;
    }
}
```

(annotation) <#1. Save view object.>

(annotation) <#2. Are we logged in?>

(annotation) <#3. Get the user record.>

(annotation) <#4. Use a view helper for the url.>

(annotation) <#5. Create string for display.>

Set `setView()` function (#1) is automatically called by the view before calling the main helper function. We use it to store a local reference to the view so that we can reference another view helper, `linkTo()` (#4). This is the same view helper that we created earlier in Listing 6.5 and is usefully re-used here.

The actual work is done within the `loggedInUser()` function. The `Zend_Auth` function `hasIdentity()` is used to check if there is a logged in user (#2). If there is, then the user's record is retrieved using the `getIdentity()` function (#3). As the user can select their own username, we use the `escape()` function to ensure that we don't accidentally introduce an XSS vulnerability and then create the string to display. If the current visitor is not logged in, a string is created with a link to allow the user to log in.

We use the view helper in our main site template to create the link in the main menu bar using the code:

```
<p><?php echo $this->loggedInUser(); ?></p>
```

The final result looks like Figure 6.3 and provides an easily discoverable log out link.



Figure 6.3. The logged in welcome message.

The final part of authentication is now upon us and we can implement the ability to log out.

6.3.4 Logging out

In comparison to logging in, logging out is very simple as all we need to do is call `Zend_Auth`'s `clearIdentity` function. Listing 6.7 shows the log out action in the `Auth` controller.

Listing 6.7: The auth/logout Controller Action

```
public function logoutAction()
{
    $auth = Zend_Auth::getInstance();
    $auth->clearIdentity();           #1
    $this->_redirect('/');           #2
}
```

(annotation) <#1. Unset the session variable.>

(annotation) <#2. Redirect to home page.>

As you can see, logging out really is simple, so it's time to move on to look at authorisation and giving logged in users more rights than visitors. This is handled by `Zend_Auth`'s sibling, `Zend_Acl`.

6.4 Implementing Authorisation

As we discussed in section 6.1, authorisation is the process of allowing a logged in user access to a specific resource. There are many ways of doing this, but a flexible and standard method is to employ role-based access control lists. The Zend Framework provide Zend_Acl to take care of this for us.

NOTE

There is a lot of jargon when talking about authorisation. The key ones are:

- Role: a grouping of users
- Resource: something to protect, such as a controller action or a data record such as a news item.
- Privilege: The type of access to required. Such as read or edit

Access control lists are a very flexible solution to access control; hence it is quite hard to understand how to apply the theory to the real world problem that you have. Figure 6.4 shows how the three main pieces of the puzzle are related.

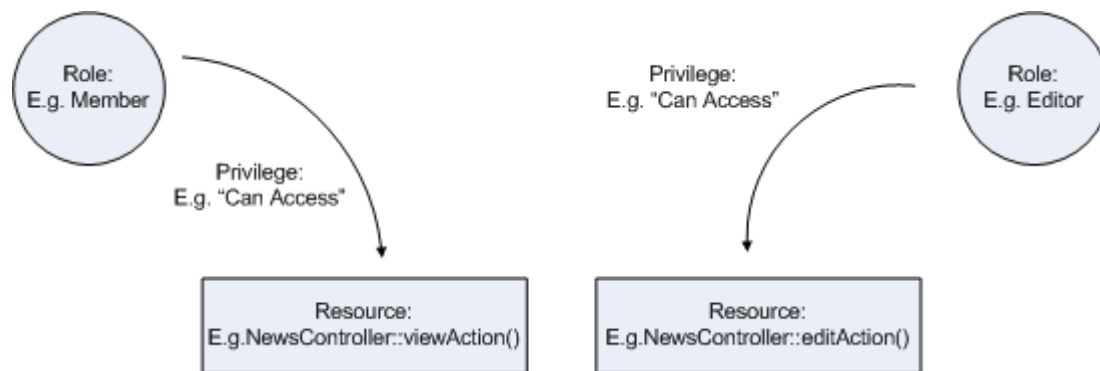


Figure 6.4. The relationship between the pieces of the ACL puzzle

This is just one way that ACL can be used to provide access to a resource, controller actions in this case. We'll start with a look at Zend_Acl and then move onto applying ACL to control access to different controller actions. Finally we will look at access control at the database level.

6.4.1 Introducing Zend_Acl

Using Zend_Acl is deceptively simple in theory. You create some roles and resources, set up the required permissions and call the `isAllowed()` function. Sounds easy enough! Let's look at that in a bit more detail, starting with a diagram of all related Zend_Acl classes as shown in figure 6.3.

Figure 6.3: Zend_Acl classes.

Zend_Acl_Role

A role is a responsibility that a user has within the system. A typical role would be "news editor" or "member". This is encapsulated with `Zend_Acl_Role` which is a very simple class that just holds the name of the role. The code to create a role is:

```
$roleMember = new Zend_Acl_Role('member');
```

Once a role is created, its name cannot be changed. Note also, that within the context of the ACL, each role name must be unique. We can assign a role to the Zend_Acl instance using:

```
$acl = new Zend_Acl();  
$acl->addRole($roleMember);
```

A given role may have a parent which means that the new role can do everything the parent role can do. A typical situation may be a forum moderator role, which would be coded like this:

```
$acl = new Zend_Acl();  
$acl->addRole(new Zend_Acl_Role('member'));  
$acl->addRole(new Zend_Acl_Role('moderator'), 'member');
```

In this situation, the moderator role has a parent of member and so a moderator can do everything that a member can do in addition to any other permissions that have been set.

Zend_Acl_Resource

A resource is something that you want to protect. A typical resource in a Zend Framework application would be a controller or action. For example you may want to protect access to the forums module controller so that only those users belonging to the member role have access to the controllers within it. Zend_Acl_Resource is as simple as Zend_Acl_Role and just holds the name of the resource, so creating one is simply a case of:

```
$forumResource = new Zend_Acl_Resource('comments');
```

A resource is attached to Zend_Acl in a similar manner to a role using the addResource() member function and again, parents are supported:

```
$acl = new Zend_Acl();  
$acl->addRole(new Zend_Acl_Role('member'));  
$acl->addRole(new Zend_Acl_Role('moderator'), 'member');  
$acl->addResource(new Zend_Acl_Resource('forum'));  
$acl->addResource(new Zend_Acl_Resource('posts'), 'forum');  
$acl->addResource(new Zend_Acl_Resource('threads'), 'forum');
```

In this example, we have created resource called posts and threads that are children of the forum resource. If we are mapping resources to modules and controllers, then the “posts” and “threads” resources represent controllers within the forum module.

Setting up Zend_Acl's Permissions

The final part of the setting up a Zend_Acl object for use is telling the object which permissions a given role has for accessing a given resources. To do this we need to look at the concept of privileges. A privilege is the type of access required. Typically, privileges are based around the operations that will be performed, so have names like “view”, “create”, “update”, etc.

Zend_Acl has two functions for setting permissions: allow() and deny(). We start off in state where all roles are denied access to all resources. The allow() function then provides access to a resource for a role and deny() and remove a subset of the allowed access for a particular case. Inheritance also comes into play here as permissions set for a parent role will cascade to child roles.

Let's look at an example:

```
$acl->allow('members', 'forum', 'view');
$acl->allow('moderators', 'forum', array('moderate', 'blacklist'));
```

In this example we have given view privileges to the member role for the forum resource and then have given the additional privileges of moderate and blacklist to the moderator role which also has view privileges as moderator is a child of member.

We have now looked at the building blocks of Zend_Acl and the key components of its API, so we can now look at how to integrate it into a real Zend Framework application. The first approach we will take is to protect specific controllers and actions for specific roles.

6.4.2 Configuring a Zend_Acl Object

As we have discovered, a fair amount of configuration of a Zend_Acl object is required before we can use it. This means that we want to make the setting up as easy as possible. We can make it easier on ourselves by storing the roles in our config.ini file. For each role, we need to store in its name and parent the INI file. Therefore we enter them into the INI file as shown in listing 6.8.

Listing 6.8: Extended Zend_Acl object

```
acl.roles.guest = null | #1
acl.roles.member = guest |
acl.roles.admin = member |
```

(annotation) <#1. All roles are a child of acl.roles for easy reading back.>

To read the configuration file, we extend Zend_Acl to read from the roles from config object and into the Acl object. The code do to this is shown in Listing 6.9.

Listing 6.9: Extended Zend_Acl object

```
class Places_Acl extends Zend_Acl
{
    public function __construct()
    {
        $config = Zend_Registry::get('config'); #1
        $roles = $config->acl->roles;
        $this->_addRoles($roles);
    }

    protected function _addRoles($roles)
    {
        foreach ($roles as $name=>$parents) { #2
            if (!$this->hasRole($name)) { #3
                if (empty($parents)) {
                    $parents = null;
                } else {
                    $parents = explode(',', $parents);
                }
                $this->addRole(new Zend_Acl_Role($name), $parents);
            }
        }
    }
}
```

(annotation) <#1. Retrieve from registry.>

(annotation) <#2. Iterate over all the roles.>

(annotation) <#3. Only add role if it doesn't already exist...>

As we have already explored `Zend_Config`, it comes as no surprise how easy it is to retrieve the data from the config object using a foreach loop. For each role, we check that it isn't already added and then create a new `Zend_Acl_Role` and add it to this object. The Acl object is going to be used to check permissions for every action and so we can create it in the bootstrap.

An especially tricky aspect is ensuring that we don't need to load the Acl object with information about every controller and action. The main issues are maintenance and performance. Maintaining a list of controllers independently of the actual controller class files will result in inaccuracies. We also do not want to spend the time and memory loading the Acl with rules that will never be required as this is a needless performance hit.

6.4.3 Checking the Zend_Acl Object

With these requirements in mind, the correct place to implement Acl checking is as an action helper. An action helper ties into the MVC dispatch system at the controller level and allows us to check the Acl rules before the controller's `preDispatch()` function is called. This happens before the actual action itself and so is an ideal place for the check. We need to populate the Acl object with the rules to be checked at `preDispatch()`. As this needs to be done earlier in the process, the `init()` function is where this needs to be done. In this case, the rules are controller-specific and so are written within each controller as required. Figure 6.5 shows the complete flow from requesting access to a controller's action to gaining access.

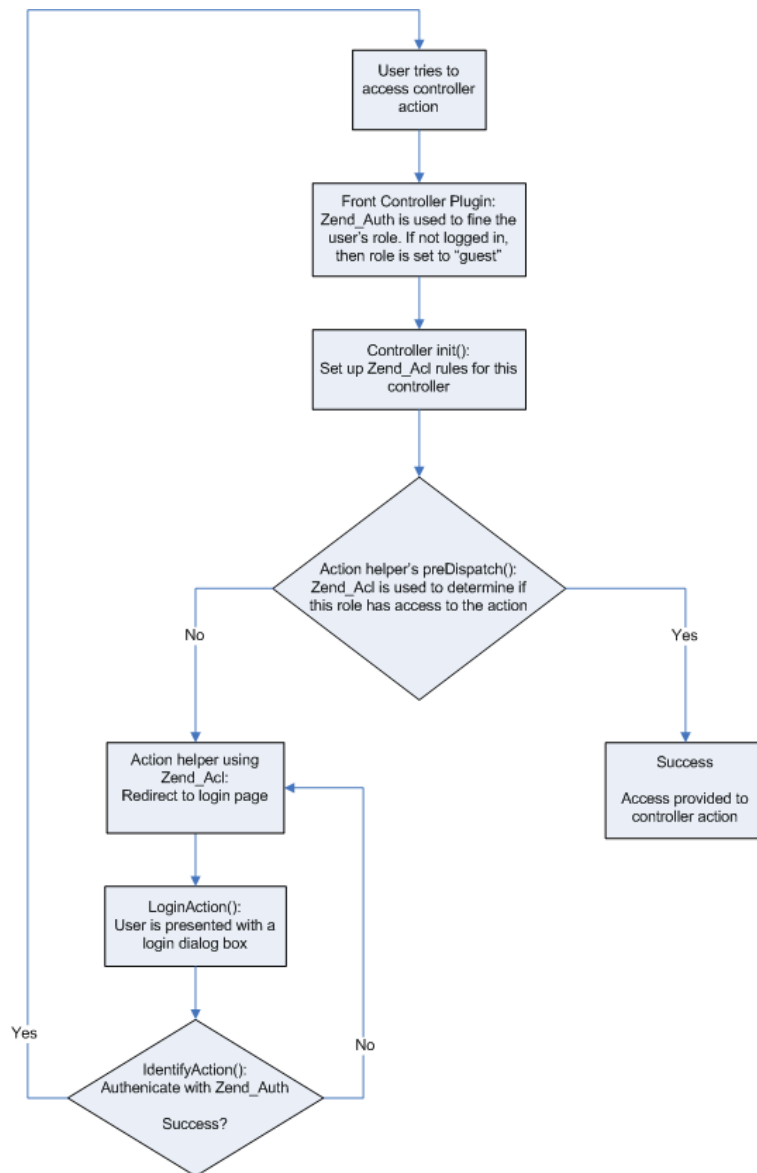


Figure 6.5. Zend_Auth and Zend_Acl working together to provide access to a controller action

The action controller is called `Places_Controller_Action_Helper_Acl` and is too long to list out completely here, so we'll look at the bits that do the real work. The class handles two important functions: providing a controller-centric interface to the underlying `Acl` object and performing the actual authentication. As the class is quite long, we'll look at it three sections. Listing 6.10 shows the class skeleton and the initial set up required.

Listing 6.10: Setting up the Acl action helper

```

class Places_Controller_Action_Helper_Acl extends
Zend_Controller_Action_Helper_Abstract
{
    protected $_action;
    protected $_auth;
    protected $_acl;
    protected $_controllerName;

    public function __construct(Zend_View_Interface $view = null,
array $options = array())

```

```

{
    $this->_auth = Zend_Auth::getInstance();           | #1
    $this->_acl = $options['acl'];                       |
}

/**
 * Hook into action controller initialization
 * @return void
 */
public function init()                                #2
{
    $this->_action = $this->getActionController();

    // add resource for this controller
    $controller = $this->_action->getRequest()->getControllerName();
    if(!$this->_acl->has($controller)) {
        $this->_acl->add(new Zend_Acl_Resource($controller));
    }
}
}

```

(annotation) <#1. Constructor.>

(annotation) <#2. Initialisation.>

(#1) The constructor is used to pass in an instance of the Acl object that we will use via the options array. We have to use the options array as the constructor signature is already set in the parent Zend_Controller_Action_Helper_Abstract class. As the Zend_Auth class is a singleton, we can use its getInstance() function to collect a reference to it.

(#2) The init() function is called immediately before the call to the controller's init() function. This makes it an ideal place to set up the required resource for the controller: the controller's name. We don't need any more resources than that, as we will use Zend_Acl's privileges as the actions, so that we don't have the performance hit of working out the names of the action functions dynamically.

Listing 6.11 shows the controller-centric rule functions, allow() and deny() which proxy through to the underlying Acl object.

Listing 6.10: Acl action helper

```

/**
 * Proxy to the underlying Zend_Acl's allow() function.
 *
 * We use the controller's name as the resource and the
 * action name(s) as the privilege(s)
 *
 * @param  Zend_Acl_Role_Interface|string|array    $roles
 * @param  string|array                            $actions
 * @uses   Zend_Acl::setRule()
 * @return Places_Controller_Action_Helper_Acl Provides a fluent interface
 */
public function allow($roles = null, $actions = null)
{
    $resource = $this->_controllerName;
    $this->_acl->allow($roles, $resource, $actions);
    return $this;
}

/**
 * Proxy to the underlying Zend_Acl's deny() function.
 *
 * We use the controller's name as the resource and the
 * action name(s) as the privilege(s)

```

```

*
* @param Zend_Acl_Role_Interface|string|array $roles
* @param string|array $actions
* @uses Zend_Acl::setRule()
* @return Places_Controller_Action_Helper_Acl Provides a fluent interface
*/
public function deny($roles = null, $actions = null)
{
    $resource = $this->_controllerName;
    $this->_acl->allow($roles, $resource, $actions);
    return $this;
}

```

(annotation) <#1. Constructor.>

(annotation) <#2. Initialisation.>

The functions `allow()` and `deny()` are used to set up the authentication rules for the controller. Typically this done in the controllers `init()` function. The view helper's versions of `allow()` and `deny()` simply fill in the resource parameter for us and change the terminology from privileges to actions. Whilst that doesn't seem like a lot, it makes much more sense when creating the rules in the controller and hence maintenance long-term is easier too.

For Places, we have different rules depending on the controller. For the index controller, we just want to give everyone access, so the `init()` function is simply:

```

class IndexController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->acl->allow(null);
    }

    //... class continues...
}

```

For other controllers, the rules will be more complex. For example listing 6.11 shows what is needed to ensure that members cannot manage records within the Places Controller.

Listing 6.10: Acl action helper

```

class PlaceController extends Zend_Controller_Action
{
    function init()
    {
        $memberActions = array('index', 'details', 'reportError'); |#1
        $this->_helper->acl->allow('member', $memberActions); |
        $adminActions = array('add', 'edit', 'delete'); |#2
        $this->_helper->acl->allow('admin', $adminActions); |
    }
    //... class continues...
}

```

As you can see, we allow the member role access to one set of actions (#1) and administrators access to another set (#2). We don't need to explicitly tell the system to allow administrators access to the member actions as the administrators role is a child of the member role and so inherits those permissions automatically.

6.5 Summary

In this chapter we have looked at the two related concepts of authorisation and authentication. Ensuing that we know the identity of the current user is the process of authentication and `Zend_Auth` is an intuitive and comprehensive component to that allows for checking against different data sources with more being added. By leveraging `Zend_Session`, `Zend_Auth` provides a one-stop-shop solution and enables us to look up the currently logged on user very easily.

Choosing an authentication strategy to ensure that only users with the correct privileges are allowed access to certain parts of the application is an art in itself. We have explored a solution that uses an action helper to easily limit access to controller actions without having to do lots of set up independently of the controller being protected. One key benefit of the flexibility of the Zend Framework is that there is no need to force a particular application's requirements into one methodology as the framework can be moulded to solve the problem in hand.

Forms are one of those aspects of building a web site that developers either love or loathe due to the complexities of validation and ensuring that the user is informed about issues in the form in a friendly manner. In the next chapter, we will look at how the Zend Framework can help make building and validating online forms easier with view helpers and the `Zend_Filter_Input` component.

Searching

One feature that marks out an excellent web site from a mediocre one is search. It doesn't matter how good your navigation system is, your users are used to Google and so expect that they can search your web site to find what they are looking for and if they can't search, or if the search doesn't return useful results, then they will try another site that does. A good search system is also hard to write but the Zend Framework provides the `Zend_Search_Lucene` component to make it easier.

In this chapter, we'll look at how search should work for the user and then look at how the `Zend_Search_Lucene` component works. We will then integrate search into an example website, looking at how to index a model and how to present results to the user.

8.1 Benefits of search

For most of us, we get the best from a web site when we successfully read the information we came for. For an e-commerce site, this generally means finding the item and purchasing it.

8.1.1 Key usability issue for users

Users want only one thing from their search: to find the right thing with the minimum amount of effort. Obviously, this isn't easy and is a major reason why Google is the number one search engine. Generally, a single text field is all that is required. The user enters their search term, presses the submit button and gets a list of answers.

8.1.2 Ranking results is important

For a search system to be any good, it is important that the relevant results are displayed. The way to do this is for each results to be ranked in order of relevance to the search query. In order to do this a full text search engine is used. Full text search engines work by creating a list of keywords for each page of the site. The list of keywords is known as the index. Along with the list of keywords, other relevant data is also stored, such as a title, date of document, author and information on how to retrieve the page, such as the url. When the runs a query, each document in the index is ranked based on how many of the requested keywords are in the document. The results are then displayed in order with (hopefully) the most useful at the top

8.2 Introducing `Zend_Search_Lucene`

The Zend Framework's search component, `Zend_Search_Lucene` is a very powerful tool. It is a full text search engine that is based on the popular Apache Lucene project which is a search engine for Java. The index files created by `Zend_Search_Lucene` are compatible with Apache Lucene and hence any of the index management utilities written for Apache Lucene will work with `Zend_Search_Lucene` too.

Zend_Search_Lucene creates an index of documents. The documents are each instances of Zend_Search_Lucene_Document. Each document contains Zend_Search_Lucene_Field objects which contain the actual data. A visual representation is shown in Figure 8.1.

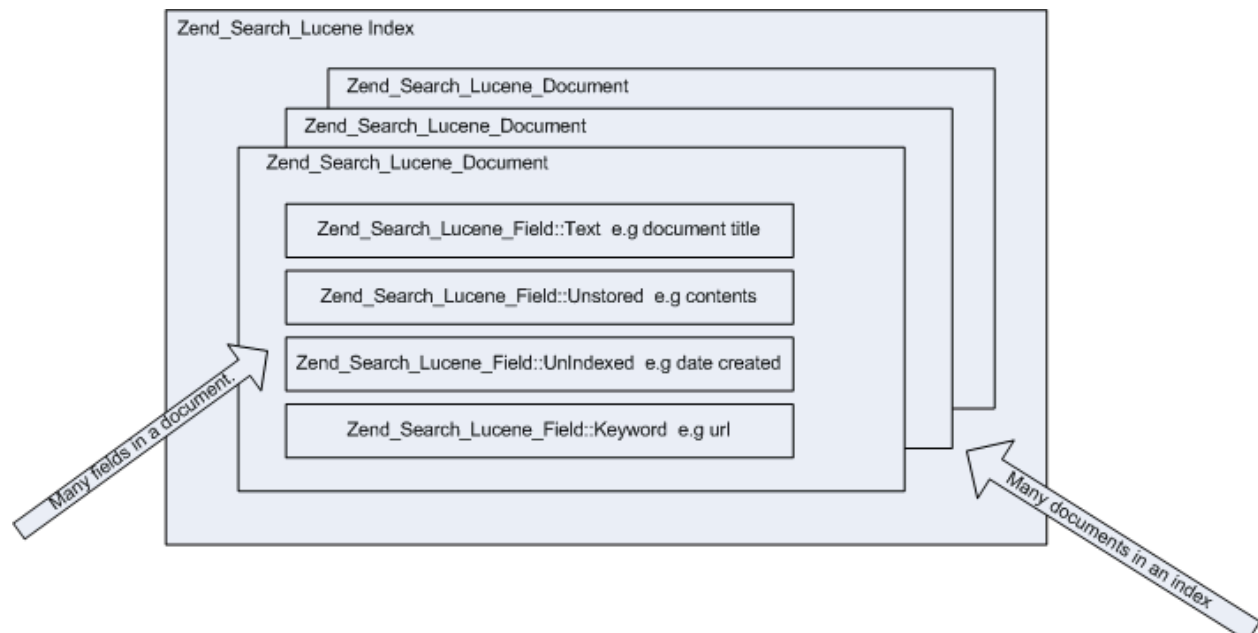


Figure 8.1 A Zend_Search_Lucene index consists of multiple documents, each containing multiple fields. Not all fields are stored and some fields may contain data for display rather than searching.

Queries can then be issued against the index and an ordered array of results (each of type Zend_Search_Lucene_Search_QueryHit) is returned. The first part of implementing a solution using it, is therefore to create an index.

8.2.1 A separate search index for your website

Creating an index for Zend_Search_Lucene is just a case of calling the create() function:

```
$index = Zend_Search_Lucene::create('path/to/index');
```

The index created is actually a directory that contains a few files in a format that is compatible with the main Apache Lucene project. This means that if you want to, you could create index files using the Java or .Net applications, or conversely, create indexes with Zend_Search_Lucene and search them using Java.

Having created the index, the next thing to do is to put data in it for searching on. This is where it gets complicated and we have to pay attention! Adding data is easily done using the addDocument() method, however you need to set up the fields within the document and each field has a type.

For example:

```
$doc = new Zend_Search_Lucene_Document();
$doc->addField(Zend_Search_Lucene_Field::UnIndexed('url', $url));
$doc->addField(Zend_Search_Lucene_Field::UnStored('contents', $contents));
$doc->addField(Zend_Search_Lucene_Field::Text('desc', $desc));

$index->addDocument($doc);
```

As you can see in this example, the url data is of field type “UnIndexed”, contents are “UnStored” and the description is “Text”. Each field type is treated differently by the search engine to decide if it needs to store the

actual data or just use it for creating the index file. Table 8.1 shows the key field types and what the differences are.

Table 8.1 Lucene field types for adding fields to an index

Name	Indexed	Stored	Tokenised	Notes
Keyword	Yes	Yes	No	Use for storing and indexing data that is already split into separate words for searching.
UnIndexed	No	Yes	No	Use for data that isn't searched on, but is displayed in the search results, such as dates or database id fields.
Text	Yes	Yes	Yes	Use for storing data that is both indexed and used in the search results display. This data is split into separate words for indexing.
Unstored	Yes	No	Yes	Use for indexing the main content of the document. The actual data isn't stored as you won't be using it for search results display.
Binary	No	Yes	No	Use for storing binary data that is associated with the document, such as a thumbnail.

There are two reasons for adding a field to the search index: providing the search data and providing for display of the results. The data fields available cover both these operations and choosing the correct field type for a given piece of information is crucial for the correct operation of the search engine. Let's look at storing the data for searching first. This is known as indexing and so the Keyword, Text and Unstored fields are relevant. The main difference between Keyword and Text/Unstored is the concept of tokenizing which is when the indexing engine needs to analyse the data to determine the actual words in use. The Keyword field is not tokenized which means that each word is used exactly as spelt for the purposes of search matching. The Text /Unstored fields however are tokenized, which means that each word is analyzed and the underlying base word is used for search matching. For example, punctuation and plurals are removed for each word.

When using a search engine, the list of results needs to provide enough information for the user to determine if a given result is the one she is looking for. The field types that are "Stored" can help with this as they are returned by the search engine. These are the Keyword, UnIndexed, Text and Binary field types. The Unindexed and Binary field types exist solely for storing data used in the results display. Typically, the Binary field type would be used for storing a thumbnail image that relates to the record and the UnIndexed field type is used to store items such as a summary of the result or data related to finding the result, such as its database table, id or URL.

Zend_Search_Lucene will automatically optimize the search index as new documents are added. You can also force optimization by calling the optimize() function if required though. Now that we have created an index file, we can now perform searches on the index. As you would expect, Zend_Search_Lucene provides a variety of powerful mechanisms for building queries that produce the desired results and we shall explore them next

8.2.2 Powerful queries

Searching a Zend_Search_Lucene index is as simple as:

```
$index = Zend_Search_Lucene::open('path/to/index');  
$index->find($query);
```

The \$query parameter may be either a string or you can build a query using Zend_Search_Lucene objects and pass in an instance of Zend_Search_Lucene_Search_Query. Clearly passing in a string is the easiest, but for maximum flexibility, using the Zend_Search_Lucene_Search_Query object can be very useful. The String is converted to a search query object using a query parser and a good rule of thumb is that you should use the query parser for data from users and the query objects directly when programmatically creating queries. This

implies that for an advanced search system that most websites provide, a hybrid approach would be used. We'll explore that a little later. Let's look first at the query parser for strings.

String queries

All search engines provide a very simple search interface for their users: a single text field. This makes it very easy to use, but at first glance seems to make it harder to provide a complex query. Like Google, Zend_Search_Lucene has a query parser that can interpret what is typed into a single text field into a powerful query. When you pass in a string to the find() function, behind the scenes the function Zend_Search_Lucene_Search_QueryParser::parse() is called. This class implements the Lucene query parser syntax as supported by v 2.0 of Apache Lucene.

To do its work, the parser breaks down the query into terms, phrases and operators. A term is a single word and a query is multiple words grouped using quotation marks, such as "hello world". An operator is a boolean word (such as AND) or symbol modified used to provide more complex queries. Wildcards are also supported using the asterisk and question mark symbols. A question mark is used to represent a single character and the asterisk represents several characters. For instance searching for frame* will find frame, framework, frameset and so on. Other

Table 8.2 lists the key modifier symbols that can be applied to a search term

Table 8.2: Modifiers for search terms can be applied to control how the parser uses the term when searching.

Modifier	Symbol	Example	Description
Wildcards	? and *	H?t h*t	? is a single character placeholder and * represents multiple characters
Proximity	~x	"php power"~10	The terms must be within a certain number of words apart (10 in the example)
Inclusive Range	fieldname:[x TO y]	category:[skiing TO surfing]	Find all documents whose field values are between the upper and lower bounds specified
Exclusive Range	fieldname:{x To y}	published:[20070101 TO 20080101]	Find all documents whose field values are greater than the specified lower bound and less than the upper bound. This does not include the bounds.
Term Boost	^x	"Rob Allen"^3	Increase the relevance of a document containing this term. The number determines how much of a increase in relevance is given.

Each modifier in table 8.1 affects only the term to which it is attached. Operators on the other hand affect the make up of the query. The key operators available are listed in Table 8.3.

Table 8.3: Boolean operators affect the make up of the query and provide the power to refine the search

Operator	Symbol	Example	Description
Required	+	+php power	The term after the + symbol must exist in the document.
Prohibit	-		Documents containing the term after the - symbol are excluded from the search results.
AND	AND or &&	php and power	Both terms must exist anywhere in the document
OR	OR or	php or html	Either term must be present in all returned documents.
NOT	NOT or !	php not java	Only include documents that contain the first term but not the second term.

Careful use of the Boolean operators can create very complex queries however, in the "real world" most users are unlikely to use more than one or two operators in any given query. The set of Boolean operators supported is pretty much the same as that used by major search engines, such as Google, and so your users will have a degree of familiarity. We will now look at the other way to creating a search query: programmatic creation.

Programmatically creating search queries

A programmatic query involves instantiating the correct objects and putting them together. There are a lot of different objects available that can be combined to make up a query. Table 8.4 lists search query objects that can be used.

Table 8.4: Search query objects

Type	Example and Description
Term Query	<p>A term query is used to search for a single term.</p> <p>Example search: category:php</p> <pre>\$term = new Zend_Search_Lucene_Index_Term('php', 'category'); \$query = new Zend_Search_Lucene_Search_Query_Term(\$term); \$hits = \$index->find(\$query);</pre> <p>Finds the term “php” within the category field. If the second term of the Zend_Search_Lucene_Index_Term constructor is omitted, then all fields will be searched.</p>
Multi-Term Query	<p>Multi-term queries allow for searching through many terms. For each term, you can optionally apply the required or prohibited modifier. All terms are applied to each document when searching.</p> <p>Example search: +php power -java</p> <pre>\$query = new Zend_Search_Lucene_Search_Query_MultiTerm(); \$query->addTerm(new Zend_Search_Lucene_Index_Term('php'), true); \$query->addTerm(new Zend_Search_Lucene_Index_Term('power'), null); \$query->addTerm(new Zend_Search_Lucene_Index_Term('java'), false); \$hits = \$index->find(\$query);</pre> <p>The second parameter controls the modifier: true for required, false for prohibited and null for no modifier. In this the query will find all documents that contain the word “php” and may contain the word “power” but do not contain the word “java”.</p>
Wildcard Query	<p>Wildcard queries are used for search for a set of terms where only some of the letters are known.</p> <p>Example search: super*</p> <pre>\$term = new Zend_Search_Lucene_Index_Term('super*'); \$query = new Zend_Search_Lucene_Search_Query_Wildcard(\$term); \$hits = \$index->find(\$query);</pre> <p>As you can see, the standard Zend_Search_Lucene_Index_Term object is used and then passed to a Zend_Search_Lucene_Search_Query_Wildcard object to ensure that the * (or ?) is interpreted correctly. If you pass a wildcard term into an instance of Zend_Search_Lucene_Search_Query_Term, then it will be treated as a literal.</p>
Phrase Query	<p>Phrase queries allow for searching for a specific multi-word phrase.</p> <p>Example search: “php is powerful”</p> <pre>\$query = new Zend_Search_Lucene_Search_Query_Phrase(array('php', 'is', 'powerful')); \$hits = \$index->find(\$query);</pre> <p>or</p> <pre>// separate Index_Term objects \$query = new Zend_Search_Lucene_Search_Query_Phrase(); \$query->addTerm(new Zend_Search_Lucene_Index_Term('php')); \$query->addTerm(new Zend_Search_Lucene_Index_Term('is')); \$query->addTerm(new Zend_Search_Lucene_Index_Term('powerful')); \$hits = \$index->find(\$query);</pre>

	<p>The words that make up the query can be either specified in the constructor of the Zend_Search_Lucene_Search_Query_Phrase object or each term can be added separately. You can also search with “word” wildcards so:</p> <pre>\$query = new Zend_Search_Lucene_Search_Query_Phrase(array('php', 'powerful'), array(0, 2)); \$hits = \$index->find(\$query);</pre> <p>will find all three word phrases that have “php” as the first word and “powerful” as the last. This is known as “slop” and you can also use the setSlop() function to achieve the same effect like this:</p> <pre>\$query = new Zend_Search_Lucene_Search_Query_Phrase(array('php', 'powerful'); \$query->setSlop(2); \$hits = \$index->find(\$query);</pre>
Range Query	<p>Find all records within a particular range, usually date, but can be anything.</p> <p>Example search: published_date:[20070101 to 20080101]</p> <pre>\$from = new Zend_Search_Lucene_Index_Term('20070101', 'published_date'); \$to = new Zend_Search_Lucene_Index_Term('20080101', 'published_date'); \$query = new Zend_Search_Lucene_Search_Query_Range(\$from, \$to, true /* inclusive */); \$hits = \$index->find(\$query);</pre> <p>Either boundary may be null to imply “from the beginning” or “until the end” as appropriate.</p>

Clearly the benefit of using the programmatic interface to Zend_Search_Lucene over the string parser is that it’s easier to clearly express search criteria exactly and allows the use of an “advanced search” web form for the user to refine their search.

8.2.3 Best practices

We have now covered all you need to know about the using Zend_Search_Lucene, but before we move on to implementing search into a real application, it will be beneficial to look at a few best practices for using Zend_Search_Lucene.

Firstly, don’t use id or score in field names as this will make them harder to retrieve again. For other field names you can do this:

```
$hits = $index->find($query);
foreach ($hits as $hit) {
    // Get 'title' document field
    $title = $hit->title;
```

But to retrieve the a field called id, you have to do:

```
$id = $hit->getDocument()->id;
```

This is only required for the fieldnames called id and score, so it’s best to use different names, such as doc_id and doc_score.

Secondly, you need to be aware of memory usage. Zend_Search_Lucene uses a lot of memory! The memory usage increases if you have a lot of unique terms which occurs if you have a lot of untokenised phases as field values. This means that you are indexing a lot of non-text data. From a practical point of view, this means that Zend_Search_Lucene works best for searching text which isn’t a problem for most websites! Note that indexing uses a bit more memory too, and can be controlled using the MaxBufferedDocs parameter. Further details are in the main Zend Framework manual.

Lastly, Zend_Search_Lucene uses UTF-8 character coding internally and if you are indexing non-ascii data, it is wise to specify the encoding of the data fields when adding to the index. This is done using the optional third parameter of the field creation methods, for example:

```
$doc = new Zend_Search_Lucene_Document();  
$doc->addField(Zend_Search_Lucene_Field::Text('body', $body, 'iso-8859-1'));
```

Let's do something interesting with Zend_Search_Lucene and integrate searching into the *Places* website.

8.3 Adding search to Places

As Places is a community site, a search facility will be expected by its members. We will implement a simple “one field” search available on every page in the side bar so that it is very easy for users to find what they want. The results will look like Figure 8.2.

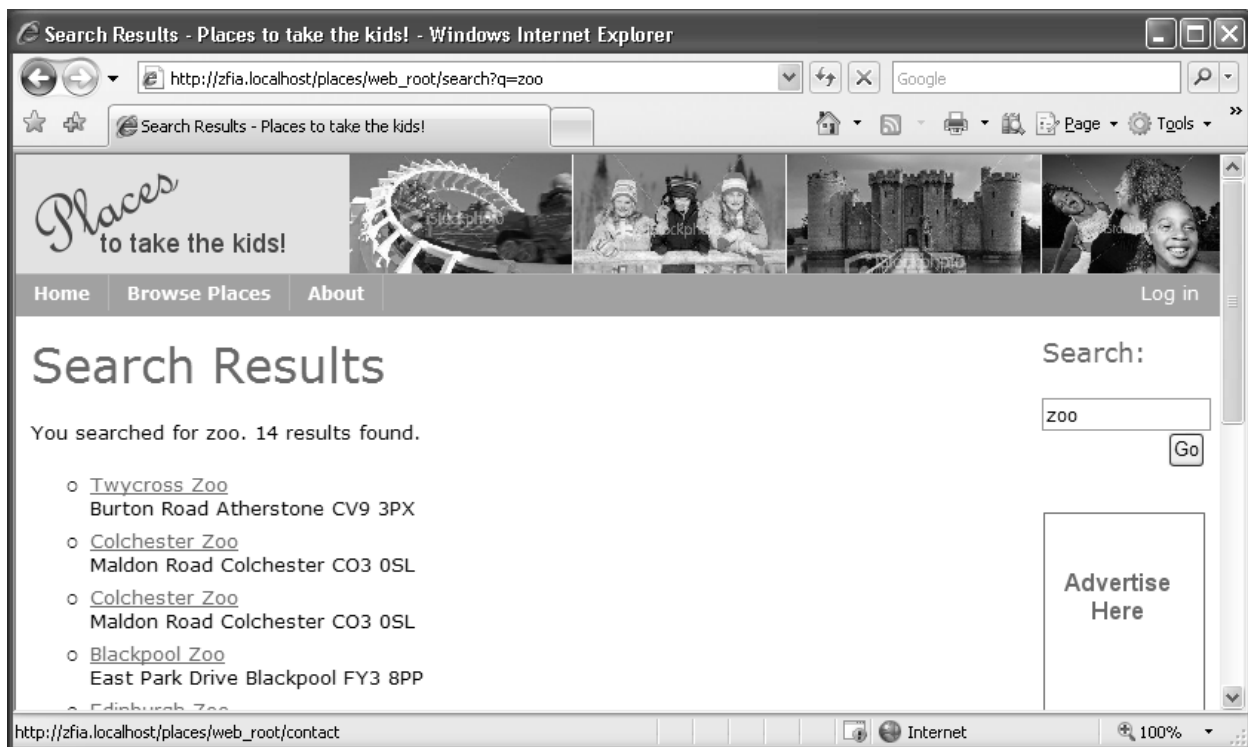


Figure 8.2 Search results page for Places. Each entry has a title which is linked to the page followed by a summary. The results are ranked by the search engine so that the most relevant are at the top.

In order to achieve this, we are going to create the index files used by the search first and then we will write the simple form and search results page, finishing with the advanced search form.

8.3.1 Updating the index as new content is added

We have two choices when to create index files: when adding data to the database or as a scheduled task using cron or similar. A lot depends on the traffic that the site receives as to when the additional stress of creating an index is done. For *Places*, we want both. Adding to the index as new data is added to the site is vital for a useful web site. We also want the ability to re-index all the data in one hit so that we can optimize the search index files as the amount of data on the site gets larger. We'll start by looking at adding to the index as we go along as we can leverage that work for the full re-index later.

Designing the index

When creating our index, we need to consider the fields that we are going to create in our index. The search index is going to contain records of different types, such as pages, reviews or user profile data but presents to the user a list of web pages to view. Therefore we unify the set of fields in the index so that the results will make sense to the user. Table 8.5 shows the set of fields we will use.

Table 8.5 Lucene field types for adding fields to an index

Field name	Type	Notes
class	UnIndexed	The class name of the stored data. We need this on retrieval in order to be able to create a URL to the correct page in the results list.
key	UnIndexed	The key of the stored data. Usually this is the id of the data record. We need this on retrieval in order to be able to create a url to the correct page in the results list.
docRef	Keyword	A unique identifier for this record. We need this so we can find it for updates or deletions.
title	Text	This is the title of the data and will be used for searching on and display within the results.
contents	UnStored	This is the main content which is used for search and is not displayed.
summary	UnIndexed	The summary contains information about the search result to display within the results. It is not used for searching.
createdBy	Text	The author of the record is used for searching and display. We use the Keyword type as we want to preserve the author's name exactly when searching.
dateCreated	Keyword	The date created is used for searching and display. We use the keyword type as we do not want Lucene to parse the data.

The basic code for creating the `Zend_Search_Lucene_Document` to be indexed is the same no matter what type of data we are going to be indexing. It looks something like the code in Listing 8.1

Listing 8.1: Adding a document to the search index.

```
function addToIndex($index, $class, $key, $title, $contents,
    $summary, $createdBy, $dateCreated)
{
    $doc = new Zend_Search_Lucene_Document();           A

    $doc->addField(Zend_Search_Lucene_Field::UnIndexed('class',
        $class));                                       1
    $doc->addField(Zend_Search_Lucene_Field::UnIndexed('key',
        $key));                                       1
    $this->addField(Zend_Search_Lucene_Field::Keyword('docRef',
        "$class:$key"));                               1
    $doc->addField(Zend_Search_Lucene_Field::Text('title',
        $title));                                     1
    $doc->addField(Zend_Search_Lucene_Field::UnStored('contents',
        $contents));                                   1
    $doc->addField(Zend_Search_Lucene_Field::UnIndexed('summary',
        $summary));                                     1
    $doc->addField(Zend_Search_Lucene_Field::Keyword('createdBy',
        $createdBy));                                   1
    $doc->addField(Zend_Search_Lucene_Field::Keyword('dateCreated',
        $dateCreated));                               1

    $index->addDocument($doc);                         2
}
```

A Create the search document.

B Add document to the index.

This function creates a `Zend_Search_Lucene` document, called `$doc` and then we add each field to the document (#1), before calling `addDocument()` (#2). As we are going to be adding documents for every page on the website, we should make the creation of the document as easy as possible. We will extend

Zend_Search_Lucene_Document so that we can instantiate the object with the right data in one line of code. This is shown in listing 8.2.

Listing 8.2: Extending Zend_Search_Lucene_Document for easier creation

```
class Places_Search_Lucene_Document extends Zend_Search_Lucene_Document
{
    public function __construct($class, $key, $title,
        $contents, $summary, $createdBy, $dateCreated)
    {
        $this->addField(Zend_Search_Lucene_Field::Keyword(
            'docRef', "$class:$key"));
        $this->addField(Zend_Search_Lucene_Field::UnIndexed(
            'class', $class));
        $this->addField(Zend_Search_Lucene_Field::UnIndexed(
            'key', $key));
        $this->addField(Zend_Search_Lucene_Field::Text(
            'title', $title));
        $this->addField(Zend_Search_Lucene_Field::UnStored(
            'contents', $contents));
        $this->addField(Zend_Search_Lucene_Field::UnIndexed(
            'summary', $summary));
        $this->addField(Zend_Search_Lucene_Field::Keyword(
            'createdBy', $createdBy));
        $this->addField(Zend_Search_Lucene_Field::Keyword(
            'dateCreated', $dateCreated));
    }
}
```

Using the new `Places_Search_Lucene_Document` class is now very easy as shown in listing 8.3.

Listing 8.3: Adding to the index

<code>\$index = Zend_Search_Lucene::open(\$path);</code>	A
<code>\$doc = new Places_Search_Lucene_Document(</code>	B
<code> \$class, \$key, \$title, \$contents,</code>	B
<code> \$summary, \$createdBy, \$dateCreated);</code>	B
<code>\$index->addDocument(\$doc);</code>	C

A Open the index

B Create the document in one line of code

C Add to the index

That's all there is to it! We now need to write this code within each model class that is to be searched over, and we immediately hit a design brick wall. The model clearly knows all about the data to be searched, however it should not be know about how to add that data to the search index. If it did, it would be tightly coupled to the search system which would make our lives much harder if we ever want to change the search engine to another one. Fortunately, programmers before us have had the same basic problem that we have and it turns out that it's so common that there's a design pattern named after the solution: the Observer pattern.

Using the Observer Pattern to decouple indexing from the model

The Observer pattern describes a solution that uses the concept of notifications. An *observer* object registers interest in an *observable* object, and then when something happens to the *observable*, the *observer* is notified and can take appropriate action. In our case, this is shown in Figure 8.3.

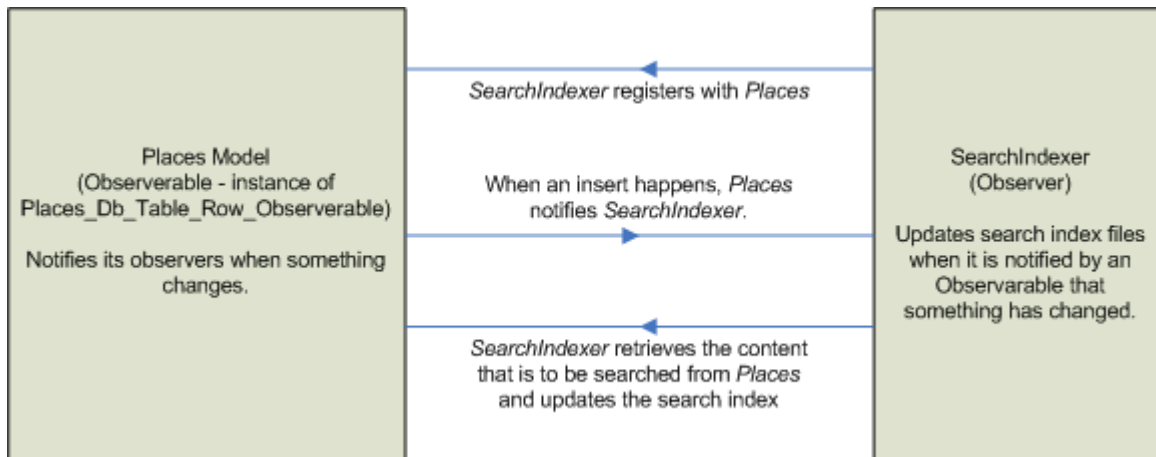


Figure 8.3 The Observer pattern allows us to decouple the search indexing from the model data. This will allow us to easily add new models to be searched or to change the way we index for searching.

Our observable classes are our models and so we want to allow observers to register themselves with the models. The data we are interested in are instances of `Zend_Db_Table_Row_Abstract`, and so we'll create an extension class called `Places_Db_Table_Row_Observable` that will contain the functions that allow us to register and notify observers. Listing 8.4 shows the class skeleton.

Listing 8.4: Places_Db_Table_Row class

```
class Places_Db_Table_Row_Observable extends Zend_Db_Table_Row_Abstract
{
    protected static $_observers = array();                                     A

    public static function attachObserver($class)
    {
        if (!is_string($class) || !class_exists($class)                       1
            || !is_callable(array($class, 'observeTableRow'))) {                1
            return false;                                                       1
        }

        if (!isset(self::$_observers[$class])) {                             2
            self::$_observers[$class] = true;                                   2
        }

        return true;
    }

    protected function notifyObservers($event)
    {
        if (!empty(self::$_observers)) {
            foreach (array_keys(self::$_observers) as $observer) {              3
                call_user_func(array($observer, 'observeTableRow'),
                               $event, $this);
            }
        }
        parent::_postInsert();
    }
}
```

```

    }
}
A Array of registered observers
1 Ensure that observer contains a function called observeTableRow()
2 Add the observer to the list if it's not already there.

```

The first two functions we need are the core of the Observer pattern. The `attachObserver()` function is used to allow an observer to attach itself. We use a static function as the list of observers because the list of observers is independent of a specific model class. Similarly, the array of observers is static as it needs to be accessible from every model class.

The `notifyObservers()` function is used to notify all observers that have been attached to the class. This function simply iterates over the list and calls a static function, `observeTableRow()` within the observer class. The observer can then use the information about what the event is and the data from the model to perform whatever action is necessary. In this case, we will update the search index. `Zend_Db_Table_Row_Abstract` provides a number of hook functions that allow us to perform processing before and after the database row is inserted, updated or deleted. We use those functions to call `notifyObservers()`; the ones we need for updating the search index are `_postInsert()`, `_postUpdate()` and `_postDelete()`. Each function is the same as `_postInsert()` is shown in Listing 8.5.

Listing 8.4: Notifying the observer after an insert

```

class Places_Db_Table_Row_Observable extends Zend_Db_Table_Row_Abstract
{
    protected function _postInsert()
    {
        self::notifyObservers('post-insert');    A
    }
}
A Pass the event type to the observer as a string

```

Let's look now at the observer class. This class is called `SearchIndexer` and, as it is a model, is stored in `application/models`. The code to register it with the observable class is in the bootstrap and looks like this:

```

SearchIndexer::setIndexDirectory(ROOT_DIR . '/var/search_index');
Places_Db_Table_Row_Observable::attachObserver('SearchIndexer');

```

As you can see, it's simply a case of setting the directory to store the search index files and then attaching the class to the `Places_Db_Table_Row_Observable` using the name of the class. There are three main functions in `SearchIndexer` and these are shown in Listings 8.5, 8.6 and 8.7.

Listing 8.5: SearchIndexer::observeTableRow(): the notification hook function

```

class SearchIndexer
{
    public static function observeTableRow($event, $row)
    {
        switch ($event) {
            case 'post-insert':                A
            case 'post-update':                A
                $doc = self::getDocument($row);    1
                if ($doc !== false) {
                    self::_addToIndex($doc);        2
                }
                break;
        }
    }
}

```

```
}
A Only add to index for certain events
1 Retrieve the data from the model
2 Update the search index
```

The notification hook function, `observeTableRow()` checks the event type and if the event indicates that new data has been written to the database, then it retrieves the data from the model (#1) and then updates the search index files (#2). All that's left to do is to retrieve the data from the Model

Adding the model's data to the index

The process of retrieving the data from the model is a separate function as it will be used when re-indexing the entire database and is shown in listing 8.6.

Listing 8.6: SearchIndexer::getDocument() retrieves the field information

```
class SearchIndexer
{
    // ...
    public static function getDocument($row)
    {
        if(method_exists($row, 'getSearchIndexFields')) {
            $fields = $row->getSearchIndexFields($row);
            $doc = new Places_Search_Lucene_Document(
                $fields['class'], $fields['key'],
                $fields['title'], $fields['contents'],
                $fields['summary'], $fields['createdBy'],
                $fields['dateCreated']);
            return $doc;
        }
        return false;
    }
    // ...
```

```
1 Does a data retrieval function exist?
2 Get the the data in search index format from the model
3 Create a new Zend_Search_Lucene_Document document
```

The SearchIndexer is only interested in models that need to be searched. As the observation system can be used for many observers, it is possible that some models may be sending out notifications that are not relevant to the SearchIndexer. To test for this we see if the model implements the function `getSearchIndexFields()` (#1). If it does, then we call the function (#2) to retrieve the data from the model in format that is suitable for our search index and then we create the document ready to be added to the index (#3).

Listing 8.7: SearchIndexer::_addToIndex() add to the search index

```
class SearchIndexer
{
    // ...
    protected static function _addToIndex($doc)
    {
        $dir = self::$_indexDirectory;
        $index = Zend_Search_Lucene::open($dir);

        $index->addDocument($doc);
        $index->commit();
    }
}
```

```
1 Open the search index
3. Add the document and store
```

Adding the document to the index is really easy. All we need to do is open the index using the directory that was set up in the bootstrap(#1) and then add the document (#2). Note that we need to commit the index to ensure that it is saved for searching on. This isn't needed if you add lots of documents as there is an automatic commit system that will sort it out for you.

One problem is that Zend_Search_Lucene doesn't handle updating a document. If you want to update a document, then you need to delete it and then re-add it. We don't want to ever end up with the same document in the index twice, and so we will create Places_Search_Lucene as an extension of Zend_Search_Lucene and override addDocument() to do a delete first if required. The code for Places_Search_Lucene is shown in Listing 8.8.

Listing 8.8: Deleting a document before adding it

```
class Places_Search_Lucene extends Zend_Search_Lucene
{
    public function addDocument(Zend_Search_Lucene_Document $document)
    {
        $docRef = $document->docRef;                                A

        $term = new Zend_Search_Lucene_Index_Term(
            $docRef, 'docRef');
        $query = new Zend_Search_Lucene_Search_Query_Term($term);
        $results = $this->find($query);                                B

        if(count($results) > 0) {                                     C
            foreach($results as $result)                             C
            {                                                         C
                $this->delete($result->id);                           C
            }                                                         C
        }                                                            C

        return parent::addDocument($document);                      D
    }
}
```

A docRef is unique in the index
 B Find the document
 C Delete current documents from index
 D Add new document to index

This function works very well and does exactly what we need, however, it doesn't get called until we change the static function as Zend_Search_Lucene::open() as this creates an instance of Zend_Search_Lucene, not Places_Search_Lucene. Therefore we need to override the open() and create() function too as shown in listing 8.9.

Listing 8.8: Overriding open() so that it all works

```
class Places_Search_Lucene extends Zend_Search_Lucene
{
    public static function create($directory)
    {
        return new Zend_Search_Lucene_Proxy(
            new Places_Search_Lucene($directory, true));            A
    }

    public static function open($directory)
    {
        return new Zend_Search_Lucene_Proxy(
            new Places_Search_Lucene($directory, false));
    }
}
```

```
// continue class...
```

A instantiate an instance of Places_Search_Lucene

The create() and open() functions in Listing 8.8 are very simple but also very necessary and now everything works as expected once we have updated SearchIndexer::_addToIndex() to reference Places_Search_Lucene. This is shown in Listing 8.9.

Listing 8.9: Correcting SearchIndexer::_addToIndex()

```
protected static function _addToIndex($doc)
{
    $dir = self::$_indexDirectory;
    $index = Places_Search_Lucene::open($dir);           1

    $index->addDocument($doc);                           2
    $index->commit();
}
```

1 open the index using Places_Search_Lucene

2. add the document removing duplicates

The function in listing 8.9 is the same as that in listing 8.7, with the exception of calling Places_Search_Lucene::open() (#1). This means that the call to addDocument() (#2) now calls our newly written function so that we can be sure that there are no duplicate pages in the search index. We now have a working system for adding updated content to our search index. We now need to implement the search system on the front end so that our users can find what they are looking for.

Re-indexing the entire site

Now we have the ability to update the index as new content is added, we can utilize all the code we have written to easily enable re-indexing all the data in the site. This is useful for supporting bulk inserting of data and also as a recovery strategy if the index is damaged or accidentally deleted. The easiest way to support re-indexing is to create a controller action, search/reindex which is shown in listing 8.10.

Listing 8.9: The re-indexing controller

```
public function reindexAction()
{
    $index = Places_Search_Lucene::create(                A
        SearchIndexer::getIndexDirectory());

    $places = new Places();                               B
    $allPlaces = $places->fetchAll();
    foreach($allPlaces as $place) {                       C
        $doc = SearchIndexer::getDocument($place);       C
        $index->addDocument($doc);                       C
    }
}
```

A Create a new index, wiping the old one

B Only the Places model has data to index

C Add each document in turn

The `reindex` action is very simple and uses `Zend_Search_Lucene`'s `create()` function to start a new index, effectively overwriting any current index. To add each document, we take advantage of the

SearchIndexer's `getDocument()` function that creates a document using the model's `getSearchIndexFields()` function, hence reusing the code and making this function very simple.

We now have complete control over creating search indexes within the Places website and can now look at creating a search form to allow the user to search and then view the results.

8.3.2 Creating the search form and displaying the results

The search form for Places is very simple as it consists only of a single input text field and a Go button. This form is available on every page and so the HTML for it is contained in the site template located in `application/views/scripts/site.phtml` as shown in listing 8.10.

Listing 8.10: A simple search form in HTML

```
<h3>Search:</h3>
<form method="get" action="<?php echo $this->baseUrl;?>/search">
    <input type="text" name="q" value=""> 1
    <input type="submit" name="search" value="Go"> 2
</form>
</div>
1 Text field for search terms
2 Go button to initiate search
```

The search form's action field is the index action in the search controller which is where the actual searching takes place. As Places follows the MVC pattern, the searching takes place in the `SearchController::indexAction()` function and the display of the search results is separated into the associated view file, `views/scripts/search/index.phtml`. Let's look at the controller first.

Processing a search request in the controller

This function performs the search and assigns the results to the view. It also validates and filters the user's input to ensure that we don't accidentally introduce cross-site security holes. The controller action is shown in Listing 8.11.

Listing 8.11: Filtering and Validation for the search form

```
public function indexAction()
{
    $this->view->title = 'Search Results';

    $filters = array('q' => array('StringTrim' , 'StripTags')); 1
    $validators = array('q' => array('presence' => 'required')); 2
    $input = new Zend_Filter_Input($filters, $validators, $_GET);
    if ($input->isValid()) { 3
        $this->view->messages = '';
        $q = $input->getEscaped('q');
        $this->view->q = $q;

        // do search
        $index = Places_Search_Lucene::open( 4
            SearchIndexer::getIndexDirectory()); 4

        $results = $index->find($q); 5
        $this->view->results = $results;

    } else {
        $this->view->messages = $input->getMessages(); 7
    }
}
```

In order to use data provided by a user, we first have to ensure that it is safe to do so. The `Zend_Filter_Input` component provides both filtering and validation functionality. We use filtering (#1) to remove any whitespace padding on the search term and also to remove any HTML with the `StripTags` filter. The only validation we do on the search term is to ensure that the user has provided it (#2) as searching for an empty string will not return useful results! `Zend_Filter_Input`'s `isValid()` function (#3) filters the data and then checks that the validation passes. On success, we collect the search query and reassign to the view to display.

Having checked that the data provided by the user is okay to use, we can now perform the search. As usual, with `Zend_Search_Lucene`, firstly we open the index (#4) and then call the `find()` function to do the work. In this case, we can use the built in string query parser as the user can provide a very simple search query (such as "zoo" to find all zoos, or a more complicated one such as "warwickshire -zoo" to find all attractions in Warwickshire except zoos.

If the validation fails, we collect the reason from `Zend_Filter_Input` using `getMessages()` (#7) and assign to the view. Now that we have either generated a result set or failed validation, we need to display this information to the user in the view.

Displaying the search results in the view

The view has two responsibilities: display any failure messages to the user and to display the search results. To display the error messages, we simply iterate over the list and echo within a list. This is shown in Listing 8.12.

Listing 8.12: Displaying error messages from `Zend_Filter_Input`

```
<?php if ($this->messages) : ?>
<div id="error">
    There was a problem:
    <ul>
        <?php foreach ($this->messages['q'] as $msg) : ?>
            <li><?php echo $msg; ?></li>
        <?php endforeach; ?>
    </ul>
</div>
```

A id is used to CSS styling

B Iterate over all messages for 'q'

This is very straightforward and the only thing to note is that we only iterate over the 'q' array within `messages` as we know that there is only one form field in this form. For a more complicated search form, we'd have to iterate over all the form fields. The second half of the view script displays the search results. The fields we have available are limited to those we set up in `Places_Search_Lucene_Document` back in listing 8.2 and this is shown in listing 8.13.

Listing 8.12: Displaying error messages from `Zend_Filter_Input`

```
<p>You searched for <?php echo $this->escape($this->q); ?>.
<?php echo count($this->results);?> results found.
</p>

<ul>
<?php foreach ($this->results as $result) : ?>
<li><a href="<?php echo $this->getSearchResultUrl(
    $result->class, $result->id); ?>">
    <?php echo $this->escape($result->title);?></a>
    <div class="summary">
        <?php echo $this->escape($result->summary);?>
        <!-- (<?php echo $result->score;?>) -->
    </div>
</li>
```



```
<?php endforeach; ?>
A escape() to handle special characters
B View helper to retrieve the URL
C Mainly for interest! The user doesn't care
```

As with any response to a user action, we provide important feedback about what the user searched for and how many results were found. We then iterate over the results array displaying each item's information within an unsigned list. The search results do not contain the url to the page containing the result, so we need to work it out from the `class` and `key` fields that we do have in the search index. This is outsourced to a view helper, `getSearchResultUrl()`, to keep the code contained which is shown in listing 8.13. The results are ordered by the `score` field which shows the weighting of each result. The user is not interested in this, but I was, so I included it as a comment so that I could inspect it using the view source command whilst investigating various search queries. Obviously, this can be removed from a production application.

Listing 8.12: Displaying error messages from Zend_Filter_Input

```
function getSearchResultUrl($class, $id)
{
    $baseUrl = $this->_view->baseUrl;
    $id = (int)$id;
    $class = strtolower($class);
    $url = $baseUrl . "/" . $class . "/index/id/" . $id;
    return $url;
}
A Ensure the parameters are "sane"
```

The initial version of `getSearchResultUrl()` is very simple as there is a one-to-one mapping from the model's class name to controller action. That is, for a model called `Places`, the controller used is `places/index`. It is likely that this would change as more models are introduced into the application. As this happens the complexity of mapping from model to the URL will increase and be completely contained within the view helper. This will help make long term maintenance that much easier.

8.5 Summary

This chapter has introduced one of the exceptional components of the Zend Framework. `Zend_Search_Lucene` is a very comprehensive full text search engine written entirely in PHP and easily enables a developer to add a search facility to a website. We have looked in detail at the way `Zend_Search_Lucene` works and how to queries can be either a simple string in the same manner as a Google search or programmatically constructed using the rich API provided to allow for very complex queries to be performed.

To put `Zend_Search_Lucene` in context, we have also integrated search into the *Places* community website. As a worked example, *Places* is relatively simplistic as it only has one model that requires indexing. However, we have future-proofed our code using the Observer pattern to separate the search indexing from the models where the data is stored. The result is a search engine that performs a ranked search algorithm which helps your users find the information they are looking for quickly with all the benefits that this brings to your website.

Email

While waiting to meet my kids off the school bus, I was chatting to a neighbour who commented that the kind of work I was doing was the "way of the future". As I pondered over that comment, I realised an irony in the fact that amongst the technologies of the future is one that, at around forty years old, is actually older than I am!

PHP has a reasonably diverse range of functions for dealing with mail, from its `mail()` function, which most all PHP programmers of any level are sure to be familiar with, to its IMAP, POP3 and NNTP functions. Unfortunately the latter need to be specifically compiled into the PHP installation which means they may not be available for some users. In contrast `Zend_Mail` is a fairly complete implementation of mail functionality that does not require any specific PHP configuration aside from the general requirements of the Zend Framework itself.

We will start the chapter with some general information about how email actually works before going into more detail about the construction of an email message with `Zend_Mail`. From there we will expand on your practical knowledge by building a simple support tracker.

9.1 Background on email

Whenever I need to describe technology to clients I start by looking for comparable real world examples and this is particularly appropriate when explaining email as it is actually modelled after the process of physical mail. In the next section we will give an overview of how email works and at the same time demonstrate the part `Zend_Mail` plays in the process.

9.1.1 Email simplified

Just as the complex routing process of physical mail can be reduced to simply dropping a letter in the mailbox and waiting for it to arrive in the letterbox of its addressee, so it is with email, albeit many times faster. In the following example, illustrated in figure 9.1, we will cover a fairly typical use of email in a web application where our user, Bert, sends a message inviting his friend, Ernie, to sign up on this fantastic new website he has just found.

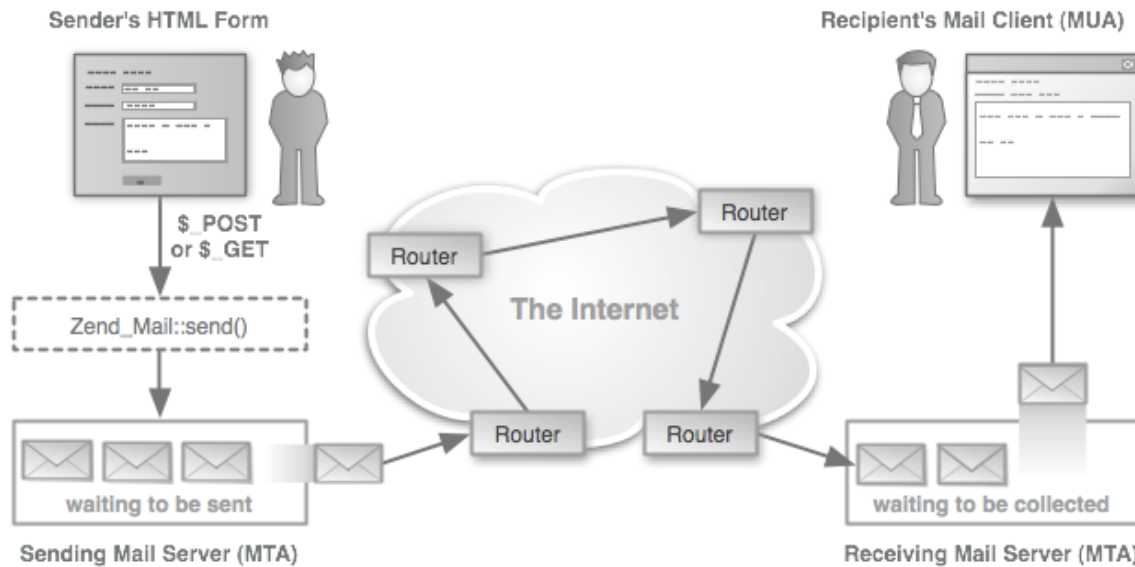


Figure 9.1 A simplified illustration of the processes involved in sending an email from an HTML form to its recipient's email client.

Bert starts by writing his message in the form and once satisfied, clicks "Submit" which sends the contents to the server in a POST, or occasionally a GET, HTTP request method. Zend_Mail then composes this information into email format and forwards it on to the mail transfer agent (MTA). Since Zend_Mail defaults to using Zend_Mail_Transport_Sendmail, which is a wrapper for PHP's own mail() function, the expected mail transfer agent is sendmail.

Having accepted the email, the mail transfer agent routes any local mail to local mailboxes (e.g. mail for domains with a DNS entry on the same machine) or, in the case of this email for Ernie, places it in its queue to be forwarded on. Once it clears the queue and is sent, the email will be bounced from server to server before finally landing in the receiving mail server where it awaits collection. From there all that is needed is for Ernie to hit "Get New Mail" and his mail client (Mail User Agent) will collect the mail using, in this instance, the Post Office Protocol (POP3).

9.1.2 Dissecting an email address

To go into a bit more detail about the routing of email we need to look at its key component, the email address. The simplest way to do this is to compare it to a physical address. Of course we're not going to attempt to go into any great detail about how real world postal addressing operates but when you look at table 9.1 it is much easier to see how the two compare. While Bert's physical location is indicated by an increasingly widening, or narrowing depending on how you read it, geographic description, his email address uses a series of suffixes to identify his network location.

Table 9.1 Comparing a physical address with an email address

Physical Address		Email Address	
Addressee Name:	Bert	Account Name:	bert@
Address	10 Some Street, Sydney, NSW 2000	Account Domain:	bertsblog
		Generic Top-Level Domain	.com
Country	Australia	Country Code Top-Level Domain	.au

Just as the responsibility for relaying physical mail lies with the sending parts of the various postal mechanisms and postal workers, so to do the sending mail transfer agents handle the transfer of email. By constant referrals to the Domain Name System (DNS) the mail transfer agents work backwards through the

email address until they arrive at the local domain. At each stage the actual path taken is determined by the availability of servers, successfully passing firewalls, spam and virus filters.

So far we've only gone through an overview of the workings of email and there are still more details we need to cover, but at this point it's a good time to start looking at the main reason we're here; Zend_Mail. As well as going into some of its features, from this point on we will focus on some of the details of email that Zend_Mail can take care of.

9.2 Introducing Zend_Mail

While the previous section portrayed Zend_Mail as having a fairly minor role in the overall process it is nonetheless a pivotal one. Not only can Zend_Mail be used to send mail via sendmail or SMTP, it can also compose them, add attachments, send HTML formatted email and even read mail messages.

9.2.1 Creating emails with Zend_Mail

The very least an email needs to have for Zend_Mail to send it is a sender, a recipient and a message body. In Listing 9.1 we can see how that bare minimum, plus a subject header, is perfectly adequate for Bert to get his invite sent to Ernie. All those familiar with PHP's own mail function will immediately appreciate the way Zend_Mail encapsulates the mail composition in a much more pleasing interface.

Listing 9.1 A simple Zend_Mail example

```
<?php
require_once 'Zend/Mail.php';
$mail = new Zend_Mail();
$mail->setFrom('welcome@greatnewsite.com', 'Support');
$mail->addTo('bert@bertsblog.com.au', 'Bert');
$mail->setSubject('An Invite to a great new site!');
$mail->setBodyText('Hi Bert, Here is an invitation to a great new website.');
```

Now that we're looking at the composition of our email the first thing worth mentioning is that the email produced by listing 9.1 is simple enough that it sits comfortably within the specification for email, otherwise known as RFC 2822. The specifications for email are handled by the official internet body; the Internet Engineering Task Force (IETF) and it is worth being a little familiar with some of this information as you will see reference to it elsewhere. For example the manual currently states that one of the validation classes "Zend_Validate_EmailAddress will match any valid email address according to RFC 2822".

Unfortunately RFC 2822 is a fairly limited specification and had Bert wanted to send an invite to his friends in non-English speaking countries he would have had problems if it wasn't for the Multipurpose Internet Mail Extensions (MIME). MIME is really just a collective name for a series of RFC (request for comment) documents that add additional functionality to the basic email specification such as character encoding, attachments, and more. Zend_Mime provides the MIME functionality in Zend_Mail and can also be used as a component on its own.

All our code in Listing 9.1 really does is to define a few headers followed by the body and then sends the email and in table 9.2 we compare the role that those headers and body have to a physical letter.

Table 9.2 Comparing a physical letter with an email

Physical Mail		Email	
Envelope	Recipient and address	Headers	Fields using the field name: field body syntax that include information about the email such as sender, recipient, and more.
Letter	The written contents	Body	The text of the message being sent

Just as you may want to add a picture of your kids in a physical letter, so can you add images as attachments with Zend_Mail by using:

```
$mail->createAttachment($pictureOfKids);
```

Having dropped in that important picture, you then write a note on the envelope with the officious looking warning "Photographs - Do NOT Bend" and with Zend_Mail we could theoretically do the same thing by adding additional headers:

```
$mail->addHeader('PhotoWarningNote', 'Photographs - Do NOT Bend');
```

Of course, just as the note on the envelope will likely be ignored and your picture will arrive complete with crease line, so too will that header be ignored as PhotoWarningNote is not a header recognised by any mail user agents!

Adding additional headers with PHP's mail() function is something a developer needs to take care with as if you are not careful it is possible for a malicious user to add headers such as further recipients. This is known as email header injection and you will be relieved to know that Zend_Mail does filter all headers to prevent such an attack.

Having gone into some detail about the creation of our email it is now time to look at the options available when sending it.

9.2.2 Sending emails with Zend_Mail

In Zend_Mail there are currently two ways to send mail; using Sendmail or SMTP. Why you would choose one over another and what options are available with each is the subject of this section.

Sending via Sendmail

It was mentioned earlier in this chapter that Zend_Mail defaults to using Zend_Mail_Transport_Sendmail which itself uses PHP's mail() function. What that means is that, unless you decide otherwise, your emails are simply being composed and passed on to PHP's mail() function and from there on to the local sendmail (or variation of) mailserver which handles the actual transfer of the mail.

To illustrate more clearly how this works we will make use of PHP mail()'s option of passing additional parameters in the command sent to the mailserver. In this case we will use it to set a header in the constructor of Zend_Mail_Transport_Sendmail like so:

```
$transportWithHeader = new Zend_Mail_Transport_Sendmail(
    '-fwelcome@greatnewsite.com'
);
Zend_Mail::setDefaultTransport($transportWithHeader);
```

What that does is pass on '-fwelcome@greatnewsite.com' as the fourth parameter in the mail() function which then sends it in the following command which sets the sender address of the email:

```
sendmail -f welcome@greatnewsite.com
```

Since all we are doing is sending a command, this method should be fast and incur little latency, however, that is dependant on the setup of the machines on which it is being called and run from. Also, as this is a *nix command, it is not even an option on Windows based servers which will default to using SMTP anyway.

Sending via SMTP

There are occasions when you do not want to burden the local mailserver with sending your mail. A good example would be when sending large volumes of mail in an environment where there may be restrictions on the amount of mail your local mailserver can send. In this case it is possible to pass on the mail via SMTP to another service provider.

Because SMTP (Simple Mail Transfer Protocol) is the standard by which all mail is sent across the internet even if we use sendmail our email is ultimately sent using SMTP. When we say "sending via SMTP"

in this case we are meaning, more specifically, sending through a specified outgoing SMTP server in much the same way that we set-up our email clients to send our mail. In Listing 9.2 we are setting up Zend_Mail to send via SMTP using the authentication required by the service provider and through a secure connection.

Listing 9.2 Setting up Zend_Mail to use an SMTP connection

```
<?php
require_once 'Zend/Mail/Transport/Smtp.php';
$authDetails = array(
    'ssl' => 'tls',
    'port' => 25,
    'auth' => 'login',
    'username' => 'myusername',
    'password' => 'mypassword'
);
$transport = new Zend_Mail_Transport_Smtp(
    'mail.our-smtp-server.com', $authDetails
);
Zend_Mail::setDefaultTransport($transport);
```

A
A
B
B
B
C
C
D

A Optional settings for the secure transport layer

B Optional Authentication details for the SMTP server

C Server and authentication details passed to the constructor

D SMTP is set as the default transport method

Having setup Zend_Mail to use the SMTP connection we are ready to use it and since we mentioned sending large volumes of email we'll start with sending multiple emails.

Sending multiple emails via SMTP

There are occasions when we may need to send several emails out in one go, such as for example when sending a newsletter to multiple users. However, in the PHP manual the mail() function has the following note:

It is worth noting that the mail() function is not suitable for larger volumes of email in a loop. This function opens and closes an SMTP socket for each email, which is not very efficient.

In contrast, when using Zend_Mail_Transport_Smtp, the SMTP connection is maintained by default until the object stops being used. Therefore it is more suitable than mail() for sending larger volumes of email such as in the following example:

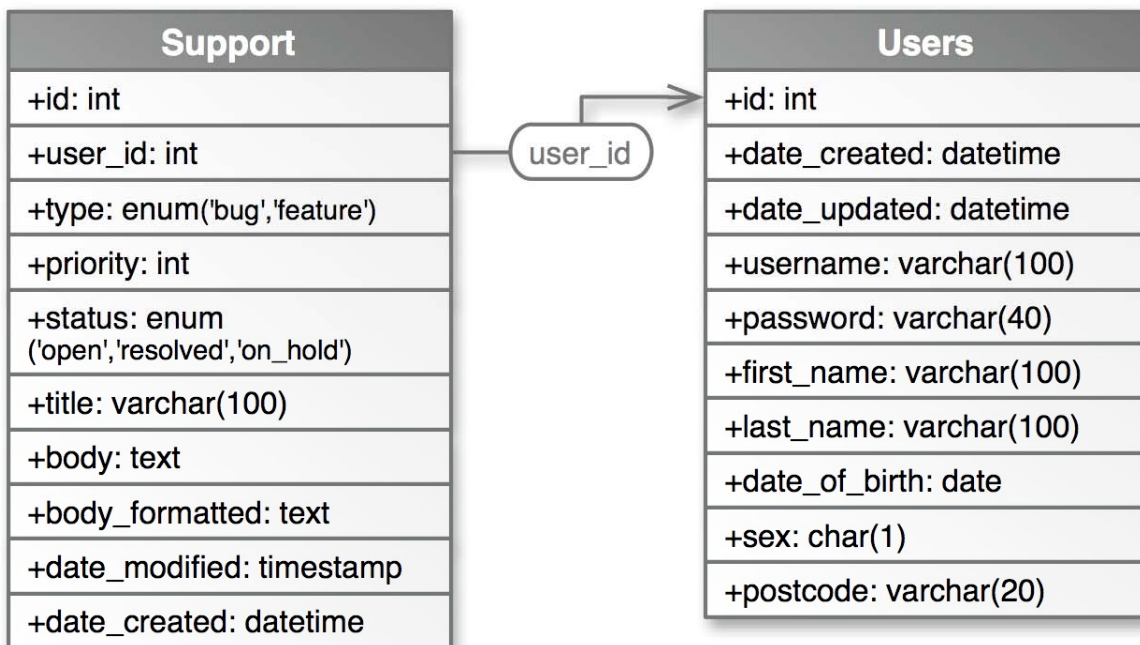
```
foreach ($users as $user) {
    $mail->addTo($user->email, $user->full_name);
    $mail->setBodyText('Hi ' . $user->first_name . ',
        Welcome to our first newsletter.');
```

A

In case you're wondering why anyone would want to send such an uninteresting newsletter, you're probably ready for the next section where we will begin to use Zend_Mail in an actual application and make better use of the email body.

9.3 Building a support tracker for Places

The more you work on a project the more you build up lists of bugs, issues, feature ideas. On top of that these lists end up all over the place; spread across emails, jotted down after phone conversations, frantic notes from a meeting that are almost illegible. There are existing options available like bug trackers and project management applications but they are often overkill for clients use and besides that would be too easy! Instead we are going to build a solution and since the key to good support is communication our support tracker is sure to be doing a lot of mailing.



9.3.1 Designing the application

After a brainstorming session we decide that our support tracker has the following requirements:

1. Simple enough that we'll want to use it rather than just email.
2. Update-able, e.g. changing the status of a bug.
3. Integrates with current user data. No new usernames and passwords.
4. Notification by email to all concerned.
5. Ability to add attachments, e.g. screenshots, and have them sent with notification emails.
6. Formatted emails for quick scanning.

The second requirement makes it clear that we are going to need to store this data and figure 9.2 shows the initial table structure of our application. Since we will have many support issues to a user this is reflected in the one-to-many relationship of the schema.

Figure 9.2 The initial database structure of our support tracker requiring the addition of a single “support” table alongside our existing “users” table

If you've read chapter 5 you should already be familiar with the use of Zend_Db_Table relationships and recognise the model that follows in listing 9.4. By specifying the `$_dependentTables` and `$_referenceMap` we are defining the relationship between the existing Users class from our *Places* application and our new Support class.

Listing 9.4: One-to-many relationship using Zend_Db_Table

```

class Users extends Zend_Db_Table_Abstract
{
    protected $_name = 'users';
    protected $_dependentTables = array('Support');
}

class Support extends Zend_Db_Table_Abstract
{
    protected $_name = 'support';
    protected $_rowClass = 'SupportTicket';
    protected $_referenceMap = array(
        'Support' => array(
            'columns' => array('user_id'),

```

```

        'refTableClass' => 'Users',
        'refColumns'    => array('id')
    );
}

```

Linking these two classes fulfills the third requirement of having the current *Places* user data integrated with the support tracker. Users will be able to submit support tickets without having to login to a separate system.

In order to add support tickets the least we are going to need is a submission form and the functionality to create the entry in the databases. Listing 9.6 reflects these needs with an `editAction` and `createAction` in our `SupportController` action controller class.

Listing 9.5: Our `SupportController` class

```

class SupportController extends Zend_Controller_Action
{
    public function editAction() {}

    public function createAction()
    {
        $filterInt = new Zend_Filter_Int;
        $filterStripTags = new Zend_Filter_StripTags;
        $filterFormat = new Zend_Filter;
        $filterFormat->addFilter(new Zend_Filter_StripTags)
            ->addFilter(new Places_Filter_Markdown);

        $support = new Support;
        $newSupport = $support->createNew();

        $newSupport->user_id = Zend_Auth::getInstance()
            ->getIdentity()->id;
        $newSupport->type = $filterStripTags->filter(
            $this->_getParam('type'));
        $newSupport->priority = $filterInt->filter(
            $this->_getParam('priority'));
        $newSupport->status = $filterStripTags->filter(
            $this->_getParam('status'));
        $newSupport->title = $filterStripTags->filter(
            $this->_getParam('title'));
        $newSupport->date_created = date('Y-m-d H:i:s');
        $newSupport->body = $filterStripTags->filter(
            $this->_getParam('body'));
        $newSupport->body_formatted = $filterFormat->filter(
            $this->_getParam('body'));

        $id = $newSupport->save();

        $this->_forward('form', 'support', null, array('id' => $id));
    }
}

```

A The action method for the support ticket submission form

B Instantiating the objects used to filter submitted data

C A custom filter that converts plain text to HTML using Markdown

D Create a new non-database row

E Get the current user id from `Zend_Auth`

F Calling the `save()` method here will trigger the email being sent

G Forward to the ticket submission form

The `editAction()` is empty as it currently doesn't require any more than having the view rendered which is taken care of by the `ViewRenderer` action helper. Of course it does require a view script in `views/scripts/edit/index.phtml` which we can see rendered in figure 9.3.

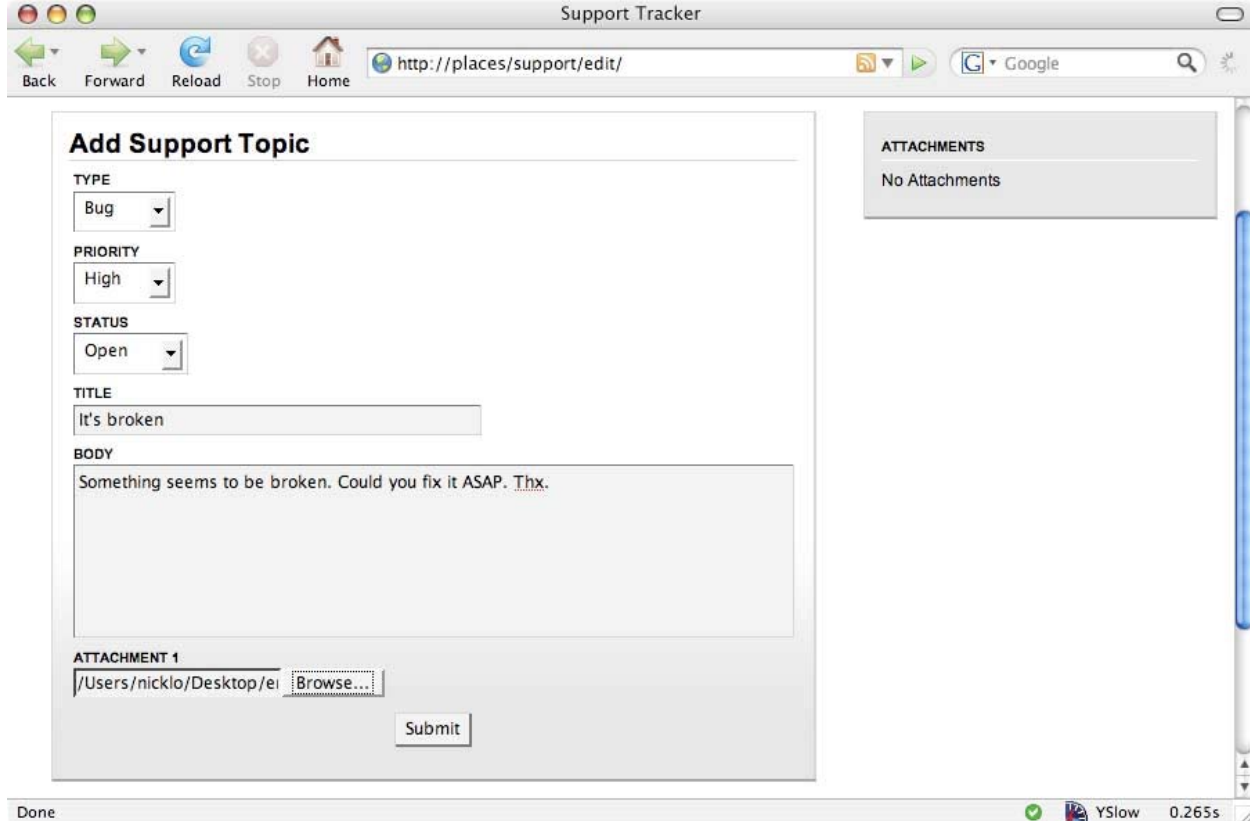


Figure 9.3 The support tracker submission form used to send bug or feature requests to the development team.

Having done that we can then move on to the fourth requirement which is to notify support team members when support tickets are added and updated. Conveniently, this also brings us back on to the topic of this chapter; email.

9.3.2 Integrating Zend_Mail into the application

You'll probably have noticed that there is no mail functionality in our SupportController and all createAction() does is filter the input and create the database row. Rather than add unnecessary bulk to the controller action method it makes more sense to attach the email notification to the save() process which is ultimately handled by Zend_Db_Table_Row::save(). Since Zend_Db_Table_Row::save() can be used for updating existing rows as well as creating new ones it is a good point to trigger the mail to the support team members.

Another thing you may have noticed was the line in our Support Zend_Db_Table subclass that set the row class to use:

```
protected $_rowClass = 'SupportTicket';
```

In listing 9.6 you can see the contents of that row class and how our save() method overrides the parent class save() method and adds the notification email to the support team.

Listing 9.6: Overriding save() in our SupportTicket subclass to trigger email

```
class SupportTicket extends Zend_Db_Table_Row_Abstract
{
    public function save()
    {
        parent::save();
        $mail = new Zend_Mail();
        $mail->setBodyText($this->body);
        $mail->setFrom('system@example.com', 'The System');
        $mail->addTo('support@example.com', 'Support Team');
        $mail->setSubject(strtoupper($this->type) . ': ' . $this->title);
        $mail->send();
    }
}
```

A Call the parent class save() method

B Instantiate Zend_Mail, set-up headers and send

Now, whenever a call is made to the `save()` method, an email will be sent to the support team with the support ticket information in the body of the email.

It isn't hard to see however that the code is far from optimal and later when, for example, we want to send a copy to the submitter, add an attachment, format the body, our `save()` method will have bloated beyond its purpose. With that in mind let's refactor out the mailing code from the `save()` method and create a `SupportMailer` class which can focus on that task.

Listing 9.6: Overriding `save()` in our `SupportTicket` subclass to trigger email

```
class SupportMailer
{
    public function send(Zend_Db_Table_Row_Abstract $supportTicket)    A
    {
        $mail = new Zend_Mail();
        $mail->setBodyText($supportTicket->body);
        $mail->setFrom('system@example.com', 'The System');
        $mail->addTo('support@example.com', 'Support Team');
        $mail->setSubject(
            strtoupper($supportTicket->type) . ': '
            . $supportTicket->title);
        $mail->send();
    }
}
```

A The `supportTicket` is passed to the `send` function

The `SupportTicket::save()` method in listing 9.6 can now call our new mailing class and pass itself as a parameter like so:

```
$id = parent::save();
SupportMailer::send($this);
```

With that done we can add some more of the functionality to the mail class itself.

9.3.3 Adding headers to the support email

The fourth requirement in our application specification was that an email be sent to “all concerned”. It turns out this does not mean just the support team at `support@example.com`. In fact what is needed is that all admin users of the system are to be emailed and the submitter is to be cc'd a copy as well. We also get the additional request to add a priority indication that email clients like Outlook and Apple's Mail can recognise. Luckily both of these are relatively easy and simply require working with email headers.

Adding recipients

For the sake of simplicity in this case our application is storing the role of users as a single field in the `Users` table so to retrieve all admin users we make the following query:

```
$users = new Users;
$where = array(
    'role = ?' => Zend_Auth::getInstance()->getIdentity()->id,
    'user_id != ?' => 'admin'
);
$rows = $users->fetchAll($where);
```

You'll notice the additional term in the argument to `fetchAll()` to filter out cases where if, as is quite likely, the submitter is also an admin user. This prevents them receiving an additional and unnecessary admin email on top of the CC (carbon copy) version they'd receive as a submitter. In some cases you may actually prefer to send a different version to the submitter from the one sent to the support team so this is largely an implementation preference.

Having retrieved all admin users we can now loop over them and add them to the mail with a “To” header like so:

```
foreach ($rows as $adminUser) {
    $mail->addTo($adminUser->email, $adminUser->name);
}
```

Next we'll retrieve the email details for the submitter using their id from Zend_Auth and add them to the mail with a CC header:

```
$users->find(Zend_Auth::getInstance()->getIdentity()->id);
$submitter = $users->current();
$mail->addCC($submitter->email, $submitter->name);
```

We also have the option of adding users with a BCC (blind carbon copy) header using Zend_Mail::addBcc() but don't need it in this case.

Adding additional headers

The addHeader() method can be used to set additional headers with its first two arguments being the name and value pair of the header and a third boolean argument to indicate whether there will be multiple values for the header. The requested priority indication is an additional header that appears in the email as a name:value pair like so:

```
X-Priority: 2
```

We were clever enough to have preempted this requirement and designed our database field to contain an integer to correspond with the value of the priority. Adding this value as a header is now simple enough:

```
$mail->addHeader('X-Priority', $supportIssue->current()->priority, false);
```

The arguments now specify that we are adding a header with the name 'X-Priority', the priority value chosen for the current support ticket and that it is just a single value. The received email should then have an "at-a-glance" indication of its priority depending on whether the email client used recognises the priority header or not.

9.3.4 Adding attachments to the support email

It would be a safe bet to say that there won't be a single developer reading this book that hasn't had bug reports with descriptions as vague as "it's broken" or "it's not working" and narrowing down the actual problem can occasionally be more frustrating than actually fixing it. In such cases a screenshot of the offending page can be a great help and that is the reason why adding attachments to the support email is the fifth requirement of our application.

We have already added a file input field to the support ticket submission form which the user will use to browse for their screenshot file and have it uploaded on submission of the form. The process of actually dealing with the uploaded file isn't something we can do justice to in this chapter but let us imagine it is something along the lines of:

```
move_uploaded_file(
    $_FILES['attachment']['tmp_name'],
    realpath(getcwd()) . '/support/' . $id . '/'
);
```

This moves the uploaded file, which we will call error-screenshot.jpg, to a sub-directory of /support/ using the support ticket id as its directory name.

Now that we have the file in place we can attach it to the email with the code shown in listing 9.7

Listing 9.7: Attaching the screenshot file to the support email

```
$file = "/home/places/public_html/support/67/error-screenshot.jpg";      A
$fileBody = file_get_contents($file);                                     A
$fileName = basename($file);                                             A
$fileType = mime_content_type($file);                                     A

$at = $mail->addAttachment($fileBody);                                    B
$at->filename = $fileName;                                                C
$at->type = $fileType;                                                    C
```

```
$at->disposition = Zend_Mime::DISPOSITION_INLINE;           C
$at->encoding     = Zend_Mime::ENCODING_BASE64;             C
A Getting the general information about the file
B Adding the attachment to the email
C Optional settings for the attachment
```

The optional settings in listing 9.7 are only really needed if your attachment deviates from the default which is a binary object transferred with base64 encoding, and handled as an attachment. In our case we've specified that the attachment be displayed inline in the email just so we don't have to open the file separately to view it. As mentioned early in the chapter settings like this are handled by Zend_Mime so you will need to look at its section of the Zend Framework manual for more information.

With the fifth requirement taken care of that leaves the sixth and final requirement which is to format the emails so they can be quickly and easily read by the busy admin team.

9.3.5 Formatting the email

The subject of whether to send text or HTML email is a contentious issue that is debated at almost every opportunity. Whatever your personal preference, as a developer you still need to know how to send either one, though ironically, Zend_Mail makes the process easier than the choice.

Sending HTML formatted mail

As part of the input filtering in the createAction() method of our SupportController action, the body text is formatted as HTML using the PHP Markdown conversion tool written by Michel Fortin and based on John Gruber's original code. The formatted body is then saved in a separate field alongside the original giving us two versions of the body to use in our email.

First we have the original body that we have been using up to now as the plain text version:

```
$mail->setBodyText($supportTicket->body);
```

Now we can use the formatted body as the HTML version of our email:

```
$mail->setBodyHtml($supportTicket->body_formatted);
```

And that is it! Well actually not quite. It's important to remember that if you are going to be sending an HTML version of your email that you also send a plain text version for those who cannot or do not want to view the HTML version.

Formatting with Zend_View

We now know how to send pre-formatted mail in plain text or HTML versions, but what if we want to do some formatting to the email body before it is sent? Earlier, while adding recipients, we mentioned that you may prefer to send a separate email to the support ticket submitter from the one sent to the support team. This submitter email is the example we will use to demonstrate how Zend_View can be used to render an email from the same kind of view script as used in our HTML pages.

What we have decided to do is send an email to the submitter that notifies them of the support ticket number, includes the body of their submission, gives a brief description of what will happen next and then thanks them. Listing 9.8 shows the plain text version.

Listing 9.8: The plain text version of the support ticket submitter email in text-email.phtml

```
Hello <?php echo $user->name; ?>,
```

```
Your support ticket number is <?php echo $supportTicket->id; ?>
```

```
Your message was:
```

```
<?php echo $supportTicket->body; ?>
```

```
We will attend to this issue as soon as possible and if we have any further questions will
contact you. You will be sent a notification email when this issue is updated.
```

Thanks for helping us improve Places,

The Places Support team.

Unless you've jumped straight into this chapter, that should look very familiar by now as it is much the same as the view scripts you'll have seen in previous chapters, although the HTML version in listing 9.10 is probably even more so.

Listing 9.10: The HTML version of the support ticket submitter email in html-email.phtml

```
<p>Hello <?php echo $user->name; ?>,</p>

<p>Your support ticket number is <b><?php echo $supportTicket->id; ?></b></p>

<p>Your message was:</p>

<?php echo $supportTicket->body_formatted; ?>

<p>We will attend to this issue as soon as possible and if we have any further questions will
contact you. You will be sent a notification email when this issue is updated.</p>

<p>Thanks for helping us improve Places,</p>

<p><b>The Places Support team.</b></p>
```

All that is needed to turn those view scripts into something we can use in our email is to have Zend_View render them like so:

```
$mail->setBodyText($this->view->render('support/text-email.phtml'));
$mail->setBodyHtml($this->view->render('support/html-email.phtml'));
```

The following figure 9.4 shows the output of the above in the resulting emails.

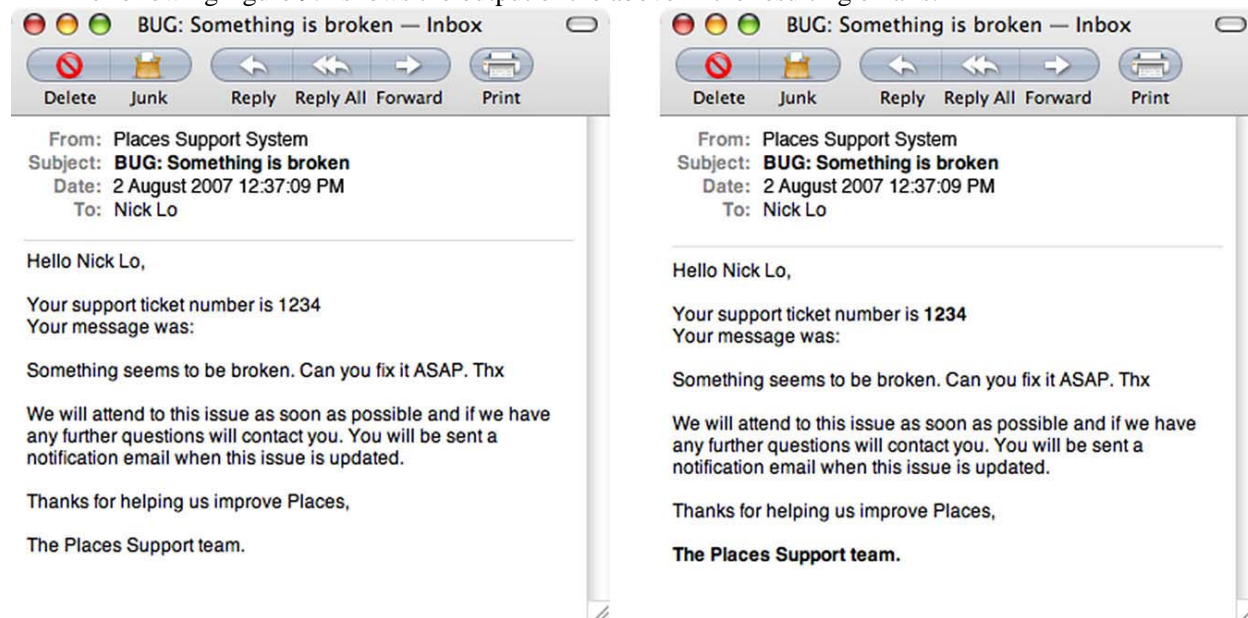


Figure 9.4 Side-by-side comparison of the output of our plain text (left) and html (right) support ticket notification emails.

Clearly this is incredibly useful and simple, largely because it is the same process used for any view script making it flexible enough to be used for anything from a lost password email to a more involved HTML newsletter.

Having completed the final requirement of our support tracker we have now also covered the composition and sending features of Zend_Mail. This leaves us with the final link in the email chain; reading mail and Zend_Mail has thoughtfully provided functionality to do that.

9.4 Reading email

Our support tracker now has a web form as a means for issues to be recorded and the support team notified, however it is very likely that issues will be coming from other sources such as sent or forwarded in an email. In order to deal with this we are going to take advantage of Zend_Mail's ability to read mail messages by monitoring a support email account and add issues to the support tracker.

Before we look at the requirements for this new feature in our application let's take a short look at some of the components of collecting and storing email.

9.4.1 Email collection and storage

In our earlier background on email we noted that Ernie collected his invite from Bert using POP3 however he could have just as likely used the other main email collection protocol IMAP (Internet Message Access Protocol). Since Zend_Mail can work with both we'll take a look at the differences between the two.

POP3 and IMAP

Our two characters Bert and Ernie both make use of these two protocols to access their mail and in figure 9.5 we can see a the basic difference between Bert's use of IMAP to get his mail and Ernie's use of POP3. As Ernie works out of a home office and rarely needs to access his mail from anywhere else, he uses POP3 while Bert, who is always on the move, relies on his mail to be available wherever he is, so he uses IMAP.

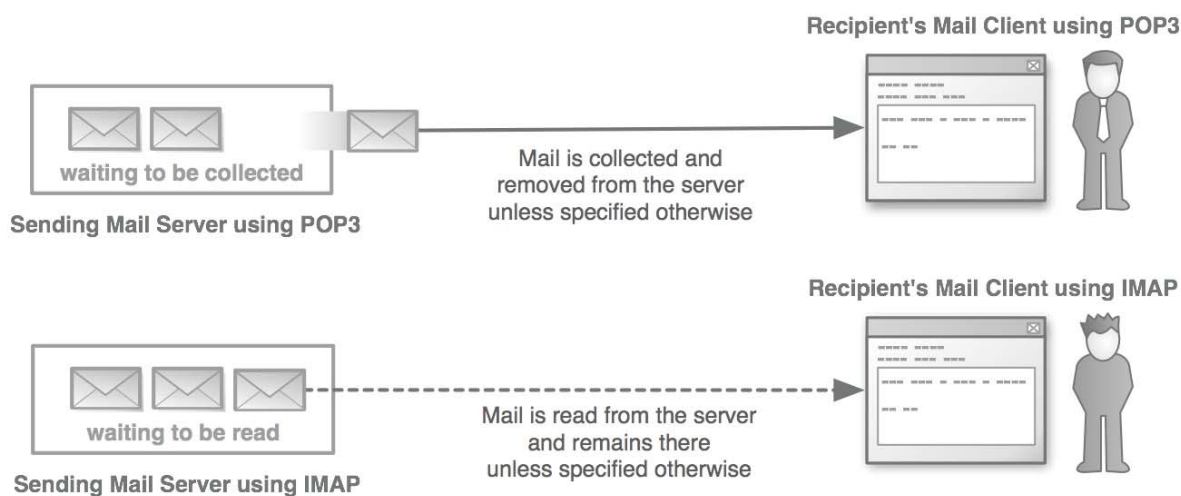


Figure 9.5 Comparing the main difference in email collection between POP3, which works on a simple connect, collect, disconnect relationship, and the more capable IMAP.

POP3 could be seen as the older and dumber of the two protocols and works on a connect, collect, disconnect relationship with the mail user agent (email client) having most of the responsibility. On the other hand the fact that it is so simple, widely supported and makes minimal use of server resources is the reason it is so widely used.

IMAP, in comparison, is a more capable protocol and has more responsibility in the relationship with the email client. Mail is kept on the server and the email client just requests a copy which it will cache locally. Among some of the features that IMAP has over POP3 include; it allows an ongoing connection with the server, multiple client access to the same mailbox, email folders, server-side searches, partial mail retrieval and more.

If you, like Bert, were needing to access your mail from multiple locations and/or from multiple devices such as a PDA, mobile phone, laptop then IMAP would be a better solution because each device would simply be accessing the mail rather than collecting it.

Email storage

In contrast to the official internet standards we have described so far, such as the email format outlined in RFC 2822, MIME, SMTP, POP3 and IMAP, the file format for email storage is not standardised. Instead it is a decision left to the developers of email clients.

Zend_Mail currently supports the Mbox and Maildir formats and the key difference between the two is the former requires application-level [file locking](#) to maintain the integrity of the message. Aside from that there isn't much benefit going into great detail about the formats which means we can now get back to working on our application.

9.4.2 Reading email with our application

Just as we did with the earlier stages in the development of our support tracker, we will again jot down a few requirements for this additional work before we start. This list makes it easier to see exactly what we are trying to achieve and how Zend_Mail helps us to accomplish it. So far the requirements we have put together are:

3. Periodically read mail from a designated mail account and store it in the support table.
4. Include some details about the reporter.
5. Save the original email as a file attachment.

Opening a connection

Before we can do anything with the stored mail the first thing we need to do is connect to it. For our support tracker it's probably likely that we'd use a local storage location and access it with either Zend_Mail_Storage_Maildir or as in the following example with Zend_Mail_Storage_Mbox:

```
$mail = new Zend_Mail_Storage_Mbox(
    array('filename' => '/home/places/mail/inbox')
);
```

If we needed to access a remote storage location we'd do so using Zend_Mail_Storage_Imap or Zend_Mail_Storage_Pop3. We'll chose the latter as it's the one most readers will have access to and can start using right away with minimal setup. In our case we'll connect using:

```
$mail = new Zend_Mail_Storage_Pop3(array('host'      => 'example.com'
                                         'user'       => 'support',
                                         'password'  => 'support123'));
```

Our mail object now has a connection open and we can start fetching messages.

Fetching and storing the support messages

Because Zend_Mail_Storage_Abstract implements one of the standard PHP library iterator classes it can be iterated over easily so let's start with the first requirement of our application and look at storing the messages in our support table. Listing 9.11 shows the first implementation without filtering.

Listing 9.11: Turning the support email into a row in the support table

```
foreach ($mail as $messageNum => $message) {
    $newSupport = $this->_supportTable->createRow();

    $newSupport->priority = isset($message->xPriority) ?           A
                          $message->xPriority : 3;                A
    $newSupport->status = 'open';
    $newSupport->title = $message->subject;                        B
}
```

```

    $newSupport->date_created = date('Y-m-d H:i:s');
    $newSupport->body = $message->getContent();
    $id = $newSupport->save();
}

```

A Using the x-priority header as the priority setting
 B Using the email subject as the issue title
 C Using the email body as the issue body

Combine the above with some way to run it periodically, such as with cron, and we've pretty well covered the first requirement. There are a few points to mention however, the first being that the we've not done anything about the body_formatted field of our support table.

When we entered the body information with the web form it was put through the text to HTML markup filter Markdown. In this case however the body of the email could be plain text or HTML or both, which is referred to as multipart. If we enter an HTML or multipart email body directly into the body field it is going to be a mess so in listing 9.12 we will check for that and reduce the body down to just the plain text content.

Listing 9.12: Reducing a multipart email down to its plain text component

```

$part = $message;
while ($part->isMultipart()) {
    $part = $message->getPart(1);
}

if (strtok($part->contentType, ';') == 'text/plain') {
    $plainTextContent = $part->getContent();
}

```

A
 B
 C
 D

A This loops as long as the Content-Type header of \$part contains the text multipart/
 B Reassigning the first part of the multipart message to the \$part variable
 C The first part is usually plain text but we'll do a check anyway
 D Assigning the plain text content to a variable

In theory we could also insert any HTML part into the body_formatted field of our support table, however, we'd then need to do some careful filtering of the data and even after doing so we'd then have to deal with all the varieties of HTML markup from the various email clients. Instead, let's just use the text to html markup tool Markdown as we previously did just to give it some basic but clean formatting. This leaves us with the following adjustment and addition to the code in listing 9.11:

```

$newSupport->body = $plainTextContent;
$newSupport->body_formatted = $filterFormat->filter($plainTextContent);

```

As it is quite possible that even if we store a well formatted version of the support email body text that the issue itself may still not provide us with all the information we need. For that reason we will also store the sender of the email so we can contact them for any further clarification.

In Figure 9.6 we've added a "reported_by" field in our support table to hold this information and adding the sender to the code in listing 9.11 is accomplished with this addition:

```

$newSupport->reported_by = $message->from;

```

We could spend a bit more time breaking up that sender string into its sender's name and email address if we needed to but for now it's sufficient for our needs.

Support
+id: int
+user_id: int
+type: enum('bug','feature')
+priority: int
+status: enum('open','resolved','on_hold')
+title: varchar(255)
+body: text
+body_formatted: text
+date_modified: timestamp
+date_created: datetime
+reported_by: varchar(255)

Figure 9.6 The support table to which the `reported_by` field has been added, allowing us to record the details of the sender of the support email.

Our support table can now have tickets inserted that have been sent via email, however, as one last measure we're going to save the email as a file attachment just in case we need to refer to it for whatever reason.

Saving the fetched mail to file

As we loop through the support mail messages in listing 9.11 we have all the information we need in the `$message` variable to write out to a file. Listing 9.13 shows how we'll fulfil the third requirement by storing it as the original email.

Listing 9.13: Writing the message out to a file

```
$messageString = '';
foreach ($message->getHeaders() as $name => $value) {
    $messageString .= $name . ': ' . $value . "\n";
}
$messageString .= "\n" . $message->getContent();
file_put_contents(getcwd() . '/support/' . $id . '/email.txt',
    $messageString
);
```

A Loop over the headers and append to `$messageString`

B Append the message content to `$messageString`

C Store in file in a directory named with the support ticket id

The next point is a security one; the email content could just as easily contain malicious code as any form submission could so before we insert it into the database we will need to add some filtering.

Filtering the email content

Just to wrap up properly in listing 9.14 we'll bring together all the code we've done so far and add in some basic data filtering using `Zend_Filter`.

Listing 9.14: Adding filtering to our support email processing code

```
$filterStripTags = new Zend_Filter_StripTags;
$filterFormat = new Zend_Filter;
$filterFormat->addFilter(new Zend_Filter_StripTags)
    ->addFilter(new Places_Filter_Markdown);
```

```

foreach ($mail as $messageNum => $message) {
    $part = $message;
    while ($part->isMultipart()) {
        $part = $message->getPart(1);
    }

    if (strtok($part->contentType, ';') == 'text/plain') {
        $plainTextContent = $filterStripTags->filter(
            $part->getContent());
    }

    $newSupport = $this->_supportTable->createRow();

    $newSupport->priority = isset($message->xPriority) ?
        $message->xPriority : 3;
    $newSupport->status = 'open';
    $newSupport->title = $filterStripTags->filter(
        $message->subject);
    $newSupport->date_created = date('Y-m-d H:i:s');
    $newSupport->body = $plainTextContent;
    $newSupport->body_formatted = $filterFormat->filter(
        $plainTextContent);
    $newSupport->reported_by = $filterStripTags->filter(
        $message->from);
    $id = $newSupport->save();

    $messageString = '';
    foreach ($message->getHeaders() as $name => $value) {
        $messageString .= $name . ': ' . $value . "\n";
    }
    $messageString .= "\n" . $message->getContent();
    file_put_contents(getcwd() . '/support/' . $id . '/email.txt',
        $messageString
    );
}

```

Having satisfied all three requirements of our support tracker feature addition we have now also given a short practical demonstration of some of Zend_Mail's read functionality.

9.5 Summary

Having gone through this chapter you may well have discovered that email is a lot deeper subject than you had anticipated. Since almost all web applications use email to one degree or another we've attempted to give enough background to the technicalities of email so that you will come away with not only a better understanding of Zend_Mail itself, but of how it fits into the bigger picture.

Adding Zend_Mail to the components we've covered in previous chapters also provides us with a good basic toolset with which to build web applications. Before we continue adding to our arsenal however, we will take a detour into some practices that will improve the way we develop such applications in our next chapter on deployment.

Deployment

Along with the ability to develop web applications rapidly using PHP and the Zend Framework, lurks the temptation to cut corners while developing. Many readers will have experienced that sudden cold sweat after having accidentally deleted an important part of some code and frantically trying to fix a live site before anyone notices! In this chapter we'll run through some development and deployment methods that can be used to ensure quality work without curbing enthusiasm in the process. By providing a kind of safety net the following practices aim to allow more freedom and confidence in the way you approach your work.

10.1 Server setup

We will kick off this chapter by looking at the setup of the environment under which development will progress to the final deployment. Just as the rehearsal stages of any theatrical performance are intended to prevent problems occurring in the final live production, so can we follow the same careful preparation through the setup of our server environment.

In table 10.1 we outline the setup of the typical development, staging and production environment, which it's worth noting, may well be on the same physical machine or spread across several. For single developers and very small teams using a development server to make local changes before committing to a production server would be a minimal set-up. Changes made and tested on the development server are only uploaded to the production server when all tests have been passed. Larger teams needing to co-ordinate local changes by several developers before moving to a the live production server would likely require a staging server.

Table 10.1 The stages of a typical development environment.

Development	This is the rehearsal stage in which the application is developed, changes are made, tested and the final production is shaped. Depending on the preferences of the individual developers this may include one or several machines. Different configurations of development machines may also have the advantage of showing up bugs that may otherwise have been missed.
Staging	Like a dress rehearsal, this stage should mimic the final production as closely as possible. No development takes place on this server so any changes still need to be done on the development server and moved across to staging. This version is accessible only by the development team and anyone else concerned with moderating the release of the application.
Production	This is the live public performance, accessible via the internet. No development takes place on this server as it would not only potentially break a live site but also the chain of quality control.

It's worth stressing that what is "best practice" is dependent on the requirements of the team and the project. Each stage can add extra administration that could stretch the resources of small teams and add to the complexity of low budget projects. Blindly following suggestions of the ideal development environment will be of little consolation when a project is delayed due to internal reasons. Keeping that in mind we'll work through an example based on how we worked with some of the code for this book and urge you to take from it what you see as relevant to your particular needs.

10.1.1 Designing for different environments

One of the goals for any kind of application is the ability to move between different environments and work with minimal reconfiguration. Other than requiring PHP 5.1.4 or greater the Zend Framework itself has few specific requirements so it is largely a matter of careful design as to how much we need to vary our own application.

Most of the differences between one application and another occur during initialisation where the configuration settings are read in. In a Zend Framework application this most commonly occurs in the “bootstrap” file. We are actually going to separate the bootstrapping into its own class to which we can pass the deployment environment as a parameter from the index.php file in the web root as shown in listing 10.1.

Listing 10.1 The contents of our index.php file which will live in the web root.

```
<?php

set_include_path(get_include_path() . PATH_SEPARATOR
    . '/path/to/zf-working-copy/trunk/incubator/library/' );      A
set_include_path(get_include_path() . PATH_SEPARATOR
    . '/path/to/zf-working-copy/trunk/library/' );                A
include '../application/bootstrap.php';                           B
$bootstrap = new Bootstrap('nick-dev');                           C
$bootstrap->run();
```

A Set paths specific to the environment under which the application is running

B Include our bootstrap file and instantiate it with the configuration section to use

C Run the application

You'll notice that an additional benefit of this is that it allows us to set any environment specific paths, such as in this case to a local working copy of the latest Zend Framework core components as well as the “incubator” components.

In listing 10.1 we specified that the configuration section to use is 'nick-dev' by passing it as a parameter to the Bootstrap constructor before the call to run. Listing 10.2 shows a stripped down version of bootstrap.php to highlight how the deployment environment is set and used when calling the configuration settings.

Listing 10.2 A stripped down version of the Bootstrap class in bootstrap.php

```
<?php
class Bootstrap
{
    protected $_deploymentEnvironment;

    public function __construct($deploymentEnvironment)            A
    {
        $this->_deploymentEnvironment = $deploymentEnvironment;
    }

    public function run()                                          B
    {
        $config = new Zend_Config_Ini(
            'configuration/config.ini',
            $this->_deploymentEnvironment);                         C
    }
}
```

A The environment parameter which we had set as being “nick-dev” in our index.php file

B The run() method which initialises and starts our application

C Zend_Config_Ini is sent the section 'nick-dev' as its second parameter

The second parameter passed to Zend_Config_Ini in listing 10.2 specifies that the section to be read from our config.ini file is the one passed from index.php, i.e. 'nick-dev'. Zend_Config_Ini not only supports ini sections but also implements inheritance from one section to another. An inheriting sections can also override values inherited from its parents. Looking for the 'nick-dev' section in our config file in listing 10.3 we can see that it inherits from the 'general' settings as indicated by the [child : parent] syntax of the ini file and therefore the database settings are being reset to those of a local development database.

Listing 10.3 The contents of our config.ini file

```
[general ]      A
```

```

database.type = PDO_MYSQL
database.host = localhost

[production : general ]           B
database.username = production
database.password = production123
database.dbname = production_main

[rob-dev : general ]             C
database.username = rob
database.password = roballen123
database.dbname = robs_dev

[nick-dev : general]             D
database.username = nick
database.password = nicklo123
database.dbname = nicks_dev

```

A The general ini section from which all other sections inherit

B Settings for the final production server

C Settings for Rob's development server

B The nick-dev section with database settings used in our example

What we have managed to do so far is to contain the implementation changes within the index.php file and the configuration file. Even better, we've managed to do it in a way that allows us to move those files around without having to change the files themselves. To demonstrate this a little further let's take a look at Rob's index.php file which never needs to move from his local machine:

```

$bootstrap = new Bootstrap('rob-dev');
$bootstrap->run();

```

Finally on the production server lies the untouched index.php with its own setting:

```

$bootstrap = new Bootstrap('production');
$bootstrap->run();

```

Having set all that up, the only file that moves between the environments is config.ini which, because it contains the varying sections, can be the same file in use by all stages. Containing our application in this way simplifies the moving of files between servers whether that be synchronising using your favourite FTP client or a scripted deployment.

10.1.2 Using virtual hosts for development

Having outlined the use of separated hosting environments for the stages of development it's worth going through an example of how the hosting itself can be setup to accommodate this. In this section I'll cover a brief example assuming the use of the Apache web server in a Unix based machine on a small local network.

Virtual hosting is a method of serving multiple web sites from a single machine. The use of a single machine reduces not only the cost of hardware but also the time required to support and maintain that machine. The two main variations of virtual hosts are name-based virtual hosts that share the same IP address and IP-based in which each virtual host has its own IP address. For the sake of simplicity we are using name-based hosting. Once setup we should be able to access our separate development stages with URL's like `http://places-dev/` and `http://places-stage/`.

As an aside, Zend Framework uses `mod_rewrite` to direct all requests via the front controller file. Combine that with its routing functionality and you increase the possibilities of errors resulting from path issues. Setting up virtual hosts is one way to reduce the time wasted on path problems as `http://127.0.0.1/~places-dev` becomes `http://places-dev/`

Setting up the hosts file

The hosts file stores the information that maps the hostname to an IP address. As mentioned above name-based hosting uses a single address for multiple names so looking at the hosts file in /etc/hosts we have the following entry:

```
127.0.0.1 places-dev
127.0.0.1 places-stage
```

The names places-dev and places-stage are both being mapped to the localhost IP address 127.0.0.1 meaning that any requests this machine gets for those names will be directed to its own web server. Rather than attempt to configure a full domain name system (DNS), we're also going to configure the hosts file of one of the networked machines to point the same names to the IP address of the host machine:

```
192.1610.1.100 places-dev
192.1610.1.100 places-stage
```

Configuring Apache

Having set-up the hosts file we now need to configure the virtual hosts in Apache's configuration file httpd.conf as shown in the example in listing 10.4.

Listing 10.4 Our virtual host settings in Apache's httpd.conf file

```
NameVirtualHost *:100                                A

<Directory "/path/to/Sites">                          B
Options Indexes MultiViews
AllowOverride All                                     C
Order allow,deny
Allow from all
</Directory>

<VirtualHost *:100>                                    D
DocumentRoot /path/to/Sites/places-dev/web_root
ServerName places-dev
</VirtualHost>

<VirtualHost *:100>                                    E
DocumentRoot /path/to/Sites/places-stage/web_root
ServerName places-stage
</VirtualHost>
```

A Setting the IP address and port on which the server will receive requests

B Some default settings for all directories to which Apache has access

C The AllowOverride setting is important to allow the use of .htaccess settings needed by the Zend Framework

D Settings for our development host

E Settings for our staging host

In listing 10.4 we can see that the NameVirtualHost directive specifies that anything coming in on port 100 will receive requests for the name-based virtual hosts.

The directory section defines a few general settings for all virtual host directories, one to note in particular is the AllowOverride setting which allows directories to override certain server settings using .htaccess files. This is a requirement for Zend Framework based sites as mod_rewrite is used to direct all incoming requests through the file index.php.

Each VirtualHost section defines the full path to the the web root and the name of the server.

Having specified the DocumentRoot in httpd.conf we then need to create those directories:

```
$ cd Sites/
$ mkdir -p places-dev/web-root
$ mkdir -p places-stage/web-root
```

Restart Apache to allow it to pick up the new settings:

```
$ sudo apachectl graceful
```

If all went well, pointing our web browser at <http://places-dev> from either of the two machines whose host files we edited should end up at whatever files are located in `/path/to/Sites/places-dev/web_root` on the hosting machine. Of course at this point there are no files in the hosting space. To get those we will need to check them out of the version control repository.

10.2 Version control

With Rob being in the United Kingdom and my being in Australia, working collaboratively on the code for this book wasn't as simple as shouting across a desk. In order to keep up with our changes we needed to have some kind of version control system. There are many different version control systems with varying functionality and the one we will use for this project is the same one used in the development of the Zend Framework; Subversion. Subversion is an open-source system that will sort, store and record changes to your files and directories over time, whilst also managing the collaborative use of that data over a network. Getting acquainted with the process of version control and with Subversion in particular, will not only allow you to work a little more closely with the framework's repository, but also introduce you to a workflow that will further improve your deployment practices.

10.2.1 Working with Subversion

In the following section we are going to run through some of the day-to-day uses of version control using Subversion. The intention is to give you an overview of the process and leave you with the confidence to investigate further. To try some of the examples out for yourself you will, of course, need to have access to a subversion host. This is admittedly a bit of a chicken and egg situation, unfortunately setting up a subversion server is beyond the scope of this chapter. We will therefore presume you have access and focus on setting up and working with a project in Subversion.

Creating the repository

To store projects in Subversion, first we need to create a repository. Because creating a repository on a network drive is not supported this must be done locally to a drive on the same machine. To create the repository for places we need to type the following command:

```
svnadmin create /path/to/repository/places/
```

The subversion project officially recommends setting up three directories, `trunk/`, `tags/` and `branches/` under any number of project roots. The trunk is fairly obviously the main directory where the bulk of the action takes place. The tag directory holds meaningfully "tagged" copies such as for example if we decided to mark a stage in the development of places in the context of the writing of this book:

```
$ svn copy http://svn.example.com/svn/places/trunk \
http://svn.example.com/svn/places/tags/chapter-04 \
-m "Tagging places as snapshot of development at chapter 4."
```

The `branches/` directory holds "branched" copies of the filesystem such as those created if we decided we needed to experiment with a significant change to the architecture of "places". We'll cover branching in a little more detail further on in this chapter.

Since we are setting these directories up locally and we want them to be under version control we'll do so using the `svn mkdir` command:

```
$ cd /path/to/svn/places/
$ svn mkdir trunk
$ svn mkdir tags
$ svn mkdir branches
```

Having created the relevant directories we can then import the partial project we had started:

```
$ svn import /path/to/places file:///path/to/svn/places/trunk/ \
-m "Initial import"
Adding places/web_root/index.php
Adding places/web_root/.htaccess
```

Adding places/application

Committed revision 1

Having created those directories and imported the start of our project into the svn repository we are ready to start using it for development. Figure 10.1 shows the svn directory structure after the first commit.

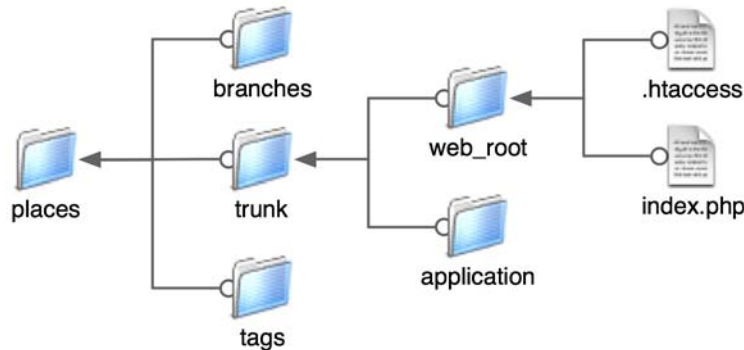


Figure 10.1 The places SVN repository after the first commit

Checkout the latest code from the repository

Having set-up the project repository to this stage we then decide to finish up for the day and leave Rob, whose day has just begun, to add his work. By the time morning comes we can see in listing 10.5 that the repository is bursting at the seams with new files, ready for us to checkout a working copy into our local virtual host setup machine.

Listing 10.5 Checking out the latest work from the repository

```
$ svn checkout http://www.example.com/svn/places/trunk/ places/ A
A places/web_root
A places/web_root/css
... B
A places/db
A places/db/test_data.sql
Checked out revision 2.
A Note the second directory which specifies to checkout the places/trunk directory to a local places directory
B Output condensed for brevity
```

Commit some changes to the repository

At this point it is worth starting to develop some good commit habits, the main one being to try and keep the purpose of the changes as focused as possible. Just think of the other members of your team having to check out one big commit that includes a multitude of scattered changes and you quickly understand what that means. You'll also notice any lack of focus when your commit messages start becoming unclear and difficult to write. Whether formal or informal, most teams will have some commit guidelines such as including issue tracker reference numbers in the commit message for example.

Going through our new working copy the first thing we notice is the need to edit the config.ini file to use our local database. For now we can just make a copy which we call config.ini.default and use the original to get the local application working. Ultimately we'll set the config file up more consistently but for now let's commit config.ini.default as shown in listing 10.6, so that any further commits won't be writing over our individual config files.

Listing 10.6 Committing our changes to the repository

```
$ cd ~/Sites/places
$ svn status | A
? application/configuration/config.ini.default A
```



```

$ svn add application/configuration/config.ini.default    B
A application/configuration/config.ini.default
$ svn status
A application/configuration/config.ini.default
$ svn commit -m "Added .default config file."          C
Adding application/configuration/config.ini.default
Transmitting file data .....
Committed revision 3.

```

A The status command gives feedback on the status of the working copy. In this case it points out that config.ini.default is not under version control

B The file is added putting it under version control indicated by the A after a second status check

C Once the working copy is ready it is committed to the repository

In the code in listing 10.6 you may have noticed the repeated status checks. To be sure that your working copy is as you expect it's a good idea to get into the habit of checking its status before performing any further action. **Updating a local working copy**

At each stage of development we need to be checking on the status of our working copy and making sure that it is up to date with the repository. Not surprisingly, the svn update command takes care of updating a local working copy with the latest changes and those changes are indicated by the output of the command. In listing 10.7 we perform an update on our working copy.

Listing 10.7 Updating our working copy from the repository

```

$ cd ~/Sites/places
$ svn status                                A
$ svn update                                B
A application/classes
A application/classes/Places
A application/classes/Places/Controller
A application/classes/Places/Controller/Action
A application/classes/Places/Controller/Action/Helper
A application/classes/Places/Controller/Action/Helper/ViewRenderer.php
Updated to revision 4.

```

A Notice the status check on our working copy before proceeding

B The update command which initiates the update action

As indicated by the A preceding each output line we can see in listing 10.7 that the file ViewRenderer.php has been added together with its parent directories. This has been a simple update with no changes conflicting with our working copy however when conflicts do occur we need to be able to deal with them. **Dealing with conflicts**

Subversion is able to manage changes even when two people have edited the same file at the same time, by simply combining changes as needed. In some cases however, conflicts may arise, if, for example, changes have been made to the same lines of a file. In this instance a later update results in a conflict with the file bootstrap.php as indicated by the C preceding the file information:

```

$ svn update
C application/bootstrap.php
Updated to revision 5.

```

In figure 10.2 we can see that the update produced four variations of the conflicting file; bootstrap.php.r1100 is the original before our local modification, bootstrap.php.r1102 is the version that Rob committed to the repository, bootstrap.php.mine is the modified version in our local working copy and finally, bootstrap.php contains the diff of both versions.

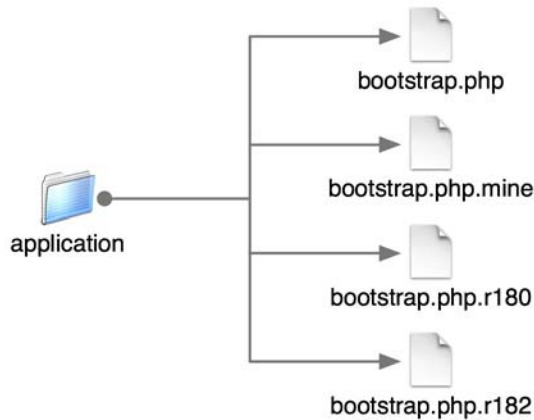


Figure 10.2 The four variations of the conflicting files after svn update

Taking a look inside bootstrap.php we can see the following conflicting lines:

```
<<<<<< .mine
Zend_Controller_Action_HelperBroker::addPrefix('Places_Controller_Action_Helper');
=====
Zend_Controller_Action_HelperBroker::addPrefix('Zend_Controller_Action_Helper_');
>>>>>> .r1102
```

Which is roughly the equivalent to the output generated if we were to run the command:

```
diff bootstrap.php.mine bootstrap.php.r1102
```

From this we can see that our edit, .mine, removed the underscore from the end of the parameter string "Places_Controller_Action_Helper" while Rob's edit was to change the start of the parameter name to "Zend_" instead of "Places_". After a quick discussion with Rob we decide that neither of our changes is needed and we should revert back to the version before either of our changes. This is a simple fix for which we could either edit bootstrap.php or copy bootstrap.php.r1100 over bootstrap.php. For the sake of the example let's edit bootstrap.php to contain only the single line:

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

Having done that a quick status check indicates that while we've corrected the file, the svn conflict still exists:

```
$ svn status
C application/bootstrap.php
```

Since we've decided that out of the four variations bootstrap.php is now the file we want to use we need to tell subversion that the issue has been resolved and which file to go with:

```
$ svn resolved application/bootstrap.php
Resolved conflicted state of 'application/bootstrap.php'
```

Having done that another svn status check indicates that the file is now marked as modified

```
$ svn status
M application/bootstrap.php
```

The other three files have also been removed as part of the resolve process, leaving only the bootstrap.php file

```
$ ls
bootstrap.php config.ini controllers views
classes configuration models
```

The final step is to commit the changes back to the repository together with a message noting the resolution:

```
$ svn commit -m "Resolved change conflict with bootstrap.php"
Sending application/bootstrap.php
Transmitting file data .
Committed revision 6.
```

Having given an overview of some of the common day-to-day tasks while using subversion there remain a few topics that are worthy of a mention.

Getting a clean copy from the repository

Subversion stores its information in its own .svn directories inside each directory in the repository. Those .svn directories are necessary while keeping content under version control but you may not want them in the final release, such as, for example if you are going to ftp the contents to a server. Listing 10.8 shows the use of the export command to get a copy cleaned of all those hidden directories from the repository.

Listing 10.8 Using the export command to get a clean working copy

```
$ svn export http://www.example.com/svn/places/trunk/ places_export/  A
A places_export
A places_export/web_root
...
A places_export/db
A places_export/db/test_data.sql
Exported revision 7.
A Note the second path indicating a location for the cleaned working copy
B Output condensed for brevity
```

Note that having performed the export in listing 10.8, we are left with the same files as we would have had if we had done a checkout, but since they do not contain the .svn directories they are not under version control.

10.2.2 Externals

While we are moving files between working copies and the repository there is one element that has not been accounted for; what about the Zend Framework code? We can avoid having to deal with that by mapping one of our local directories to the external URL of the Zend Framework repository:

```
svn propedit svn:externals .
application/library/Zend \
http://framework.zend.com/svn/framework/trunk/library/Zend/
```

Now any checkout of the repository that includes that local directory will also will also checkout the Zend Framework code.

10.2.3 Branches

We gave an example of how tagging could be used for marking the repository at specific points in the progress of the chapters of this book but what about a bigger occasion such as when we actually finish it? That would be an occasion to use the branching capabilities of Subversion:

```
$ svn copy http://www.example.com/svn/places/trunk/ \
http://svn.example.com/places/branches/its-finished \
-m "Woohoo, we've finished the book!"
```

Committed revision 200.

As the name suggests and figure 10.3 illustrates, branching creates a line of history independent of the main trunk. There are numerous reasons why we may decide to branch our code, Zend Framework branches on each official release, but we could as equally decide to branch based on a need for a custom version of the main code or an experimental version.

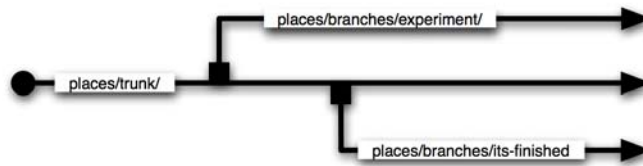


Figure 10.3 Branches in an SVN repository

The subject of branching is the topic of a whole chapter into itself so this brief mention is really just to indicate its use and hope that you the reader follow it up with your own research.

10.3 Functional testing

Throughout the chapters of this book we have made a point of unit testing the code we have been working on. The combination of the thorough code coverage of Zend Framework's unit tests with any application specific unit tests should, from the programmer's perspective, provide ongoing feedback that the code is performing as expected. However, even a system with all unit tests passing successfully may fall short in other areas such as how the system functions when in actual use. To test this we need to move onto what what can be somewhat broadly referred to as "functional testing".

While testing the functionality of the system can be narrowed into more specific areas such as its usability, security, performance, etc, in this case we're going to outline more general methods of testing the system from the perspective of the end user, that is, we will be attempting to test what the user can do with the system. These methods can then be applied more specifically as needed, for example, in security testing we could test whether a user that is not logged-in is able to access a restricted area.

10.3.1 Functional testing with Selenium

Selenium is a tool for testing web applications the same way you do; with a web browser. Anyone familiar with recording a macro will recognise that, at its simplest, Selenium IDE is a tool to record, edit and playback actions. Once recorded you can run, walk or step through actions with or without start points and breakpoints.

What makes Selenium particularly useful is the fact that the unit testing tool PHPUnit used by the Zend Framework has a Selenium RC extension which means we can integrate the Selenium tests into our overall testing procedure. **Installing Selenium IDE**

Since Selenium IDE is an add-on for the Firefox web browser you need to have downloaded and installed Firefox from <http://www.mozilla.com/firefox/> before you can install the extension.

Selenium IDE can be downloaded from the Firefox add-ons page at <https://addons.mozilla.org/firefox> or from the project's own download page at <http://www.openqa.org/selenium-ide/download.action>. The easiest method of installing the extension is to click the links to the .xpi files from within Firefox and allow Firefox to handle the download and install.

Recording a Selenium test

We start by choosing Selenium IDE from the Firefox tools menu which sets it automatically in recording mode. Next we simply interact with the web pages as usual. For this example let's record a very simple test that queries Google for "zend framework" then follows the links to the Zend Framework manual page.

7. Navigate to google.com.au
8. Enter "zend framework" in the search field and click the "Google Search" button.

9. After the list of results appear follow the link to <http://framework.zend.com>
10. On the Zend Framework home page follow the link to the manual

Once we've finished performing those steps we can stop recording and if all went well Selenium IDE will have the recorded actions visible in its "Table" view shown in figure 10.4.

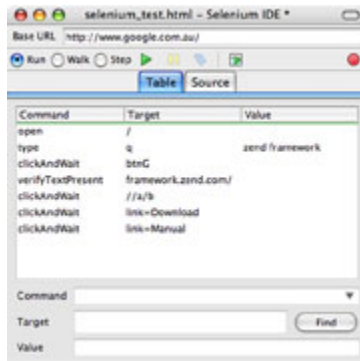


Figure 10.4 Selenium IDE in Table view with the results of our recording

Editing the test

If you're following this so far you'll probably notice that "Downloads" was accidentally clicked on in the Zend Framework home page. Since this was not the aim of our test we want to edit out that step and we can do so by switching to the "Source" view of the Selenium IDE window which will display the HTML shown in Figure 10.5.

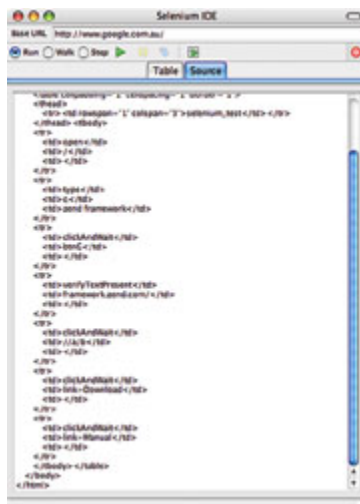


Figure 10.5 Selenium IDE in Source view with the results of our recording

The offending lines that we need to remove from the above source are of course these which can be edited directly in the "Source" view:

```
<tr>
  <td>clickAndWait</td>
  <td>link=Download</td>
  <td></td>
</tr>
```

Having removed those lines we can give it a quick run through by clicking the play button and if it performs as expected we can save it for future use. **Saving the test**

Having recorded and edited our test we can now save it for later use. With Selenium IDE active choosing File from the menu bar presents several options for saving tests. In this instance we will choose "Save Test As..."

which will save the test in the default HTML format we saw above. It can then be opened later and the test repeated as needed.

10.3.2 Automating Selenium tests

In the previous example we recorded a single test and ran it completely inside the browser under what is called Selenium's "test runner" mode. This is fine for single tests but to make ourselves truly productive, we need to use Selenium RC which allows us to run multiple Selenium tests written in a programming language.

One of the supported languages is of course PHP which is available as "PHP - Selenium RC" when choosing "Export Test As..." from Selenium IDE. Unfortunately, at the time of writing, the resulting exported file contains code that is not only a little wordy but does not make use of the newer PHPUnit extension `PHPUnit_Extensions_SeleniumTestCase`. Listing 10.9 shows a more efficient rewrite of our Selenium test as a PHPUnit test case.

Listing 10.9 Our Selenium test as a PHPUnit test case

```
<?php
require_once 'PHPUnit/Extensions/SeleniumTestCase.php';

class seleniumExampleTest extends PHPUnit_Extensions_SeleniumTestCase
{
    protected function setUp()                                A
    {
        $this->setBrowser('*firefox');
        $this->setBrowserUrl('http://www.google.com.au/');
    }

    function testMyTestCase()                                  B
    {
        $this->open('http://www.google.com.au/');
        $this->type('q', 'zend framework');
        $this->click('btnG');
        $this->waitForPageToLoad('30000');
        try {
            $this->assertTrue($this->isTextPresent('framework.zend.com/'));
        } catch (PHPUnit_Framework_AssertionFailedError $e) {
            array_push($this->verificationErrors, $e->toString());
        }
    }
}
```

A Setting up the default settings such as the browser to use and the domain name to work under

B Our test case which replicates the test we did in Selenium IDE

If you are being observant you'll notice that our test case is actually missing the part of the test we made with Selenium IDE which operates within the zend.com domain. Because Selenium RC runs partly outside the browser we run into issues with the "same origin policy" which prevents a document or script "loaded from one origin from getting or setting properties of a document from a different origin." The original test we recorded with Selenium IDE worked because it was running completely inside the browser as an extension. Selenium RC runs pages through a proxy which partly circumvents the same origin issue however support for switching domains is currently experimental.

Having written our test we are almost ready to run it, however since it relies on the Selenium RC Server that needs to be downloaded from <http://www.openqa.org/selenium-rc/download.action> and started using:

```
java -jar /path/to/server/selenium-server.jar
```

Once the server is running successfully, we are ready to run our test



Figure 10.6 Running our Selenium example unit test

Great! It passed. Now that our selenium test is written as a PHPUnit unit test we can continue adding tests and incorporating them as part of an overall test suite that can be run from a single command.

10.3.3 Functional testing with Zend_Http

As the manual states "Zend_Http_Client provides an easy interface for performing Hyper-Text Transfer Protocol (HTTP) requests." At its most basic level that is what a web browser like Firefox is and with that in mind listing 10.10 shows an example of the tests we did with Selenium rewritten to use Zend_Http_Client.

Listing 10.10 Our Selenium test rewritten to use Zend_Http

```
<?php

set_include_path(get_include_path() . PATH_SEPARATOR . '/path/to/zend/library/');
include_once 'Zend/Http/Client.php';

class ZendHttpTest extends PHPUnit_Framework_TestCase
{
    public function testLoadGoogle()
    {
        $client = new Zend_Http_Client('http://www.google.com.au/');
        $response = $client->request();
        $this->assertContains('</from>', $response->getBody());
    }

    public function testQueryGoogle()
    {
        $client = new Zend_Http_Client(
            'http://www.google.com.au/search?q=zend+framework');
        $response = $client->request();
        $this->assertContains('framework.zend.com/',
            $response->getBody());
    }
}
```

A Test makes a request to Google and then makes sure it contains some html

B Test makes a search query request to Google and makes sure a string appears in the results

The similarities between the code for this and the previous Selenium test are pretty evident so there really isn't any need to do much more than run the test. We can see the results in figure 10.7.

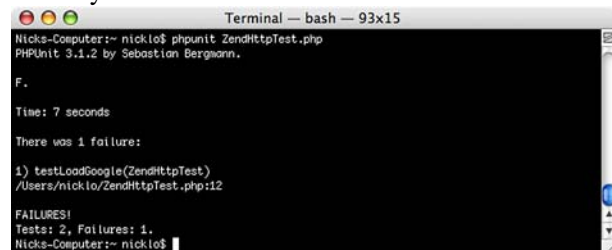


Figure 10.7 One failure on the first run of our unit test.

One failure due to the typo "</from>" instead of "</form>", so after correcting that let's check the results of running the test again in figure 10.8

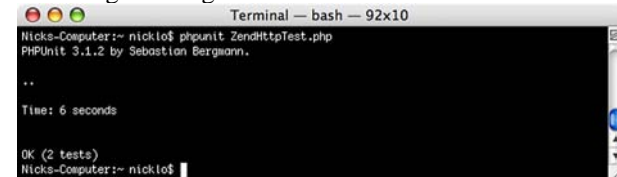


Figure 10.8 The final unit test is successful.

Great that passed and just like the Selenium based tests it can be easily incorporated into a test suite. Zend_Http does not currently have all the capabilities that Selenium has, such as for example the ability to test javascript, however, as we can see it is a relatively simple way to make functional tests.

10.4 Scripting the deployment

We briefly mentioned using the svn export command to get a clean version of an SVN repository which we could transfer via FTP to a production server. That solution would be useful if, for example, you are working with a client's production server over which you have no control. If you did have control over the server there is no reason why the production server could not also be under version control and updated with the svn update command. Since we've already spent a lot of time discussing methods of automating tests the natural follow on would be to automate the testing and deployment of your code between the different environments.

Unfortunately, as with svn branching, scripting your deployment is not only too deep a topic for a few paragraphs of this chapter but also one quite specific to each team's work requirements. We can, however, point you to the Zend Framework build-tools/Makefile for inspiration. This file contains a series of shell scripts which export from the repository, generate the documentation and generate the zip and tarball archives that can be downloaded from the Zend Framework download page. Another option in the process would be to run all automated tests and only continue with the deployment if all tests pass. Even the triggering of the running of these deployment scripts can be automated, perhaps on a scheduled basis, resulting in a form of "continuous integration".

10.5 Summary

This chapter has presented a bit of a whirlwind tour of deployment practices and as a result has probably raised more questions than provided answers. Our reason for including a chapter on the topic of deployment was to illustrate that producing a quality product does not stop with just writing quality code.

We also wanted to demonstrate the use of some of the Zend Framework components in the process. How much of these practices you decide to implement is dependent not only on your working environment but also on the needs of the projects you are working on. Ultimately, if any part of it has inspired you to investigate the topic further armed with a clearer overview of the process then it will have done its job.

Talking with Other Applications

While working on this chapter I was flicking through some of the XML and PHP books I picked up several years ago and noted that despite being around five years old they are still in almost perfect condition. The truth is that while I read them from cover to cover, I never really needed to use them the same way I have my dog-eared book on design patterns. Like any idiot with a new piece of technology I made the mistake of trying to use XML at every possible and often inappropriate opportunity. However, once over that initial burst of enthusiasm I settled on using it in much more measured ways, such as, for example, using RSS feeds to supplement a directory site with news from the websites of its listings.

This chapter will cover the use of various components of the Zend Framework that can be loosely lumped together under the term “web services”, which the World Wide Web Consortium (W3C) defines as “a software system designed to support interoperable machine-to-machine interaction over a network”. For the sake of simplicity we will base our use of the term web services in this chapter on that definition rather than the more specific W3C focus on it being a combination of SOAP and WSDL (Web Services Description Language).

Part of that interoperation includes the use of XML, hence my exploring the books mentioned above, however one of the benefits of using these components is that we don’t need to focus on XML itself. Zend Framework provides a series of tools that take care of the formatting and protocol of the interaction, leaving you the developer to focus on the logic using only PHP. If you were to pause and make a list of all the formats available just for web newsfeeds the benefits of not having to deal with that range and rate of change would be clearly illustrated.

Before we get into using the Zend Framework components we’ll take a look at the how’s and why’s of integrating applications using web services.

11.1 Integrating Applications Together

It is interesting just how much web services have become a part of our offline and online existence. Every time I start up my computer my newsreader fires up and digests a list of XML formatted news from sites that I could never have been able to keep up with otherwise. Recently my wife sold some of the clutter from our garage using GarageSale, a Mac OS X desktop application that talks to eBay using its XML based API (Application Programming Interface) over HTTP. The key to all of these actions is the exchange of information across a network and the distribution of the computing.

11.1.1 Exchanging structured data

XML stands for Extensible Markup Language and originates from Standard Generalized Markup Language (SGML) which is one of the many markup languages whose role is simply to describe text. Probably the most well know is HTML (HyperText Markup Language) which describes text documents that are intended to be transmitted by HTTP.

Data does not need to be marked up in order to exchange it however, but in most all cases it does need to have some kind of structure. Whether that be comma/tab separated values (csv or tsv) structured text which

bases its structure on a regular sequence of data separated by consistent delimiters, or serialized data such as created by PHP's own `serialize()` function, or other formats like Javascript Object Notation (JSON) which can be used as an alternative to XML in AJAX (and incidentally is accommodated for in the Zend Framework by `Zend_Json`).

Little of this information should be new to readers of this book but the fact to take away is that what we are trying to achieve is the passing of information from one system to another such that the recipient system knows how to handle that data.

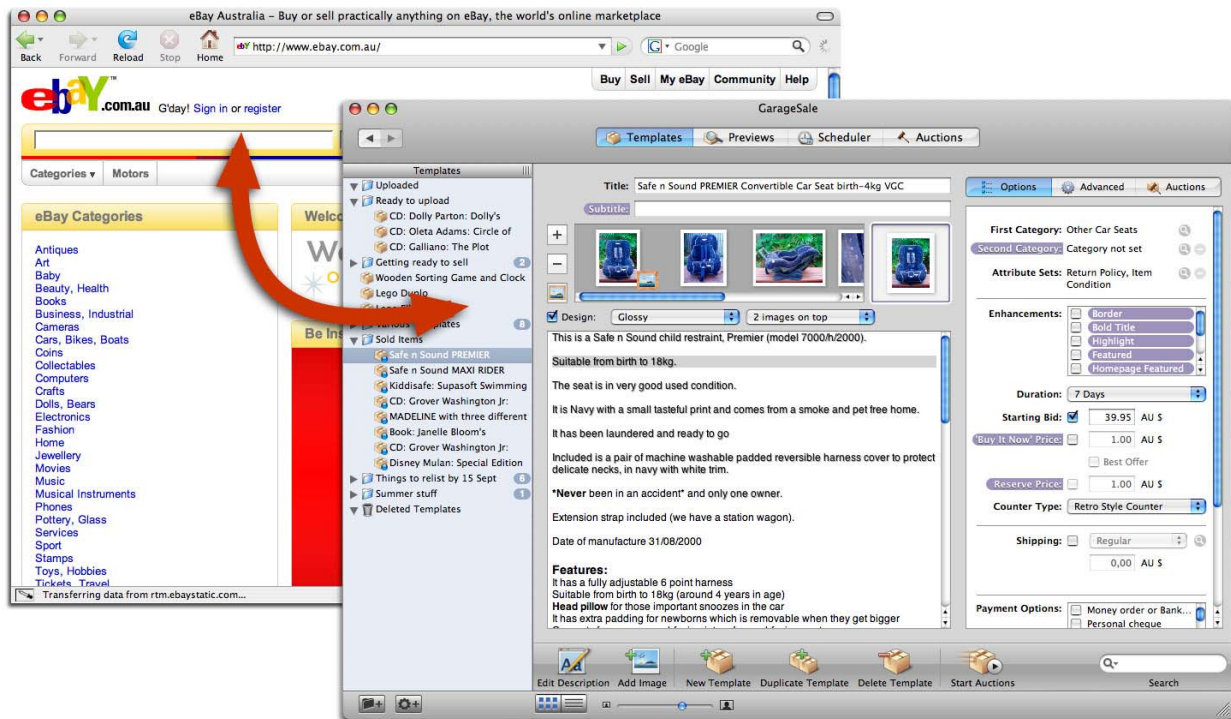


Figure 11.1 Mac OS X desktop application GarageSale which converses with eBay's XML based API.

If we take a look in Figure 11.1 at the application GarageSale that was mentioned earlier, it is clearly a fairly complex application whose data could not be exchanged unless it was suitably structured in order for eBay to process it and carry out whatever request it makes, such as creating a new item for auction.

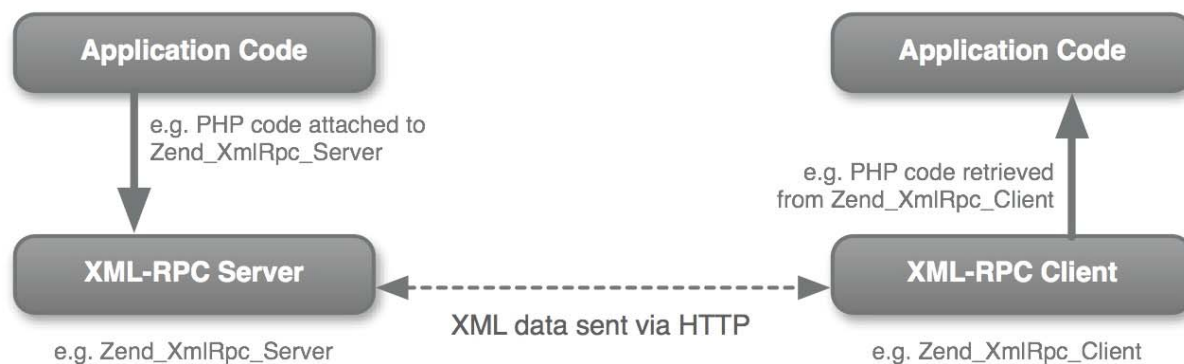
Having looked at the language involved in this discussion between applications the next question is how the applications talk to each other.

11.1.2 Producing and consuming structured data

The earlier chapter on email serves as a good example of how applications talk to each other using a structured data format (Internet e-mail format) that is produced at one end, sent by a mail server (Mail Transfer Agent) and consumed by the receiving email client (Mail User Agent) at the other. The application conversations in this chapter take that basic concept a step further by using that exchange to trigger actions at either end, through what is known as a remote procedure call (RPC).

Later in this chapter we'll be covering `Zend_XmlRpc` which uses `Zend_XmlRpc_Server` to send and receive XML encoded remote procedure calls over HTTP. Originally created by Dave Winer, XML-RPC is a surprisingly simple and flexible specification which allows it to be used in a wide variety of situations.

As functionality was added, XML-RPC evolved into SOAP (originally an acronym but now just a word in itself) and has been adopted by the World Wide Web Consortium. However, you don't need to look too hard to find complaints about how this added functionality also added a lot more complexity, which partly explains



why XML-RPC is still in use despite the official W3C adoption of SOAP. In some ways SOAP is itself being superseded by other protocols like ATOM. You can see the status of these protocols reflected in Zend Framework itself with the fact that at the time of writing Zend_Soap is still in the incubator while XML-RPC and ATOM are both included in the core.

11.1.3 How web services work

To illustrate how web services work let's follow the steps of an imaginary XML-RPC example in which a desktop application needs to get the updated price of a particular item from an online service as illustrated in figure 11.2:

Figure 11.2 The basic transaction between one system and another using XML-RPC

11. The desktop application gathers the data required to make a procedure call that includes the id of the requested item and the remote procedure which gets the prices for items.

7. 2. The XML-RPC client component of the desktop application encodes this remote procedure call into XML format and sends it to the online service like so:

```

<?xml version="1.0"?>
<methodCall>
  <methodName>onlineStore.getPriceForItem</methodName>
  <params>
    <param>
      <value><i4>123</i4></value>
    </param>
  </params>
</methodCall>
  
```

6. The XML-RPC server of the online service receives the XML encoded procedure call and decodes it into a format that the system code can process, e.g. \$store->getPriceForItem(123).

7. The system code returns the requested price to its XML-RPC server which encodes that as an XML response and sends it back to the requesting desktop application:

```

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>19.95</double></value>
    </param>
  </params>
</methodResponse>
  
```

8. The desktop application receives the response and decodes it into a format it can process and updates the price for item 123 to \$19.95.

11.1.4 Why do we need web services?

The simple answer to this question is also the most ironic; we need web services so that applications running on different platforms and/or frameworks can talk to each other in standard way. This chapter has already

started to point out the irony in that concept by, in all likelihood, confusing you with what is only a small selection of the variations in the protocols that make up these “standards”.

Pushing this cynicism aside however and returning to the example of our desktop application fetching updated prices from the online service, you’ll notice that there was little detail about how each end of the transaction actually performed their procedure calls and the reason was that it didn’t matter. It didn’t matter because as long as each application was able to convert its internal processes into a standard form of communication, the XML-RPC protocol, then the transaction could be completed.

Clearly this can lead to some very powerful interactions such as the one between GarageSale and eBay. We will continue this chapter and the next with a look into some examples of how the Zend Framework components can bypass some of the complexities of web services to take advantage of such interaction. We’ll start by one that almost all readers are probably most familiar with; web feeds and the benefits of using Zend_Feed to produce and consume them.

11.2 Zend_Feed

The Zend Framework manual describes Zend_Feed as providing “functionality for consuming RSS and Atom feeds.” However, we are going to start from the opposite end of that transaction by showing how it can also be used to produce RSS and ATOM feeds. We’ll then show some examples of consuming web feeds.

11.2.1 Producing a feed

If you took up the challenge mentioned at the start of this chapter to list all the formats available for web feeds you’d have started with the RDF or RSS 1.* branch that includes RSS 0.9, 1.0 and 1.1 then moved to the RSS 2.* which includes RSS 0.91, 0.92 and RSS 2.0.1 then you’d have followed that to the ATOM syndication format. If you did that while under the pressure of a deadline and realised that all of those formats are potentially in use amongst the millions of syndicated sites you’d have probably sat down in a cold sweat!

Of course, once you calmed down, you’d likely realise that all you need do is to pick the latest and greatest formats and concentrate on outputting those. Thankfully, even that isn’t needed as Zend_Feed can take care of the format for you leaving you to simply pass it the data it needs for your feed. In listing 11.1 we demonstrate a very simple controller action that produces an RSS (2.0) or ATOM feed.

Listing 11.1 A feed producing controller action

```
require_once 'Zend/Feed.php';
require_once 'models/ArticleTable.php';

class FeedController extends Zend_Controller_Action
{
    public function indexAction()
    {
        $format = $this->_request->getParam('format');
        $format = in_array($format,
            array('rss', 'atom')) ? $format : 'rss';           A

        $articlesTable = new ArticleTable();                  B
        $rowset = $articlesTable->fetchAll();                  B

        $channel = array(                                     C
            'title'      => 'Places',
            'link'       => 'http://places/',
            'description' => 'All the latest articles',
            'charset'    => 'UTF-8',
            'entries'    => array()
        );
    }
}
```

```

foreach ($rowset as $item) {
    $channel['entries'][] = array(
        'title'      => $item->title,
        'link'       => 'http://places/article/index/id/'
            . $item->id . '/',
        'description' => $item->body
    );
}
$feed = Zend_Feed::importArray($channel, $format);
$feed->send();
$this->_helper->viewRenderer->setNoRender();
$this->_helper->layout()->disableLayout();
}
}

```

A If neither choice of RSS or ATOM feed format is requested default to RSS

B Grab a selection of articles from the database to insert into the feed (this would likely be limited but is simplified for this example)

C Build up the required <channel> elements into an array

D Build up each <item> from the rowset retrieved from the Articles table with some minimal elements within the above array

E Import the array into Zend_Feed along with the format in which it is to be encoded

F Set the content type of the HTTP header string and output the feed

H Disable the view and layout rendering as this is not an HTML page.

What we've done in listing 11.1 is to retrieve some data from the database, put it into a multidimensional array, then used that array to build either an RSS or ATOM feed which is then output as an XML string ready to be digested by a newsfeed reader or aggregator. Note that in our example we use the send() method which sets the content type of the HTTP header string for us to something like this:

Content-Type: text/xml; charset=UTF-8

If we were using the feed in some other way we could just use the following to get the XML string without the HTTP headers:

```
$feed->saveXml()
```

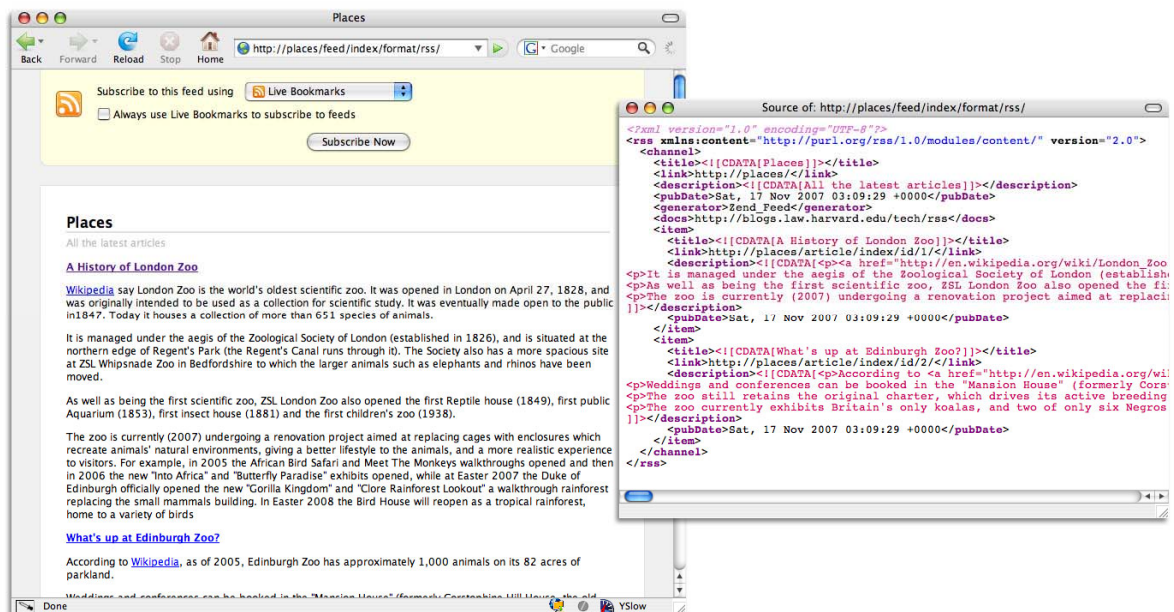


Figure 11.3 The feed we produced as it appears in the Firefox web browser together with the XML source.

In figure 11.3 we can see that Firefox recognises this as a web feed, shows its parsed content and asks if we'd like to subscribe to it through its Live Bookmarks. It should be noted, however, that the feed we've produced is a little too minimal and would likely need further elements added for it to work with other readers and aggregators. We have kept it simple for the sake of clarity.

11.2.2 Consuming a feed

Earlier in this chapter we gave an example of the consumption of web feeds retrieved from the websites of listings in an online directory. In that particular case each listing had the URL of its feed stored along with its other data. If that site had been built using Zend Framework storing the specific feed URL would have been unnecessary as `Zend_Feed` is able to parse any HTML page searching for the same link elements that modern browsers use to indicate the presence of a feed:

```
<link rel="alternate" type="application/rss+xml"
      title="Places RSS Feed" href="/feed/index/format/rss/" />
<link rel="alternate" type="application/atom+xml"
      title="Places Atom Feed" href="/feed/index/format/atom/" />
```

All that is required is a single line of code:

```
$feedArray = Zend_Feed::findFeeds('http://places/');
```

In the current case we do know the URL for the feed we produced earlier and the code to consume that feed is straightforward as it imports directly from the URL:

```
$this->view->feed = Zend_Feed::import('http://places/feed/index/format/rss/');
```

The elements of that feed could then be presented in a view like so:

```
<h1>
  <a href="<?php echo $this->feed->link(); ?>
  <?php echo $this->feed->title(); ?></a>
</h1>
<div class="channeldesc"><?php echo $this->feed->description(); ?></div>
<?php foreach ($this->feed as $item): ?>
  <h2>
    <a href="<?php echo $item->link; ?>"><?php echo $item->title(); ?></a>
  </h2>
  <div class="itemdesc"><?php echo $item->description(); ?></div>
<?php endforeach; ?>
```

While we're covering the methods of consuming feeds it would be remiss not to mention the remaining methods which include importing from a text file:

```
$cachedFeed = Zend_Feed::importFile('cache/feed.xml');
```

Importing from a PHP string variable:

```
$placesFeed = Zend_Feed::importString($placesFeedString);
```

That covers some of the basic features of `Zend_Feed` and of course there is more that we could have covered if it weren't for the limitations of chapter size and the fact that we're eager to move onto the next section on `Zend_XmlRpc`.

11.3 Zend_XmlRpc

We already described how XML-RPC makes XML encoded remote procedure calls through HTTP requests and responses. The relative youth of XML might suggest this is yet another new technology being thrust upon us by marketing departments, but remote procedure calls are not a new concept. Written over thirty years ago, the memo RFC 707: "A High-Level Framework for Network-Based Resource Sharing", describes the RPC protocol in a slightly quaint way:

"Given such a protocol, the various remote resources upon which a user might wish to draw can indeed be made to appear as a single, coherent workshop by interposing between him and them a command language interpreter that transforms his commands into the appropriate protocol utterances"

RFC# 707: A High-Level Framework for Network-Based Resource Sharing, 14th Jan 1976

The start of that document makes an interesting challenge to ARPANET, the predecessor to today's internet, in the the sentence beginning "This paper outlines an alternative to the approach that ARPANET system builders have been taking". While it is interesting to consider how the internet would be now had that

alternative approach been taken, the key point is that RPC is one solution among many, including the internet itself, to the need to have disparate applications talking to each other.

Just as a further note, because RPC mediates between one application and another, it can be classified under the term middleware, which, partly due to the fact that it sounds like marketing-speak, isn't particularly interesting until you notice that amongst the others in that classification is SQL.

Clearly XML-RPC has therefore enough credentials to add to any proposal: It is based on technology established over thirty years ago, which was partly proposed as an alternative to today's internet and shares the same problem solving area as the language through which we converse with databases. Thus having determined that XML-RPC has a suitable lineage and with the addition of XML, enough youth to keep it vibrant, we can move on to actually using Zend Framework's implementation.

The example we are going to work through is an implementation of the various blog application programming interfaces (API) that allow blog editors an alternative method of adding, editing and deleting blog entries using desktop or other remote applications. We will start by setting up an XML-RPC server using `Zend_XmlRpc_Server`.

11.3.1 *Zend_XmlRpc_Server*

`Zend_XmlRpc_Server` is used to implement the single point of entry for XML-RPC requests and in that respect acts in much the same way that Zend Framework's Front Controller does. In fact you can make your XML-RPC server a controller action that receives its request via the front controller but that would involve a lot of unnecessary processing overhead. Instead we're going to separate out the bootstrapping process a little.

Setting up the bootstrapping

If you've read previous chapters, you'll already be aware that the Zend Framework MVC structure relies on `mod_rewrite` settings in a `.htaccess` file to pass all requests via a front controller file like `index.php`. Since our XML-RPC server has its own single point of entry we need to exclude it from that rewrite rule. In listing 11.2 we do that by adding a rewrite condition that excludes any requests for `/xmlrpc` from the final rewrite rule that passes requests to `index.php`.

Listing 11.2 Rewrite rules modified to allow requests through to the XML-RPC server

```
RewriteEngine on
RewriteCond %{REQUEST_URI} !^/css
RewriteCond %{REQUEST_URI} !^/img
RewriteCond %{REQUEST_URI} !^/js
RewriteCond %{REQUEST_URI} !^/xmlrpc
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php/$1
```

A The rewrite condition that excludes the `/xmlrpc/` directory from the final rewrite rule

Alternatively, in figure 11.4 we have illustrated a solution which you will appreciate if, like me, you are a coward when it comes to the mystical art of `mod_rewrite`.

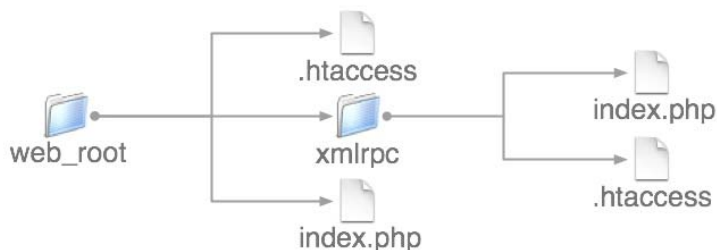


Figure 11.4 The directory structure with our xmlrpc controller file

What we've done is added a .htaccess file with one line within our xmlrpc directory:

```
RewriteEngine off
```

This means we can drop our xmlrpc directory into any of the applications we are working on and any requests to /xmlrpc/ will go to our xmlrpc/index.php file. This means that we don't have to tamper with what could be a finely tuned .htaccess file belonging to the main application or several applications each with potentially varying rewrite settings.

Now that requests are successfully getting to index.php we can add the code to forward those requests on to the bootstrap file in our application directory. Readers of the previous chapter will already be familiar with this setup and note the call to a new method; runXmlRpc() shown in listing 11.3.

Listing 11.3 The contents of our xmlrpc/index.php

```
include '../application/bootstrap.php';  
$bootstrap = new Bootstrap('general');  
$bootstrap->runXmlRpc();
```

A Set the configuration section to use
B Run the xmlrpc function in the bootstrap

Having got all that together we're now finally able to get to the main topic of this section; using Zend_XmlRpc_Server. Listing 11.4 shows a stripped down version of the contents of the bootstrap file which includes the runXmlRpc() method that was called from our index.php file.

Listing 11.4 Zend_XmlRpc_Server used in our bootstrap file

```
class Bootstrap  
{  
    public function __construct($deploymentEnvironment)  
    {  
        // Set up any configuration settings here  
    }  
  
    public function runXmlRpc()  
    {  
        $server = new Zend_XmlRpc_Server();  
  
        require_once 'Blogger.php';  
        require_once 'Metaweblog.php';  
        require_once 'MovableType.php';  
  
        $server->setClass('Blogger', 'blogger');  
        $server->setClass('Metaweblog', 'metaWeblog');  
        $server->setClass('MovableType', 'mt');  
  
        $response = $server->handle();  
        $response->setEncoding('UTF-8');  
        header('Content-Type: text/xml; charset=UTF-8');  
        echo $response;  
    }  
}
```

A Application initialisation occurs here
B Initialise the XML-RPC server
C Include any class files that will be attached to the server
D Attach class methods as XMLRPC method handlers together with their namespaces
E Handle the remote procedure call
F Set the character encoding of the response and echo that when setting the HTTP header string
G Output the response

We've intentionally left out the contents of the constructor in this example but have included the method to show where things like introducing configuration settings, setting include paths, establishing a database connection and any other application specific initialisation would occur. Setting up the server is then a fairly straightforward process of instantiating the server object, supplying that server with the class methods that will become its method handlers, handling the XML remote procedure call and echoing back the response.

One thing that you might notice is when we set the method handlers we also passed in a namespace string. The example classes that we are working with demonstrate exactly why this is important, as both the Blogger and Metaweblog classes contain an editPost() method which would clash without the ability to namespace them as

metaWeblog.editPost and blogger.editPost.

With the server setup and ready to go we can now elaborate on the class methods that the server will be using.

Creating the XML-RPC method handlers

Since the purpose of our XML-RPC server is to receive remote procedure calls, our next step is to create those procedures. As already mentioned, we will be implementing some of the API's (Application Programming Interfaces) for several of the various weblog applications so that a desktop application can work with articles on our Places application.

First let's take a look at the application we will be working with; Ecto, which describes itself as "a feature-rich desktop blogging client for MacOSX and Windows, supporting a wide range of weblog systems". The main reason we chose Ecto was because it has a "console" which makes it very useful when debugging our XML-RPC transactions, however most of what we discuss in this section is applicable to other similar applications. In Figure 11.5 we can see Ecto's main window on the left, with a list of Places articles and its editing window on the right.

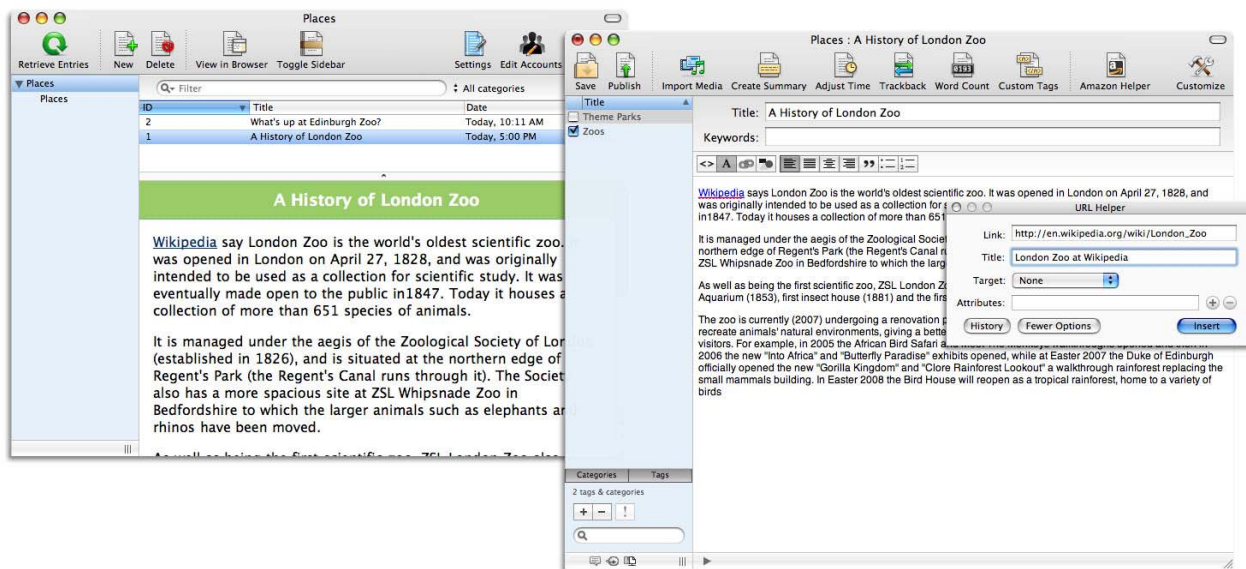


Figure 11.5 Ecto, the desktop blogging client we will be using to make XML-RPC requests, shown editing some Places content

After setting up the connection details and selecting the API we want to connect to, Ecto runs through a series of method calls to establish the account. In our case we have chosen to use the MovableType API, however doing so still involves calls to methods that belong to other API's, such as blogger.getUsersBlogs and blogger.getUserInfo which belong to the Blogger API or metaWeblog.editPost which belongs to the MetaWeblog API. For this reason we're obligated to provide those methods for our XML-RPC server.

Creating the interfaces

In order to work with any of the API's already mentioned, our application must implement the methods required by those API's. If we pick an example method like `metaWeblog.editPost` the requirement is that it returns a boolean value and consumes the following parameters:

```
metaWeblog.editPost (postid, username, password, struct, publish)
```

How our application actually carries out the processing of this request is dependant on the application itself, but the fact that it must implement the required methods clearly indicates an appropriate use for object interfaces. According to the PHP manual “object interfaces allow you to create code which specifies which methods a class must implement, without having to define how these methods are handled”. Listing 11.5 shows one of our interfaces establishing all the methods of the Metaweblog API.

Listing 11.5 Our interface for the MetaWeblog API

```
interface Places_Service_Metaweblog_Interface
{
    public function newPost($blogid, $username, $password, $content, $publish);
    public function editPost($postid, $username, $password, $content, $publish);
    public function getPost($postid, $username, $password);
    public function newMediaObject ($blogid, $username, $password, $struct);
    public function getCategories($blogid, $username, $password);
    public function getRecentPosts($blogid, $username, $password, $numposts);
}
```

We should note that this interface was quite straightforward to set up as it's clearly detailed in the MetaWeblog spec. The XML-RPC Blogger API and Movable Type API's are a bit more fiddly as they have actually been deprecated despite still being used in many current web and desktop applications.

Having set up an example interface we can then use it in our concrete classes where it will ensure that those classes will adhere to the requirements of the original API.

Creating the concrete classes

Since our original shot of Ecto in figure 11.5 showed us editing an article on Places we'll continue that theme and use the MetaWeblog `editPost()` method to demonstrate both the use of our interface and how methods need to be set up so `Zend_XmlRpc_Server` can use them.

Listing 11.6 shows a stripped down version of the Metaweblog class which resides in our models/ directory. The final version would, of course, have to include all the methods in the interface it implements, but for the sake of space they have not been included here.

Listing 11.6 Our MetaWeblog model implementing the interface from listing 11.4

```
include_once 'ArticleTable.php';
include_once 'ServiceAuth.php';

class Metaweblog implements Places_Service_Metaweblog_Interface
{
    protected $_auth;
    protected $_articleTable;

    public function __construct()
    {
        $this->_auth = new ServiceAuth;
        $this->_articleTable = new ArticleTable();
        // Full version would have further code here...
    }

    /**
     * Changes the articles of a given post
     * Optionally, publishes after changing the post
     *
     * @param string $postid Unique identifier of the post to be changed
     * @param string $username Login for a user who has permission to edit
```

```

* the given post (either the original creator or an admin of the blog)
* @param string $password Password for said username
* @param struct $struct New content of the post
* @param boolean $publish If true, the blog will be published
* immediately after the post is made
* @return boolean D
*/
public function editPost($postId, $username, $password, $struct, $publish) E
{
    $identity = $this->_auth->authenticate($username, $password); F
    if(false === $identity) {
        throw new Exception('Authentication Failed');
    }

    // Set up filters
    $filterInt = new Zend_Filter_Int; G
    $filterStripTags = new Zend_Filter_StripTags;

    $id = $filterInt->filter($postId);

    $data = array( H
        'publish' => $filterInt->filter($struct['publish']),
        'title' => $filterStripTags->filter($struct['title']),
        'body' => $struct['description']
    );

    $where = $this->_articleTable->getAdapter()->quoteInto('id = ?', $id);
    $rows_affected = $this->_articleTable->update($data, $where); I

    if(0 == $rows_affected) {
        throw new Exception('Somehow your post failed to be updated');
    }

    return true; J
}
// Add all the other required methods here... K
}

```

A This model class implements our Metaweblog interface

B Instantiating the objects we will need for our editPost method. Note the full version of this class would do more here.

C Establishing the method parameters.

Docblocks are the key to getting methods working with Zend_XmlRpc_Server. We'll go into more detail in this section but their key role is to determine the method help text and method signatures. In our example you can see they indicate the type, variable name and description of each parameter.

D The docblock return value is also a requirement.

E The parameters must match our interface and the docblocks above.

F Here we are doing some rudimentary security checks with a custom auth class and throwing an exception if they fail.

G Filtering the data is important for security as it is in most cases.

H The \$struct array is used to build the data to be used in the update query.

I The post is updated

J As specified in the return value of the docblock we return a boolean on success.

K The full version would be required to implement all methods in the interface.

After a simple authentication check using the username and password passed in the parameters this editPost() method filters and formats an array from the received data and updates the database row that corresponds with the provided id. On success it returns the required boolean and on failure an exception is thrown that will be handled by Zend_XmlRpc_Server.

You'll notice that the editPost() method is really only minimally different from any standard method in that it has the docblock parameter data type "struct", which isn't a native PHP data type. The reason for this is that when you use Zend_XmlRpc_Server::setClass() or Zend_XmlRpc_Server::addFunction(), Zend_Server_Reflection checks all methods or functions and determines their method help text and signatures using the docblocks.

Referring to table 11.1 we can see that in the case of the @param \$struct the datatype has been set to the XML-RPC type “struct” that corresponds with the PHP type “associative array” which is processed using the Zend_XmlRpc_Value object Zend_XmlRpc_Value_Struct.

Using these mappings we can at any time call on an Zend_XmlRpc_Value object to prepare values for XML-RPC such as is often needed when preparing values in an array, for example:

```
array('dateCreated' => new Zend_XmlRpc_Value_DateTime(
    $row->date_created, Zend_XmlRpc_Value::XMLRPC_TYPE_DATETIME);
```

The above example formats a date from a database table row into the ISO8601 format required by XML-RPC. The second parameter refers to the class constant XMLRPC_TYPE_DATETIME which is unsurprisingly defined as “dateTime.iso8601”.

PHP Native Type	XML-RPC Type	Zend_XmlRpc_Value Object
boolean	<boolean>	Zend_XmlRpc_Value_Boolean
integer	<int> or <i4>	Zend_XmlRpc_Value_Integer
double	<double>	Zend_XmlRpc_Value_Double
string	<string> (the default type)	Zend_XmlRpc_Value_String
dateTime.iso8601	<dateTime.iso8601>	Zend_XmlRpc_Value_DateTime
base64	<base64>	Zend_XmlRpc_Value_Base64
array	<array>	Zend_XmlRpc_Value_Array
associative array	<struct>	Zend_XmlRpc_Value_Struct

Table 11.1 Mapping PHP native types against their XML-RPC Type and the corresponding Zend_XmlRpc_Value object

If you are thinking that all this introspection by Zend_Server_Reflection must come at a price then you would be right, particularly when a lot of classes or functions are attached to the server. Thankfully there is a solution in the form of Zend_XmlRpc_Server_Cache which, as the name implies, can be used to cache the information gathered by Zend_Server_Reflection. We don’t need to vary too much from the example given in the Zend Framework manual as its use is really very simple as we can see in listing 11.7.

Listing 11.7 Our Zend_XmlRpc_Server with caching implemented

```
$cacheFile = ROOT_DIR . '/cache/xmlrpc.cache';           A
$server = new Zend_XmlRpc_Server();

if (!Zend_XmlRpc_Server_Cache::get($cacheFile, $server)) {           B
    require_once 'Blogger.php';
    require_once 'Metaweblog.php';
    require_once 'MovableType.php';

    $server->setClass('Blogger', 'blogger');
    $server->setClass('Metaweblog', 'metaWeblog');
    $server->setClass('MovableType', 'mt');

    Zend_XmlRpc_Server_Cache::save($cacheFile, $server);           C
}
```

A The file to use to store the cached information

B Checking for the presence of a cache file before attaching classes

C If there was no cache file a new one is saved with the introspection information

With Zend_XmlRpc_Server_Cache in place we can cut out all that resource intensive introspection and as long as we don’t need to change the attached classes we can leave it as it is. If we do change any of those classes we need only delete the cache file so that a new version can be rewritten that reflects the changes.

Now that we’ve set-up our XML-RPC server we can look at the client side of the exchange with Zend_XmlRpc_Client.

11.3.4 Zend_XmlRpc_Client

For our Places application our intention was to set-up an XML-RPC server so that we could remotely edit articles with any of the desktop applications that support the weblog API’s. In doing so we have covered a lot

of the functionality of Zend_XmlRpc already. In this section we'll demonstrate those a little further by simulating a client request to the editPost() method we demonstrated in listing 11.6 using Zend_XmlRpc_Client.

Unlike Zend_XmlRpc_Server which was used within its own front controller we are using Zend_XmlRpc_Client within a controller action as shown in listing 11.8.

Listing 11.8 Zend_XmlRpc_Client used within a controller action

```
public function editPostAction()
{
    $xmlRpcServer = 'http://places/xmlrpc/';           A
    $client = new Zend_XmlRpc_Client($xmlRpcServer);   B

    // Set up filters
    $filterInt = new Zend_Filter_Int;                 C
    $filterStripTags = new Zend_Filter_StripTags;

    $id = $filterInt->filter($_POST['id']);
    $publish = (bool) $_POST['publish'];

    $structData = array(                             D
        'title' => $filterStripTags->filter($_POST['title']),
        'dateCreated' => new Zend_XmlRpc_Value_DateTime(time(),
            Zend_XmlRpc_Value::XMLRPC_TYPE_DATETIME),
        'description' => $filterStripTags->filter($_POST['body'])
    );
    $client->call('metaWeblog.editPost',
        array($id, 'myusername', 'mypassword', $structData, $publish)); E
}
```

A Assign the URL of the XML-RPC server that the client will send its requests to

B Instantiate the client object with the server URL

C Filter input from the HTML form

D Setup the data that will become an XML-RPC struct

E Make the remote procedure call with the assembled data

In this example we are filtering the data from an HTML form and preparing it for use by the client that has been set-up with the URL of the XML-RPC server we will be sending requests to. The Zend_XmlRpc_Client::call() method then handles the rest by compiling the XML encoded request and sending it via HTTP to the XML-RPC server it was instantiated with.

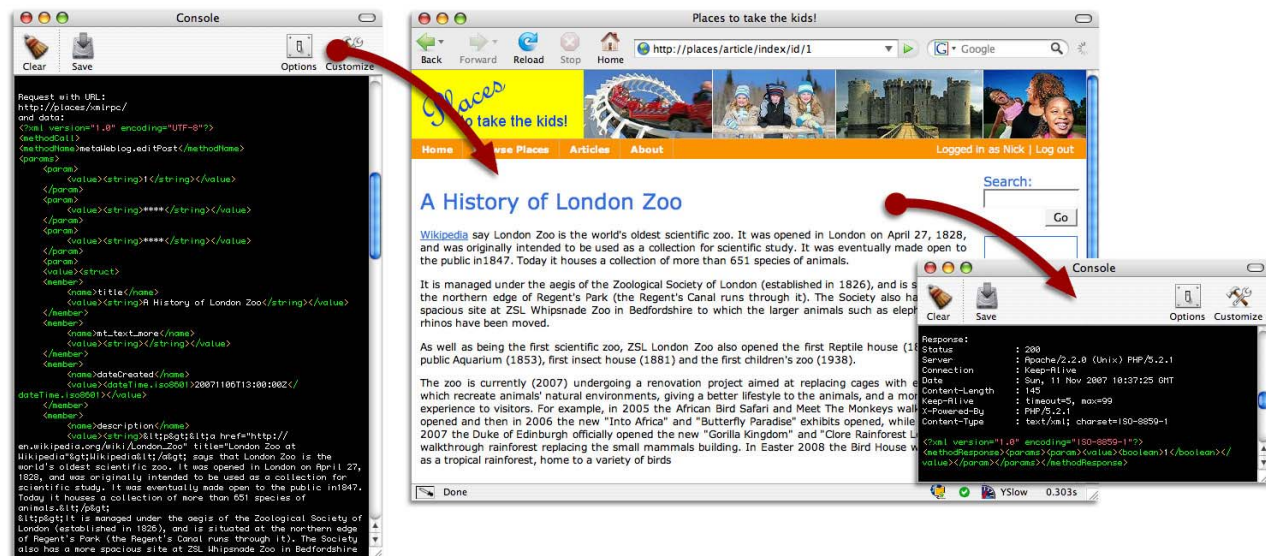


Figure 11.6 Demonstrating the editPost() method call with the response on the left, the updated article in the middle and the response returned from the XML-RPC server on the right.

Borrowing Ecto's console window we can illustrate the process a little more thoroughly in figure 11.6. There we can see the XML encoded request in the left hand console window, which is processed by the XML-RPC server and if successful will update the article in the middle and return an XML encoded response with the boolean value true, back to the client.

Not all XML-RPC requests are successful of course and for that reason `Zend_XmlRpc_Client` has the ability to handle HTTP and XML-RPC errors through exceptions. In listing 11.9 we add some error handling to the `editPost` procedure call from listing 11.8.

Listing 11.9 Adding error handling to our XML-RPC client

```
try {  
    client->call('metaWeblog.editPost',  
        array($id, 'myusername', 'mypassword', $structData, $publish));  
} catch (Zend_XmlRpc_HttpException $e) {  
    // echo $e->getCode() . ': ' . $e->getMessage() . "\n";  
} catch (Zend_XmlRpc_FaultException $e) {  
    // echo $e->getCode() . ': ' . $e->getMessage() . "\n";  
}
```

A First we try the remote procedure call

B If the call fails we handle any HTTP request error

B If the error was not an HTTP error we handle any XML-RPC error

Our client will now attempt to make the remote procedure call and on failure will be able to handle HTTP and XML-RPC errors respectively. Note that since we're not actually using the client in our Places application we've not gone into any detail about what we'd do with those error messages.

As we've hopefully demonstrated in this section, XML-RPC is really quite straightforward to set-up and use and `Zend_XmlRpc` makes it even easier. However, like any technology, XML-RPC has its critics and in the next section we will look at another approach to web services with `Zend Rest`.

11.4 Zend_Rest

At the start of this chapter we mentioned that the various Zend Framework components we'd be working with would include XML but that we wouldn't be needing to deal with it directly. The `Zend_Rest` section of the manual opens by saying "REST Web Services use service-specific XML formats" which is true, but needs a little clarification as REST web services don't actually care whether they use XML or not. `Zend_Rest`, does use XML as its format of choice but it is possible to circumvent this if you dig around a bit which we'll do after looking at REST in a bit more detail.

11.4.1 What is REST?

REST stands for Representational State Transfer and was originally outlined in "Architectural Styles and the Design of Network-based Software Architectures" by Roy Fielding, whose role as "primary architect of the current Hypertext Transfer Protocol" explains some of the background to REST. With regards to the use of XML; "REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.". An "A-ha!" moment came for me when I read the line in the document that says "the motivation for developing REST was to create an architectural model for how the Web should work, such that it could serve as the guiding framework for the Web protocol standards". In other words, every time we make an HTTP request we are using a transfer protocol based on the REST concept.

While the key element of RPC is the command, usually accessed via a single point of entry, the key element in REST is the resource, an example of which could be Places login page whose resource identifier is the URL `http://places/auth/login/`. Resources can change over time, for example our login page could have interface updates, but their resource identifiers should remain valid.

Of course having a lot of resources doesn't hold a great deal of value unless we can actually do something with them and in the case of web services we have the HTTP methods. To illustrate the value that such apparently simple operations have Table 11.2 compares the HTTP methods POST, GET, PUT and DELETE with the common generic database operations; Create, Read, Update and Delete (CRUD).

HTTP Methods	Database Operations
POST	Create
GET	Read
PUT	Update
DELETE	Delete

Table 11.2 Comparing the HTTP Methods used in REST with the common generic database operations

All readers of this book will be familiar with the POST and GET HTTP request methods whereas PUT and DELETE are less well known, partly as they are not often implemented by all HTTP servers. This limitation is reflected in Zend_Rest with the fact that Zend_Rest_Client is able to send all of these request methods while Zend_Rest_Server will only respond to GET and POST.

Presuming that readers will already have enough knowledge of HTTP to be able to “join the dots” on the fundamentals, we have kept this introduction to REST intentionally brief. Hopefully, as we start to explore some of the REST components of the Zend Framework, things will become clearer. That exploration will start with Zend_Rest_Client

11.4.1 Zend_Rest_Client

The start of this section mentioned that Zend_Rest uses XML to serialize the data it processes in the body of the HTTP request or response, however, not all RESTful web services use XML. One example is the Akismet spam filtering service which we'll use to demonstrate the various ways of accessing REST based web services. Figure 11.6 shows a good example of where we could use the service to check the reviews submitted by users of our Places application.

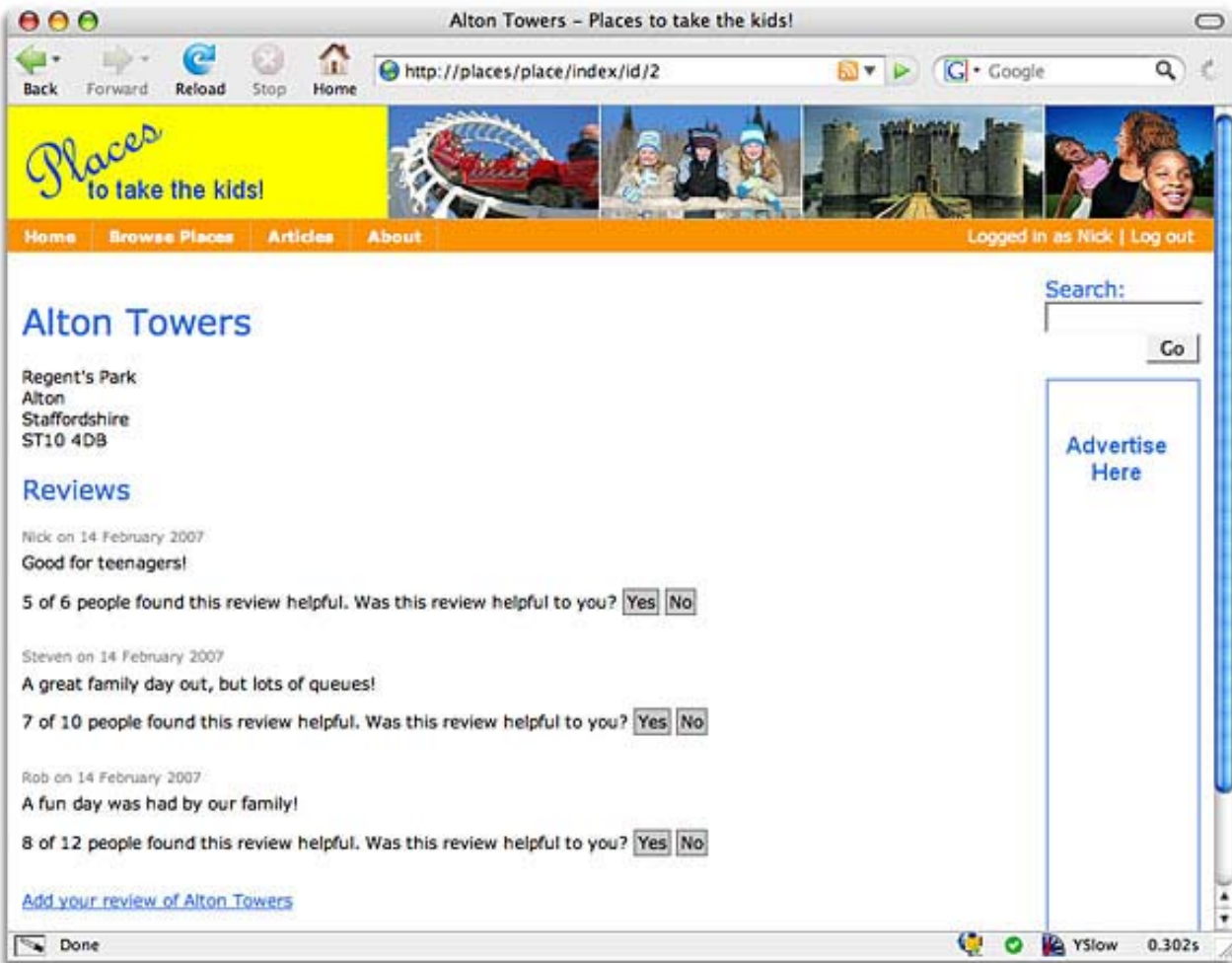


Figure 11.6 The reviews in our Places application which we could filter using the Akismet spam filter service

Another reason we've chosen Akismet is that Zend Framework has a `Zend_Service_Akismet` component which means readers won't be left with a partially implemented solution but will hopefully be able to understand how that component works a little more.

Resource Identifier	Description
<code>rest.akismet.com/1.1/verify-key</code>	Verify the required API key
<code>api-key.rest.akismet.com/1.1/comment-check</code>	Identify the submitted comment is spam or not
<code>api-key.rest.akismet.com/1.1/submit-spam</code>	Submitting missed spam
<code>api-key.rest.akismet.com/1.1/submit-ham</code>	Submitting content incorrectly marked as spam

Table 11.3 The resource provided by the Akismet API

The Akismet API provides a small choice of resources, shown in table 11.3, from which we will choose the resource to verify that the API key obtained from Wordpress.com is valid. The first thing we will do is attempt to use `Zend_Rest` to access this resource in the method demonstrated in the manual as we have in listing 11.10.

Listing 11.10 A REST request to Akismet using `Zend_Http_Client`

```
$client = new Zend_Rest_Client('http://rest.akismet.com/1.1/verify-key');
echo $client->key('f6k3apik3y');
    ->blog('http://places/')
    ->post();
```

A
B
C
D

A Instantiate the client with the URL of the resource
B Set the key 'key' to the value of the API key

C Set the key 'blog' to the URL of the blog
D Send the request as an HTTP POST

In this example we construct our client with the URL of the verify key resource then, using the fluent interface that is common in many Zend Framework components, we set the two required variables “key” and “blog” using the built in magic methods, finally we send them via an HTTP POST request. Unfortunately, despite the pleasing simplicity of this code the resulting request will fail because Akismet doesn’t return the XML response that is expected by Zend_Rest_Client when used this way.

Since all Akismet needs is a simple HTTP request and Zend_Rest itself uses Zend_Http_Client why then can’t we just use Zend_Http_Client in the first place? The answer, as demonstrated in listing 11.11, is that we can.

Listing 11.11 A REST request to Akismet using Zend_Http_Client

```
$client = new Zend_Http_Client('http://rest.akismet.com/1.1/verify-key'); A
$data = array( B
    'key' => 'f6k3apik3y',
    'blog' => 'http://places/'
);

$client->setParameterPost($data); C

try {
    $response = $client->request(Zend_Http_Client::POST); D
    var_dump($response);
} catch (Zend_Http_Client_Exception $e) {
    echo $e->getCode() . ': ' . $e->getMessage() . "\n";
}
```

A Instantiate the client with the URL of the resource

B Build an array with the data required for the verify key request

C Format the data into an HTTP POST

D Make the request (wrapped in a try/catch exception block)

Zend_Http_Client will not attempt to parse the returned response from Akismet as if it was XML and we will receive the plain text response “valid” or “invalid” depending on whether our data is verified successfully or not, respectively.

If Zend_Http_Client can successfully make the request surely there is a way that Zend_Rest can do the same? Of course there is and in listing 11.12, which should be looking familiar by now, we bypass having Akismet’s plain text response parsed as XML by calling the restPost() method directly. Unlike our first attempt this method returns the body of the HTTP response rather than sending it through Zend_Rest_Client_Result.

Listing 11.12 A REST request to Akismet using Zend_Rest

```
$client = new Zend_Rest_Client('http://rest.akismet.com'); A
$data = array( B
    'key' => 'f6k3apik3y',
    'blog' => 'http://places/'
);

try {
    $response = $client->restPost('/1.1/verify-key', $data); C
    var_dump($response);
} catch (Zend_Rest_Client_Exception $e) {
    echo $e->getCode() . ': ' . $e->getMessage() . "\n";
}
```

A Instantiate the client with the URL to make requests to

B Build an array with the data required for the verify key request

C Make the HTTP POST request with the path of the resource and data to verify

Having solved the problem we were having with Akismet’s service we now know we can use Zend_Rest_Client with plain text and XML based RESTful web services. If we wanted to work with the rest of Akismet’s resources it would obviously make more sense to use Zend_Service_Akismet, but if there was not a pre-built Zend Framework component we are now aware of several options. One of those is the option to interact with REST based web services provided by our own application server using Zend_Rest_Server.

11.4.1 Zend_Rest_Server

Let's imagine that we've convinced one of the advertisers on Places; Edinburgh Zoo, to take part a joint promotion that will be hosted separately from both of our sites. The idea is to make a short lifespan "mash-up" site based on content from Places, Edinburgh Zoo and other interested sites.

As we did with our XML-RPC server earlier we will start by making a simple interface to make sure our server has a consistent API. Listing 11.13 shows our interface with two methods; one to get a place and one to get the reviews for a place.

Listing 11.13 Our Places service application interface

```
interface Places_Service_Place_Interface
{
    public function getPlace($id);
    public function getReviews($id);
}
```

Next in listing 11.14 we make a concrete implement of our interface using queries similar to those from our Zend_Db chapter. Note that the database results are returned as an array rather than the default objects which would have failed when processed by Zend_Rest_Server. Another thing that you may notice is that unlike Zend_XmlRpc_Server, Zend_Rest_Server does not require parameters and return values specified in docblocks.

Listing 11.14 Our Places service concrete class

```
class ServicePlaces implements Places_Service_Place_Interface
{
    public function getPlace($id)
    {
        $placesFinder = new Places();
        $place = $placesFinder->find($id);
        return $place->current()->toArray();
    }

    public function getReviews($id)
    {
        $reviewsFinder = new Reviews();
        $rowset = $reviewsFinder->fetchByPlaceId($id);
        return $rowset->toArray();
    }
}
```

A Note that we change the query results from an object to an array as Zend_Rest_Server will not process the object

Now that we have the concrete class we can attach it to Zend_Rest_Server with the same approach that we covered with Zend_XmlRpc_Server.

Listing 11.15 Our REST server

```
class RestController extends Zend_Controller_Action
{
    protected $_server;

    public function init()
    {
        $this->_server = new Zend_Rest_Server();
        $this->_helper->viewRenderer->setNoRender();
    }

    public function indexAction()
    {
        require_once 'ServicePlaces.php';
        $this->_server->setClass('ServicePlaces');
        $this->_server->handle();
    }
}
```

Listing 11.15 shows our REST server set-up within an action controller and accessible via an HTTP GET or POST that must supply the name of the service method you wish to invoke. Figure 11.7 shows the XML formatted results of some example GET requests to the resources <http://places/rest/?method=getPlace&id=6> (on the left) and <http://places/rest/?method=getReviews&id=6> (on the right) using Firefox.

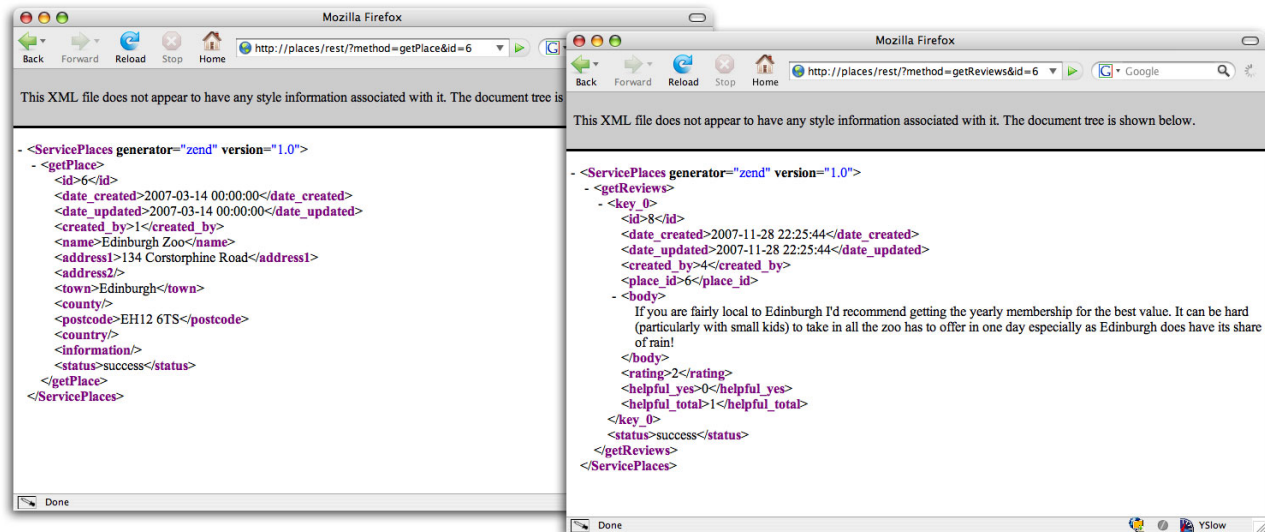


Figure 11.7 The results of our REST server query; `getPlace` on the left and `getReviews` on the right

All our mashup site needs to do is use `Zend_Rest_Client` to make requests like the following:

```
$client = new Zend_Rest_Client('http://places/rest/?method=getPlace');
$response = $client->id('6')->get();
```

Which will return a `Zend_Rest_Client_Result` object that allows us to access the elements of the response as properties that can be used in our site like so:

```
echo $response->name; // Outputs "Edinburgh Zoo"
```

Having worked through the implementation of `Zend_XmlRpc_Server`, itself relatively simple when compared to say SOAP, you will hopefully have found `Zend_Rest_Server` very easy to follow. As with any brief introduction there is a lot that has been left untouched such as PUT and DELETE HTTP requests which are not handled by `Zend_Rest_Server` and authentication for example. However, what we have set-up is a REST server whose strength lies in the relationship between `Zend_Rest_Server` and `Zend_Rest_Client` and the simplicity of implementation.

11.5 Summary

This chapter has been quite a concentrated look at some of the web service components of the Zend Framework. The emphasis has been on the client/server relationships which we set up first through using `Zend_Feed` to generate a newsfeed of our Places articles and then consume that newsfeed. Next we set-up `Zend_XmlRpc_Server` to allow remote editing of Places articles through the weblog API's and followed that with an example of how `Zend_XmlRpc_Client` can make the remote procedure calls to that server. Finally we covered `Zend_Rest`; first to filter comments through Akismet's REST based spam filtering service using `Zend_Rest_Client` and then to provide our own API to Places content with `Zend_Rest_Server`.

Hopefully you should leave this chapter with a decent understanding of how web services work and be able to implement useful services using the various Zend Framework components. You should also be well prepared for the next chapter where we will use the more specific components with some of the publicly available web services.

Internationalization & Localization

Most websites are written in a single language for a single country and this makes life easier for both the designers and developers. Some projects however require more than this. For instance, some countries have more than one language (e.g. in Wales, both English and Welsh are used) and some websites are intended to target all the countries that the company operates in. To create a website targeted at different countries and cultures, significant changes to the application are required in terms of supporting this and also the different formats of dates, times, currency, etc. which each country uses.

We are going to look at what needs to be done to make a multi-lingual website and then show how the `Zend_Locale` and `Zend_Translate` components of the Zend Framework help to make the process easier. Finally we will implement a second language into the *Places* website to show how to practically achieve a localized application.

14.1 What do we need to do?

Before making a multi-lingual website, we first need to look at what needs to be done. There are two main areas to consider: language and customs. Intuitively, most people think about changing the language when they consider supporting another country on their website; clearly we need to display all the text in the correct language. However, for some locales, there are cultural issues to consider too. The most common are the formatting of dates and currency. For dates, the most infamous issue is that the US uses mm/dd/yy whereas the UK uses dd/mm/yy and hence it gets tricky to determine what date 02/03/08 actually is. Is it the 2nd March or the 3rd February? Similarly, for currency, in France, they use the comma where the UK uses a decimal place and they use a decimal place where we would use a comma. To make a French user feel at home, €1,234.56 should be displayed as €1 234,56.

The key controller on a computer system for this is called the locale. The locale is a string that defines the current language and region used by the user. For example, the locale “en_GB” means English language, in the GB region. Similarly, “es_PR” is Spanish language in Puerto Rico. Generally, for language localization, the first part of the locale is used as it is rare to find a website that provides both US and UK English.

14.1.1 Translating languages

Translation of languages involves changes to both the HTML design and build of the website site and to the PHP code that runs it. The most obvious change that’s required is that every string displayed to the user has to be in the correct language and so correct sizing of areas for text in the design is required. This also includes any text that is embedded in a graphical image and so in order to provide for multiple languages, the image files need to be separated out into generic and language specific if text is used in images.

There are multiple methods of doing the actual translation but they all boil down to the same thing. Every string displayed on the site needs to be mapped to a string in the target language. It follows that it is important that the strings are rewritten using a professional translator as there is rarely a one-to-one mapping from one language to the next. Industry standard systems such as gettext() have many tools available dedicated to

making it simple for a translator to perform the translation of phrases used in the application without having to know anything about programming.

14.1.2 Translating Idioms

The most obvious idioms that need translation are the formatting of currency and dates. PHP has locale support built in that is set using the `setlocale()` function. Once the locale is set, then all the locale-aware functions will use it. This means that `strftime()` will use the correct language for the months of the year and `money_format()` will use the comma and period characters in the right place for the language involved. One gotcha is that `setlocale()` is not thread-safe and the strings you need to set are inconsistent across operating systems, so care must be taken using it. Also, some functions like `money_format()` aren't available on all operating systems, for example Windows. The `Zend_Locale` component is intended to mitigate these issues and also provides additional functionality like normalization.

Now that we know a little about what localization and internationalization is, let's look at what the Zend Framework provides to make the process of creating an international website easier. We will start by looking at `Zend_Locale`'s ability to convert numbers and dates before moving on to investigate `Zend_Translate`'s functionality for providing translated text.

14.2 Using Zend_Locale and Zend_Translate

`Zend_Locale` and `Zend_Translate` are the key components of the Zend Framework for providing a multi-lingual worldwide website. Other components that are aware of localization are `Zend_Date`, `Zend_Calendar` and `Zend_Currency`. Let's look at `Zend_Locale` first.

14.2.1 Zend_Locale

To select the correct locale is as easy as:

```
$locale = new Zend_Locale('en_GB');
```

This will create a locale object for the English language, GB region. We can also create a `Zend_Locale` object for the locale of the user's browser using:

```
$locale = new Zend_Locale();
```

The locale object can then be used for translating lists of common strings such as countries, units of measurement and time information such as month names and days of the week. We can also retrieve the language and region using:

```
$language = $locale->getLanguage();  
$region = $locale->getRegion();
```

Clearly, we can then use this information to provide websites in the correct language with the right formatting of dates, times and currencies which will make our user feel right at home. Let's look at numbers first.

Dealing with numbers

The most significant regional problem with numbers are that some countries use the comma to separate out the decimal places from the whole number and some countries use the point character. If your website allows the user to enter a number, then you may have to convert appropriately. This is known as normalization. For example, consider a form that asks someone to enter their monthly insurance costs in order to try and provide a cheaper quotation. A German user may type in the number 3.637,34 (three thousand, six hundred and thirty seven Euros and thirty four cents) which you need normalize to 3637.34. This is achieved using the code shown in listing 14.1.

Listing 14.1: Number normalization with Zend_Locale

```
$locale = new Zend_Locale('de_DE');           A
$number = Zend_Locale_Format::getNumber('3.637,34',
    'locale' => $locale);

print $number;                                B
```

A German locale
B Prints the number "3637.34"

We can then process the number as appropriate and may need to display a number to the user. In this case, again we need to format the number appropriately for the user's location and we can use Zend_Locale's `toNumber()` function as shown in listing 14.2..

Listing 14.2: Number localization with Zend_Locale

```
$locale = new Zend_Locale('de_DE');           A
$number = Zend_Locale_Format::toNumber(2435.837,
    array('precision' => 2,                     B
    'locale' => $locale));

print $number;                                C
```

A German locale
B Round to 2 decimal places
C Prints the number "2.435,84"

The precision parameter is optional and is used to round the provided number to the given number of decimal places. This covers the basics of what Zend_Locale can do with numbers. However, Zend_Locale provides complete number handling though including translation of numbers between different numeral systems, for example from Arabic to Latin. There is also support for integer and floating point number normalization and localization. The manual gives full information on these functions.

Date and Time with Zend_Locale

Handling the formatting of dates and times is also within the province of Zend_Locale. This class operates in conjunction with Zend_Date to provide comprehensive support for reading and writing dates. Let's start by looking at normalizing dates because, like numbers, different regions of the world write their dates in difference formats and obviously, use their local language for the names of the months and days of the week.

Consider, the date 2nd March 2007. In the UK, this may be written as 2/3/1007 whereas in the USA it would be written 3/2/2007. To use the date supplied by our users, we need to normalize it and `getDate()` is the function to use as shown in listing 14.3..

Listing 14.3: Date normalization with Zend_Locale

```

$locale = new Zend_Locale('en_US');
$date = Zend_Locale_Format::getDate('3/2/2007',
    array('locale' => $locale));

print $date['month'];

```

A
B

A USA locale
B Prints "3" for March

As usual, we create a locale object for the correct language and region and then use it with the `getDate()` function. We have used the `en_US` locale for the USA and so `getDate()` correctly determines that the month is March. If we changed the locale to `en_GB`, then the month would be February. Similarly, we can use `checkDateFormat()` to ensure that the date string received is valid for the locale and obviously, once we have the date information separated into its components, we can manipulate it in any way we like.

Now that we know the basics of using `Zend_Locale` to help our international visitors feel at home, let's have a look at `Zend_Translate`'s ability to help us present our site in different languages.

14.2.2 Zend_Translate

As we have already seen, the basics of website translation is to take a every string and provide a translated version. The most common way to do this is `gettext()`, which is powerful but fairly complicated. `Zend_Translate` supports the `gettext()` format, but also supports other popular formats such as CSV, TBX, Qt and Xliff. `Zend_Translate` is also thread-safe which can be very helpful if you are running a multi-threaded webserver such as IIS.

`Zend Translate` supports multiple input formats using an adapter system. This approach is very common in the Zend Framework and allows for easy additions of further adapters as required. We will look at the CSV adapter first as that is a very simple format and hence very quick to use. Possibly the biggest benefit to CSV input files is that it is easy for the client to provide such a file using a text editor or a spreadsheet program like Excel. One downside is that support for UTF8 is quite patchy with text editors

Using `Zend_Translate` is simple enough. In Listing 14.4, we output some text using pretty ordinary PHP and then repeat the exercise in Listing 14.5 using `Zend_Translate`'s array adapter. In this case, we make life easier by "translating" to upper case, but we could equally have translated to German, if I knew enough!

Listing 14.4: Standard PHP output

```

print "Welcome\n";
print "=====\n";
print "Today's date is " . date("d/m/Y") . "\n";

```

A
B

A Standard PHP print statement
B UK format date

Listing 14.4. is a very simple piece of code that displays three lines of text, maybe for a command line script. To provide a translation we need to create an array of translation data for the target language. The array consists of identifying keys mapped against the actual text to be displayed. The keys can be anything, but it makes it easier if it is essentially the same as the source language.

Listing 14.5: Translated version of Listing 14.4

```

$data = array();

```

A


```

$data['hello'] = 'WELCOME';                                A
$data['today %1$s'] = 'TODAY\'S DATE IS %1$s';            A

$translate = new Zend_Translate('array', $data, 'en');      B

print $translate->_("hello")."\n";                          1
print "=====\n";
printf($translate->_('today %1$s')."\n", date('d/m/Y'));    2

```

A Translation array
B Create an instance of Zend_Translate
C Print the “hello” translation text
C printf() placeholders also work

In this example, we use the `_()` function to do the translation (1). This is a very common function name in multiple languages and frameworks for translation as it is a very frequently used function and it is less distracting in the source code if it is short. As you can see in (2) the `_()` function also supports the use of `printf()` placeholders so that you can embed some dynamic text in the correct place. The current date is a good example as in English we say “Today’s date is {date}” whereas in another language, the idiom may be “{date} is today’s date”. By using `printf()` placeholders, we are able to move the dynamic data to the correct place for the language construct used.

The array adapter is mainly useful for very small projects. For a large project, the `gettext()` format or `csv` format is much more useful. For `gettext()`, the translation text is stored in `.po` files which are best managed using a specialized editor, such as the open source `poEdit` application.. These applications provide a list of the source language strings and next to each one, the translator can type the target language equivalent string. This makes creating translation files relatively easy and completely independent of the website source code.

The process of using `gettext()` source files with `Zend_Translate` is as simple as picking a different adapter as shown in listing 14.6.

Listing 14.6: Zend_Translate using the gettext() adapter

```

$filename = 'translations/uppercase.po';                    A
$translate = new Zend_Translate(gettext, $filename, 'en');  A

print $translate->_("hello")."\n";                          B
print "=====\n";                                            B
printf($translate->_('today %1$s')."\n", date('d/m/Y'));    B

```

A Change the adapter to gettext
B Everything else is the same

As you can see, the use of the translation object is exactly the same, regardless of the translation source adapter used. Let’s look at integrating what we have learnt into a real application. We will use our *Places* application and adapt it to support two languages.

14.3 Adding a second language to Places

The initial goal for making *Places* multi-lingual is to present the user interface in the language of the viewer. We will use the same view templates for all languages, and ensure that all phrases are translated appropriately. Each language will have its own translation file stored, in our case using the array adapter to keep things simple and if the user requests a language for which we do not have a translation, then we will use English. The result can be seen in Figure 14.1 which shows the German version of *Places*. (Note that the translation was done using Google translate – a human translator would do a much better job of it!)



Figure 14.1 The text on the German version of Places is translated, but the same view templates are used to ensure that adding additional languages does not require too much work.

The key steps we will be taking to make Places multi-lingual are:

- Change the default router to support a language element.
- Create an Action Helper to create the Zend_Translate object and load the correct language file.
- Update the controllers and views to translate all text.

We will start by looking at how to make the Front Controller's router multi-language aware so that the user can select a language.

14.3.1 Selecting the language

The first decision that needs to be made is how to determine the user's language choice. The easiest solution is to ask the web browser using Zend_Locale's `getLanguage()` function and then store this into the session. This has a few problems that need to be considered. Firstly, sessions rely on cookies and so the user would have to have cookies enabled in order to view the site in another language. Related to this, creating a session for every user involves overhead that we may not want to bear. The final problem is that search engines like Google would only see the English version of the site.

To solve these problems, the code for the language of choice should be held within the URL and there are two places we can put it: the domain or the path. To use the domain, we would need to buy the relevant domains, such as `placestotakethekids.de` and `placestotakethekids.fr` etc. These country specific domains are a very easy solution and for a commercial operation can show your customers that you are serious about doing business in their country. Problems that may arise are the domain name may not be available for the country of choice (e.g. `apple.co.uk` is not owned by Apple Inc.) and also for some country specific domains you need to have proof of business incorporation within that country in order to purchase the domain name.. The alternative is to have the language code as part of the path, such as `www.placestotakethekids.com/fr` for French and `www.placestotakethekids.com/de` for the German language. We shall use this approach.

As we wish to use full locales for each language code we need to map from the language code used in the URL to the full locale code. For example, /en will be mapped to en_GB, /fr to fr_FR, and so on for all supported languages. We will use our configuration INI file to store this as shown in listing 14.7.

Listing 14.7: Setting locale information in config.ini

```
languages.en = en_GB           A
languages.fr = fr_FR           A
languages.de = de_DE           A
```

A The short form text is the key to the full locale

The list of valid language codes and their associated locales are now available in the \$config object that is loaded in the bootstrap and then stored in the Registry. We can retrieve the list of supported language codes using:

```
$languages = array_keys($config->languages->toArray());
```

To allow the user of the language codes within the address, we need to alter the routing system to account for the additional parameter. The standard router interprets paths of the form: `{controller}/{action}/{other_parameters}` or `{module}/{controller}/{action}/{other_parameters}`. A typical path for *Places* is: `/place/index/id/4` which calls the index action of the place controller with the id parameter set to 4. For our multi-lingual site, we need to introduce the language as the first parameter, so that the path now looks like: `{language}/{controller}/{action}/{other_parameters}`. We'll use the standard two character forms for the language, so that a typical path for the German language version of *Places* is `/de/place/index/id/4`.

To change this, we need to implement a new routing rule and then replace the default route with it. The Front Controller will then be able to do its magic and ensure that the correct controller and action are called. This is done in the bootstrap part of the code as shown in listing 14.8.

Listing 14.8: Implementing a new routing rule for language support

```
$languages = array_keys($config->languages->toArray());           A
$zl = new Zend_Locale();
$lang = in_array($zl->getLanguage(), $languages)                   B
        ? $zl->getLanguage() : 'en';                               B

// add language to default route
$route = new Zend_Controller_Router_Route(                          1
    ':lang/:controller/:action/*',                                  1
    array('controller'=>'index',                                    1
        'action' => 'index',                                       1
        'module'=>'default',                                       1
        'lang'=>$lang));                                           1

$router = $frontController->getRouter();                             C
$router->addRoute('default', $route);                                C
$frontController->setRouter($router);                               C
```

A Load the list of allowed language codes from the config
 B Use the browser's language code if it is in the allowed list
 1 Create new route
 C Update the router with the new route

We use `Zend_Controller_Router_Route` to define the route (#1) using the `:` character to define the variable parts with language first, then controller, then action and then the asterisk meaning “all other parameters”. We also define the defaults for each part of the route for when it is missing. As we are replacing the default route with our new route, we keep the same defaults so that the index action of the index controller is called for an empty address. We set the default language to the browser’s default language as determined by `Zend_Locale`’s `getLanguage()` function. However, as we set this as a default, it means that if the user chooses a specific language, the choice will take precedence. This allows for people using a Spanish browser to view the site in English, for instance.

Now that we have routing working, we need load the translation files. We need to do this after routing has happened, but before we get to the action function. An action helper is the ideal vehicle for this.

14.3.2 The language action helper

An action helper has a number of different hooks into the various stages of the dispatch process. In our case, we are interested in the `init()` hook as we want to load the language files after routing has happened and we only need it to be called once per request. Our action helper, `Language` will be stored in the `Places` library and in order to take advantage of `Zend_Loader`, the class is called `Places_Controller_Action_Helper_Language` and so lives in `library/Places/Controller/Action/Helper/Language.php`. The main functionality that it performs is:

- Load language file containing array of translations
- Instantiate `Zend_Translate` object for the selected language
- Assign language string and `Zend_Translate` object to the controller and view.

All this is done in the `init()` function, however we need to ensure that the action helper knows the directory where the language files are stored and also the list of languages available. This information is known in the bootstrap file and so we pass it to the action helper using the constructor. This is shown in listing 14.9.

Listing 14.9: Language action helper to load `Zend_Translate` object

```
class Places_Controller_Action_Helper_Language
    extends Zend_Controller_Action_Helper_Abstract
{
    protected $_languages;
    protected $_directory;

    public function __construct($languages, $directory)
    {
        $this->_dir = $directory;           A
        $this->_languages = $languages;     B
    }

    public function init()
    {
        // actual work done here
    }
}
```

A Translation files are stored here

B List of languages available

The action helper is loaded in the bootstrap in just the same way as the ACL helper in chapter 6, only this time we need to pass in the list of languages and the directory where the translation files are stored. For *Places*,

we are using php files stored in the application/configuration/translations directory. The code the load the action helper is therefore just:

```
$languageHelper = new Places_Controller_Action_Helper_Language($languages,
    ROOT_DIR . '/application/configuration/translations');
Zend_Controller_Action_HelperBroker::addHelper($languageHelper);
```

The \$languages array has already been loaded in listing 14.7 and we take advantage of the ROOT_DIR define to absolutely specify the translations directory. Now that we have loaded the action helper with the required data, we can write the init() function that loads the language strings. This is shown in listing 14.10.

Listing 14.10: Places_Controller_Action_Helper_Language::init()

```
public function init()
{
    $lang = $this->getRequest()->getParam('lang');

    if(!in_array($lang, array_keys($this->_languages))) {           1
        $lang = 'en';                                             1
    }                                                            1

    $localeString = $this->_languages[$lang];                      A

    // setup translate object
    $file = $this->_dir . '/' . $lang . '.php';                    2
    if(file_exists($file))                                         2
    {                                                              2
        include $file;                                           2
    } else {                                                       2
        include $this->_dir . '/en.php';                          2
    }                                                            2

    if(!isset($translationStrings)) {
        throw new Exception('Missing $translationStrings');
    }

    $translate = new Zend_Translate('array',                        3
        $translationStrings, $lang);                             3

    $this->_actionController->_localeString = $localeString;       C
    $this->_actionController->_translate = $translate;             C

    $viewRenderer = Zend_Controller_Action_HelperBroker           D
        ::getStaticHelper('viewRenderer');                       D
    $viewRenderer->view->localeString = $localeString;             D
    $viewRenderer->view->translate = $translate;                   D
};
```

1 Ensure that language chosen is allowed
A Collect the locales string from the config
2 Load translation file or English one if file missing
3 Create Zend_Translate object
C Assign to action controller
D Assign to view

The code in init() consists of as much error checking as actual code which is not uncommon. Before we load a language file, we first check that the selected language is available (#1) as the user could, in theory, type any text within the language element of the address path. As we only have a limited number of language files

in the translations directory, we check that the choice of the user is available. If not, then we pick English instead. Similarly, although the language is valid, we double check that the file actually exists (2) in order to avoid errors later and load the file using a simple include statement. An assumption is made that the language file contains an array called `$translationStrings` which contains the actual translations and so we throw an exception if this array doesn't exist. Having completed our error checking, we then instantiate a new `Zend_Translate` object (#3) and assign it to the action controller and the view for use within the rest of the application.

As we discovered when we looked at `Zend_Translate`, the system supports multiple adapters to allow for a variety of input sources for the translation strings. For *Places*, we have chosen arrays as they are the simplest to get going, however if the site grows significantly, then moving to `gettext` would be easy to do and would require changes to just this `init()` function. The next stage is to use the `translate` object to translate our website

14.3.3 Translating the view

For our website to be multi-lingual, we need every piece of English text on every page to be changed to run through the `_()` function of `Zend_Translate`. We have already assigned the `translate` object to the view, so we can change the original `<h2>Recent reviews</h2>` on the home page to `<?php echo $this->translate->_('Recent reviews'); ?>`. That's quite a lot of typing though, so we will create a view helper called `_()` to reduce our typing and make the view templates a little clearer. The “`_`” view helper is shown in listing 14.11.

Listing 14.11: The `_()` view helper

```
class Zend_View_Helper__
{
    protected $_view;

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

    public function _($string)
    {
        return $this->_view->translate->_($string);    A
    }
}
```

A Call through to the `translate` object's function

The oddest thing about this class is its name; due to the naming convention, it looks like it isn't finished! However, other than that, this helper is very simple as it exists solely to reduce typing. Let's look at it in use on the home page. Listing 14.12 shows the top part of the `index.phtml` view template for the home page before localization.

Listing 14.12: Top part of the non-localized `index.phtml`

```
<h1><?php echo $this->escape($this->title); ?></h1>

<p>Welcome to <em>Places to take the kids</em>! This site will
help you to plan a good day out for you and your children. Every
place featured on this site has been reviewed by people like you,
so you'll be able to make informed decisions with no marketing
waffle!</p>
```

```
<h2>Recent reviews</h2>
```

As you can see, with the exception of the title, all the text in the view template is hard coded directly and so we have to change this as shown in listing 14.13.

Listing 14.13: Localized version of index.phtml

```
<h1><?php echo $this->escape($this->title);?></h1>
<p><?php echo $this->_('welcome-text'); ?></p>
1

<h2><?php echo $this->_('Recent reviews'); ?></h2>
```

1 use a simple key for the long text

With the localized template, every string is passed through the `_()` view helper. For the very long body of text, we have simplified to a simple key (`#1`) in order to make the template easier to understand and the language file simpler. The language files for English and German are shown in Listing 14.14.

Listing 14.14: English translation file, en.php

```
<?php
$translationStrings = array(
    'welcome'=>'Welcome to <i>Places to take the kids</i>! This site will help you to plan a good
day out for you and your children. Every place featured on this site has been reviewed by people
like you, so you\'ll be able to make informed decisions with no marketing waffle!',
    'Recent reviews' => 'Recent reviews',
);
```

And the equivalent file in German is shown in listing 14.15.

Listing 14.15: English translation file, en.php

```
<?php
$translationStrings = array(
    'welcome'=>'Willkommen bei <i>Places to take the kids</i>! Diese Website wird Ihnen helfen,
zu planen ein guter Tag für Sie und Ihre Kinder. Jedem Ort auf dieser Website wurde vom Menschen
wie Sie, damit Sie in der Lage, fundierte Entscheidungen treffen können, ohne Marketing Waffel!',
    'Recent reviews' => 'Jüngste Bewertungen',
);
```

Unfortunately, I can't vouch for the actual translation as I asked Google's translate system for it and I doubt that it is as good as a human translator would have done. It is quite clear, when looking at listings 14.12 and 14.13 how easy array based translations are to create.

Similarly, we have access to the `Zend_Translate` object from within the action controller and so we can translate those strings that are created at that level. This is most commonly used for the title string as this is used in the outer site template, `site.phtml` and changes per action. Listing 14.16 shows how the About page's title is created within the `IndexController's` `aboutAction()` function.

Listing 14.16: Translating within action controllers

```
public function aboutAction()
{
    $this->view->title = $this->_translate->_('About');
}
A
```

A Translate the title string

The final area we need to look at is creation of links from one page to another. Fortunately, Zend_Framework provides a url builder for us in the shape of the url() view helper. This is shown in listing 14.17.

Listing 14.17: Creating urls using the url() view helper

```
<a href="<?php echo $this->url(array(
    'lang'=>$this->localeString,           A
    'module'=>'default'                    B
    'controller'=>'review',                B
    'action'=>'add',                       B
    'placeId'=>$this->place->id             B
    ), null, true);?>">
    Add review</a>
```

A Set the language parameter

B Set the other parameters for routing

As can be seen in listing 14.16, creating a url is simplified using the url() view helper. It can be further simplified by not passing `true` as the last parameter (\$reset) in which case the helper will “remember” the state of any parameters you do not override. As the lang parameter would never be overridden, it only has to be specified if the \$reset parameter is set to true.

We have now provided comprehensive support for all aspects of translation using Zend_Translate and adding additional languages can be done very easily by adding the language to the config.ini and writing a translation file. It is traditional to provide a mechanism for allowing users to choose the language that they wish to view the site in. One common mechanism is the flag as it works in all languages, although can annoy UK citizens when they see the USA flag indicating English. Another alternative is to use text in each language, but that can be quite hard to integrate into a site’s design.

To complete our conversion of *Places* into a multi-lingual website and make our visitors feel a home, we need to ensure that we need to translate the dates displayed throughout the site using Zend_Locale.

14.3.4 Tidying up with Zend_Locale

If you look closely at Figure 14.1, you will notice that the dates are in the wrong format and are displaying the English month name rather than the German one. This is due to our view helper, displayDate() which is not correctly localized. As a reminder, the code to display dates is shown in Listing 14.18.

Listing 14.18: Naïve localization of dates

```
class Zend_View_Helper_displayDate
{
    function displayDate($timestamp, $format='%d %B %Y')
    {
        return strftime($format, strtotime($timestamp));    A
    }
}
```

A strftime() is locale aware

We are using strftime() in listing 14.17 which, according to the PHP manual is locale aware. Unfortunately, whilst we know which locale we want, strftime() uses the locale of the server unless you tell it otherwise. There are two solutions: use setlocale() or Zend_Date.

At first glance, using `setlocale()` is very tempting; we just need to add `setlocale(LC_TIME, $lang);` to our Language action helper. However it doesn't work as expected. The first problem with `setlocale()` is that it is not thread-safe. This means that If you are using a threaded webserver, then you need to call `setlocale()` before every use of a locale-aware PHP function. The second issue is that on Windows, the string that is passed to the `setlocale()` function is not the same as the string we are using in our config.ini file. That is, we are using "de" for German, however the Windows version of `setlocale()` expects "deu". `Zend_Date`, as a member of the Zend Framework works more predictably for us.

`Zend_Date` is a class that comprehensively handles date and time manipulation and display. It is also locale-aware and if you pass it a `Zend_Locale` object, it will use it to translate all date and time related strings. Our immediate requirement is simply to retrieve dates in the correct format for the user, so we use `Zend_Date`'s `get()` function as shown in listing 14.19

Listing 14.19: Localization of dates using `Zend_Date`

```
class Zend_View_Helper_displayDate
{
    protected $_view;
    protected $_locale;

    function setView($view)
    {
        $this->_view = $view;
        $this->_locale = new Zend_Locale($view->localeString);           1
    }

    function displayDate($dateString, $format = Zend_Date::DATE_LONG)
    {
        $date = new Zend_Date($dateString, null, $this->_locale);       2

        return $date->get($format);                                     3
    }
}
```

1 Create and store the locale

2 Created `Zend_Date` object

3 Get the date as a formatted string

As we already know, if a view helper has a `setView()` function, then the view will call it before calling the view helper's main function. The `setView()` function is also only called once and so is ideal for creating a `Zend_Locale` object from the language string and storing as a variable in the class(#1). Display of a locale-aware date is a case of creating the `Zend_Date` object (#2) from the date to be displayed and then calling `get()` (#3). The `get()` function takes a format parameter that can be a string or a `Zend_Date` constant. These constants are locale aware so that `DATE_LONG` will display the month name in the correct language. Similarly, `Zend_Date::DATE_SHORT` knows that the format of the short date is dd/mm/yy in the UK and mm/dd/yy in the USA. In general, I recommend staying away from short dates though as they can lead to confusion by users who aren't used to locale aware websites.

The German version of Places, including localized dates can be seen in Figure 14.2.



Figure 14.2 Using the locale-aware `Zend_Date`, we can display the date in the correct language

14.4 Summary

`Zend_Locale` and `Zend_Translate` help in making a multi-lingual, locale aware website much easier. Of course, creating a website in multiple languages is not *easy* as care needs to be taken to ensure that the text fits in the spaces provided and of course, you need to do translations that work!

`Zend_Locale` is the heart of localization with the `Zend_Framework`. It allows for normalizing numbers and dates written in different formats so that they can be stored consistently and so used within the application. `Zend_Translate` is used for translating text into a different language using the `_()` function. It supports multiple input formats, including the widespread `gettext()` format, so that you can choose the most appropriate format for your project. Small projects will use array and csv formats, whereas larger projects are more likely to use `gettext`, `Tmx` or `XmlTm`. Through the flexibility of the `Zend_Translate`'s adapter system, migrating from a simpler system to a more robust one does not affect the main code base at all.

Stuff you (should) already know

This is the chapter where you can catch up on your knowledge of those bits of PHP that you haven't had a chance to deal with until now. The Zend Framework is written in PHP5 and is fully Object Oriented and you need to have a reasonable amount of knowledge about object oriented programming in PHP to make sense of it. This chapter is no substitute for a full book, such as *PHP in Action*, but it should help you enough to get going with the Framework.

We shall also look at some software design patterns. A design pattern is an approach used by many people to the same problem. I use the word approach deliberately, as the actual code used to solve the problem is usually different but all solutions share the same design. Design patterns give us a vocabulary to be able to share solutions with one another and to also understand each other. They also have the benefit of having been tested in real applications and so you can be sure that the design pattern is a good solution to the class of problems that it helps to solve.

Let's dive right in and look at object orientation.

17.1 Object Orientation in PHP

Object oriented programming is a way to group related functions and variables together so that they don't get lost. The fundamental goal is to produce more maintainable code – that is, code that doesn't accidentally break when you are working on something else.

17.1.1 Classes, objects and inheritance

In order to understand object oriented programming, you need to understand classes and objects. While we are here, we'll also look at interfaces which are closely related. A class is just the code that groups functions and variables together. That's all. It's not scary! Figure 17.1 shows a simple class.

Figure 17.1 A simple class

```
class ZFiA_Person      A
{                      B
                      B
                      B
}
```

A "class" keyword defines the grouping of functions and variables
B All code for this class is between the braces

As you can see, the keyword `class` is used to declare the class and it is followed by its name, "ZFiA_Person", in this case. The name of the class has to be unique across all your files, so it is a good idea to prefix every class name with a unique identifier. In our case, we'll use the prefix "ZFiA" and while this introduces a modicum of hassle when writing the code, the benefits down the line when you need to integrate your code with someone else's far outweighs any initial pain.

An object is a runtime instantiation of a class. This means that it has a name and is created using the `new` keyword:

```
$rob = new ZFiA_Person();
```

The variable `$rob` is an object and it follows that you can have many objects of the same type. For instance, I can create a `$nick` object using the same syntax:

```
$nick = new ZFiA_Person();
```

I now have two objects that are of the same class. That is, you only have one class, but can have many instance objects of it.

Constructing and destructing

Usually an object needs to set up its internal state; that is, set the initial values of its member variables. To do this, a special function, called a constructor is used, which has the name `__construct()`. This function is called automatically when a new object is instantiated. It is very common for a class to have a constructor and usually one or more function parameters are used in order to set the object's internal state. This is shown in listing 17.2.

At the opposite end of the object's lifetime, another function is called, `__destruct()`, just as the object is about to be deleted. This function is called the destructor and is rarely used in PHP scripts intended for the web due to the "set-up and tear-down every request" nature of PHP.

Figure 17.2 Adding a constructor

```
<?php

class ZFiA_Person
{
    private $_firstName;           1
    private $_lastName;           1

    public __construct($firstName, $lastName)  2
    {
        $this->_firstName = $firstName;       3
        $this->_lastName = $lastName;         3
    }
}
```

1 Member variables, only visible to this class

2 Constructor function

3 Assign function parameters to member variables

Listing 17.2 introduces the `private` and `public` keywords that control visibility of functions and variables within a class. These are discussed in a little more depth in the next section. As you can see, from Listing 17.2, we have created two member variables (1) which are set to their initial values using the constructor (3).

Public, protected and private visibility

As we have noted, a class can have functions and variables. This grouping is useful in its own right to help organize code, but really comes into its own when we add *information hiding* to the mix. Information hiding is the ability to mark class functions and variables as invisible outside the class boundary. This leads to more

maintainable code as each class “publishes” an API for other classes to use, but is free to implement that API however it would like to.

There are three keywords involved: public, protected and private and follow the same usage as other Object Oriented languages such as C++, Java or C#:

- **public**: available from any scope
- **protected**: only available from with the class or any of its children
- **private**: only available within the class itself

The **final** keyword is similar to the public keywords, except that child classes cannot override it. Let’s look at this in use and flesh out the ZFiA_Person class in listing 17.3.

Listing 17.3 A simple class

```
<?php

class ZFiA_Person
{
    protected $_firstName;           A
    protected $_lastName;           A

    public __construct($firstName, $lastName)           B
    {
        $this->_firstName = $firstName;           C
        $this->_lastName = $lastName;           C
    }

    public fullName()           D
    {
        return $this->_firstName . ' ' . $this->_lastName;           D
    }
}
```

A Variables only visible to class and its children

B Constructor class called on new

C Assign to member variables

D Public function used by consumers of class

17.1.3 Extending classes

Extending classes is extremely powerful and can save you a lot of time by facilitating code reuse. When one class extends another it will inherit the properties and methods of the class it extends provided they are not “private”. The class being extended is called the parent and the class that is extending the parent is called the child. When you extend another class, you are creating an “is a” relationship. That is your child class “is a” parent class with additional functionality. For example, we could create a ZFiA_Author class that extends ZFiA_Person which means that an Author is a Person. This is shown in Listing 17.4.

Listing 17.4 Extending ZFiA_Person to create ZFiA_Author

```
class ZFiA_Author extends ZFiA_Person           A
{
    protected $_title;           1

    public function setTitle($title)           2
    {
        $this->_title = $title;           2
    }
}
```

A extends is used to inherit functionality

1 This variable is only available in ZFiA_Author
2 Member function

The ZFiA_Author class inherits all the properties and methods of the ZFiA_Person class. It can also have its own properties (#1) and functions (#2). You can overwrite the methods by creating new methods with the same name in the child class, or you can use the methods of the parent class simply by calling them, you do not need to redefine them.

17.1.4 Abstract classes and interfaces

Two new features of PHP5 are interfaces and abstract classes. Some classes are designed purely to be extended by other classes. These classes are known as abstract classes and act as a skeleton implementation of a concept that is then fleshed out by child classes. Abstract classes contain their own member variables and functions along with declarations of functions that must be defined by the concrete child classes. For instance, we could make ZFiA_Person an abstract class where we determine that all people will have a first name and surname, but leave the definition of how to present their formal name to the child classes, as shown in listing 17.6.

Listing 17.5 An Abstract class contains undefined functions that must be implemented by child classes.

```
abstract class ZFiA_Person                                     A
{
    protected $_firstName;
    protected $_lastName;

    public __construct($firstName, $lastName)                  B
    {
        $this->_firstName = $firstName;
        $this->_lastName = $lastName;
    }

    abstract public function fullName();                        C
}
```

A Abstract keyword in class definition

B Standard member function

C Abstract function is not defined in this class

As ZFiA_Person is now declared as an abstract class, it can no longer be instantiated directly using `new`; it must be extended the child class instantiated. As the function `fullName()` is defined as abstract, all child classes (that are not themselves declared as abstract) must provide an implementation of this function. We can therefore provide an implementation of ZFiA_Author in listing 17.6.

Listing 17.6 An Abstract class contains undefined functions that must be implemented by child classes.

```
class ZFiA_Author extends ZFiA_Person
{
    public function fullName()
    {
        $name = $this->_firstName . ' ' . $this->_lastName;
        $name .= ', Author';
        return $name;
    }
}
```

As you can see, the ZFiA_Author class provides a specific implementation of `fullName()` that is not appropriate to any other child class of ZFiA_Person.

One limitation of the PHP object model is that a class may only inherit from one parent class. There are lots of good reasons for this, but it does mean that another mechanism is required to enable a class to conform to more than one “template”. This is achieved using interfaces.

Interfaces

Interfaces are a mechanism for defining a template for classes that implement the interface, that is, an API or contract that the class satisfies. In practical terms, this means that an interface is a list of functions that must be defined by implementing classes. Interfaces are used to allow a class to be used by a function that is depending upon the class having certain functionality implemented. If we were to make ZFiA_Person an interface, it would be declared as shown in listing 17.7.

Listing 17.7 An interface is a class template defining which functions must be defined.

```
interface ZFiA_Person
{
    public function fullName();
}

class ZFiA_Author implements ZFiA_Person
{
    protected $_firstName;           A
    protected $_lastName;           A

    public __construct($firstName, $lastName)           B
    {
        $this->_firstName = $firstName;           C
        $this->_lastName = $lastName;           C
    }

    public fullName()           D
    {
        return $this->_firstName . ' ' . $this->_lastName;           D
    }
}
```

If the `fullName` method is not defined in ZFiA_Author it will result in a fatal error. As PHP is not a statically typed language, Interfaces are a convenience mechanism for the programmer to help prevent logical errors. This is helped by using type hinting in functions. Type hinting is used to tell the PHP interpreter information about the parameters passed to a function. Consider the function in listing 17.8.

Listing 17.8 Type hinting provides information about function parameters

```
function displayPerson(ZFiA_Person $person)           1
{
    echo $person->fullName();
}
```

1 Provide ZFiA_Person type hint in function declaration

To create a type hint, all we need to do is include the type of the function parameter, ZFiA_Person in this case (#1). If we try to pass any other kind of object to this function then we get this error message:

Catchable fatal error: Argument 1 passed to displayPerson() must be an instance of ZFiA_Person, string given, called in type_hinting.php on line 18 and defined in type_hinting.php on line 11

As should be obvious, interfaces are not required for PHP coding as PHP is not a statically typed language and will do the right thing in the majority of cases. The main benefit of interfaces is for the programmer as it helps make code self-documenting.

Another feature of PHP 5 that is used by the Zend Framework to make programmers' lives easier is the so-called "magic methods". These are special methods that PHP calls automatically when required, allowing the programmer to provide a cleaner interface to the class.

17.1.5 Magic methods

Magic methods are special methods used by PHP in certain circumstances. Any method beginning with “__” are considered reserved by PHP and so all magic methods are prefixed with __. The most common method is the constructor, __construct, which we have looked at “Constructing and Deconstructing” earlier. Table 17.1 shows all the available magic methods that you can use in your class.

Table 17.1 PHP5's magic methods are automatically called by PHP when required

Method name	Prototype	When called
__construct	void __construct()	Object is instantiated
__destruct	void __destruct()	Object is cleaned up and removed from memory
__call	mixed __call(string \$name, array \$arguments)	Member function does not exist. Used to implement method handling that depends on the function name.
__get	mixed __get(string \$name)	Member variable does not exist when retrieving.
__set	void __set(string \$name, mixed \$value)	Member variable does not exist when setting.
__isset	bool __isset(string \$name)	Member variable does not exist when testing if it is set
__unset	void __unset(string \$name)	Member variable does not exist when unsetting
__sleep	void __sleep()	Object about to be serialized. Used to commit any pending data that the object may be holding.
__wakeup	void __wakeup()	Object has been unserialized. Used to reconnect to external resources (e.g. a database).
__toString	void __toString()	Object is converted to a string. Used to output something sensible on echo().
__set_state	static void __set_state(array \$properties)	var_export() has been called on the object. Used to enable recreation of an object using eval().
__clone	void __clone()	Object is being copied. Usually used to ensure that the copy of the object handles references.
__autoload	void __autoload(\$class_name)	Class cannot be found when instantiating. Used to load the class using include() by mapping the class's name to a file on disk.

Let's look at one use of __set() and __get within Zend_Config. Zend_Config stores all of its variables within a protected member variable called \$data and using the magic __set() and __get() to provide public access to the information as shown in listing 17.9

Listing 17.9 Zend_Config's use of __get() and __set()

```
class Zend_Config implements Countable, Iterator           A
{
    protected $_data;                                     B

    public function get($name, $default = null)           1
    {
        $result = $default;
        if (array_key_exists($name, $this->_data)) {
            $result = $this->_data[$name];
        }
        return $result;
    }

    public function __get($name)                           2
    {
        return $this->get($name);
    }

    public function __set($name, $value)                   3
    {
        if ($this->_allowModifications) {
            if (is_array($value)) {
                $this->_data[$name] = new Zend_Config($value, true);
            } else {
                $this->_data[$name] = $value;
            }
        }
    }
}
```



```

        $this->_count = count($this->_data);
    } else {
        throw new Zend_Config_Exception('Zend_Config is read only');
    }
}
}

```

A Uses interfaces

B data stored in a protected array

1 Function to retrieve data value

2 Call through to get()

3 Special conditions on setting a variable

As you can see, Zend_Config has a public function, get(), that is used to retrieve a variable and provides for a default if the variable is not set (#1). The magic function __get() uses the get() function that's already written (#2) and so allows for accessing a variable as if it was a native member variable. For example:

```
$adminEmail = $config->adminEmail
```

This is identical to calling:

```
$adminEmail = $config->get('adminEmail');
```

except that the first way is “eleaner” and easier to remember!

Similarly, __set() is used for setting a variable (#3). This function implements private business logic to only allow a variable to be set if the object has its __allowModifications variable set. If the object has been created as read-only, then an exception is thrown instead.

This concludes our whistle-stop tour through the highlights of objects in PHP. For more in depth information, I recommend reading chapters 2 through 6 of *PHP in Action* by Dagfinn Reiersøl. Another area of PHP 5 that is used by the Zend Framework is the Standard PHP Library, known as the SPL.

17.2 The SPL

The Standard PHP Library, aka SPL, is a library of useful interfaces designed to make data access easier. The SPL provides the following categories of interfaces:

- Iterators – for looping over elements in a collection, files, directories or XML.
- Array Access – for making objects act like arrays
- Counting – to allow objects to work with count()
- Observer – an implementation of the Observer software design pattern

17.2.1 Iterators

Iterators are objects that loop (or traverse) a structure such as an array, database result set or a directory listing. You need an iterator for each type of structure that you wish to iterate over and the most common (arrays, directories) are built into PHP and the SPL directly. A common use-case is to read a directory and this is done using the SPL's DirectoryIterator object as shown in listing 17.10.

Listing 17.10 Iteration over a directory listing using DirectoryIterator

```

$dirList = new DirectoryIterator('/');           1
foreach ($dirList as $item) {                    A
    echo $item . "\n";
}

```

```

}
1 Create iterator
A Use iterator in foreach() to loop

```

As you can see, using an iterator is as easy as using a `foreach()` loop and that's the whole point! Iterators make accessing collections of data very very easy and they come into their own when used with custom classes. This can be seen within the Zend Framework in the `Zend_Db_Table_Rowset_Abstract` class and is used to allow traversal over all the results returned from a query. Let's look at some highlights of the implementation. Firstly, we use `Zend_Db_Table_Rowset` as shown in listing 17.11.

Listing 17.11 Using iteration with `Zend_Db_Table_Rowset`

```

class Users extends Zend_Db_Table {}           A

$users = new Users();
$userRowset = $users->fetchAll();              B

foreach ($userRowset as $user) {              C
    echo $user->name . "\n";
}

```

A Gateway to the Users database table
 B Get Rowset
 C Iterator over the Rowset class

As we expect, using a Rowset is as easy as using an array. This is made possible because `Zend_Db_Table_Rowset_Abstract` implements the Iterator SPL interface as shown in listing 17.12.

Listing 17.12 Iterator implementation within `Zend_Db_Table_Rowset_Abstract`

```

abstract class Zend_Db_Table_Rowset_Abstract implements Iterator, Countable
{
    protected $_rows = array();                A

    public function current()                  1
    {
        if ($this->valid() === false) {
            return null;                      B
        }

        return $this->_rows[$this->_pointer];
    }

    public function key()                     2
    {
        return $this->_pointer;
    }

    public function next()                    3
    {
        ++$this->_pointer;
    }

    public function rewind()                  4
    {
        $this->_pointer = 0;
    }

    public function valid()                   5
    {
        return $this->_pointer < $this->_count;
    }
}

```

A Private array of Rows created by select function

- 1 Iterator's `current()` returns current item
- 2 return null at the end to break out of a `foreach()` or `while()`
- 3 Iterator's `key()` returns a point to the current item
- 4 Iterator's `next()` moves the pointer forward by one item
- 5 Iterator's `rewind()` resets the pointer to the start
- 6 Iterator's `valid()` checks that the pointer is valid

`Zend_Db_Table_Rowset_Abstract` holds its data in an array called `_rows` and holds the count of how many items are in the array in `_count`. We also need to keep track of where we are in the array as the `foreach()` is progressing and the `_pointer` member variable is used for this. To implement Iterator, a class needs to provide 5 functions, `current()`, `key()`, `next()`, `rewind()` and `valid()`. These functions are very simple as you can see in listing 17.12 as they simply manipulate the `_pointer` variable as required and return the correct data in `current()`.

One limitation of Iterator is that it is forward traversal of the data and does not allow for random access of the data. That is, you cannot do:

```
$aUser = $userRowset[4];
```

with an instance of `Zend_Db_Table_Rowset_Abstract`. This is by design as the underlying technology behind the class is a database that may not allow random access to a dataset. Other type of objects may require this though and for these cases, the `ArrayAccess` Interface is used.

17.2.2 Array Access and Countable

The `ArrayAccess` iterator allows a class to behave like an array. This means that a user of the class can randomly access any element within the collection using the `[]` notation like this:

```
$element = $objectArray[4];
```

In a similar manner to Iterator, this is achieved using a set of functions that must be implemented by the class. In this case the functions are: `offsetExists()`, `offsetGet()`, `offsetSet()` and `offsetUnset()`. The names are self-explanatory and, as shown in listing 17.13, allow for the following array operations to be carried out:

Listing 17.13 Array operations with an `ArrayAccess` capable class

```
if(isset($rob['firstName'])) {           A
    echo $rob['firstName'];             B
    $rob['firstName'] = 'R.';           C
}
unset($rob['firstName']);                D
```

A calls `offsetExists()`
 B calls `offsetGet()`
 C calls `offsetSet()`
 D calls `offsetUnset()`

The only other array based functionality that is useful to implement in the context of an object is the `count()` function. This is done via the `Countable` interface which contains just one method, `count()`. As you would expect, this function needs to return the number of items in the collection and then we have all the tools to make a class behave like an array.

The SPL has many other iterators and other interfaces available for more complicated requirements; however they are not, yet at least, used by the Zend Framework.

17.3 PHP4

By now, everyone should be using PHP 5 as it is over three years old and PHP4 will be unsupported after August 2008. If your first steps into PHP 5 are with the Zend Framework then you really should read the excellent migration guide available on the PHP website at <http://www.php.net/manual/en/migration5.php>.

17.4 Software design patterns

After producing software for a while you soon discover that you encounter similar problems to ones you have encountered in the past. As a result, you probably solved them with similar approaches. It turns out that this happens across the whole gamut of software development and there tends to be a best practice solution to certain classes of problems that appear regularly. In order to make it easier to discuss these approaches to solving problems, the term design pattern has been coined and catalogues of design patterns are available to help you solve software design problems.

The key thing to understand about design patterns is that they are not about the code and they are all about guidelines on how to solve the problem. Let's look at a couple of the common patterns to see how it works in practice.

17.4.1 The Singleton pattern

The Singleton design pattern is a very simple pattern that is intended to ensure that only one instance of an object may exist. This is used by the Zend Framework's Zend_Controller_Front front controller to ensure that there is only one front controller in existence for the duration of the request. Let's look at the code:

Listing 17.14 The Singleton as implemented by Zend_Controller_Front

```
class Zend_Controller_Front
{
    protected static $_instance = null;                                1

    private function __construct()                                    2
    {
        $this->_plugins = new Zend_Controller_Plugin_Broker();
    }

    public static function getInstance()                                3
    {
        if (null === self::$_instance) {                                4
            self::$_instance = new self();                                4
        }

        return self::$_instance;                                        5
    }
}
```

1 Static variable holds the one instance of this object

2 private constructor

3 method to allow creation

4 create one instance if not already created

5 return instance to user

There are three pieces in the Singleton jigsaw as implemented by Zend_Controller_Front, although every other implementation is similar. A static variable is used to store the instance of the front controller (#1). This is protected to prevent anyone outside of this class (or its descendants) from accessing it directly. To prevent use of the new keyword, the constructor is private (#2) and so the only way to create an instance of this class is to call the static function getInstance() (#3). getInstance() is static as we don't have an instance of the class

when we call it; it returns an instance to us. Within `getInstance()` we only instantiate the class once (#4), otherwise we just return the previously created instance (#5).

The net result is that the code to access the front controller is:

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

This code will work wherever it is required and will create a front controller for us if we don't have one, otherwise, it returns a reference to the one that has already be created. Whilst the Singleton is easy to implement, there is a rather large caveat with it: it's a global variable by the back door. As the Singleton has a static function that returns a reference to the object, it can be called *anywhere* and so if used unwisely can introduce coupling between disparate sections of a program.

Coupling

Coupling is the term used when two sections of code are linked in some way. More coupling makes it harder to track down bugs there are more places where the code could have been changed from.

As a result, care should be taken when designing a class to be a singleton and also when choosing to use the `getInstance()` function elsewhere in the code. Usually, it is better to pass an object around so that it is easier to control access to it than to access the Singleton directly. An example of where we can do this is within the action controller class: `Zend_Controller_Action` provides a member variable `_request` which is a reference to the `Request` object and this member variable should be used in preference to `Zend_Controller_Front::getInstance()->getRequest()`.

Having said that, it is not uncommon to need to access commonly used information from multiple places, for example configuration data. To help control this, the Registry pattern provides the tool to ensure that managing such data works well.

17.4.2 The Registry pattern

The registry pattern allows us to take any object and treat it as a Singleton and also allows us to centrally manage an object. The biggest improvement over the Singleton is that is possible to have two instances of an object if required. A typical scenario is a database connection object. Usually, you connect to one database for the duration of the request and so it is tempting to make the object as a Singleton. However, you may need to connect to a second database every so often, such as for an export or import and for those times only you need two instances of the database connection object. A Singleton will not allow this, but using a Registry will give you the benefits of the easily accessible object without the drawbacks. This problem is so common that the Zend Framework provides the `Zend_Registry` class that implements the Registry pattern. For simple access, there are three main functions that form the basic API of `Zend_Registry` and usage is shown in listings 17.15 and 17.16.

Listing 17.15 Storing an object to the Zend_Registry

```
// bootstrap class
$config = $this->loadConfig();
Zend_Registry::set('config', $config);           A
A Static function for easy usage
```

Listing 17.16 Retrieving an object from the Zend_Registry

```
// bootstrap class
$config = Zend_Registry::get('config');          A
A Retrieve via key name: 'config'
```

The only complication to be aware of with Zend_Registry is that you must ensure that you a unique key when you store the object into the registry, otherwise you will overwrite the previously stored object of the same name. Other than that, as Zend_Registry provides a pair of static functions that makes using a registry easier than modifying a class to make it a Singleton. The key components of the implementation are shown in listing 17.17. Let's examine it.

Listing 17.17 The Registry as implemented by Zend_Registry

```
class Zend_Registry extends ArrayObject
{
    public static function getInstance()                                1
    {
        if (self::$_registry === null) {
            self::init();
        }

        return self::$_registry;
    }

    public static function set($index, $value)
    {
        $instance = self::getInstance();
        $instance->offsetSet($index, $value);                            2
    }

    public static function get($index)
    {
        $instance = self::getInstance();

        if (!$instance->offsetExists($index)) {                          A
            require_once 'Zend/Exception.php';                          A
            throw new Zend_Exception("No entry is [CA]                  A
            registered for key '$index'");                                A
        }

        return $instance->offsetGet($index);                            3
    }
}
```

1 Internally, a Singleton is used
 2 Set element via ArrayObject interface
 A Check that key exists in ArrayObject
 3 Retrieve and return

Internally, the Zend_Registry uses a Singleton-like function, getInstance() to retrieve an instance of itself (#1). This is used in both get() and set() and ensures that both functions are using the same registry when setting and retrieving items. Zend_Registry implements ArrayObject which means that you can treat an instance of it as if it was an array and so iterate over it or directly set and get elements. As such, the class implements the functions required by the interface, including offsetGet(), offsetSet() and offsetExists() which are the same as the ArrayAccess functions we looked at in section 17.2.2. The static functions set() and get()

need to perform the same actions and so set called `offsetSet()` to set an item into the registry (#2) and `offsetGet` is used to retrieve an item from the registry(#3).

Note that by using `offsetGet/Set`, the code is oblivious of the underlying storage mechanism used to hold the items. This future-proofing allows for the storage mechanism to be changed if required and this code will not need to be updated.

We've looked at two of the common design patterns used in web applications and as I've said, there are many others. The Zend Framework itself implements more too and so is a good code-base for studying them. Other patterns in use include:

- Model-View-Controller in the `Zend_Controller` family of functions
- Table Gateway in `Zend_Db_Table`
- Row Gateway in `Zend_Db_Table_Row`
- Strategy in `Zend_Layout_Controller_Action_Helper_Layout`
- Observer in `Zend_XmlRpc_Server_Fault`

There are a lot of patterns out there and I recommend a read of *Patterns of Enterprise Application Architecture* by Martin Fowler and *php/architect's Guide to PHP Design Patterns* by Jason Sweat for further information on many of the common web-oriented patterns. Also, you picked up Dagfinn Reiersøl's *PHP in Action* a very accessible introduction into the finer points of using design patterns with PHP.

17.5 Summary

This concludes our look at the PHP knowledge required to understand the Zend Framework. The Zend Framework is a modern PHP5 framework and, as such, takes advantage of all the new goodies that PHP5 provides. Every part of the Framework is written in Object Oriented PHP with careful use of the SPL interfaces such as `ArrayAccess` and its sibling `ArrayObject`, `Countable` and `Iterator`. In this chapter we briefly looked at how PHP handles Object Oriented programming and also looked at how the SPL can be used to provide interfaces to classes to make them behave more like native PHP arrays. Software design patterns are an important part of modern web development and we have covered what they are and looked at how the Zend Framework uses the Singleton and Registry patterns.

This chapter only provided an overview and for more information, I strongly recommend looking at other books that cover these topics in depth. Having said that, armed with the knowledge gleaned here, you are well placed to dive head first into the Zend Framework.