wjgilmore.com

EASY PHP WEBSITES WITH THE ZEND FRAMEWORK

GAME NOMAD

Master the popular Zend Framework by following along with the creation of a social networking website for the video gaming community.

by W. Jason Gilmore

Easy PHP Websites with the Zend Framework

W. Jason Gilmore

Easy PHP Websites with the Zend Framework

W. Jason Gilmore

Acknowledgements

Whew. Although I recently celebrated the tenth anniversary of the publication of my first book, and have somehow managed to pen six more since, this process really isn't any easier than when I put my very first words to paper back in 2000. Writing anything, let alone books about fast moving technology, is a difficult, tedious, and often frustrating process. Yet paradoxically writing this particular book has also a deeply gratifying experience, particularly because it's a major update to the very first book published through my namesake company W.J. Gilmore, LLC back in early 2009. In the years since I've had the pleasure of communicating directly with thousands of readers around the globe, and although the self-publishing process has been occasionally a rocky road, the experience has been nothing short of extraordinary.

This particular project has been a difficult one, notably because it's actually comprised of two major projects, including the book and the companion GameNomad project. Throughout, I've been very keen on trying to do things *the right way*, both in my writing and the process used to develop a proper Zend Framework website complete with an emphasis on models, testing, and other best practices such as deployment.

In terms of acknowledgements, I'd like to make special mention of the people and teams who have (most of them unknowingly) had a major influence on this book. Thanks to project lead Matthew Weier O'Phinney and the rest of the Zend Framework team for putting together a truly fantastic web framework solution. Sebastian Bergmann for his work on PHPUnit, and EdgeCase co-founder and friend Joe O'Brien for his steadfast advocacy of test-driven development. Andreas Aderhold, Michiel Rook, and the rest of the Phing team. Martin Fowler for his amazing book "Patterns of Enterprise Application Architecture". The entire Doctrine team for what is easily one of the coolest PHP technologies on the planet. Capistrano creator Jamis Buck. The GitHub crew. Bob Stayton for his amazing patience and boundless Docbook knowledge. This list could literally go on for pages, as the number of great programmers who have influenced my thinking particularly in recent years seems infinite.

Last but certainly not least, I'd also like to thank you dear readers, each and every one of you mean more to me than you'll ever know.

Jason Gilmore
Columbus, Ohio
March 8, 2011
wj@wjgilmore.com

Table of Contents

List of Figures

List of Tables

Introduction

The Web Ain't What It Used to Be

The World Wide Web's technical underpinnings are incredibly easy and intuitive to understand, a characteristic which has contributed perhaps more than anything else to this revolutionary communication platform's transformational growth over the past 15 years or so. Its also this trait which I believe have led so many developers horribly astray, because while the web's plumbing remains decidely free of complexity even today, the practice of developing web sites has evolved into something decidely more complex than perhaps ever would have been imagined even a decade ago.

Despite this transformation, far too many developers continue to treat web development as something separate from software development. Yet with the Web having become an indispensable part of much of the planet's personal and business affairs, it is no longer acceptable to treat an enterprise-level website as anything but an application whose design, development, deployment, and lifecycle is governed by rigorous process. Embracing a rigorous approach to designing, developing, testing and deploying websites will make you a far more productive and worry-free developer, because your expectations of *what should be* and realization of *what is* are identical.

If you quietly admit to not having yet embraced a formalized development process, I can certainly empathize. For years I too grappled with tortuous code refactoring, unexpected side effects due to ill-conceived updates, and generally found the testing and deployment process to be deeply steeped in voodoo. After having been burned by yet another problematic bit of code, a few years ago I decided to step back from the laptop and take the time to learn how to *develop software* rather than merely *write code*. One of the first actionable steps I took in this quest was to embrace what was at the time a fledgling project called the Zend Framework. This step served as the basis for reevaluating practically everything I've come to know about the software development process, and it has undoubtedly been the most reinvigorating experience of my professional career.

If you too have grown weary of writing code in a manner similar to Shakespeare's typing monkeys, hoping that with some luck a masterpiece will eventually emerge, and instead want to start developing software using the patterns, practices, and strategies of developers who seem to be unable to do any wrong, you'll find the next 12 chapters not only transformational, but rather fun.

Book Contents

This book introduces several of the most commonly used features of the Zend Framework, organizing these topics into the following twelve chapters:

Chapter 1. Introducing Framework-Driven Development

It's difficult to fully appreciate the convenience of using a tool such as the Zend Framework without understanding the powerful development paradigms upon which such tools are built. In this chapter I'll introduce you to several key paradigms, notably the concepts of convention over configuration, the power of staying DRY, and problem solving using design patterns.

Chapter 2. Creating Your First Zend Framework Project

In this chapter you'll learn how to install and configure the Zend Framework, and use the framework's command line tool to create your first Zend Framework-powered website. You'll also learn how to expand the website by creating and managing key application components such as controllers, actions, and views.

Chapter 3. Managing Layouts, Views, CSS, Images, and JavaScript

Modern website user interfaces are an amalgamation of templates, page-specific layouts, CSS files, images and JavaScript code. The Zend Framework provides a great number of features which help reduce the complexities involved in effectively integrating and maintaining these diverse components, and in this chapter you'll learn all about them.

Chapter 4. Managing Configuration Data

Most websites rely upon a great deal of configuration data such as database connection parameters, directory paths, and web service API keys. The challenges of managing this data increases when you consider that it will often change according to your website's lifecycle stage (for instance the production website's database connection parameters will differ from those used during development). The Zend Framework's Zend_Config component was created to address these challenges in mind, and in this chapter you'll learn how to use this component to maintain configuration data for each stage of your website's lifecycle.

Chapter 5. Creating Web Forms with Zend_Form

HTML forms are one of the most commonplace features found on a website, yet their implementation is usually a chaotic and undisciplined process. The Zend Framework's Zend_Form component brings order to this important task, providing tools for not only auto-generating your forms, but also making available clear procedures for validating and processing the data. In this chapter you'll learn how

Zend_Form can remove all of the implementational vagaries from your form construction and processing tasks.

Chapter 6. Talking to the Database with Zend_Db

These days it's rare to create a website which doesn't involve some level of database integration. Although PHP makes it easy to communicate with a database such as MySQL, this can be a double-edged sword because it often leads to a confusing mishmash of PHP code and SQL execution statements. Further, constantly donning and removing the PHP developer and SQL developer hats can quickly become tiresome and error prone. The Zend Framework's MVC implementation and Zend_Db component goes a long way towards removing both of these challenges, and in this chapter you'll learn how.

Chapter 7. Integrating Doctrine 2

The Zend_Db component presents a significant improvement over the traditional approach to querying databases using PHP, however an even more powerful solution named Doctrine 2 is now at your disposal. A full-blown object-relational mapping solution, Doctrine provides developers with an impressive array of features capable of not only interacting with your database using an object-oriented interface, but can also make schema management almost enjoyable.

Chapter 8. Managing User Accounts

Whether you're building an e-commerce site or would prefer readers of your blog register before adding comments, you'll need an effective way to create user accounts and allow users to easily login and logout of the site. Further, you'll probably want to provide users with tools for performing tasks such as changing their password. Accomplishing all of these tasks is easily done using the Zend_Auth component, and in this chapter I'll show you how to use Zend_Auth to implement all of these features.

Chapter 9. Creating Rich User Interfaces with JavaScript and Ajax

What's a website without a little eye candy? In a mere five years since the term was coined, Ajax-driven interfaces have become a mainstream fixture of websites large and small. Yet the challenges involved in designing, developing and debugging Ajax-oriented features remain. In this chapter I'll introduce you to JavaScript, the popular JavaScript library jQuery, and show you how to integrate a simple but effective Ajax-based username validation feature into your website.

Chapter 10. Integrating Web Services

Every web framework sports a particular feature which sets it apart from the competition. In the Zend Framework's case, that feature is deep integration with many of the most popular web services, among them Amazon's EC2, S3, and Affiliate services, more than ten different Google services including Google Calendar and YouTube, and Microsoft Azure. In this chapter I'll introduce you to Zend_Service_Amazon (the gateway to the Amazon Product Advertising API), a Zend Framework component which figures prominently into GameNomad, and also show you how easy it is to integrate the Google Maps API into your Zend Framework application despite the current lack of a Zend Framework Google Maps API component.

Chapter 11. Unit Testing Your Zend Framework Application

Most of the preceding chapters include a special section devoted to explaining how to use PHPUnit and the Zend Framework's Zend_Test component to test the code presented therein, however because properly configuring these tools is such a source of pain and confusion, I thought it worth devoting an entire chapter to the topic.

Chapter 12. Deploying Your Website with Capistrano

Lacking an automated deployment process can be the source of significant pain, particularly as you need to update the production site to reflect the latest updates and bug fixes. In this chapter I'll show you how to wield total control over the deployment process using a great deployment tool called Capistrano.

Reader Expectations

You presumably expect that I possess a certain level of knowledge and experience pertaining to PHP and the Zend Framework. The pages which follow will determine whether I've adequately met those expectations. Likewise, in order for you to make the most of the material in this book, you should possess a basic understanding of the PHP language, at least a conceptual understanding of object-oriented programming and preferably PHP's particular implementation, and a basic grasp of Structured Query Language (SQL) syntax, in addition to fundamental relational database concepts such as datatypes and joins.

If you do not feel comfortable with any of these expectations, then while I'd imagine you will still benefit somewhat from the material, chances are you'll have a lot more to gain after having read my book *Beginning PHP and MySQL, Fourth Edition*, which you can purchase from WJGilmore.com.

About the Companion Project

Rather than string together a bunch of contrived examples, an approach which has become all too common in today's programming books, you'll see that many examples are based on a social networking website for video gamers. This website is called GameNomad (http://gamenomad.wjgilmore.com), and it embodies many of the concepts and examples found throughout the book. All readers are able to download *all* of the GameNomad source code at WJGilmore.com. Once downloaded, unarchive the package and read the `INSTALL.txt` file to get started.

Like any software project, I can guarantee you'll encounter a few bugs, and encourage you to e-mail your findings to support@wjgilmore.com. Hopefully in the near future I'll make the project available via a private Git repository which readers will be able to use in order to conveniently obtain the latest updates.

About the Author

W. Jason Gilmore is a developer, trainer, consultant, and author of six books, including the bestselling "Beginning PHP and MySQL, Fourth Edition" (Apress, 2010), "Easy PHP Websites with the Zend Framework" (W.J. Gilmore LLC, 2011), and "Easy PayPal with PHP" (W.J. Gilmore LLC, 2009). He is a regular columnist for Developer.com, JS Magazine, and PHPBuilder.com, and has been published more than one hundred times over the years within leading online and print publications. Jason has instructed hundreds of developers in the United States and Europe.

Jason is co-founder of the popular CodeMash Conference http://www.codemash.org), and was a member of the 2008 MySQL conference speaker selection board.

Contact the Author

I love responding to reader questions and feedback. Get in touch at wj@wjgilmore.com

Chapter 1. Introducing Framework-Driven Development

Although the subject of web development logically falls under the larger umbrella of *computer science*, mad science might be a more fitting designation given the level of improvisation, spontaneity and slapdashery which has taken place over the last 15 years. To be fair, the World Wide Web doesn't have a stranglehold on the bad software market, however in my opinion bad code and practices are so prevalent within the web development community is because many web developers tend not to identify a website as software in the first place.

This misinterpretation is paradoxical, because websites are actually software of a most complex type. User expectations of perpetual uptime, constant exploitation attempts by a worldwide audience of malicious intruders, seamless integration with third-party web services such as Amazon, Facebook and Twitter, availability on all manner of platforms ranging from the PC to mobile devices and now the iPad, and increasingly complex domain models as businesses continue to move sophisticated operations to the web are all burdens which weigh heavily upon today's web developer.

To deal with this growing complexity, leading developers have devoted a great deal of time and effort to establishing best practices which help the community embrace a formalized and rigorous approach to website development. The *web application framework* is the embodiment of these best practices, providing developers with a foundation from which a powerful, secure, and scalable website can be built.

Introducing the Web Application Framework

While I could come up with my own definition of a web application framework (heretofore called a web framework), it would likely not improve upon Wikipedia's version (http://en.wikipedia.org/wiki/Web_application_framework):

> *A web application framework is a software framework that is designed to support the development of dynamic websites, web applications and web services. The framework aims to alleviate the overhead associated with common activities used in web development. For example, many frameworks provide libraries for database access, templating frameworks and session management, and often promote code reuse.*

That's quite a mouthful. I'll spend the remainder of this chapter dissecting this definition in some detail in order to provide you with a well-rounded understanding of what solutions such as the Zend Framework have to offer.

Frameworks Support the Development of Dynamic Websites

Dynamic websites, like any software application, are composed of three components: the data, the presentation, and the logic. In the lingo of web frameworks, these components are referred to as the *model*, *view*, and *controller*, respectively. Yet most websites intermingle these components, resulting in code which might be acceptable for small projects but becomes increasingly difficult to manage as the project grows in size and complexity. As you grow the site, the potential for problems due to unchecked intermingling of these components quickly becomes apparent:

- *Technology Shifts*: MySQL has long been my preferred database solution, and I don't expect that sentiment to change anytime soon. However, if another more attractive database comes along one day, it would be foolhardy to not eventually make the switch. But if a site such as GameNomad were created with little regard to tier separation, we'd be forced to rewrite every MySQL call and possibly much of the SQL to conform to the syntax supported by the new database, in the process potentially introducing coding errors and breaking HTML output due to the need to touch nearly every script comprising the application.

- *Presentation Maintainability and Flexibility*: Suppose you've stretched your graphical design skills to the limit, and want to hire a graphic designer to redesign the site. Unfortunately, this graphic designer knows little PHP, and proceeds to remove all of those "weird lines of text" before uploading the redesigned website, resulting in several hours of downtime while you recover the site from a backup. Furthering your problems, suppose your site eventually becomes so popular that you decide to launch a version optimized for handheld devices. This is a feature which would excite users and potentially attract new ones, however because the logic and presentation are so intertwined it's impossible to simply create a set of handheld device-specific interfaces and plug them into the existing code. Instead, you're forced to create and subsequently maintain an entirely new site!

- *Code Evolution*: Over time it's only natural your perspective on approaches to building websites will evolve. For instance, suppose you may initially choose to implement an OpenID-based authentication solution, but later decide to internally host the authentication mechanism and data. Yet because the authentication-specific code is sprinkled throughout the entire website, you're forced to spend a considerable amount of time updating this code to reflect the new authentication approach.

- *Testability*: If I had a dollar for every time I wrote a bit of code and pressed the browser reload button to see if it worked properly, this book would have been written from my yacht. Hundreds of dollars would have piled up every time I determined if a moderately complex form was properly passing data, verified that data retrieved from a SQL join was properly format, and ensured that a user registration feature sent the new registrant a confirmation e-mail. Sound familiar? The time, energy, and frustration devoted to this inefficient testing strategy can literally add weeks to the development schedule, not to mention make your job a lot less fun.

So how can you avoid these universal problems and hassles? The solution is to separate these components into distinct parts (also known as *tiers*), and write code which loosely couples these components together. By removing the interdependencies, you'll create a more manageable, testable, and scalable site. One particularly popular solution known as an *MVC architecture* provides you with the foundation for separating these tiers from the very beginning of your project!

Let's review the role each tier plays within the MVC architecture.

The Model

You can snap up the coolest domain name and hire the world's most talented graphic designer, but without content, your project is going nowhere. In the case of GameNomad that data is largely user- and game-related. To manage this data, you'll logically need to spend some time thinking about and designing the database structure. But there's much more to effectively managing an application's data than designing the schema. You'll also need to consider characteristics such as session state, data validation, and other data-related constraints. Further, as your schema evolves over time, it would be ideal to minimize the number of code modifications you'll need to make in order to update the application to reflect these schema changes. The model tier takes these sorts of challenges into account, acting as the conduit for all data-related tasks, and greatly reducing the application's underlying complexity by centralizing the data-specific code within well-defined classes.

The View

The second tier comprising the MVC architecture is the view. The view is responsible for formatting and displaying the website's data and other visual elements, including the CSS, HTML forms, buttons, logos, images, and other graphical features. Keep in mind that a view isn't restricted to solely HTML, as the view is also used to generate RSS, Flash, and printer-friendly formats. By separating the interface from the application's logic, you can greatly reduce the likelihood of mishaps occurring when the graphic designer decides to tweak the site logo or a table layout, while also facilitating the developer's ability to maintain the code's logical underpinnings without getting lost in a mess of HTML and other graphical assets.

Try as one may, a typical view will almost certainly not be devoid of PHP code. In fact, as you'll see in later chapters, even when using frameworks you'll still use simple logic such as looping mechanisms and if statements to carry out various tasks, however the bulk of the complex logic will be hosted within the third and final tier: the controller.

The Controller

The third part of the MVC triumvirate is the controller. The controller is responsible for processing events, whether initiated by the user or some other actor, such as a system process. You can think of the controller like a librarian, doling out information based on a patron's request, be it the date of Napoleon's birth, the location of the library's collection of books on postmodern art, or directions to the library. To do this, the librarian reacts to the patron's input (a question), and forms a response thanks to information provided by the model (in this case, either her brain, the card catalog, or consultation of a colleague). In answering these questions, the librarian may dole out answers in a variety of formats (which in MVC parlance would comprise the view), accomplished by talking to the patron in person, responding to an e-mail, or posting to a community forum.

A framework controller operates in the same manner as a librarian, accepting incoming requests, acquiring the necessary resources to respond to that request, and returning the response in an appropriate format back to the requesting party. As you've probably already deduced, the controller typically responds to these requests by invoking some level of logic and interacting with the model to produce a response (the view) which is formatted and returned to the requesting party. This process is commonly referred to as an *action*, and they're generally referred to as *verbs*, for example "add game", "find friend", or "contact administrator".

MVC in Action

So how do these three components work in unison to power a website? Consider a scenario in which the user navigates to GameNomad's video game listing for the PlayStation 3 console (http://gamenomad.wjgilmore.com/games/console/ps3). The model, view, and controller all play important roles in rendering this page. I'll break down the role of each in this section, interweaving the explanation with some Zend Framework-specific behavior (although the process is practically identical no matter which MVC-based web framework solution you use):

- *The Controller*: Two controllers are actually involved with most requests. The *front controller* is responsible for routing incoming requests to the appropriate *application controller* which is tasked with responding to requests associated with a specific URL. The controller naming convention and class structure usually (but is not required to) corresponds with the URL structure, so the URL `http://gamenomad.wjgilmore.com/games/console/ps3` maps to an application controller

named `Games`. Within the `Games` controller you'll find a method (also known as an *action*) named `console` which is passed the parameter `ps3`. The `console` action is responsible for retrieving a list of video games associated with the specified console, in this case the PS3, and then passing that list to the associated view. The video games are retrieved by way of the model, discussed next.

- *The Model*: As you'll learn in later chapters, GameNomad's model consists of a number of object-oriented classes, each representative of a data entity such as a gaming console, video game, or user account. Two models are actually required to retrieve a list of games supported on the PS3 console, namely `Console` and `Game`. By using the `Console` class to create an object representative of the PS3 console, we can in turn retrieve a list of all video games associated with that console, making this list available to the controller as an array of `Game` objects. Each `Game` object contains attributes which are named identically to the associated database table's columns. Therefore the `Game` object includes attributes named `name`, `price`, and `description`, among others. Don't worry about the mechanics behind this process, as you'll be introduced to this subject in great detail in later chapters.

- *The View*: Once the controller receives the array of `Game` objects back from the model, it will pass this array to the view, which will then iterate over the objects and embed them into the view template. Doing this will logically require a bit of PHP syntax, but only a looping mechanism such as a `foreach` statement and basic object-oriented syntax.

Frameworks Alleviate Overhead Associated with Common Activities

Web frameworks were borne from the understanding that all dynamic websites, no matter their purpose, share common features which can be abstracted into generally reusable implementations. For instance, almost every website will need to validate user input, communicate with a data source such as a relational database, and rely upon various configuration settings such as mail server addresses and other data such as API developer keys. A web framework removes many of the design decisions you'll need to make regarding how to approach data validation and configuration data management by embracing two powerful paradigms known as *convention over configuration* and *staying DRY*.

Convention Over Configuration

The number of decisions a developer must make when starting a new project is seemingly endless. Conclusions must be drawn regarding how approaches to tasks such as manage templates and configuration parameters, validate forms, and cache data and static pages, to say nothing of

more mundane decisions such as file- and database table-naming conventions, documentation processes, and testing policy. Making matters worse, it's not uncommon for a developer to vary the implementation of these decisions from one project to the next, introducing further chaos into the development and maintenance process.

Frameworks attempt to reduce the number of decisions a developer has to make throughout the development process by advocating an approach of *convention over configuration*. In reducing the number of decisions you have to make by offering implementation solutions right out of the box, you'll logically have more time to spend building those features which are specific to your application's problem domain. As you'll learn in the chapters that follow, the Zend Framework removes the bulk of the decisions you'll need to make regarding all of the matters mentioned in the previous paragraph. I believe this alleviation of uncertainty is one of the strongest points to consider when weighing the advantages of a framework against creating a website from scratch. Ask yourself, should you be spending valuable time doing the middling tasks which will invariably come up every time you set out to create a new website, or should you simply let a framework do the thinking for you in those regards while you concentrate on building the most compelling website possible? I think you know the answer.

Staying DRY

Avoiding repetition within your code, also known as staying DRY (Don't Repeat Yourself), is one of programming's oldest and most fundamental tenets, with constructs such as the *function* having made an appearance within even the earliest languages. Frameworks embrace this concept on multiple levels, notably not only allowing you to reduce redundancy within the application logic, but also within the presentation. For instance, the Zend Framework offers a feature known as a *view helper* which operates in a manner similar to a function, and is useful for eliminating redundancy within your page templates.

As an example, GameNomad allows registered users to assign a star rating to various technology products. This starred rating is displayed as a series of one to five star icons, and appears not only on the product detail page, but also as a sortable visual cue within category listings. The average rating will be stored in the database as an integer value, meaning some logic is required for converting that integer value into a corresponding series of star icons. While the logic is simplistic, it's nonetheless significant enough that avoiding repeating it throughout your application would be ideal. You can avoid the repetition by bundling this logic within a view helper, and then referencing that view helper much like you would a PHP function within your presentational code. Contrast this with redundantly embedding the logic wherever needed within the website, and then struggling to update each repetitive instance following a decision to update the location of your website images. You'll learn how to create and implement both action and view helpers in Chapter 3.

Frameworks Provide a Variety of Libraries

Beyond helping you to quickly surpass the myriad of implementation decisions which need to be made with the onset of each project, many mainstream frameworks provide a wide assortment of libraries which assist in the implementation of key features such as database integration and user authentication. In this section I'll provide three examples of the power these libraries can bring to your projects.

Database Integration

The practice of repeatedly jumping from one language such as PHP to SQL within a web page is a rather inefficient affair. For instance, the following sequence of statements is something you'll typically encounter in a PHP- and MySQL-driven web page:

```
$sql = "SELECT id, platform_id, title, price FROM games ORDER BY title";
$query = $db->prepare($sql);
$query->execute();
$query->store_result();
$query->bind_result($id, $platform_id, $title, $price);
```

What if you could write everything in PHP? Using the Zend Framework's Zend_Db component, you can achieve an identical result while foregoing altogether the need to write SQL statements:

```
$game = new Application_Model_Game();
$query = $game->select();
$query->from(array('id', 'platform_id', 'title', 'price'));
$query->order('title');
$result = $game->fetchAll($query);
```

This programmatic approach to interacting with the database has an additional convenience of giving you the ability to move your website from one database to another with minimum need to rewrite your code. Because most frameworks abstract the database interaction process, you're free to switch your website from one supported database to another with minimum inconvenience.

User Authentication

Whether your website consists of just a small community of friends or is an enormous project with international reach, chances are you'll require a means for uniquely identify each user who interacts with your site at some level (typically done with user accounts). Zend_Auth (discussed in Chapter 8) not only provides you with a standardized solution for authenticating users, but also provides you with interfaces to multiple authentication storage backends, such as a relational database, LDAP, and OpenID. Further, while each backend depends upon custom options for configuration, the

authentication process is identical for all solutions, meaning that even when switching authentication solutions you'll only have to deal with configuration-related matters.

Web Services

Today's website is often hybridized a construct created from the APIs and data of other online destinations. GameNomad is a perfect example of this, relying upon the Amazon Associates web Service for gaming data and the Google Maps API for location-based features, among others. Without this ability to integrate with other online services such as these, GameNomad would be a far less compelling project.

While many of these services are built using standardized protocols and data formats, there's no doubt that writing the code capable of talking to them is a time-consuming and difficult process. Recognizing this, many frameworks provide libraries which do the heavy lifting for you, giving you the tools capable of connecting to and communicating with these third-party services. For its part, the Zend Framework offers Zend_Gdata, for interacting with Google services such as Book Search, Google Calendar, Google Spreadsheets, and YouTube. You'll also find Zend_Service_Twitter, for talking to the Twitter service (http://www.twitter.com/), Zend_Service_Amazon, for retrieving data from Amazon's product database through its web Services API (http://aws.amazon.com/), and Zend_Service_Flickr, for creating interesting photo-based websites using Flickr (http://www.flickr.com/), one of the world's largest photo sharing services.

Test-Driven Development

Suffice to say that even a total neophyte is acutely aware of the programming industry's gaffe-filled history. Whether we're talking about last Tuesday's emergency security fix or the high-profile crash of the Mars Orbiter due to a simple coding error, it seems as if our mistakes are chronicled in far more detail than those made within other professions. And for good reason, given that computing affects practically every aspect of people's lives, both personal and professional.

Given the significant role played by today's computing applications, why are programmers so seemingly careless? Why does the industry remain so prone to blunders large and small? Although I'd love to offer some complicated scientific explanation or convincing conspiracy theory, the answer is actually quite elementary: *programming is hard*.

So hard in fact, that some of the professional programming community has come to the grips with the fact that mistakes are not only likely, but that they are inevitable. They have concluded that the only reasonable way to lessen the frequency of mistakes creeping into the code is by integrating testing into the development process, rather than treating it as something which occurs after the

primary development stage is over. In fact, a growing movement known as *test-driven development* emphasizes that tests should be written *before* the application itself!

To help developers out with the testing process, the Zend Framework comes with a component called Zend_Test which integrates with the popular PHPUnit testing framework. Using this powerful combination, you can create tests which verify your website is working exactly as intended. Further, you can automate the execution of these tests, and even create a variety of reporting solutions which provide immediate insight into the proper functioning of your site.

I believe this to be such an important part of the development process that subsequent chapters conclude with a section titled "Testing Your Work". This section presents several common testing scenarios which relate to the material covered in the chapter, complete with a sample test. Further, Chapter 11 is entirely devoted to the topic of configuring PHPUnit to work with Zend_Test.

Hopefully this opening section served as a compelling argument in favor of using a framework instead of repeatedly building custom solutions. And we've hardly scratched the surface in terms the advantages! My guess is that by this chapter's conclusion, you'll be wondering how you ever got along without using a framework-centric approach.

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Identify and describe the three tiers which comprise the MVC architecture.
- How does the concept of "convention over configuration" reduce the number of development decisions you need to make?
- Name two ways the Zend Framework helps you keep your code DRY.

Chapter 2. Creating Your First Zend Framework Project

Getting started building your first Zend Framework-powered website is surprisingly easy thanks to a great utility called zf which not only automates the process of creating new Zend Framework applications, but also automates the generation of key project components such as controllers, actions, models, and views. In this chapter I'll show you how to get started using the framework by guiding you through the framework installation and configuration process, followed by a thorough introduction to the powerful zf tool. After creating your first project, I'll help you navigate through the various directories and files comprising the default project structure, and show you how to generate project controllers, actions, and views. Following that, you'll learn how to pass and retrieve data from one controller action to another using the GET and POST methods, in addition to modify the framework's default URL routes behavior to your specific needs. Finally, I'll present several examples which demonstrate how to test various aspects of the features introduced in this chapter.

Downloading and Installing the Zend Framework

Open your browser and navigate to http://framework.zend.com/download/latest. Scroll to the bottom of the page where you'll find links to the full and minimal versions of the Zend Framework. I recommend downloading the full version as it contains everything you could possibly need to follow along with the rest of this book, and a whole lot more. Download the `tar.gz` or `zip` package depending on your operating system's capabilities (the ZIP format is typical for Windows users, whereas the TAR format is common for Linux users, although you may be able to use either depending on what decompression software is available on your computer).

Tip

If you're familiar with Subversion, consider retrieving the latest stable version by checking the project out from Zend's Subversion repository. In doing so you'll be able to easily update your framework source files to the latest stable version using Subversion's `UPDATE` command.

Within the decompressed directory you'll find a directory named `library`. The `library` directory contains the files which together make the Zend Framework run. Because you'll likely soon be in the position of simultaneously building or maintaining multiple Zend Framework-driven websites, I recommend placing this directory within a location where it won't later be disturbed, and then add this location to PHP's `include_path` configuration directive. For instance, if you store the

`library` directory in `/home/wjgilmore/src/zfw110/library` then you should open the `php.ini` configuration file and add the path to your `include_path` directive like this:

```
include_path = ".:/usr/share/php/:/home/wjgilmore/src/zfw110/library"
```

Once the change has been made, save the file and restart the web server.

Configuring the zf Tool

One really interesting feature of the Zend Framework is a component known as Zend_Tool_Framework. This component acts as an API of sorts to many features of the framework, allowing you to create custom utilities useful for managing the framework tooling in powerful new ways. This component has already been extended to provide developers with a command line interface typically referred to as *zf*, which can be used to not only create a new project, but also to extend your project by adding new controllers, actions, views, models, and other features. While you're not required to use zf to manage your projects, I guarantee it will be a significant timesaver and so highly recommend doing so.

To configure zf, return to the decompressed Zend Framework directory where you'll find a directory named `bin`. This directory contains the scripts which you'll access via the command line, including `zf.bat`, `zf.php`, and `zf.sh`.

On Windows, copy the `zf.bat` and `zf.php` files into the same directory where your `php.exe` file is located (the directory where PHP was installed). Next, make sure the directory where `php.exe` is located has been added to your system path. Once added, you'll be able to execute the `zf` command from any location within your file system.

On Linux the process is essentially the same; just add the framework directory's `bin` directory location to your system path:

```
%>PATH=$PATH:/path/to/your/zend/framework/bin/directory
%>export PATH
```

Of course, you'll probably want to make this path modification permanent, done by adding a line similar to the following to your `.bash_profile` file:

```
export PATH=$PATH:/path/to/your/zend/framework/bin/directory
```

Next, confirm zf is working properly by executing the following command from your prompt:

```
%>zf show version
Zend Framework Version: 1.11.2
```

If you do not see your framework version number, and instead receive an error, it's likely because the wrong path was used within the system path variable or when defining the `library` directory's location within the `include_path` directive. So be sure to double-check those settings if you encounter a problem. Presuming your framework version number has indeed been displayed, move on to the next section!

Tip

If you're using any version of Microsoft Windows, you're probably aware that the native terminal window is a piece of trash. As you'll presumably be spending quite a bit of time using zf, typing commands into this nightmarish interface will quickly become tiresome. Save yourself some pain and consider installing Console2 (http://sourceforge.net/projects/console/), a fantastic command prompt replacement which lets you run multiple prompts using a tabbed interface, and perform useful tasks such as changing the font size and color, and resizing the window.

Creating Your First Zend Framework Project

With the Zend Framework installed and zf configured, it's time to create a project. Open a terminal window and navigate to your web server's document root (or wherever else you choose to manage your websites). Once there, execute the following command:

```
%>zf create project dev.gamenomad.com
Creating project at /var/www/dev.gamenomad.com
Note: This command created a web project, for more
information setting up your VHOST,
please see docs/README
```

The project name is completely up to you, however for organizational purposes I prefer to name my projects similarly to the URL which will be used to access them via the browser. Because the project is hosted on my development laptop, I'd like to reference the project via the URL `http://dev.gamenomad.com` and so have named the project accordingly.

Adjust Your Document Root

You might recall how in the previous chapter we talked about how the Zend Framework uses a front controller to process all incoming requests. This front controller is contained within a file named `index.php`, and it resides in a directory named `public`. You don't need to create this directory or file, because zf automatically created both for you when the project was generated. In order for the front controller to be able to intercept these requests, the `public` directory must be identifiable by Apache as the site's root directory. Precisely how this is done will depend upon your system's particular

configuration, however presuming you're running the recommended latest stable version of Apache 2.2.X let's do things the proper way and configure a virtual host for the new site. Doing so will give you the ability to easily maintain multiple websites on the same web server.

What is a Virtual Host?

A virtual host is a mechanism which makes it possible to host multiple websites on a single machine, thereby reducing hardware and support costs. If you host your website at a *shared hosting provider*, then your site is configured as a virtual host alongside hundreds, and perhaps even thousands of other websites on the same server. This feature is also useful when developing websites, because you can simultaneously develop and maintain multiple sites on your development machine, and even reference them by name within the browser rather than referring to localhost.

Configuring a Virtual Host on Windows

Setting up an Apache-based virtual host on Windows is a pretty easy process, accomplished in just a few steps. First you want to configure Apache's virtual hosting feature. Open your `httpd.conf` file and uncomment the following line:

```
#Include conf/extra/httpd-vhosts.conf
```

This `httpd-vhosts.conf` file will contain your virtual host definitions. Open this file and you'll find the following block of text:

```
<VirtualHost *:80>
  ServerAdmin webmaster@dummy-host.localhost
  DocumentRoot "C:/apache/docs/dummy-host.localhost"
  ServerName dummy-host.localhost
  ServerAlias www.dummy-host.localhost
  ErrorLog "logs/dummy-host.localhost-error.log"
  CustomLog "logs/dummy-host.localhost-access.log" common
</VirtualHost>
```

This `VirtualHost` block is used to define a virtual host. For instance to define the virtual host `dev.gamenomad.com` (which will be used for a Zend Framework-powered website) you should copy and paste the block template, modifying it like so:

```
<VirtualHost *:80>
  ServerAdmin webmaster@dummy-host.localhost
  DocumentRoot "C:/apache/docs/dev.gamenomad.com/public"
  ServerName dev.gamenomad.com
  ServerAlias dev.gamenomad.com
```

```
  ErrorLog "logs/dev.gamenomad.com-error.log"
  CustomLog "logs/dev.gamenomad.com-access.log" common
</VirtualHost>
```

The `ServerAdmin` setting is irrelevant because Windows machines are not by default configured to send e-mail. The `DocumentRoot` should define the absolute path pointing to the Zend Framework project's `public` directory (more on this in a bit). The `ServerName` and `ServerAlias` settings should identify the name of the website *as you would like to access it locally*. Finally, the `ErrorLog` and `CustomLog` settings can optionally be used to log local traffic.

Save the `httpd-vhosts.conf` file and restart Apache. Finally, open the `hosts` file, which on Windows XP and Windows 7 is located in the directory `C:\WINDOWS\system32\drivers\etc`. Presuming you've never modified this file, the top of the file will contain some comments followed by this line:

```
127.0.0.1 localhost
```

Add the following line directly below the above line:

```
127.0.0.1 dev.gamenomad.com
```

Save this file and when you navigate to `http://dev.gamenomad.com`, your machine will attempt to resolve this domain *locally*. If Apache is running and you access this URL via your browser, Apache will look to the virtual host file and server the domain's associated website.

Configuring a Virtual Host on Ubuntu

Ubuntu deviates from Apache's default approach to virtual host management in a very practical way, defining each virtual host within a separate file which is stored in the directory `/etc/apache2/sites-available/`. For instance, a partial listing of my development machine's `sites-available` directory looks like this:

```
dev.gamenomad.com
dev.wjgilmore.com
```

All of these files include a `VirtualHost` container which defines the website's root directory and default behaviors as pertinent to Apache's operation. This is a fairly boilerplate virtual host definition, insomuch that when I want to create a new virtual host I just copy one of the files found in `sites-available` and rename it accordingly. What's important is that you notice how the `DocumentRoot` and `Directory` definitions point to the website's `public` directory, because that's where the front controller resides. For instance, the `dev.gamenomad.com` file looks like this:

```
<VirtualHost *>
  ServerAdmin webmaster@localhost
   ServerName dev.gamenomad.com

  DocumentRoot /var/www/dev.gamenomad.com/public
  <Directory />
    Options FollowSymLinks
    AllowOverride All
  </Directory>
  <Directory /var/www/dev.gamenomad.com/public/>
    Options Indexes FollowSymLinks MultiViews
    AllowOverride All
    Order allow,deny
    allow from all
  </Directory>

  ErrorLog /var/log/apache2/error.log

  LogLevel warn

  CustomLog /var/log/apache2/access.log combined

</VirtualHost>
```

With the virtual host defined, you're not able to access the site just yet. The `sites-available` directory only contains the sites which you have defined. To *enable* a site, you'll need to execute the following command:

```
%>sudo a2ensite dev.gamenomad.com
```

Attempting to access this site from within the browser will cause your machine to actually attempt to resolve the domain, because your machine doesn't yet know that it should instead resolve the domain name locally. To resolve the name locally, open your `/etc/hosts` file and add the following line:

```
127.0.0.1    dev.gamenomad.com
```

Once this file has been saved, all subsequent attempts to access `dev.gamenomad.com` will result in your machine resolving the domain locally! You may need to clear the browser cache if you had attempted to access `dev.gamenomad.com` before modifying your `hosts` file.

Navigate to the Project Home Page

Presuming your project has been correctly configured, you should see the image displayed in Figure 2.1.

Figure 2.1. A Zend Framework Project's Home Page

If this page doesn't appear, double-check both the changes you made to Apache's configuration file and your system's `hosts` file to make sure there aren't any spelling mistakes, and that the directory you reference in the virtual host is indeed the correct one.

The Project Structure

A Zend Framework project structure consists of quite a few directories and files, each of which plays an important role in the website's operation. Taking some time to understand their specific roles is going to help you to swiftly navigate among and modify these files as your site begins to take shape. Open a terminal window and list the contents of the newly created project's home directory. There you'll find five directories and one file, each of which is introduced next:

- `application`: The `application` directory contains the bulk of your website's domain-specific features, including the actions, configuration data, controllers, models, and views. Additionally, this directory contains a file named `Bootstrap.php`, which is responsible for initializing data and other resources specific to your website. I'll return to this file throughout the book as needed.
- `docs`: The `docs` directory is intended to store your website's developer documentation, including notably documentation generated using an automated solution such as PHPDoc.

- `library`: Empty by default, the `library` directory is intended to host third-party libraries which supplement your website's behavior. I'll return to this directory in later chapters as the example website grows in complexity.

- `public`: The `public` directory contains the website files which should not be processed via the front controller, including notably the site's CSS stylesheets, images, and JavaScript files. Additionally in this directory you'll find the front controller `index.php`) and `.htaccess` file responsible for redirecting all client requests to the front controller, which in turn identifies the appropriate application controller to contact. A newly created project's `public` directory contains nothing but the `.htaccess` and `index.php` files, meaning you'll need to create directories for organizing other site assets such as the images and JavaScript. In Chapter 3 I'll talk more about best practices for managing this data.

- `tests`: The `tests` directory contains the website's test suite. I'll talk about this directory in some detail in Chapter 11.

- `.zfproject.xml`: This file contains a manifest of all changes made by the zf's command line interface, organized in XML format. While it's quite unlikely you'll ever need to view or modify this file's contents, under no circumstances should you delete it because doing so will negate your ability to continue using zf in conjunction with your project.

Incidentally, although this is the most common way to organize a Zend framework project, it's not the only supported structure. You'll occasionally see project's organized as a series of *modules*, because it's possible to build Zend Framework-driven applications which can be plugged into another site as a module. I suspect that as it becomes easier to create and distribute these modules, you'll see this alternative structure gain in popularity however for the time being I suggest using the default structure until the growing complexity of your project warrants exploring other options.

Extending Your Project with Controllers, Actions, and Views

Following the project skeleton generation, zf will remain a constant companion throughout the lifetime of your project thanks to its ability to also create new project controllers, actions, and views (it can also create models, but that's a subject for chapters 6 and 7).

Warning

At the time of this writing zf was incapable of recognizing changes made to the project which were not carried out using the command-line interface. This is because zf considers

the `.zfproject.xml` manifest introduced in the previous section to be the sole determinant in regards to the current project state. Therefore if you manually create a project component such as a controller and then later try to add an action to the controller using zf, you will be greeted with a warning stating that the controller does not exist, because there is no record of it existing as determined by the `.zfproject.xml` file.

Creating Controllers

When a new project is generated, zf will also create the `Index` and `Error` controllers, so you can go about modifying the `Index` controller right away. As you expand the site, you'll logically want to create additional controllers. For instance, we might create a controller named `About` which will visitors a bit more about your organization. To do this, use the `create controller` command:

```
%>zf create controller About
```

Executing this command will result in the creation of the `About` controller containing one action named `IndexAction`, a corresponding `index` view, and an `About` controller test file. The project profile (`.zfproject.xml`) is also updated to reflect the latest changes to the project.

The generated `AboutController.php` (located in `application/controllers/`) contains the following contents:

```
<?php

class AboutController extends Zend_Controller_Action
{

  public function init()
  {
      /* Initialize action controller here */
  }

  public function indexAction()
  {
      // action body
  }


}
```

First and foremost, note that the controller extends the `Zend_Controller_Action` class. In doing so, the controller class will be endowed with the special characteristics and behaviors necessary to function within the Zend Framework environment. One such special characteristic is the `init()`

method, located at the top of the class. This method will execute prior to the execution of any action found in the controller, meaning you can use `init()` to initialize parameters or execute tasks which are relevant to more than one action.

You'll also find a method named `IndexAction`. When generating a new controller this action and its corresponding view (named `index.phtml`) will also be created. The index action is special because the Zend Framework will automatically refer to it when you access the controller via the browser with no corresponding action. For instance, if you were to access `http://dev.gamenomad.com/about`, the `About` controller's `index` action will automatically execute. If you want zf to skip creating an `index` view, pass a second parameter of `0` to the `create controller` command, like so:

```
%>zf create controller About 0
```

Navigate to `http://dev.gamenomad.com/about/` and you'll see that the `About` controller has indeed been created, along with an corresponding view which contains some placeholder text. Consider opening `index.phtml` (located in `application/views/scripts/about/`) and replacing the placeholder text with some background information about your website. Remember that in your view you can use HTML, so format the information however you please.

Creating Actions

You can add an action to an existing controller using the `create action` command. For instance, to add an action named `contact` to the `About` controller, use the following command:

```
%>zf create action contact About
```

The default behavior of this command is to also create the corresponding `contact.phtml` view. To override this default, pass a third parameter of `0` like so:

```
%>zf create action contact About 0
```

Creating Views

You can use the `create view` command to create new views. At the time of writing, this command works a bit differently than the others, prompting you for the controller and action:

```
%>zf create view
Please provide a value for $controllerName
zf> About
Please provide a value for $actionNameOrSimpleName
zf> contact
```

```
Updating project profile '/var/www/dev.gamenomad.com/.zfproject.xml'
```

Keep in mind this command only creates the view. If you want to create an action and a corresponding view, use the `create action` command.

Passing Data to the View

Recall that the view's primary purpose is to display data. This data will typically be retrieved from the model by way of the corresponding controller action. To pass data from the action to its corresponding view you'll assign the data to the `$this->view` object from within the action. For instance, suppose you wanted to associate a specific page title with the `About` controller's `index` action. The relevant part of that action might look like this:

```
public function indexAction()
{
    $this->view->pageTitle = "About GameNomad";
}
```

With this variable defined, you'll be able to reference it within your view like this:

```
<title><?= $this->pageTitle; ?></title>
```

Retrieving GET and POST Parameters

The dynamic nature of most websites is dependent upon the ability to persist data across requests. For instance a video game console name such as `ps3` might be passed as part of the URL (e.g. `http://dev.gamenomad.com/games/console/ps3`). The requested page could use this parameter to consult a database and retrieve a list of video games associated with that console. If a visitor wanted to subscribe to your newsletter, then he might pass his e-mail address through an HTML form, which would then be retrieved and processed by the destination page.

Data is passed from one page to the next using one of two methods, either via the URL (known as the GET method) or as part of the message body (known as the POST method). I'll spare you the detailed technical explanation, however you should understand that the POST method should always be used for requests which add or change the world's "state", so to speak. For instance, submitting a user registration form will introduce new data into the world, meaning the proper method to use is POST. On the contrary, the GET method should be used in conjunction with requests which would have no detrimental effect if executed multiple times, such as a web search effected through a search engine. Forms submitted using the GET method will result in the data being passed by way of the URL. For instance, if you head on over to Amazon.com and search for a book, you'll see the search keywords passed along on the URL.

The distinction is important because forms are often used to perform important tasks such as processing a credit card. Browser developers presume such forms will adhere to the specifications and be submitted using the POST method, thereby warning the user if he attempts to reload the page in order to prevent the action from being performed anew (in this case, charging the credit card a second time). If GET was mistakenly used for this purpose, the browser would logically not warn the user, allowing the page to be reloaded and the credit card potentially charged again (I say potentially because the developer may have built additional safeguards into the application to prevent such accidents). Given the important distinction between these two methods, keep the following in mind when building web forms:

- Use GET when the request results in an action being taken that no matter how many times it's submitted anew, will not result in a state-changing event. For instance, searching a database repeatedly will not affect the database's contents, making a search form a prime candidate for the GET method.
- Use POST when the request results in a state-changing event, such as a comment being posted to a blog, a credit card being charged, or a new user being registered.

In the sections that follow I'll show you how to retrieve data submitted using the GET and POST methods. Understanding how this is accomplished will be pivotal in terms of your ability to build dynamic websites.

Retrieving GET Parameters

The Zend Framework's default routing behavior follows a simple and intuitive pattern in which the request's associated controller and action are specified within the URL. For instance, consider the following URL:

```
http://dev.gamenomad.com/games/list/console/ps3
```

The framework's default behavior in this instance would be to execute the `Games` controller's `list` action. Further, a GET parameter identified by the name `console` has been passed and is assigned the value `ps3`. To retrieve this parameter from within the `list` action you'll use a method named `getParam()` which is associated with a globally available `_request` object:

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

If the `list` action was capable of paging output (see Chapter 6 for more information about pagination), you might pass the current page number along as part of the URL:

```
http://dev.gamenomad.com/games/list/console/ps3/page/4
```

The framework supports the ability to pass along as many parameters as you please, provided each follows the pattern of `/key/value`. Because the above URL follows this pattern, retrieving both the `console` and `page` values is trivial:

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

Retrieving POST Parameters

Although Chapter 5 is dedicated to forms processing, the matter of passing form data from one action to another is of such fundamental importance that I wanted to at least introduce the syntax in this early chapter. The syntax is only slightly different from that used to retrieve a GET parameter, involving the `_request` object's `getPost()` method. For example, suppose you wanted to provide visitors with a simple contact form which can be used to get in touch with the GameNomad team. That form syntax might look like this:

```
<form action="/about/contact" method="post">
<label for="email">Your E-mail Address:</label><br />
<input type="text" name="email" value="" size="25" /><br />
<label for="message">Your Message:</label><br />
<textarea name="message" cols="30" rows="10"></textarea><br />
<input type="submit" name="submit" value="Contact Us!" />
</form>
```

The form's action points to the `About` controller's `contact` method, meaning the form data will be made available to this action once the form has been submitted. The form method is identified as POST , so to retrieve the data, you'll use the `_request` object's `getPost()` method as demonstrated here:

```
$email = $this->_request->getPost('email');
$message = $this->_request->getPost('message');
```

Keep in mind that the `getPost()` method does not filter nor validate the form data! The Zend Framework offers a powerful suite of input validation features which I'll introduce in Chapter 5, along with a much more efficient way to create forms and process data than the approach demonstrated here.

Creating Custom Routes

As you've seen throughout this chapter, the Zend Framework employs a straightforward and intuitive routing process in which the URL's composition determines which controller and action will execute.

This URL may also be accompanied by one or more parameters which the action may accept as input. To recap this behavior, consider the following URL:

```
http://dev.gamenomad.com/games/view/asin/B000TG530M/
```

When this URL is requested, the Zend Framework's default behavior is to route the request to the `Games` controller's `view` action, passing along a GET parameter named `asin` which has been assigned the value `B000TG530M`. But what if you wanted the URL to look like this:

```
http://dev.gamenomad.com/games/B000TG530M
```

Tip

The parameter asin stands for Amazon Standard Identification Number, which uniquely identifies products stored in the Amazon.com product database. See Chapter 10 for more information about how GameNomad retrieves video game data from Amazon.com.

It's possible to allow this URL to continue referring to the `Games` controller's `view` action by creating a *custom route*. You can use the Zend Framework's custom routing feature to not only override the framework's default behavior, but also to set default parameter values and even use regular expressions which can route requests to a specific controller action whenever the defined expression pattern is matched.

To create a custom route open the `Bootstrap.php` file, located in your project's `application` directory. You might recall that earlier in the chapter I mentioned the `Bootstrap.php` file was useful for initializing data and other resources specific to your website, including custom routes. The `Bootstrap.php` file's behavior is a tad unusual, as any method embedded within the `Bootstrap` class found in this file will automatically execute with each invocation of the framework (with every request). Further, these method names must be prefixed with `_init`, otherwise they will be ignored. Therefore in order for the custom routes to work, you'll need to embed them within an appropriately named method named `_initRoutes`, for instance.

Let's create a custom route which makes it very easy for users to login by navigating to `http://dev.gamenomad.com/login`. Doing so will actually result in the execution of the `Account` controller's `login` action (which we'll talk about in detail in Chapter 8). The code is presented next, followed by an explanation:

```
01 public function _initRoutes()
02 {
03   $frontController = Zend_Controller_Front::getInstance();
04   $router = $frontController->getRouter();
05
06   $route = new Zend_Controller_Router_Route_Static (
```

```
07     'login',
08     array('controller' => 'Account', 'action' => 'login')
09   );
10
11   $router->addRoute('login', $route);
12 }
```

Let's review each line of this example:

- Line 01 defines the method used to host the custom route definitions. You can name this method anything you please, provided it is prefixed with `_init`.
- Lines 03-04 retrieve an instance of the framework router. This is needed because we'll append the custom routes to it so the framework is aware of their existence. You only need to execute these two lines once regardless of the number of custom routes you create, typically at the very beginning of the method.
- Lines 06-09 define a static custom route. A custom route of type `Zend_Controller_Router_Route_Static` is preferable for performance reasons when no regular expression patterns need to be evaluated. The constructor accepts two parameters. The first, found on Line 07, defines the custom route, while the second (Line 08) determines which controller and action should be executed when this route is requested.
- Line 11 activates the route by adding it to the framework router's list of known routes. The first parameter of the `addRoute()` method assigns a unique name to this route. Be sure that each route is assigned a unique name, otherwise naming clashes will cause the previously defined route to be canceled out.

After saving the `Bootstrap.php` file, you should be able to navigate to `http://dev.gamenomad.com/login` and be served the placeholder text found in the `Account` controller's `login` action's corresponding view (obviously you'll need to create this action if you haven't already done so).

Defining URL Parameters

This section's opening example discussed streamlining the URL by removing the explicit referral to the `view` action. This is easily accomplished using a custom route which overrides the framework's default behavior of presuming the action name appears after the controller within the URL. Because a minimal level of pattern matching is required to identify the parameter location, we can no longer use the `Zend_Controller_Router_Route_Static` custom route class, and instead need to use the `Zend_Controller_Router_Route` class. The custom route which satisfies the goal of removing the reference to the `view` action follows:

```
$route = new Zend_Controller_Router_Route (
    'games/:asin/',
    array('controller' => 'Games',
          'action'     => 'view'
         )
);

$router->addRoute('game-asin-view', $route);
```

Notice how the parameter location is prefixed with a colon `:asin`). With the parameter name and location within the route defined, you can retrieve it from within the `view` action using the `getParam()` method introduced in the previous section:

```
$asin = $this->getRequest()->getParam('asin');
```

It's also possible to define default parameter values should they not be defined within the URL. Although this feature is particularly useful for values such as dates and page numbers, it can be applied to any parameter, including the game ASIN. For instance, suppose you modified the previously used route to include the parameter name `asin`, and wanted to set this route to a default ASIN value should the user somehow delete it from the URL. You can set a default `asin` value within the custom route definition:

```
$route = new Zend_Controller_Router_Route (
    'games/asin/:asin',
    array('controller' => 'Games',
          'action'     => 'view',
          'asin'       => 'B000TG530M'
         )
);

$router->addRoute('game-asin-view', $route);
```

Once defined, any attempt to request `http://dev.gamenomad.com/games/asin/` (notice the missing ASIN), will result in the `asin` parameter being populated with the string `B000TG530M`.

This section really only scratches the surface in terms of what you can do with the Zend Framework's custom routing feature. Be sure to check out the Zend Framework manual for a complete breakdown of what's possible.

Testing Your Work

Testing is such an important part of the development process that I didn't want to treat the topic as an afterthought. At the same time, it seems illogical to put the cart before the horse and discuss the fairly

complicated topic of the Zend Framework's Zend_Test testing component before acquainting you with the framework's fundamentals. Regardless, I didn't want to decouple the testing discussion from the other topics discussed throughout this book, and so have opted to conclude each chapter with a section covering testing strategies relevant to the chapter's subject matter. If you don't yet understand how to configure Zend_Test, no worries, just skip these sections until you've read Chapter 11, and then return to each section as desired.

Many of these tests are purposefully pedantic, with the goal of showing you how to cut a wide swath when testing your application. There will probably never be a need to even execute some of the tests as I present them, such as the first test which merely verifies controller existence, however the syntax found within each test could easily be combined with others to form useful testing scenarios. Ultimately, in regards to these sections each chapter will build upon what you've learned in previous chapters in order to provide you with a well-rounded understanding of how to create tests to suit your specific needs.

Verifying Controller Existence

To test for controller existence, you'll want to send (dispatch) a request to any route mapped to the desired controller, and then use the `assertController()` method to identify the controller name:

```
public function testDoesAccountControllerExist()
{
  $this->dispatch('/about');
  $this->assertController('about');
}
```

Verifying Action Existence

To test for the existence of an action, you'll use the `assertAction()` method, identifying the name of the action which should be served when the specified route is dispatched. Rather than separately test for the existence of controller and action, consider bundling both tests together, as demonstrated here:

```
public function testDoesAccountIndexPageExist()
{
  $this->dispatch('/about');
  $this->assertController('about');
  $this->assertAction('index');

}
```

Verifying a Response Status Code

A resource request could result in any number of events, such as a successful response, a redirection to another resource, or the dreaded internal server error. Each event is associated with a response code which is returned to the client along with any other event-driven information. For instance, a successful response will include the requested information and a response code of 200. An internal server error will return a response code of 500. You can test to ensure your actions are properly responding to a request by verifying the response code using the `assertResponseCode()` method:

```
public function testDoesAccountIndexPageExistAndReturn200ResponseCode()
{
  $this->dispatch('/about');
  $this->assertController('about');
  $this->assertAction('index');
  $this->assertResponseCode(200);

}
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- What command-line tool is used to generate a Zend Framework project structure?
- What file should you never remove from the project directory, because it will result in the aforementioned tool not working properly?
- What is a virtual host and why does using virtual hosts make your job as a developer easier?
- What two files are found in the `public` directory when a new project is generated? What are the roles of these files? What other types of files should you place in this directory?

Chapter 3. Managing Layouts, Views, CSS, Images and JavaScript

While a great deal of attention is devoted to compelling features such as web services integration, the lion's share of productivity stems from the Zend Framework's ability to manage more mundane details such as configuration data and website assets such as templates, CSS, images, and JavaScript. The lack of attention is unfortunate given the enormous amount of time and effort you and your team will spend organizing these resources over the course of a project's life cycle.

In this chapter I'll break from this pattern by introducing you to many of the Zend Framework's asset management features, beginning with showing you how to effectively manage your website templates (referred to as *layouts* in Zend Framework vernacular), as well as override the default behaviors of layouts and views. You'll also learn about the framework's *view helpers*, and even learn how to create custom *view helpers*, which can go a long way towards keeping your code DRY. Next, we'll talk about several native features which can help you manage your website images, CSS, and JavaScript. Finally, the chapter concludes with a variety of example tests which can help you to avoid common implementation errors.

Managing Your Website Layout

One common approach to managing website layouts involves creating a series of files with each containing a significant portion of the site, such as the header and footer. Several `require_once` statements are used within each page to include the header and footer, which when requested results in the entire page being assembled by the PHP scripting engine. However this approach quickly becomes tedious because of the inherent difficulties which arise due to breaking HTML and scripting elements across multiple files.

The Zend Framework offers a far more convenient solution which allows you to manage a website layout within a single file. This file contains the site's header, footer, and any other data which should be made available within every page. Each time a page is requested, the framework will render the layout, injecting the action's corresponding view into the layout at a predefined location. I'll talk about precisely where that location is in a moment. First you'll need to enable the framework's layout feature for your application by executing the following command from the application's root directory:

```
%>zf enable layout
Layouts have been enabled, and a default layout created at
/var/www/dev.gamenomad.com/application/layouts/scripts/layout.phtml
A layout entry has been added to the application config file.
```

As the command output indicates, the layout file is named `layout.phtml` and it resides in the directory `application/layouts/scripts/`. Open this file and you'll see a single line:

```
<?php echo $this->layout()->content; ?>
```

This is the location where the action's view will be injected into the layout. Therefore you can add your site's global page elements simply by building around this command. For instance, one of the first things you'll probably want to do is add the standard HTML header and closing tags:

```
<html>
<head>
<title>GameNomad</title>
</head>
<body>
<h1>Welcome to GameNomad</h1>
<?php echo $this->layout()->content; ?>
<p>
Questions? Contact the GameNomad team at support@gamenomad.com!
</p>
</body>
</html>
```

Once saved, navigate to any page within your site and you'll see that the header and footer are now automatically added, as depicted in Figure 3.1.

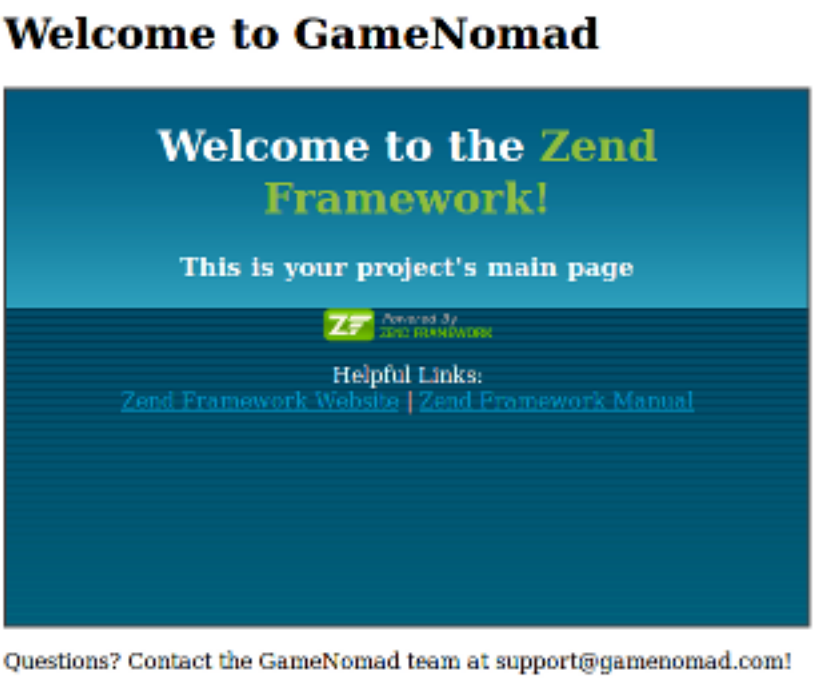


Figure 3.1. Using the Zend Framework's layout feature

Using Alternative Layouts

Although the typical website embraces a particular design theme, it's common to use multiple layouts in order to accommodate the organization of different data sets. Consider for instance the layout of any major media website. The site's home page and several of the category home pages might use a three-column layout, whereas the pages which display an article employ a two-column layout. You can change an action's layout file by retrieving an instance of the layout using a feature known as the *helper broker*, and then calling the `setLayout()` method, passing in the name of the alternative layout:

```
$layout = $this->_helper->layout();
$layout->setLayout('three-column');
```

Like the default layout, any alternative layout should also reside in the `application/layouts/scripts/` directory, and should use the `.phtml` extension.

If you wanted to change the layout for all actions in a particular controller, just insert the above two lines into the controller's `init()` method, which will execute prior to the invocation of any action found in the controller. See the last chapter for more information about the `init()` method.

Disabling the Layout

To prevent the layout from rendering, call the `disableLayout()` helper at the top of the action:

```
$this->_helper->layout()->disableLayout();
```

Keep in mind that disabling the layout will *not* disable the action's corresponding view. If you want to create an action which neither renders a layout nor a view, you'll also need to explicitly disable the view. You'll learn how to disable an action's view in the later section "Disabling the View".

Tip

If you would like to disable the layout and view in order to process an AJAX request, then chances are you won't need to call either of these helpers because the framework's `encodeJson()` helper will automatically disable rendering of both for you. See Chapter 9 for more information about processing AJAX requests.

Managing Views

When a controller action is invoked, the Zend Framework's default behavior is to look for an appropriately named action to return as the response. However, there are situations which you might wish to override this default behavior, either by using a different view or by disabling view rendering altogether.

Overriding the Default Action View

By default the framework will search for a view script named identically to the action being invoked. For instance, if the `About` controller's `contact` action is called, then the framework will expect an action named `contact.phtml` to exist and reside in the `application/views/scripts/about` directory. You can override this behavior by passing the name of a different controller into the `render()` helper:

```
$this->view->render('alternate.phtml');
```

If the view script resides in a directory different than that where the currently executing controller's views reside, you can change the view script path using the `setScriptPath()` method:

```
$this->view->setScriptPath('/application/scripts/mobile/about/');
$this->view->render('contact.phtml');
```

Disabling the View

Should you need to prevent an action's view from being rendered, add the following line to the top of the action body:

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

Presumably you'll also want to disable the layout, therefore you'll also need to call the `disableLayout()` helper as introduced earlier in this chapter:

```
$this->_helper->layout()->disableLayout();
$this->_helper->viewRenderer->setNoRender();
```

View Helpers

The Zend Framework supports a feature known as a *view helper* which can be used to manage the placement and formatting of a wide variety of site assets and other data, including page titles, CSS and JavaScript files, images, and even URLs. You can even create custom view helpers which can be immensely useful for minimizing the amount of repetitive logic which would otherwise be spread throughout the view templates. In this section I'll introduce you to one of the framework's most commonly used view helpers, and even show you how to create your own. Later in the chapter I'll introduce other native view helpers relevant to managing your site's CSS, JavaScript, and other key page elements.

Managing URLs

The framework supports a URL view helper which can be used to programmatically insert URLs into a page. For instance, suppose you wanted to create a hyperlink which points to `http://dev.gamenomad.com/games/platform/console/ps3`. Using the URL view helper within your view, you'll identify the controller, action, and lone parameter like this:

```
<a href="<?= $this->url(array(
    'controller' => 'games',
    'action' => 'platform',
    'console' => 'ps3')); ?>">View PS3 games</a>
```

Executing this code will result in a hyperlink being added to the page which looks like this:

```
<a href="/games/platform/console/ps3">View PS3 games</a>
```

But isn't this more trouble than its worth? After all, writing the hyperlink will actually require less keystrokes than using the URL view helper. The primary reason you should use the URL view helper is for reasons of maintainability. What if you created a site which when first deployed was placed within the web server's root document directory, but as the organization grew needed to be moved into a subdirectory? This location change would require you to modify every link on the site to accommodate the new location. Yet if you were using the URL view helper, no changes would be necessary because the framework is capable of detecting any changes to the base URL.

The secondary reason for using the URL view helper is that it's possible to reference a custom named route within the helper instead of referring to a controller and action at all, allowing for maximum flexibility should you later decide to point the custom route elsewhere. For instance, you might recall the custom route created in the last chapter which allowed us to use a more succinct URL when viewing information about a specific game. This was accomplished by eliminating the inclusion of the action within the URL, allowing us to use URLs such as `http://dev.gamenomad.com/games/B000TG530M` rather than `http://dev.gamenomad.com/games/asin/B000TG530M`. To refresh your memory, the custom route definition is included here:

```
$route = new Zend_Controller_Router_Route (
    'games/asin/:asin',
    array('controller' => 'Games',
          'action'     => 'view',
          'asin'       => 'B000TG530M'
         )
);

$router->addRoute('game-asin-view', $route);
```

Notice how the name `game-asin-view` is associated with this custom route when it's added to the framework's router instance. You can pass this unique name to the URL view helper to create URLs:

```
<a href="<?= $this->url(array(
    'asin' => 'B000TG530M'),
    'game-asin-view'); ?>">Call of Duty 4: Modern Warfare</a>
```

Executing this code will produce the following hyperlink:

```
http://dev.gamenomad.com/games/B000TG530M
```

Table 3-1 highlights some of the Zend Framework's other useful view helpers. Keep in mind that this is only a partial listing. You should consult the documentation for a complete breakdown.

Table 3.1. Useful View Helpers

Name	Description
Currency	Displays currency using a localized format
Cycle	Alternates the background color for a set of values
Doctype	Simplifies the placement of a DOCTYPE definition within an HTML document
HeadLink	Links to external CSS files and other resources, such as favicons and RSS feeds
HeadMeta	Defines meta tags and setting client-side caching rules
HeadScript	Adds client-side scripting elements and links to remote scripting resources. You'll learn more about this helper later in the chapter
HeadStyle	Adds CSS declarations inline

Creating Custom View Helpers

You'll often want to repeatedly perform complex logic within your code, such as formatting a user's birthday in a certain manner, or rendering a certain icon based on a preset value. To eliminate the redundant insertion of this code, you can package it within classes known as custom view helpers, and then call each view helper as necessary.

To create a custom view helper, you'll create a new class which extends the framework's `Zend_View_Helper_Abstract` class. For instance, the following helper is used on the GameNomad website in order to easily associate the appropriate gender with the user's gender designation:

```
01 <?php
02
03   class Zend_View_Helper_Gender extends Zend_View_Helper_Abstract
04   {
05
06     /**
07      * Produces string based on whether value is
08      * masculine or feminine
09      *
10      * @param string $gender
11      * @return string
12      */
13     public function Gender($gender)
14     {
15
```

```
16       if ($gender == "m") {
17         return "he";
18       } else {
19         return "she";
20       }
21
22     }
23
24   }
25
26 ?>
```

The code breakdown follows:

- Line 03 defines the helper class. Notice the naming convention and format used in the class name.
- Line 13 defines the class method, `Gender()`. This method must be named identically to the concluding part of your class name `Gender`, in this case). Likewise, the helper's file name must be named identically to the method, include the `.php` extension `Gender.php`, and be saved to the `application/views/helpers` directory.

Once created, you can execute the helper from within your views like so:

```
Jason owns 14 games, and <?= $this->Gender("m"); ?> is
currently playing Call of Duty: World at War.
```

Partial Views

Many web pages are built from snippets which are found repeatedly throughout the website. For instance, you might insert information about the best selling video game title within a number of different pages. The HTML might look like this:

```
<p>
 Best-selling game this hour:<br />
 <a href="/games/title/call_of_duty_black_ops">Call of Duty: Black Ops</a>
</p>
```

So how can we organize these templates for easy reuse? The Zend Framework makes it easy to do so, calling them *partials*. A partial is a template which can be retrieved and rendered within a page, meaning you can use it repeatedly throughout the site. If you later decide to modify the partial to include for instance the current Amazon sales rank, the change will immediately occur within each location the partial is referenced. Let's turn the above snippet into a partial:

```
<p>
 Best-selling game this hour:<br />
```

```
 <a href="/games/title/<?= $this->permalink;?>"><?= $this->title; ?></a>
</p>
```

However partials have an additional useful feature in that they can contain their own variables and logic without having to worry about potential clashing of variable names. This is useful because the variables `$this->permalink` and `$this->title` may already exist in the page calling the partial, but because of this behavior, we won't have to worry about odd side effects.

For organizational purposes, I prefix partial file names with an underscore, and store them within the `application/views/scripts` directory. For instance, the above partial might be named `_hottestgame.phtml`. To insert a partial into a view, use the following call:

```
<?= $this->partial('_hottestgame.phtml',
    array('permalink' => $game->getPermalink(), 'title' => $game->getTitle())); ?>
```

Notice how each key in the array corresponds to a variable found in the referenced partial.

The Partial Loop

The Zend Framework offers a variation of the partial statement useful for looping purposes. Revising the hottest game partial, suppose you instead wanted to provide a list containing several of the hottest selling games. You can create a partial which represents just one entry in the list, and use the `PartialLoop` construct to iterate through the games and format them accordingly. The revised partial might look like this:

```
<a href="/games/<?= $this->asin; ?>"><?= $this->title; ?></a>
```

Using the `PartialLoop` construct, you can pass along a partial and a multi-dimensional array, prompting the loop to iterate until the array values have been exhausted:

```
<ul id="hottest">
<li><?= $this->partialLoop('_hottestgames.phtml',
    array(
      array('asin' => 'B000TG530M', 'title' => 'Call of Duty 4: Modern Warfare'),
      array('asin' => 'B000FRU1UM', 'title' => 'Grand Theft Auto IV'),
      array('asin' => 'B000FRU0NU', 'title' => 'Halo 3')
    )
  )
</li>
</ul>
```

Executing this partial loop within a view produces the following output:

```
<ul id="hottest">
```

```
  <li><a href="/games/B000TG530M">Call of Duty 4: Modern Warfare</a></li>
  <li><a href="/games/B000FRU1UM">Grand Theft Auto IV</a></li>
  <li><a href="/games/B000FRU0NU">Halo 3</a></li>
</ul>
```

Managing Images

There really isn't much to say regarding the integration of images into your website views, as no special knowledge is required other than to understand that the framework will serve images from the `public` directory. However, the Zend_Tool utility does not generate a directory intended to host your site images when the application structure is created, so I suggest creating a directory named `images` or similar within your `public` directory. After moving the site images into this directory, you can reference them using the typical `img` tag:

```
<img src="/images/logo.png" alt="Welcome to GameNomad" />
```

Managing CSS and JavaScript

As is the case with images, no special knowledge is required to begin integrating Cascading Style Sheets (CSS) and JavaScript into your Zend Framework-powered website, other than the understanding that the CSS and JavaScript files should be placed somewhere within the `public` directory. For organizational purposes I suggest creating directories named `css` and `javascript` or similar, and placing the CSS and JavaScript files within them, respectively.

For instance, with the directory and CSS files in place, you'll typically use the HTML `link` tag within your site layout in order to make the CSS styles available:

```
<link rel="stylesheet" href="/css/screen.css"
      type="text/css" media="screen, projection">
```

Testing Your Work

This chapter is primarily devoted to user interface-specific features. Just as you'll want to thoroughly test the programmatic features of your website, so will you want to not only ensure that the user interface is rendering the expected page elements, but that the application behaves properly as the user navigates the interface. While PHPUnit was not intended to test user interfaces, the Zend_Test component bundles several useful features which allow you to perform rudimentary user interface tests, several of which I'll demonstrate in this section.

Before presenting the example tests, keep in mind that thorough user interface testing is much more involved than merely verifying the existence of certain page elements. Notably, you'll want to use

a tool such as Selenium which can actually navigate your website interface using any of several supported web browsers (among them Firefox, Internet Explorer, and Safari). In Chapter 11 you'll learn how to configure PHPUnit to execute Selenium tests.

Verifying Form Existence

Using the `assertQueryCount()` method, you can ensure that a page element is found within a retrieved page. This is useful when you want to make sure a certain image, form, or other HTML element has been rendered as expected. For instance, the following test ensures that exactly one instance of a form identified by the ID `login` is found within the `Account` controller's `login` view:

```
public function testLoginActionShouldContainLoginForm()
{
  $this->dispatch('/account/login');
  $this->assertQueryCount('form#login', 1);
}
```

Verifying the Page Title

The Zend Framework offers a view helper named `headTitle()` which when output within the view will generate the `title` tag and insert into it the value passed to `headTitle()`. You'll execute `headTitle()` somewhere between the layout's `head` tags in order to properly render the `title`:

```
...
<head>
<?php echo $this->headTitle('Welcome to GameNomad'); ?>
</head>
...
```

Executing this view helper will result in the following `title` tag being inserted into the layout:

```
<head>
<title>Welcome to GameNomad</title>
</head>
```

Personally, I find this feature to be superfluous, as it's just as easy to add the `title` tag manually, and then pass the desired view title in from the associated action, like this:

```
<title>
<?= (!is_null($this->pageTitle)) ? $this->pageTitle : "Welcome to GameNomad"; ?>
</title>
```

If you'd like to use a custom page title in conjunction with a specific view, all you need to do is define `$this->view->pageTitle` within the action:

```
public function loginAction()
{

  $this->view->pageTitle = 'GameNomad: Login to Your Account';
  ...
}
```

No matter which approach you take, you can execute a test which ensures the page title is set properly by using the `assertQueryContentContains()` method:

```
public function testLoginViewShouldContainLoginTitle()
{
  $this->dispatch('/account/login');
  $this->assertQueryContentContains('title', 'GameNomad: Login to Your Account');
}
```

Testing a PartialLoop View Helper

Earlier in this chapter a convenient formatting feature known as the PartialLoop view helper was introduced. You can use a PartialLoop to separate the presentational markup from the logic used to iterate over an array when displaying the array contents to the browser. The example used to demonstrate the PartialLoop view helper involved iterating over an array containing three video games, creating a link to their GameNomad pages and inserting the link into an unordered list identified by the ID `hottest`. You can create a test which verifies that exactly three unordered list items are rendered to a page:

```
public function testExactlyThreeHotGamesAreDisplayed()
{
  $this->dispatch('/games/platform/360');
  $this->assertQueryCount('ul#hottest > li', 3);
}
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- The Zend Framework's convenient layout feature is not enabled by default. What ZF CLI command should you use to enable this feature?
- From which directory does the Zend Framework expect to find your website CSS, images, and JavaScript?

- What is the name of the Zend Framework feature which can help to reduce the amount of PHP code otherwise found in your website views?
- Which Zend Framework class must you extend in order to create a custom view helper? Where should your custom view helpers be stored?
- Name two reasons why the Zend Framework's URL view helper is preferable over manually creating hyperlinks?

Chapter 4. Managing Configuration Data

Your website will likely require a fair amount of configuration-related data in order to function properly, including database connection parameters, cache-related directory paths, and SMTP addresses. Further, the site may refer to certain important bits of information which may occasionally need to be changed, such as a support-related e-mail address. Making matters more difficult, this configuration data may change according to the application's life cycle stage; for instance when your website is in the development stage, the aforementioned support-related e-mail address might be set to `bugs@gamenomad.com`, whereas when the site is public the address might be set to `support@gamenomad.com`. You'll also want to adjust PHP-specific settings according to the life cycle stage such as whether to display errors in the browser. So what's the most efficient way to manage this data?

In the interests of adhering to the DRY principle the Zend Framework offers a great solution which not only allows you to maintain this data in a central location, but also to easily switch between different sets of stage-specific configuration data. In this chapter I'll introduce you to this feature which is made available via the Zend_Config component, showing you how to use it to store and access configuration data from a central location.

Introducing the Application Configuration File

The `application.ini` file (located in the `application/configs` directory) is the Zend Framework's default repository for managing configuration data. Open this file and you'll see that several configuration parameters already exist, using a category-prefixed dotted notation syntax similar to that found in your `php.ini` file. For instance, the variables beginning with `phpSettings` will override any settings found in the `php.ini` file, such as the following variable which will prevent the display of any PHP-related errors:

```
phpSettings.display_errors = 0
```

You're free to override other PHP directives as you see fit, provided you follow the above convention, and that the directive is indeed able to be modified outside of the `php.ini` file. Consult the PHP manual for more information about each configuration directive's scope.

Note

Managing your configuration data within the `application.ini` file forms one of two approaches currently supported by the Zend_Config component. It's also possible to manage this data using an XML format, and even using an external resource such as a MySQL database. Of the three approaches the one involving INI-formatted data seems to be the most commonly used, and so this chapter will use INI-specific examples, although everything you learn here can easily be adapted to the other approaches.

You can create your own configuration parameters, even grouping them according to their purpose using intuitive category prefixes. For instance, I group my web service API keys like this:

```
webservice.amazon.affiliates.key = KEY_GOES_HERE
webservice.amazon.ec2.key = KEY_GOES_HERE
webservice.google.maps.key = KEY_GOES_HERE
```

The second way configuration parameters are organized is according to the application's life cycle stage. For instance, notice that the `phpSettings.display_errors` parameter is set to `0` within the `[production]` section. This is because when the website is deployed in a live environment, you don't want to display any ugly errors to the end user. If you scroll down the file to the section `[development:production]`, you'll find the very same variable defined again, but this time `phpSettings.display_errors` is set to `1` (enabled). This is because when your website is in the development stage, you'll want to see these errors in real-time as they occur while you develop the site.

To save unnecessary repetition, life cycle stages can be configured to inherit from another. For instance, the syntax `[development: production]` indicates that the development stage will inherit any configuration variables defined within the production stage. You can override those settings by redefining the variable, as we did with the `display_errors` variable.

You'll also see other default variables defined in the `application.ini` file. For example, the following variable identifies the location where your application controllers are found:

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

As you might imagine, these sorts of variables are useful if you wanted to change the Zend Framework's default settings, although in most cases you won't need to tinker with them. Unfortunately there's currently no definitive list of all of the available variables, however as you explore other features of the Zend Framework you'll undoubtedly come across the variables you need to add the feature. Throughout this book I'll occasionally be referencing other variables as needed.

Setting the Application Life Cycle Stage

The `.htaccess` file introduced in Chapter 2 serves a primary role of forwarding all requests to the front controller. However, it also serves a secondary role of providing a convenient location to define your application's life cycle stage. For instance, to define the stage as `development`, open the `.htaccess` file (located in the `/public/` directory) and add the following line at the top of the file:

```
SetEnv APPLICATION_ENV development
```

Once saved, the framework will immediately begin using the configuration parameters defined within the `[development]` section of the `application.ini` file.

Tip

You're not constrained to using solely the four default stages defined within the `application.ini` file. Feel free to add as many custom stages as your please!

While defining the `APPLICATION_ENV` in the `.htaccess` file is no doubt convenient, you'll still need to modify this variable when migrating your website from one staging server to another. Neglecting to do so will logically result in unexpected consequences, such as continuing to display website errors within the browser on your production server because you forgot to update the `APPLICATION_ENV` variable to `production`. You can eliminate such gaffes entirely by automating the migration process using a utility such as Phing.

Accessing Configuration Parameters

Naturally you'll want to access some of these configuration parameters within your controllers, and in the case of end-user parameters such as e-mail addresses, within your views. There are several different approaches available for accessing this data. In this section, I'll introduce you to each approach, concluding with the solution which I believe to be most practical for most applications.

Accessing Configuration Data From a Controller Action

Most newcomers to the Zend Framework are happy with simply understanding how to access the configuration data from within a controller action, which is certainly understandable although in most cases it's the most inefficient approach because it results in duplicating a certain amount of code each time you want to access the data from within a different action. Nonetheless it's useful to understand how this is accomplished because if anything it will demonstrate the syntax employed by all approaches. You can use the following command to load all parameters defined within `application.ini` file into an array:

```
$options = $this->getInvokeArg('bootstrap')->getOptions();
```

The `getInvokeArg()` call retrieves an instance of the bootstrap object which is invoked every time the front controller responds to a request. This object includes the `getOptions()` method which can be used to retrieve a multidimensional array consisting of the defined stage's configuration parameters. For instance, you can retrieve the Google Maps API key referenced in an earlier example using the following syntax:

```
echo $options['webservices']['google']['maps']['api'];
```

I think the multidimensional array syntax is a bit awkward to type, and instead prefer an object-oriented variant also supported by the Zend Framework. To use this variant, you'll need to load the parameters into an object by passing the array into the `Zend_Config` class constructor:

```
$options = new Zend_Config($this->getInvokeArg('bootstrap')->getOptions());
```

This approach allows you to use object notation to reference configuration parameters like so:

```
$googleMapsApiKey = $options->webservices->google->maps->api->key;
```

Using the Controller's init() Method to Consolidate Code

If you plan on using configuration parameters throughout a particular controller, eliminate the redundant calls to the `getOptions()` method by calling it from within your controller's `init()` method.

```
public function init()
{
  $this->options = new Zend_Config($this->getInvokeArg('bootstrap')->getOptions());
}
...
public function contactAction()
{
  $this->view->email = $this->options->company->email->support;
}
```

Accessing Configuration Parameters Globally Using Zend_Registry

While retrieving the options within the `init()` method is an improvement over the first approach, we're still not as DRY as we'd like to be if it's necessary to access configuration parameters within multiple controllers. Therefore my preferred approach is to automatically make the options globally

available by assigning the object returned by `Zend_Config` to a variable stored within the application registry. This registry is managed by a Zend Framework component called Zend_Registry. You can use this registry to set and retrieve variables which are accessible throughout the entire application. Therefore by assigning the configuration parameters object to a registry variable from within the bootstrap, this variable will automatically be available whenever needed from within your controllers.

As discussed in the Chapter 2, tasks performed within the application bootstrap are typically grouped into methods, with each method appropriately named to identify its purpose. Each time the bootstrap runs (which occurs with every request), these methods will automatically execute. Therefore to load the configuration object into the registry, you should create a new method within the bootstrap, and call the appropriate commands within, as demonstrated here:

```
protected function _initConfig()
{
  $config = new Zend_Config($this->getOptions());
  Zend_Registry::set('config', $config);
  return $config;
}
```

With the configuration object now residing in a registry variable, you'll be able to retrieve it within any controller action simply by calling the Zend_Registry component's static `get` method. This means you won't have to repetitively retrieve the configuration data from within every controller `init()` method! Instead, you can just retrieve the configuration parameters like this:

```
$this->view->supportEmail =
  Zend_Registry::get('config')->company->email->support;
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Which Zend Framework component is primarily responsible for simplifying the accessibility of project configuration data from a central location?
- What is the name and location of the default configuration file used to store the configuration data?
- Describe how the configuration data organized such that it is possible to define stage-specific parameters.
- What is the easiest way to change your application's life cycle stage setting?

Chapter 5. Creating Web Forms with Zend_Form

When I was first acquainted with the Zend Framework back in 2008, the Zend_Form component was the lone feature which I was convinced was a horribly misguided implementation. I simply could not understand why any sane developer would want to programmatically generate HTML forms when they are so easy to write manually.

As it turns out, it was I who was horribly misguided. While HTML forms can indeed be created in mere minutes, the time and effort required to write the code used to populate, process, validate, and test these forms can be significant. It is *here* where Zend_Form's power is apparent, as it can greatly reduce the time and effort needed to carry out these tasks. Further, you won't lose any control over the ability to format and stylize forms!

In my experience Zend_Form is the most difficult of the Zend Framework's components, largely because of the radical shift towards the programmatic creation of forms. Therefore you'll likely need to remain patient while making your first forays into creating and validating forms using this component. I'll do my best to guide you through the process and make you aware of potential gotchas as we work through the chapter.

> ### Caution
> While this chapter will indeed provide a detailed introduction to `Zend_Form`, I've decided to spend little time talking about the many rendering options at your disposal. Instead, I'm going to focus upon what I believe to be the rendering solution which will appeal to the vast majority of readers who wish to balance Zend_Form's convenient form processing and validation features with the ability to maintain control over the form's layout and styling. However, in order to ensure you fully understand many of the most confusing issues surrounding `Zend_Form`'s approach to rendering forms, several of this chapter's early examples will employ a trial-and-error approach, showing you how the form's appearance changes with each iteration.

Creating a Form with Zend_Form

You'll use Zend_Form's class methods to not only create the form, but also validate form data and even determine how the form is presented to the user. To create a form you'll invoke the `Zend_Form`

class, create the form field objects using a variety of classes such as `Zend_Form_Element_Text`, and then add those form field objects to the form using methods exposed through the `Zend_Form` class.

I'd imagine this sounds pretty elementary, however there's a twist to the approach which causes a great deal of confusion among newcomers to framework-driven development. You'll actually want to encapsulate each form within a model! This is a preferable approach because the model can contain all of the functionality required to manage the form data and behavior, not only resulting in easier maintainability but also allowing you to easily reuse that model within multiple applications. Let's use this approach to create the model used to sign registered GameNomad users into their accounts.

Begin by using the ZF CLI to create the model. You're free to name the model however you please, although I suggest choosing a name which clearly identifies the model as a form. For instance I preface all form-specific models with the string `Form` (for instance `FormLogin`, `FormRegister`, and `FormForgetPassword`):

```
%>zf create model FormLogin
Creating a model at /var/www/dev.gamenomad.com/application/models/FormLogin.php
Updating project profile '/var/www/dev.gamenomad.com/.zfproject.xml'
```

Next open up the `FormLogin.php` file, located in the directory `application/models/`, and add a constructor method containing the following elements (also note that the class definition has also been modified so that it extends the `Zend_Form` class:

```
01 <?php
02
03 class Application_Model_FormLogin extends Zend_Form
04 {
05
06   public function __construct($options = null)
07   {
08
09     parent::__construct($options);
10     $this->setName('login');
11     $this->setMethod('post');
12     $this->setAction('/account/login');
13
14     $email = new Zend_Form_Element_Text('email');
15     $email->setAttrib('size', 35);
16
17     $pswd = new Zend_Form_Element_Password('pswd');
18     $pswd->setAttrib('size', 35);
19
20     $submit = new Zend_Form_Element_Submit('submit');
21
22     $this->setDecorators( array( array('ViewScript',
```

```
23                          array('viewScript' => '_form_login.phtml'))));
24
25     $this->addElements(array($email, $pswd, $submit));
26
27    }
28
29 }
```

Let's review this example:

- Line 03 defines the model. Note how this model extends the `Zend_Form` class. When you generate a model using the zf tool, this extension isn't done by default so you'll need to add the extension syntax manually.
- Line 06 defines a class constructor method. All of the remaining code found in this example is encapsulated in this constructor because we want the code to automatically execute when the model object is created within the controller. Note how this constructor can also accept a lone parameter named `$options`. I'll talk more about the utility of this parameter in the section "Passing Options to the Constructor".
- Line 09 calls the class' parent constructor, which is required in order to properly initialize the `Application_Model_FormLogin` class.
- Line 10 defines the form's name, which can be used to associate CSS styles and Ajax-based functionality. Line 11 defines the form method, which can be set to `get` or `post`. Line 12 defines the form action, which points to the URL which will process the form data. In order to ensure maximum model portability, you may not want to hard wire these values and instead want to pass them through the constructor. I'll show you how this is done in the section "Passing Options to the Constructor".
- Lines 14 and 15 define the text field which will accept the user's e-mail address. The `email` value passed into the `Zend_Form_Element_Text` constructor will be used to set the field's `name`.
- Lines 17 and 18 define the password field which will accept the user's password. The `Zend_Form_Element_Password` class is used instead of `Zend_Form_Element_Text` because the former will present a text field which masks the password as the user enters it into the form.
- Line 20 defines a submit field used to represent the form's `Submit` button.
- Lines 22-23 defines the view script which will be used to render this form. I'll talk more about this form in the next section.

- Line 25 adds all of the form elements defined in lines 12-18 to the form object. It is also possible to add each separately using the `addElement()` method however using `addElements()` will save you a few keystrokes.

Notice how this model places absolutely no restrictions on how the form will actually be presented to the user, other than to reference a script named `_form_login.phtml` which contains the form's formatting instructions (more on this in the next section). Let's move on to learn how the form is rendered.

Rendering the Form

To render a form, all you need to do is instantiate the class within your controller, and then assign that object to a variable made available to the view, as demonstrated here:

```
public function loginAction()
{

  $form = new Application_Model_FormLogin();

  $this->view->form = $form;

}
```

Within the `application/views/account/login` view you'll need to echo the `$this->view`:

```
<?= $this->form; ?>
```

Finally, create the file named `_form_login.phtml` (placing it within `application/views/scripts`) which was referenced within the `FormLogin` model. This file is responsible for rendering the form exactly as you'd like it to appear within the browser.

```
<form id="login" action="<?= $this->element->getAction(); ?>"
                 method="<?= $this->element->getMethod(); ?>">
<p>
E-mail Address<br />
<?= $this->element->email; ?>
</p>
<p>
Password<br />
<?= $this->element->pswd; ?>
</p>

<p>
<?= $this->element->submit; ?>
```

```
</p>

</form>
```

Calling `http://dev.gamenomad.com/account/login` within the browser, you should see the form presented in Figure 5.1.

Figure 5.1. Creating a form with Zend_Form

The form rendered just fine, however you might notice that the spacing seems a bit odd. To understand why, use your browser's *View Source* feature to examine the form HTML. I've reproduced it here for easy reference:

```
<form id="login" action="/account/login" method="post">
<p>
E-mail Address<br />
<dt id="email-label">&#38;</dt>
<dd id="email-element">
<input type="text" name="email" id="email" value="" size="35"></dd></p>
<p>
Password<br />
<dt id="pswd-label">&#38;</dt>
<dd id="pswd-element">
<input type="password" name="pswd" id="pswd" value="" size="35"></dd></p>

<p>
<dt id="submit-label">&#38;</dt><dd id="submit-element">

<input type="submit" name="submit" id="submit" value="Login"></dd></p>

</form>
```

Where did all of those `dt` and `dd` tags come from? They are present because Zend_Form is packaged with a number of default layout *decorators* which will execute even if you define a view script within the model. A decorator is a design pattern which makes it possible to extend the capabilities of an object. In the case of Zend_Form, these decorators determine how each form field will be rendered within the browser. Why the developers chose the `dt` and `dd` tags over others isn't clear, although one would presume it has to do with the ability to easily stylize these tags using CSS. Even so, I doubt you want these decorators interfering with your custom layout, so you'll want to suppress them. This is accomplished easily enough using the `removeDecorator()` method. Because the decorator is associated with each form field object, you'll need to call `removeDecorator()` every time you create a form field, as demonstrated here:

```
$email = new Zend_Form_Element_Text('email');
$email->setAttrib('size', 35)
      ->removeDecorator('label')
      ->removeDecorator('htmlTag');
```

In this example I'm removing the decorator used to remove the default label formatting in addition to the label used to format the field itself. Execute `/account/login` again and you'll see the form presented in Figure 5.2.

Figure 5.2. Removing the default Zend_Form decorators

This is clearly an improvement, however if you again examine the source code underlying this form, you'll see that the submit button is still rendered using a default decorator, even if you explicitly removed the `htmlTag` decorator from the `Zend_Form_Element_Submit` object. This is because the `Zend_Form_Element_Submit` does not support the `htmlTag` decorator. Instead, you'll want to remove the `DtDdWrapper` decorator:

```
$submit = new Zend_Form_Element_Submit('submit');
$submit->setLabel('Login');
$submit->removeDecorator('DtDdWrapper');
```

With this change in place, call `/account/login` anew and you'll see the form presented in Figure 5.3.

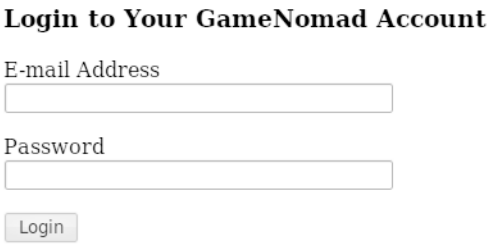


Figure 5.3. Controlling form layout is easy after all!

This is just one approach to maintaining control over your form's presentation when using Zend_Form, and in fact more sophisticated solutions are available. In fact, the easiest solution might involve simply stylizing the `dt` and `dd` tags using CSS. However, for the majority of readers, present party included, the approach described here is quite satisfactory.

Passing Options to the Constructor

I mentioned earlier in this chapter the utility of being able to reuse models across applications. In fact, you'll probably want to reuse models several times within the same application, because of the need to not only insert data, but also later modify it. Although multiple actions will be involved in carrying out these tasks `/location/insert` and `/location/update` for instance), there's no reason you should maintain separate forms! Fortunately, changing the form model's `action` setting is easy, done by passing the desired setting through the form's object constructor:

```
$form = new Application_Model_FormLocation(array('action' => '/locations/add'));
```

You'll also need to modify the form model so that the `setAction()` method refers to the passed associative array value rather than a hardwired setting:

```
$this->setAction($options['action']);
```

Of course, you're not limited to setting solely the form action; just expand the number of associative array keys and corresponding values as you see fit.

Processing Form Contents

Now that you know how to define a form object and render its contents, let's write the code used to process the form input and return feedback to the user. The execution path this task takes depends on whether the user has submitted the form:

- Form not submitted: If the form has not been submitted, render it to the browser, auto-populating the fields if necessary.

- Form submitted: If the form has been submitted, validate the input. If any of the input is deemed invalid, notify the user of the problem, display the form again, and populate the form with the previously submitted input as a convenience to the user. If the form input is valid, process the data and notify the user of the outcome.

Let's tackle each of these problems independently, and then assemble everything together at the conclusion of the section.

Determining if the Form Has Been Submitted

The Zend Framework's request object offers a useful method called `getPost()` which can determine if the incoming request has been made using the POST method. If it has, you can use the request object's `getPost()` method to retrieve the input values (this method was introduced in Chapter 2). The request object's `isPost()` method returns TRUE if the request has been POSTed, and FALSE if not, meaning you can evaluate it within an if-conditional statement, like this:

```
public function loginAction()
{

  $form = new Application_Model_FormLogin();

  if ($this->getRequest()->isPost()) {

    $email = $form->getValue('email');
    $pswd  = $form->getValue('pswd');

    echo "<p>Your e-mail is {$email}, and password is {$pswd}</p>";

  }

  $this->view->form = $form;

}
```

Try executing the revised `/account/login` action, completing and submitting the form. When submitted, you should see your e-mail address and password echoed back to the browser.

Validating Form Input

Of course, the previous example doesn't prevent you from entering an invalid e-mail address or password, including none at all. To make sure a form field isn't blank, you can associate the `setRequired()` method with the form field, like this:

```
$email = new Zend_Form_Element_Text('email');
$email->setAttrib('size', 35);
$email->setRequired(true);
```

Merely adding the validator to your model won't result in it being enforced. You also need to adjust the `login` action so that the `isValid()` method is called, passing the POSTed data as the method's lone parameter:

```
public function loginAction()
{

  $form = new Application_Model_FormLogin();

  if ($this->getRequest()->isPost()) {

    if ($form->isValid($this->_request->getPost()))
    {
      echo "<p>VALID INPUT!</p>";
    }

  }

  $this->view->form = $form;

}
```

With the validator and action logic in place, the Zend Framework will automatically associate an error message with the invalid field even if you override the default form layout using the `setDecorators()` method as we did earlier in the chapter. As an added bonus, it will automatically retain the entered form values (whether valid or not) as a convenience for the user. The error message associated with the e-mail address form field is demonstrated in Figure 5.4.

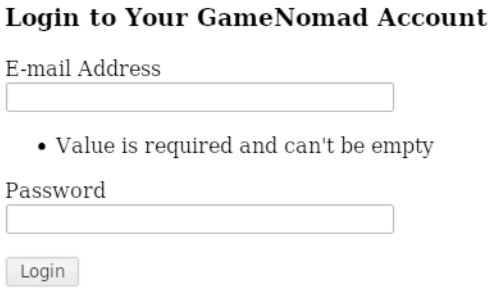


Figure 5.4. Displaying a validation error message

Tip

Chances are you'll want to modify the error messages' default text, or perhaps group all messages elsewhere rather than next to each form field. If so, see the later section "Displaying Error Messages".

While ensuring a field isn't blank is a great idea, you'll often need to take additional validation steps. Zend_Form takes into consideration the vast majority of your validation needs by integrating with another powerful Zend Framework component named Zend_Validate. The Zend_Validate component is packaged with over two dozen validators useful for verifying the syntactical correctness of an e-mail address, credit card number, IP address or postal code, determining whether a string consists solely of digits, alphanumerical characters, and ensuring numbers fall within a certain range. You can also use Zend_Validate to compare data to a regular expression and can even define your own custom validators. A partial list of available validators is presented in Table 5-1.

Table 5.1. Useful Zend_Form Validators

Name	Description
Alnum	Determines whether a value consists solely of alphabetic and numeric characters
Alpha	Determines whether a value consists solely of alphabetic characters
Between	Determines whether a value falls between two predefined boundary values

Name	Description
CreditCard	Determines whether a credit card number meets the specifications associated with a given credit card issuer. All major issuing institutions are supported, including American Express, MasterCard, Solo and Visa.
Date	Determines whether a value is a valid date provided in the format `YYYY-MM-DD`
Db_RecordExists	Determines whether a value is found in a specified database table
Digits	Determines whether a value consists solely of numeric characters
EmailAddress	Determines whether a value is a syntactically correct e-mail address as defined by RFC2822. This validator is also capable of determining whether the domain exists, whether MX records exist, and whether the domain's server is accepting e-mail.
Float	Determines whether a value is a floating-point number
GreaterThan	Determines whether a value is greater than a predefined a minimum boundary
Identical	Determines whether a value is identical to a predefined string
InArray	Determines whether a value is found within a predefined array
Ip	Determines whether a value is a valid IPv4 or IPv6 IP address
Isbn	Determines whether a value is a valid ISBN-10 or ISBN-13 number
NotEmpty	Determines whether a value is not blank
Regex	Determines whether a value meets the pattern defined by a regular expression

You can associate these validators with a form field using the Zend_Form `addValidator()` method. As an example, consider GameNomad's user registration form `/account/register`). Obviously we'll want the user to provide a valid e-mail address when registering, and so define the form field within the registration form model `/application/models/FormRegister.php` like this:

```
$email = new Zend_Form_Element_Text('email');
$email->setAttrib('size', 35);
$email->setRequired(true);
$email->addValidator('emailAddress');
```

Submitting an invalid e-mail address produces the error message depicted in Figure 5.5.

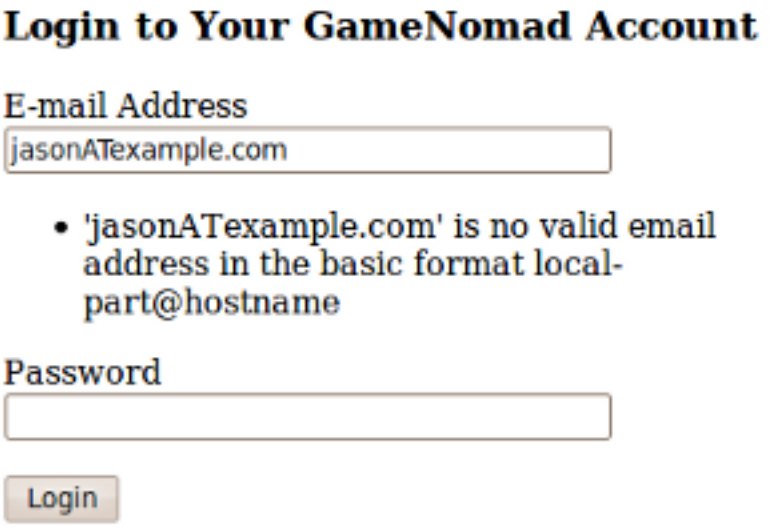


Figure 5.5. Notifying the user of an invalid e-mail address

Several validators require you to specify boundaries in order for the validator to work properly. For instance, the `StringLength` validator will ensure that a string consists of a character count falling between a specified minimum and maximum. This can be useful for making sure that the user chooses a password consisting of a certain number of characters. The following example can be used to make sure that a registering user's password consists of 4-15 characters:

```
$pswd = new Zend_Form_Element_Password('pswd');
$pswd->setAttrib('size', 35);
$pswd->setRequired(true);
$pswd->addValidator('StringLength', false, array(4,15));
```

You might be wondering about the mysterious second parameter in the above reference to `addValidator()`. When specifying boundary values, you'll need to also supply the `addValidator()`'s "chain break" parameter, which is by default set to `false`. This parameter determines whether the next validator will execute if the previous validator fails. Because the default is `false`, the Zend Framework will attempt to execute all validators even if one fails. If you change this value to `true`, validation will halt immediately should one of the validators fail.

Displaying Error Messages

As you witnessed from previous examples, default error messages are associated with each validator. However these messages aren't particularly user friendly, and so you'll probably want to override these messages with versions more suitable to your website audience. To create a custom error message, use the `addErrorMessage()` as demonstrated here:

```
$pswd = new Zend_Form_Element_Password('pswd');
$pswd->setAttrib('size', 35);
$pswd->setRequired(true);
$pswd->addValidator('StringLength', false, array(4,15));
$pswd->addErrorMessage('Please choose a password between 4-15 characters');
```

Customizing Your Messages' Visual Attributes

To further customize these messages, use your browser's *View Source* feature to examine how the error messages are rendered and you'll see that each message is associated with a CSS class named `errors`:

```
<ul class="errors"><li>Please provide a valid e-mail address</li></ul>
```

You can use this CSS class to customize the color, weight and other attributes of these messages.

Grouping Messages

If you prefer to group error messages together rather than intersperse them throughout the form, use `Zend_Form`'s `getErrors()` method. This method returns an associative array consisting of form element names and error messages. This method does behave a bit odd in that it will always return an array associated with the form's submit button, meaning you'll need to account for the blank value when formatting the errors. For instance, the following output is indicative of what you'll find when using PHP's `var_dump()` method to display the array contents:

```
array(3) {
  ["email"]=>  array(1) { [0]=>
    string(37) "Please provide a valid e-mail address" }
  ["pswd"]=>  array(1) {
    [0]=>  string(28) "Please provide your password" }
  ["submit"]=>  array(0) { }
}
```

Of course in order to access this error message array you'll need to pass it to the view. To do so, modify the `Account` controller's `login` action so that the errors are retrieved if the `isValid()` method returns FALSE:

```
if ($form->isValid($this->_request->getPost()))
{
  echo "<p>VALID INPUT!</p>";
} else {
  $this->view->errors = $form->getErrors();
}
```

Using a custom view helper (custom view helpers were introduced in Chapter 3) you can conveniently encapsulate the error message format and display logic, producing output such as that presented in Figure 5.6.

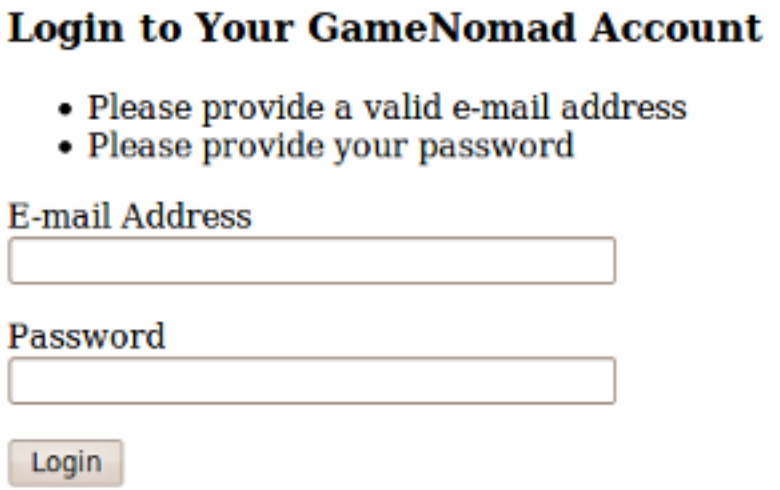


Figure 5.6. Displaying a validation error message

To create the message format shown in Figure 5.6 I've created the following `Errors` view helper (name this file `Errors.php` and place it in your `application/views/helpers/` directory):

```
class Zend_View_Helper_Errors extends Zend_View_Helper_Abstract
{

 /**
  * Outputs errors using a uniform format
  *
  * @param Array $errors
  * @return nil
  */
 public function Errors($errors)
```

```
 {

    if (count($errors) > 0) {
    echo "<div id='errors'>";
      echo "<ul>";
      foreach ($errors AS $error) {
       if ($error[0] != "") {
          printf("<li>%s</li>", $error[0]);
        }
     }
     echo "</ul>";
      echo "</div>";
    }
 }

}
```

With the view helper created, all that's left is to modify the `login.phtml` view to output the errors if any exist:

<h3>Login to Your GameNomad Account</h3> <?= $this->Errors($this->errors); ?> <?= $this->form; ?>

Completing the Process

Should the `isValid()` method return TRUE, meaning that all fields have met their validation requirements, then you'll need to process the data. Exactly what this entails depends upon what you intend on doing with the form data. For instance, you might insert the data into a database, initiate an authenticated user session, or e-mail the form data to a technical support team. All of these tasks are topics for later chapters so while I'd prefer to avoid putting the cart ahead of the horse and dive into concepts that have yet to be introduced, it would be nonetheless useful to offer a complete example which shows you just how succinct your code can really be when taking full advantage of `Zend_Form` and models such as `FormLogin`. The following example presents a typical login action, responsible for presenting the login form, validating submitted form data, attempting to authenticate the user and initiate a new session if the form data is valid, updating the user's account record to reflect the latest successful login timestamp, and displaying errors or other notifications based on the authentication attempt outcome. All of these tasks are accomplished in 50 lines of succinct, user-friendly code! The code is presented, followed by a *brief* summary. Don't worry about understanding all of the syntax for now as I'll be introducing it in great detail in later chapters; instead just marvel at the simple, straightforward approach used to accomplish these tasks.

```
01 public function loginAction()
02 {
```

```
03
04  $form = new Application_Model_FormLogin();
05
06  if ($this->getRequest()->isPost()) {
07
08    if ($form->isValid($this->_request->getPost())) {
09
10      $db = Zend_Db_Table::getDefaultAdapter();
11      $authAdapter = new Zend_Auth_Adapter_DbTable($db);
12
13      $authAdapter->setTableName('accounts');
14      $authAdapter->setIdentityColumn('email');
15      $authAdapter->setCredentialColumn('pswd');
16      $authAdapter->setCredentialTreatment('MD5(?) and confirmed = 1');
17
18      $authAdapter->setIdentity($form->getValue('email'));
19      $authAdapter->setCredential($form->getValue('pswd'));
20
21      $auth = Zend_Auth::getInstance();
22      $result = $auth->authenticate($authAdapter);
23
24      // Did the user successfully login?
25      if ($result->isValid()) {
26
27       $account = new Application_Model_Account();
28
29        $lastLogin = $account->findByEmail($form->getValue('email'));
30
31        $lastLogin->last_login = date('Y-m-d H:i:s');
32
33       $lastLogin->save();
34
35        $this->_helper->flashMessenger->addMessage('You are logged in');
36        $this->_helper->redirector('index', 'index');
37
38      } else {
39       $this->view->errors["form"] = array("Login failed.");
40      }
41
42    } else {
43      $this->view->errors = $form->getErrors();
44    }
45
46  }
47
48  $this->view->form = $form;
49
50 }
```

Let's review the code:

- Line 04 instantiates the `FormLogin` model as has been demonstrated throughout this chapter.
- Line 06 determines whether the form has been submitted. If so, form validation and subsequent attempts to authenticate the user will ensue.
- Lines 10-22 attempt to authenticate the user by consulting a database table named `accounts`. This topic is discussed in great detail in Chapter 8.
- If authentication was successful (as determined by line 25), lines 27-33 update the successfully authenticated user's `last_login` attribute within his database record to reflect the current timestamp.
- Lines 35-36 are responsible for letting the user know he has successfully logged into the site and redirecting him to the home page. This is known as a *flash message*, a great feature I'll introduce in the next section.
- Lines 38 and 44 account for any errors which have cropped up as a result of attempting to authenticate the user. Notice how line 38 in particular embraces the same format used by Zend_Form.

Introducing the Flash Messenger

Your users are busy people, and so will appreciate for any steps you can take to reduce the number of pages they'll need to navigate when creating a new account or logging into the website. For instance, following a successful login it would be beneficial to automatically transport users to the page they will most likely want to visit first. At the same time you'll want to make it clear to the user that he did successfully login to his account. So how can you simultaneously complete both tasks?

Most modern web frameworks, the Zend Framework included, solve this dilemma by offering a feature known as a *flash messenger*. The flash messenger is a mechanism which allows you to create a notification message within one action and then display that message when rendering the view of another action. This feature was demonstrated in lines 35-36 of the previous example:

```
$this->_helper->flashMessenger->addMessage('You are logged in');
$this->_helper->redirector('Index', 'index');
```

The first line of this example uses the built-in flash messenger's `addMessage()` method to define the notification message. Next, the user is redirected to the `Index` controller's `index` action.

Although flash messages may be defined within any action and conceivably displayed in any other, chances are there will only be a select few where the latter will occur. Therefore you could either embed the following code in the action whose view will display a flash message, or within a controller's `init()` method:

```
if ($this->_helper->FlashMessenger->hasMessages()) {
  $this->view->messages = $this->_helper->FlashMessenger->getMessages();
}
```

You can however consolidate the view-specific code to your `layout.phtml` file, adding the following code wherever you would like the messages to appear:

```
<?php
if (count($this->messages) > 0) {
 printf("<div id='flash'>%s</div>", $this->messages[0]);
}
?>
```

This presumes you're only interested in the first message. While it's possible to pass and display several messages, I've not had to do so and therefore am only worried about the first array element.

With the flash messenger integrated, you'll see a flash message displayed after successfully logging into your GameNomad account. This feature is depicted in Figure 5.7.

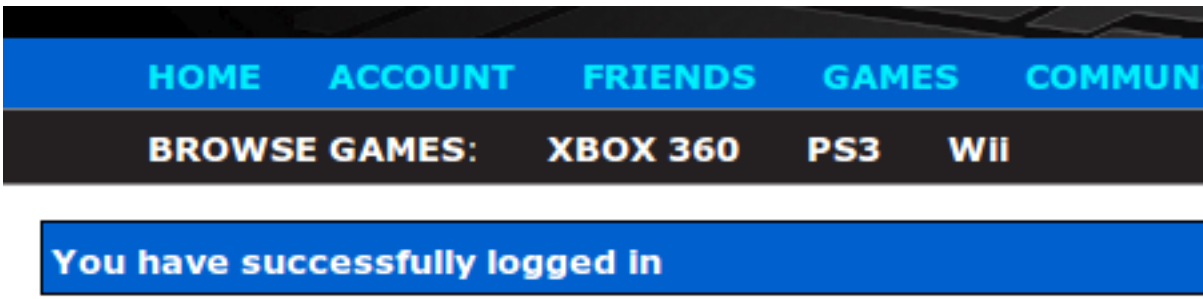


Figure 5.7. Using the flash messenger

Populating a Form

Whether you're creating administrative interfaces for managing product information, or would like to provide registered users with the ability to manage their account profiles, you'll need to know how to prepopulate forms with data retrieved from some data source, presumably a database. As it turns out, populating Zend Framework forms is surprisingly easy, requiring you to simply create an

associative array containing keys which match the form field names, and the keys' respective values which you'd like to prepopulate the fields. With this array created, you'll pass it to the form object's `setDefaults()` method:

```
$form = new Application_Model_FormProfile();

$data = array(
  'username' => 'wjgilmore',
  'email' => 'wj@example.com',
  'zip_code' => '43201'
);

$form->setDefaults($data);

$this->view->form = $form;
```

Populating Select Boxes

All of the examples provided so far in this chapter involve the simplest of form controls, namely text fields and submit buttons. However, many real-world forms will often be much more complex, incorporating a selection of more advanced controls such as check boxes, radio buttons and select boxes. The latter is often a source of confusion to new Zend_Form users because of the need to populate the control with eligible values. As it turns out, once you know the syntax the task is quite easy, requiring you to create an associative array containing the set of select box keys and corresponding values, and then pass that array to the `Zend_Form_Element_Select` object's `AddMultiOptions()` method:

```
$status = new Zend_Form_Element_Select('status');

$options = array(
  1 => "On the Shelf",
  2 => "Currently Playing",
  3 => "For Sale",
  4 => "On Loan"
);

$status->AddMultiOptions($options);
```

While the above approach is useful when you're certain the select box values won't change, its commonplace for these values to be more fluid and therefore preferably retrieved from a database. Therefore at risk of getting ahead of myself, this nonetheless seems an appropriate time to show you at least one of several easy ways to populate a select box dynamically using data retrieved from a database by way of the Zend_Db component. The Zend_Db component includes a useful method called `fetchPairs()` which can retrieve a result set as a series of key-value pairs. This feature is

ideal for populating a select box, since this particular data format is precisely what we want to pass to the `addMultiOptions()` method:

```
$db = Zend_Db_Table_Abstract::getDefaultAdapter();

$options = $db->fetchPairs(
   $db->select()->from('status', array('id', 'name'))
     ->order('name ASC'), 'id');

$status = new Zend_Form_Element_Select('status');

$status->AddMultiOptions($options);
```

Don't worry if this syntax doesn't make any sense, as it will be thoroughly introduced in Chapter 6.

Testing Your Work

There are few tasks more time-consuming and annoying than testing web forms to determine whether they are working correctly. Fortunately, it's possible to automate a great deal of the testing using unit tests. We can create tests which ensure that the form is rendering correctly, that input is properly validated, and that the form data is saved to the database, among others. In this section I'll show you how to write tests which carry out these tasks.

Making Sure the Contact Form Exists

To make sure the contact form exists and includes the expected fields, you can use the `assertQueryCount()` method to confirm that a particular element and associated DIV ID exist within the rendered page, as demonstrated here:

```
public function testContactActionContainsContactForm()
{
  $this->dispatch('/about/contact');
  $this->assertQueryCount('form#contact', 1);
  $this->assertQueryCount('input[name~="name"]', 1);
  $this->assertQueryCount('input[name~="email"]', 1);
  $this->assertQueryCount('input[name~="message"]', 1);
  $this->assertQueryCount('input[name~="submit"]', 1);
}
```

Testing Invalid Form Values

The Zend Framework's input validators have been thoroughly tested by the development team, so when testing your forms the concern doesn't lie in making sure validators such as the `EmailAddress` validator are properly detecting invalid e-mail addresses, but rather in making sure that you have

properly integrated the validators into your form model. Let's create a test which determines whether GameNomad's contact form (`http://www.gamenomad.com/about/contact`) is properly validating the supplied input before e-mailing the contact request to the support staff. This form is presented in Figure 5.8.

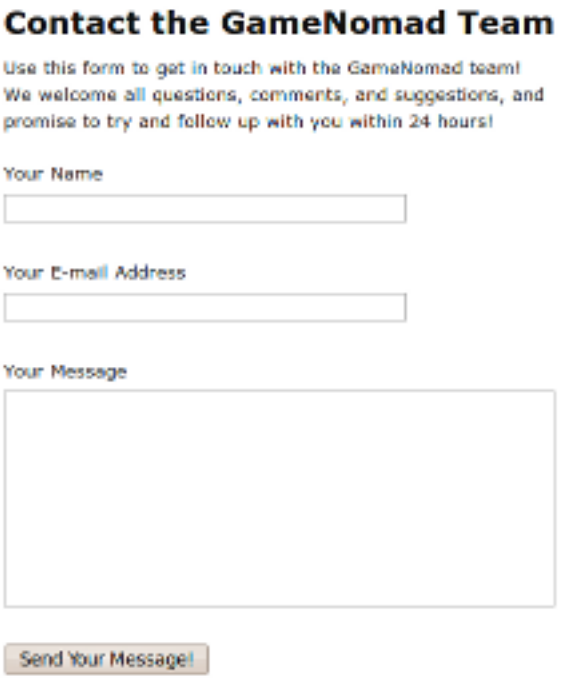


Figure 5.8. GameNomad's Contact Form

This form requires the visitor to supply a name, e-mail address, and a message, with the validators ensuring the name and message fields aren't blank, and that the e-mail address field contains a syntactically valid e-mail address. Should any of these validations fail, the errors will be rendered to the page using the custom `Errors` view helper introduced earlier in this chapter. If the validations all pass, then the e-mail will be sent and the user will be redirected to the home page. Therefore we can generally determine whether any validations fail by asserting the redirection hasn't occurred, or more specifically by examining the `errors` DIV element used to display the error messages. Let's run with the former scenario and in later chapters I'll show you how to focus on specific error messages to determine exactly what failed.

Because there exists far more than one set of invalid data, we're going to use a great PHPUnit feature known as a *data provider* to iterate over multiple sets of invalid data in order to ensure the validators are properly detecting multiple errant fields. You'll place these invalid permutations within an array found in an aptly-named method, placing the method within the `AboutControllerTest.php` class:

```
public function invalidContactInfoProvider()
{
  return array(
    array("Jason Gilmore", "", "Name and Message but no e-mail address"),
    array("", "wj@example.com", "E-mail address and message but no name"),
    array("", "", "No name or e-mail address"),
    array("No E-mail address or message", "", ""),
    array("Jason Gilmore", "InvalidEmailAddress", "Invalid e-mail address")
  );
}
```

Notice how this array is returned the moment this method is executed. This is required in order for PHPUnit's data provider feature to operate properly. Next, we'll define the test which uses this data provider:

```
01 /**
02 * @dataProvider invalidContactInfoProvider
03 */
04 public function testIsInvalidContactInformationDetected($name, $email, $message)
05 {
06
07   $this->request->setMethod('POST')
08        ->setPost(array(
09            'name'    => $name,
10            'email'   => $email,
11            'message' => $message
12          ));
13
14   $this->dispatch('/about/contact');
15
16   $this->assertNotRedirectTo('/');
17
18 }
```

This example includes several important test-related features, so let's review the code:

- Line 02 defines the data provider method used by the test using the `@dataProvider` annotation. You *must* include this line in order for PHPUnit to be able to access the array of test values found in the data provider method!

- Notice how line 04 is passing in three input parameters which correspond to the three elements found in each instance of the data provider's multidimensional array. Obviously you will adjust this number upwards or downwards depending upon the number of test values found in other data providers.
- Lines 07-12 set the request method to POST, and assign the three input parameters to an associative array which will be sent along with the POST request.
- Line 14 issues the resource request, identifying the `About` controller's `contact` action.
- Finally, line 16 ensures that the user is not redirected to the GameNomad home page, which would mean that at least one of the provided data sets validated properly.

Testing Valid Form Values

The previous test ran the GameNomad contact feature through a battery of scenarios involving invalid user data. Likewise, we'll also want to verify that the feature will properly accept and process valid input. This test behaves identically to the previous, except that this time we can just pass one set of valid input, and additionally use the `assertRedirectTo()` method rather than the `assertNotRedirectTo()` method to ensure the redirection occurs as expected:

```
public function testIsValidContactInformationEmailedToSupport()
{

  $this->request->setMethod('POST')
       ->setPost(array(
           'name'        => 'Jason Gilmore',
           'email'       => 'wj@wjgilmore.com',
           'message'     => "This is my test message."
         ));

  $this->dispatch('/about/contact');

  $this->assertRedirectTo('/');

}
```

When the provided user input is deemed valid, GameNomad's `contact` action will send an e-mail containing the visitor's message and contact information to the e-mail address defined within the application configuration file's `email.support` parameter. Because you'll presumably be regularly running the test suite, consider pointing the e-mails to a specially designated testing account by overriding the `email.support` parameter within the configuration file's `[testing : production]` section.

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Name two reasons why the Zend_Form component is preferable to creating and processing forms manually.
- Describe in a paragraph how Zend_Form can be configured so as to allow certain forms to be used for both inserting and later modifying data.
- How does the Flash Messenger feature streamline a user's website interaction?
- What is the role of PHPUnit's data provider feature?

Chapter 6. Talking to the Database with Zend_Db

Even the simplest website will likely rely upon a database for data management, meaning you're going to devote a great deal of time writing code which passes data between the PHP logic responsible for driving your application and the SQL code which interacts with the database. This common practice of mixing SQL with the rest of your website's logic is counteractive to the goal of separating the application's data, logic, and presentation tiers. So what's a developer to do? After all, it's clearly not possible to do away with the database, but using one at the cost of sacrificing efficiency and sound development practices seems an impractical tradeoff.

The Zend Framework attempts to lessen the pain by providing an object-oriented interface named Zend_Db which you can use to talk to a database without having to intermingle SQL and application logic. In this chapter you'll learn all about Zend_Db, along the way gaining valuable experience building key features which will provide you with a sound working understanding of this important aspect of the Zend Framework.

Because the Zend_Db component is packed with features, this chapter introduces a great deal of material. Therefore I thought it would be worthwhile to summarize the sections according to their order of appearance so as to provide you with an easy way to later reference the material:

- *Introducing Object-relational Mapping*: This chapter kicks off with an introduction to *object-relational mapping*, an approach to database access upon which the Zend Framework's Zend_Db is built.

- *Introducing Zend_Db*: The Zend_Db component is the Zend Framework's primary conduit for talking to a database. In this step I'll introduce you to this component, which is so powerful that it almost manages to make database access fun.

- *Creating Your First Model*: When using Zend_Db, you'll rely upon a series of classes (known as models) which serve as the conduits for talking to your database. These models expose the underlying database tables via a series of attributes and methods, meaning you'll be able to query and manage data without having to write SQL queries! In this step I'll show you how to create a model for managing video game data.

- *Querying Your Models*: With the video game model created, we can begin retrieving data from the database using the query syntax exposed through the Zend_Db component. In this step I'll

introduce you to this syntax, showing you how to retrieve data from the database in a variety of useful ways.

- *Creating a Row Model*: Row models give you the ability to query and manipulate specific rows within your database tables. In this step I'll show you how to configure and use this powerful feature.

- *Inserting, Updating, and Deleting Data*: Just as you can retrieve data through the Zend_Db component, so can you use it to insert, update, and delete data. In this step I'll show you how.

- *Creating Model Relationships*: Zend_Db supports the ability to model table relationships, allowing you to deftly interact with the database in amazingly convenient ways. In my experience this is one of the component's most compelling features, and in this step I'll show you how to take advantage of this feature by defining a model which we will use to manage gaming platforms (such as Xbox 360 and Nintendo Wii), and tying it to the video game model so we can associate each game with its platform.

- *JOINing Your Data*: Most of your time will be spent dealing with simple queries, however you'll occasionally require a more powerful way to assemble your data. In this step I'll introduce you to the powerful SQL statement known as the *join*, which will open up a myriad of new possibilities to consider when querying the database.

- *Creating and Managing Views*: As the complexity of your data grows, so will the SQL queries used to interact with it. Rather than repeatedly refer to these complex queries within your code, you can bundle them into what's known as *a view*, which stores the query within the database. You'll then be be able to call the query associated with the view using a simple alias rather than repeatedly insert the complex query within your code. In this section I'll show you how to create and manage views within your Zend Framework-driven websites.

- *Paginating Results with Zend_Paginator*: When dealing with large amounts of data, you'll probably want to spread the data across several pages, or *paginate* it, so the user can easily navigate the data without having to endure long page loading times. But manually splitting retrieved data into multiple pages is a more difficult task than you might think; thankfully the Zend_Paginator component can do the dirty work for you, and in this step I'll show you how to use it.

Before continuing, I'd like to point out that while the Zend Framework's Zend_Db component does a fine role of encapsulating the application's database functionality, it lacks many of the conveniences enjoyed by similar implementations found within other frameworks, including notably Ruby on Rails (http://rubyonrails.org/). This isn't due to oversight, but is rather the result of the Zend Framework developers' particular philosophy in terms of providing a solution which serves as *the starting point*

for building more sophisticated features. Notably, Zend_Db supports the ability to create table gateways, table mappers, and associated model classes, however this approach can quickly become time-consuming, complex, and tedious. Therefore I've made the perhaps controversial although I believe pragmatic decision to spend this chapter introducing you to only Zend_Db's fundamental features, and will not guide you down the path of creating your own ORM implementation as described in the Zend Framework Quickstart. Instead, if you require a more sophisticated solution than what's available by way of the fundamentals discussed in this chapter, I suggest you consider Doctrine, a full-blown ORM solution introduced in Chapter 6. Although at the time of this writing Doctrine was not natively supported by the Zend Framework, there exists a relatively straightforward way to use Doctrine in conjunction with your Zend Framework applications, and in the next chapter I'll show you how this is accomplished.

Introducing Object-Relational Mapping

Object-relational mapping (ORM) tools provide the developer with an object-oriented database interface for interacting with the underlying database. Although the implementation details vary according to each ORM solution, generally speaking an ORM will map a class to each database table, with the former serving as the programmatic conduit for manipulating the latter. This class not only provides a seamless table interface, performing tasks such as selecting, inserting, updating, and deleting data, but can also be extended to incorporate other features such as data validation. Best of all, because the chosen ORM is typically written in the same language as that used for the rest of the application, the developer is no longer inconvenienced by the need to repeatedly digress. Further, the isolation of database-related actions allows you to effectively maintain the desired tier separation espoused by web frameworks.

If like me your PHP knowledge outweighs your SQL acumen, the object-oriented database interface is a welcome feature. Further, you'll no longer have to haphazardly intermingle PHP and SQL syntax, an approach which is probably the largest contributor to the mess known as "spaghetti code". Instead, you'll use a series of exposed methods and other features made available through the ORM to cleanly and concisely integrate the database into your website.

Because ORM is such an attractive solution, a variety of open source and commercial ORM projects are currently under active development. PHP is no exception, and in the next chapter I'll introduce you to Doctrine, which is widely considered to be one of the PHP community's most prominent ORM solutions. The Zend Framework too offers its own native ORM solution, packaged into a component called Zend_Db. Let's take an introductory look at Zend_Db, examining a typical bit of code used to retrieve a video game according to its Amazon.com ASIN (Amazon Standard Identification Number):

```
$gameTable = new Application_Model_DbTable_Game();
```

```
$select = $gameTable->select()->where('asin = ?', 'B002BSA20M');
$this->view->game = $gameTable->fetchOne($select);
```

From within the associated view you can access the record's columns using the familiar object-oriented syntax:

```
<h3><?= $this->game->name; ?></h3>
<img src="/images/games/<?= $this->game->image_medium; ?>" style="float: left;" />
<p>
Release date: <?= date("F j, Y", strtotime($this->game->release_date)); ?>
</p>
```

Using this intuitive object-oriented interface, it's easy to create video game profile pages such as the one presented in Figure 6.1.

Figure 6.1. Building a game profile page using Zend_Db

Let's move on to consider how an ORM solves a more sophisticated problem. Most ORM solutions, Zend_Db included, are capable of intuitively handling the often complex relations defined within a database schema. For instance, suppose you were updating a table named `ranks` with the daily sales rankings of the video games stored within your database as determined by Amazon.com's sales volume. Because the `sales_ranks` table includes a foreign key which points to a record found within the `games` table, we're dealing with a *one-to-many* relationship, meaning that one game is related to multiple sales rank entries. You want to provide users with a summary highlighting these historical rankings, and so want to create a page which identifies the game and provides a tabular summary of

the rankings over a given period. Provided you've properly configured the relationship within your models (a subject we'll discuss in some detail later in the chapter), retrieving these rankings using Zend_Db is trivial:

```
$gameTable = new Application_Model_DbTable_Game();

$select = $gameTable->select()->where('asin = ?', 'B002BSA20M');
$this->view->game = $gameTable->fetchOne($select);

$this->view->rankings =
   $this->view->game->findDependentRowset('Application_Model_DbTable_Rank')
```

With the `$rankings` view scope variable defined, you can iterate over it within the view using PHP's `foreach` statement:

```
foreach ($this->view->rankings AS $ranking) {
  printf("Date: %s, Rank: %i<br />", $ranking->created_on, $ranking->$rank);
}
```

These examples only provide a taste of what Zend_Db's capabilities. Throughout the remainder of this chapter I'll introduce you to a vast selection of other useful Zend_Db features.

Introducing Zend_Db

The Zend_Db component provides Zend Framework users with a flexible, powerful, and above all, easy, solution for integrating a database into a website. It's easy because Zend_Db almost completely eliminates the need to write SQL statements (although you're free to do so if you'd like), instead providing you with an object-oriented interface for retrieving, inserting, updating, and deleting data from the database.

Connecting to the Database

Built atop PHP's PDO extension, Zend_Db supports a number of databases including MySQL, DB2, Microsoft SQL Server, Oracle, PostgreSQL, SQLite, and others. Connecting to the desired database is typically done by defining the desired database adapter and connection parameters within the `application.ini` file (the purpose of this file was introduced in Chapter 5), so let's begin by using the ZF CLI to configure your application to use a MySQL database. Enter your project's root directory and execute the following command:

```
%>zf configure db-adapter \
> "adapter=PDO_MYSQL&amp; \
> host=localhost&amp; \
```

```
> username=gamenomad_user&amp; \
> password=secret&amp; \
> dbname=gamenomad_dev" development
A db configuration for the development section has been written to the
application config file.
```

Executing this command will result in the following five parameters being added to the `development` section of the `application.ini` file:

```
resources.db.adapter               = PDO_MYSQL
resources.db.params.host           = localhost
resources.db.params.username       = gamenomad_user
resources.db.params.password       = secret
resources.db.params.dbname         = gamenomad_dev
```

Believe it or not, adding these parameters to your `application.ini` file is all that's required to configure your database. Next, create the database which you've associated with the `resources.db.params.dbname` parameter if you haven't already done so, and within it create the following table:

```
CREATE TABLE games (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  asin VARCHAR(255) NOT NULL,
  name VARCHAR(255) NOT NULL,
  price DECIMAL(5,2) NOT NULL,
  publisher VARCHAR(255) NOT NULL,
  release_date DATE NOT NULL
);
```

We'll use this table as the basis for several initial examples in order to acquaint you with Zend_Db's fundamental features.

Creating Your First Model

You'll use Zend_Db to interact with the database data via a series of classes, or *models*. Each model is configured to represent the database tables and even the rows associated with a table. I'll show you how to create and interact with row-level models later in this chapter, so for now let's concentrate on table-level models, which extend Zend_Db's `Zend_Db_Table_Abstract` class.

As usual, the best way to learn how all of this works is by using it. So let's begin by creating a class which will serve as the model for interacting with the `games` table. You can generate this model using the ZF CLI, which is always the recommended way to create new application components when possible:

```
%>zf create db-table Game
```

At the time of this writing the zf utility was capable of doing little more than creating the class skeleton and saving the file to the `application/models/DbTable` directory, although I expect its capabilities to improve in future versions. Nonetheless, the tool serves as a useful tool for getting started, so once the model is created open it `application/models/DbTable/Game.php`) and update the class so it looks exactly like the following:

```
01 class Application_Model_DbTable_Game extends Zend_Db_Table_Abstract
02 {
03     protected $_name = 'games';
04     protected $_primary = 'id';
05 }
```

Although just five lines of code, there are some pretty important things happening in this listing:

- Line 01 defines the name of the model `Application_Model_DbTable_Game`), and specifies that the model should extend the `Zend_Db_Table_Abstract` class. The latter step is important because in doing so, the `Application_Model_DbTable_Game` model will inherit all of the traits the Zend_Db grants to models. As for naming your model, I prefer to use the singular form of the word used for the corresponding table name (in this case, the model name is `Application_Model_DbTable_Game` (although the first two parts of the name are just prefixes, so when discussing your models with others it's common to just refer to the model name, in this case, as `Game`), and the table name is `games`). It's very important you understand that this model's `Application_` prefix identifies it as being intended for the website's default module, which is the module created when a new Zend Framework project is created using the ZF CLI. If you wanted to create a model intended for a blog module, you would name the model something like `Blog_Model_Entry`. You would place this model in the blog module's model directory rather than the default model directory. See Chapter 2 for more information about the Zend Framework's modular architecture feature.

- Because of my personal preference for using singular form when naming models, line 03 overrides the Zend Framework's default behavior of presuming the model name exactly matches the name of the database table. Neglecting to do this will cause an error, because the framework will presume your database table name is `game`, rather than `games`.

- Line 04 identifies the table's primary key. By default the Zend framework will presume the primary key is an automatically incrementing integer named `id`, so this line is actually not necessary in the case of the `games` table; I prefer to include the line simply as a matter of clarification for fellow developers. Of course, if you were using some other value as a primary key, for instance a person's social security number, you would need to identify that column name instead.

Congratulations, you've just created an interface for talking to the database's `games` table. What next? Let's start by retrieving some data.

Querying Your Models

It's likely the vast majority of your time spent with the database will involve retrieving data. Using the Zend_Db component selecting data can be done in a variety of ways. In this section I'll demonstrate several of the options at your disposal.

Querying by Primary Key

The most commonplace method for retrieving a table row is to query by the row's primary key. The following example queries the database for the row associated with the primary key 1:

```
$gameTable = new Application_Model_DbTable_Game();
$game = $gameTable->find(1);
echo "{$game[0]->name} (ASIN: {$game[0]->asin})";
```

Returning:

```
Call of Duty 4: Modern Warfare (ASIN: B0016B28Y8)
```

But why do we even have to deal with index offsets in the first place? After all, using the primary key implies there should only be one result anyway, right? This is because the `find()` method also supports the ability to simultaneously query for multiple primary keys, like this:

```
$game = $gameTable->find(array(1,4));
```

Presuming both of the primary keys exist in the database, the row associated with the primary key 1 will be found in offset 0, and the row associated with the primary key 4 will be found in offset 1.

Because in most cases you'll probably use the `find()` method to retrieve just a single value, you'll likely want to eliminate the need to refer to an index offset by using the `current()` method:

```
$gameTable = new Application_Model_DbTable_Game();
$game = $gameTable->find(1)->current();
echo "{$game->name} (ASIN: {$game->asin})";
```

Querying by a Non-key Column

You'll inevitably want to query for rows using criteria other than the primary key. For instance, various features of the GameNomad site retrieve games according to their ASIN. If you only need to search by ASIN at a single location within your site, you can hardcode the query, like so:

```
$gameTable = new Application_Model_DbTable_Game();
$query = $gameTable->select();
$query->where("asin = ?", "B0016B28Y8");
$game = $gameTable->fetchRow($query);
echo "{$game->name} (ASIN: {$game->asin})";
```

Note that unlike when searching by primary key, there's no need to specify an index offset when referencing the result. This is because the `fetchRow()` method will always return only one row.

Because it's likely you'll want to search by ASIN at several locations within the website, the more efficient approach is to define a `Game` class method for doing so:

```
function findByAsin($asin) {
  $query = $this->select();
  $query->where('asin = ?', $asin);
  $result = $this->fetchRow($query);
  return $result;
}
```

Notice the use of the `$this` object when executing the `select()` method. This is because we're inside the `Application_Model_DbTable_Game` class, so `$this` can be used to refer to the calling object, saving you a bit of additional coding.

Now searching by ASIN couldn't be easier:

```
$gameTable = new Application_Model_DbTable_Game();
$game = $gameTable->findByAsin('B0016B28Y8');
```

Retrieving Multiple Rows

To retrieve multiple rows based on some criteria, you can use the `fetchAll()` method. For instance, suppose you wanted to retrieve all games with a price higher than $44.99:

```
$game = new Application_Model_DbTable_Game();
$query = $game->select();
$query->where('price > ?', 44.99);
$results = $this->fetchAll($query);
```

The `fetchAll()` method returns an array of objects, meaning to loop through these results you can just use PHP's native `foreach` construct:

```
foreach($results AS $result) {
  echo "{$result->name} ({$result->asin})<br />";
```

```
}
```

Custom Search Methods in Action

Your searches don't have to be restricted to retrieving records based on a specific criteria. For instance, the following class method retrieves all games in which the title includes a particular keyword:

```
function getGamesMatching($keywords)
{
  $query = $this->select();
  $query->where('name LIKE ?', "%$keywords%");
  $query->order('name');
  $results = $this->fetchAll($query);
  return $results;
}
```

You can then use this method within a controller action like this:

```
// Retrieve the keywords
$this->view->keywords = $this->_request->getParam('keywords');

$game = new Application_Model_DbTable_Game();
$this->view->games = $game->getGamesMatching($this->view->keywords);
```

Counting Rows

All of the examples demonstrated so far have presumed one or more rows will actually be returned. But what if the primary key or other criteria aren't found in the database? Zend_Db allows you to use standard PHP syntactical constructs to not only loop through results, but count them. Therefore, the easiest way to count your results is using PHP's `count()` function. I typically use `count()` within the view to determine whether entries have been returned from a database query:

```
<?php if (count($this->games) > 0) { ?>

  <h3>New Games</h3>

  <?php foreach($this->games AS $game) { ?>
      <p><?= $game->name; ?></p>
  <?php } ?>

<?php } else { ?>

  <p>
    No new games have been added over the past 24 hours.
```

```
  </p>

<?php } ?>
```

Selecting Specific Columns

So far we've been retrieving all of the columns in a given row, but what if you only wanted to retrieve each game's name and ASIN? Using the `from()` method, you can identify specific columns for selection:

```
$gameTable = new Application_Model_DbTable_Game();
$query = $gameTable->select();
$query->from('games', array('asin', 'title'));
$query->where('asin = ?', 'B0016B28Y8');
$game = $gameTable->fetchRow($query);
echo "{$game->name} (\${$game->price})";
```

Ordering the Results by a Specific Column

To order the results according to a specific column, use the `ORDER` clause:

```
$game = new Application_Model_DbTable_Game();
$query = $game->select();
$query->order('name ASC');
$rows = $game->fetchAll($query);
```

To order by multiple columns, pass them to the `ORDER` clause in the order of preferred precedence, with each separated by a comma. The following example would have the effect of ordering the games starting with the earliest release dates. Should two games share the same release date, their precedence will be determined by the price.

```
$query->order('release_date ASC, price ASC');
```

Limiting the Results

To limit the number of returned results, you can use the `LIMIT` clause:

```
$game = new Application_Model_DbTable_Game();
$query = $game->select();
$query->where('name LIKE ?', $keyword);
$query->limit(15);
$rows = $game->fetchAll($query);
```

You can also specify an offset by passing a second parameter to the clause:

```
$query->limit(15, 5);
```

Executing Custom Queries

Although Zend_Db's built-in query construction capabilities should suffice for most situations, you might occasionally want to manually write and execute a query. To do so, you can just create the query and pass it to the `fetchAll()` method, however before doing so you'll want to filter it through the `quoteInto()` method, which will filter the data by delimiting the string with quotes and escaping special characters.

In order to take advantage of this feature you'll need to add the following line to your `application.ini` file. I suggest adding it directly below the five lines which were generated when you executed the ZF CLI's `configure db-adapter` command:

```
resources.db.isDefaultTableAdapter = true
```

This line will signal to the Zend Framework that the database credentials found within the configuration file should be considered the application's default. You'll then obtain a database connection handler using the `getResource()` method, as demonstrated in the first line of the following example:

```
$db = $this->getInvokeArg('bootstrap')->getResource('db');
$name = "Cabela's Dangerous Hunts '09";
$sql = $db->quoteInto("SELECT asin, name FROM games where name = ?", $name);
$results = $db->fetchAll($sql);
echo count($results);
```

You can think of the `quoteInto()` method as a catch-all for query parameters, both escaping special characters and delimiting it with the necessary quotes.

Querying Your Database Without Models

Before moving on to other topics, I wanted to conclude this section with an introduction to an alternative database query approach which might be of interest if you're building a fairly simple website. As of the Zend Framework 1.9 release, you can query your tables without explicitly creating a model. Instead, you can just pass the database table name to the concrete `Zend_Db_Table` constructor, like this:

```
$gameTable = new Zend_Db_Table('games');
```

You'll then be able to take advantage of all of the query-related features introduced throughout this section. This approach can contribute towards trimming your project's code base, so be sure to use it

for those models you won't need to extend via custom methods. Because it's typical to extend most models with at least one custom feature, I'll continue using the more advanced approach throughout the book.

Creating a Row Model

It's important that you understand the `Game` model created in the previous section represents the `games` table, and not each specific record (or row) found in that table. For example, you might use this `Game` model to retrieve a particular row, determine how many rows are found in the table, or figure out what row contains the highest priced game. However, when performing operations *specific* to a certain row, such as finding the most recent sales rank of a row you've retrieved using the `Game` model, you'll want to associate a row-specific model with the corresponding table-specific model. To do so, add this line to the `Application_Model_DbTable_Game` model defined within `Game.php`:

```
protected $_rowClass = 'Application_Model_DbTable_GameRow';
```

Next, create the `Application_Model_DbTable_GameRow` model using the ZF CLI:

```
%>zf create db-table GameRow
```

A class file named `GameRow.php` will be created and placed within the `application/models/DbTable` directory. Just as when you created the `Game` table model, you'll need to do a bit of additional work before the `GameRow` model is functional. Open the `GameRow` model and replace the existing code with the following contents (note in particular that this class extends `Zend_Db_Table_Row_Abstract` as compared to a table-level model which extends `Zend_Db_Table_Abstract`):

```
class Application_Model_DbTable_GameRow extends Zend_Db_Table_Row_Abstract
{
  function latestSalesRank()
  {
    $rank = new Application_Model_DbTable_Rank();
    $query = $rank->select('rank');
    $query->where('game_id = ?', $this->id);
    $query->order('created_at DESC');
    $query->limit(1);
    $row = $rank->fetchRow($query);
    return $row->rank;
  }
}
```

This row-level model contains a single method named `latestSalesRank()` which will retrieve the latest recorded sales rank associated with a specific game by querying another table represented by

the `Rank` model. To demonstrate this feature, suppose you wanted to output the sales ranks of all video games released to the market before January 1, 2011. First we'll use the `Game` model to retrieve the games stored in the database. Second we'll iterate through the array of games (which are objects of type `Application_Model_DbTable_GameRow`), calling the `latestSalesRank()` method to output the latest sales rank:

```
$gameTable = new Application_Model_DbTable_Game();
$query = $gameTable->select()->where("release_date < ?", "2011-01-01");
$results = $gameTable->fetchAll($query);

foreach($results AS $result)
{
  echo "{$result->name} (Sales Rank: {$result->latestSalesRank()})<br />";
}
```

Executing this snippet produces output similar to the following:

```
Call of Duty 4: Modern Warfare (Sales Rank: 14)
Call of Duty 2 (Sales Rank: 2,208)
NBA 2K8 (Sales Rank: 475)
NHL 08 (Sales Rank: 790)
Tiger Woods PGA Tour 08 (Sales Rank: 51)
```

Inserting, Updating, and Deleting Data

You're not limited to using Zend_Db to simply retrieve data from the database; you can also insert new rows, update existing rows, and delete them.

Inserting a New Row

To insert a new row, you can use the `insert()` method, passing an array of values you'd like to insert:

```
$gameTable = new Application_Model_DbTable_Game();

$data = array(
  'asin' => 'B0028IBTL6',
  'name' => 'Fallout: New Vegas',
  'price' => '59.99',
  'publisher' => 'Bethesda',
  'release_date' => '2010-10-19'
);

$gameTable->insert($data);
```

Updating a Row

To update a row, you can use the `update()` method, passing along an array of values you'd like to change, and identifying the row using the row's primary key or another unique identifier:

```
$gameTable = new Application_Model_DbTable_Game();

$data = array(
  'price' => 49.99
);

$where = $game->getAdapter()->quoteInto('id = ?', '42');

$gameTable->update($data, $where);
```

Alternatively, you can simply change the attribute of a row loaded into an object of type `Zend_Db_Table_Abstract`, and subsequently use the `save()` method to save the change back to the database:

```
$gameTable = new Application_Model_DbTable_Game();

// Find NBA 2K11
$game = $gameTable->findByAsin('B003IME9UO');

// Change the price to $39.99
$game->price = 39.99;

// Save the change to the database
$game->save();
```

Deleting a Row

To delete a row, you can use the `delete()` method:

```
$gameTable = new Application_Model_DbTable_Game();

$where = $gameTable->getAdapter()->quoteInto('asin = ?', 'B003IME9UO');

$gameTable->delete($where);
```

Creating Model Relationships

Because even most rudimentary database-driven websites rely upon multiple related tables, it's fair to say you'll spend a good deal of time writing code which manages and navigates these relations.

Recognizing this, the Zend developers integrated several powerful features capable of dealing with related data. Notably, these features allow you to transparently treat a related row as another object attribute. Of course, these relations are only available when a normalized database is used, meaning you'll need to properly structure your database schema using primary and foreign keys. To demonstrate how Zend_Db can manage relations, start by altering the `games` table to include a foreign key which will reference a row within a table containing information about available gaming platforms:

```
mysql>ALTER TABLE games ADD COLUMN platform_id TINYINT UNSIGNED
    ->NOT NULL AFTER id;
```

Next create the `platforms` table:

```
CREATE TABLE platforms (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  abbreviation VARCHAR(10) NOT NULL
);
```

Finally, create the `Platform` model, which we'll use to access the newly created `platforms` table:

```
%>zf create db-table Platform
```

Once created, update the class found in `Platform.php` in the same manner we did with the `Game` model at the beginning of this chapter.

With the schema updated and necessary models in place you'll next need to configure the relationships within your models. The `games` table is dependent upon the `platforms` table, so let's start by defining the `Game` model's subservient role within the `Platform` model. Update the `Platform` model to include the protected attribute presented on Line 07 of the following listing:

```
01 class Application_Model_DbTable_Platform extends Zend_Db_Table_Abstract
02 {
03
04    protected $_name = 'platforms';
05    protected $_primary = 'id';
06
07    protected $_dependentTables = array('Application_Model_DbTable_Game');
08 }
```

Line 07 defines the relationship, informing Zend_Db of the existence of a column within the `Game` model which stores a foreign key pointing to a row managed by the `Platform` model. If a model happens to be a parent for more than one other model, for instance the `Game` model is a parent to

the `Rank` and the `AccountGame` models, you would revise the `$_dependentTables` attribute to look like this:

```
protected $_dependentTables = array('Application_Model_DbTable_Rank', 'Application_Model_DbTab
```

Returning to defining the relationship between the `Game` and `Platform` models, you'll also need to reciprocate the relationship within the `Game` model, albeit with somewhat different syntax because this time we're referring to the parent `Platform` model:

```
01 protected $_referenceMap = array (
02   'Platform' => array (
03     'columns'       => array('platform_id'),
04     'refTableClass' => 'Application_Model_DbTable_Platform'
05   )
06 );
```

In this snippet we're identifying the foreign keys found in the `Game` model's associated schema, identifying both the column storing the foreign key (`platform_id`), and the model that foreign key represents (`Platform`). Of course, it's entirely likely for a model to store multiple foreign keys. For instance, a model named `Account` might refer to three other models (`State`, `Country`, and `Platform`, the latter of which is used to identify the account owner's preferred platform):

```
protected $_referenceMap = array (
  'State' => array (
    'columns'       => array('state_id'),
    'refTableClass' => 'Application_Model_DbTable_State'
  ),
  'Country' => array (
    'columns'       => array('country_id'),
    'refTableClass' => 'Application_Model_DbTable_Country'
  ),
  'Platform' => array (
    'columns'       => array('platform_id'),
    'refTableClass' => 'Application_Model_DbTable_Platform'
  )
);
```

With the models' relationship configured, quite a few new possibilities suddenly become available. For instance, you can retrieve a game's platform name using this simple call:

```
$game->findParentRow('Application_Model_DbTable_Platform')->name;
```

Likewise, you can retrieve dependent rows using the `findDependentRowset()` method. For instance, the following snippet will retrieve the count of games associated with the Xbox 360 platform (identified by a primary key of 1):

```
$platformTable = new Application_Model_DbTable_Platform();

// Retrieve the platform row associated with the Xbox 360
$xbox360 = $platformTable->find(1)->current();

// Retrieve all games associated with platform ID 1
$games = $xbox360->findDependentRowset('Application_Model_DbTable_Game');

// Display the number of games associated with the Xbox 360 platform
echo count($games);
```

Alternatively, you can use a "magic method", made available to related models. For instance, dependent games can also be retrieved using the `findGame()` method:

```
$platformTable = new Application_Model_DbTable_Platform();

// Retrieve the platform row associated with the Xbox 360
$xbox360 = $platformTable->find(1)->current();

// Retrieve all games associated with platform ID 1
$games = $xbox360->findGame();

// Display the count
echo count($games);
```

The method is named `findGame()` because we're finding the platform's associated rows in the `Game` model. If the model happened to be named `Games`, we would use the method `findGames()`.

Finally, there's still another magic method at your disposal, in this case, `findGameByPlatform()`:

```
$platformTable = new Application_Model_DbTable_Platform();

// Retrieve the platform row associated with the Xbox 360
$xbox360 = $platformTable->find(1);

// Retrieve all games associated with platform ID 1
$games = $xbox360->findGameByPlatform();

// Display the count
echo count($games);
```

Note

The Zend_Db component can also automatically perform cascading operations if your database does not support referential integrity. This means you can configure your website model to automatically remove all games associated with the PlayStation 2 platform should

you decide to quit supporting this platform and delete it from the `platforms` table. See the Zend Framework documentation for more information about this feature.

Sorting a Dependent Rowset

When retrieving a a dependent result set (such as games associated with a particular platform), you'll often want to sort these results according to some criteria. To do so, you'll need to pass a query object into the `findDependentRowset()` method as demonstrated here:

```
$platformTable = new Application_Model_DbTable_Platform();
$gameTable = new Application_Model_DbTable_Game();
$games = $platformTable->findDependentRowset(
           'Application_Model_DbTable_Game', null,
           $gameTable->select()->order('name')
         );
```

JOINing Your Data

ORM solutions because they effectively abstract much of the gory SQL syntax that I've grown to despise over the years. But being able to avoid the syntax doesn't mean you should be altogether ignorant of it. In fact, ultimately you're going to need to understand some of SQL's finer points in order to maximize its capabilities. This is no more evident than when you need to retrieve related data residing within multiple tables, a technique known as *joining tables*.

Join Scenarios

If you're not familiar with the concept of a join, this section will serve to acquaint you with the topic by working through several common scenarios which appear within any data-driven website of moderate complexity.

Finding a User's Friends

The typical social networking website offers a means for examining a user's list of friends. There are many ways to manage a user's social connections, however one of the easiest involves simply using a table to associate each user's primary key with the friend's primary key. This table might look like this:

```
CREATE TABLE friends (
    id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    account_id INTEGER UNSIGNED NOT NULL,
    friend_id INTEGER UNSIGNED NOT NULL,
    created_on TIMESTAMP NOT NULL
```

```
);
```

Let's begin by examining the most basic type of join, known as the *inner join*. An inner join will return the desired rows whenever there is at least one match in both tables, the match being determined by a shared value such as an account's primary key. So for example, you might use a join to retrieve a list of a particular account's friends

```
mysql>SELECT a.username FROM accounts a
    ->INNER JOIN friends f ON f.friend_id = a.id WHERE f.account_id = 44;
```

This join requests the username of each account owner found in the `friends` table who is mapped to a friend of the account owner identified by 44.

Determine the Number of Copies of a Game Found in Your Network

Suppose you would like to borrow a particular game, but know your friend John had already loaned his copy to Carli. Chances are however somebody else in your network owns the game, but how can you know? Using a simple join, it's possible to determine the number of copies owned by friends, a feature integrated into GameNomad and shown in Figure 6.2.

Figure 6.2. Determining whether an account's friend owns a game

You might notice in Figure 6.2 this feature is actually used twice; once to determine the number of copies found in your network, and a second time to determine the number of copies found in your network which are identified as being available to borrow. To perform the former task, use this SQL join:

```
mysql>SELECT COUNT(gu.id) FROM games_to_accounts gu
    ->INNER JOIN friends f ON f.friend_id = gu.account_id
    ->WHERE f.account_id = 1 AND gu.game_id = 3;
```

As an exercise, try modifying this query to determine how many copies are available to borrow.

Determining Which Games Have Not Been Assigned a Platform

In an effort to increase the size of your site's gaming catalog, you've acquired another website which was dedicated to video game reviews. While the integration of this catalog has significantly bolstered the size of your database, the previous owner's lackadaisical data management practices left much to be desired, resulting in both incorrect and even missing platform assignments. To review a list

of all video games and their corresponding platform (even if the platform is NULL), you can use a join variant known as a left join.

While the inner join will only return rows from both tables when a match is found within each, a left join will return all rows in the leftmost table found in the query even if no matching record is found in the "right" table. Because we want to review a list of all video games and their corresponding platforms, even in cases where a platform hasn't been assigned, the left join serves as an ideal vehicle:

```
mysql>SELECT games.title, platforms.name FROM games
    ->LEFT JOIN platforms ON games.platform_id = platforms.id
    ->ORDER BY games.title LIMIT 10;
```

Executing this query produces results similar to the following:

```
+-----------------------------------+---------------+
| title                             | name          |
+-----------------------------------+---------------+
| Ace Combat 4:  Shattered Skies    | Playstation 2 |
| Ace Combat 5                      | Playstation 2 |
| Active Life Outdoor Challenge     | Nintendo Wii  |
| Advance Wars: Days of Ruin        | Nintendo DS   |
| American Girl Kit Mystery Challenge | Nintendo DS   |
| Amplitude                         | Playstation 2 |
| Animal Crossing: Wild World       | Nintendo DS   |
| Animal Genius                     | Nintendo DS   |
| Ant Bully                         | NULL          |
| Atelier Iris Eternal Mana         | Playstation 2 |
+-----------------------------------+---------------+
```

Note how the game "Ant Bully" has not been assigned a platform. Using an inner join, this row would not have appeared in the listing.

Counting Users by State

As your site grows in terms of registered users, chances are you'll want to create a few tools for analyzing statistical matters such as the geographical distribution of users according to state. To create a list tallying the number of registered users according to state, you can use a right join, which will list every record found in the right-side table, even if no users are found in that state. The following example demonstrates the join syntax used to perform this calculation:

```
mysql>SELECT COUNT(accounts.id), states.name
    ->FROM accounts RIGHT JOIN states ON accounts.state_id = states.id
    ->GROUP BY states.name;
```

Executing this query produces output similar to the following:

```
...
|   145 | New York             |
|    18 | North Carolina       |
|     0 | North Dakota         |
|    43 | Ohio                 |
|    22 | Oklahoma             |
|    15 | Oregon               |
|    77 | Pennsylvania         |
...
```

As even these relatively simple examples indicate, join syntax can be pretty confusing. The best advice I can give you is to spend an afternoon leisurely experimenting with the data, creating and executing joins which allow you to view the data in new and interesting ways.

Creating and Executing Zend_Db Joins

Now that you have a better understanding of how joins work, let's move on to how the Zend_Db makes it possible to integrate joins into your website. To demonstrate this feature, consider the following join query, which retrieves a list of a particular account's (identified by the primary key 3) friends:

```
mysql>SELECT a.id, a.username FROM accounts a
    ->JOIN friends ON friends.friend_id = a.id
    ->WHERE friends.account_id = 3;
```

Using Zend_Db's join syntax, you might rewrite this join and place it within a method named `getFriends()` found in the `Application_Model_DbTable_AccountRow` model:

```
01 function getFriends()
02 {
03     $accountTable = new Application_Model_DbTable_Account();
04     $query = $accountTable->select()->setIntegrityCheck(false);
05     $query->from(array('a' => 'accounts'), array('a.id', 'a.username'));
06     $query->join(array('f' => 'friends'), 'f.friend_id = a.id', array());
07     $query->where('f.account_id = ?', $this->id);
08
09     $results = $accountTable->fetchAll($query);
10     return $results;
11 }
```

Let's break this down:

- The `setIntegrityCheck()` method used in Line 04 defines the result set as read only, meaning any attempts to modify or delete the result set will cause an exception to be thrown. Although

most developers find this Zend Framework-imposed requirement confusing, it does come with the benefit of reminding you that any result set derived from a join is read-only.

- Line 05 identifies the left side of the join, in this case the `accounts` table. You'll also want to pass along an array containing the columns which are to be selected, otherwise all column will by default be selected.

- Line 06 identifies the joined table, and join condition. If you'd like to select specific columns from the joined table, pass those columns along in an array as was done in line 05; otherwise pass in an empty array to select no columns.

- Line 07 defines a `WHERE` clause, which will restrict the result set to a specific set of rows. In this case, we only want rows in which the `friends` table's `account_id` column is set to the value identified by `$this->id`.

You'll come to find the Zend_Db's join capabilities are particularly useful as your site grows in complexity. When coupled with Zend_Db's relationship features, it's possible to create impressively powerful data mining features with very little code.

Creating and Managing Views

You've seen how separating the three tiers (Model, View, and Controller) can make your life much easier. This particular chapter has so far focused on the Model as it relates to Zend_Db, along the way showing you how to create some fairly sophisticated SQL queries. However there's still further you can go in terms of separating the database from the application code.

Most relational databases offer a feature known as a *named view*, which you can think of as a simple way to refer to a complex query. This query might involve retrieving data from numerous tables, and may evolve over time, sometimes by the hand of an experienced database administrator. By moving the query into the database and providing the developer with a simple alias for referring to the query, the administrator can manage that query without having to necessarily also change any code found within the application. Even if you're a solo developer charged with both managing the code and the database, views are nonetheless a great way to separate these sorts of concerns.

Creating a View

Producing a list of the most popular games found in GameNomad according to their current sales rankings is a pretty commonplace task. Believe it or not, the query used to retrieve this data is fairly involved:

```
mysql>SELECT MAX(ranks.id) AS id, games.name AS name, games.asin AS asin,
    ->games.platform_id AS platform_id,
    ->ranks.rank AS rank
    ->FROM games
    ->JOIN ranks
    ->ON games.id = ranks.game_id
    ->GROUP BY ranks.game_id
    ->ORDER BY ranks.rank LIMIT 100;
```

Although by no means the most complex of queries, it's nonetheless a mouthful. Wouldn't it be much more straightforward if we can simply call this query using the following alias:

```
mysql>SELECT view_latest_sales_ranks;
```

Using MySQL's view feature, you can do exactly this! To create the view, login to MySQL using the mysql client or phpMyAdmin and execute the following command:

```
mysql>CREATE VIEW view_latest_sales_ranks AS
    ->SELECT MAX(ranks.id) AS id, games.name AS name, games.asin AS asin,
    ->games.platform_id AS platform_id,
    ->ranks.rank AS rank
    ->FROM games JOIN ranks
    ->ON games.id = ranks.game_id
    ->GROUP BY ranks.game_id
    ->ORDER BY ranks.rank LIMIT 100;
```

Tip

View creation statements are not automatically updated to reflect any structural or naming changes you make to the view's underlying tables and columns. Therefore if you make any changes to the tables or columns used by the view which reflect the view's SQL syntax, you'll need to modify the view accordingly. Modifying a view is demonstrated in the section "Reviewing View Creation Syntax".

Adding the View to the Zend Framework

The Zend Framework recognizes views as it would any other database table, meaning you can build a model around it!

```
<?php

  class Application_Model_DbTable_ViewLatestSalesRanks extends Zend_Db_Table_Abstract
  {
```

```
    protected $_name = 'latest_sales_ranks';
    protected $_primary = 'id';

    protected $_referenceMap = array (
      'Platform' => array (
          'columns'       => array('platform_id'),
          'refTableClass' => 'Application_Model_DbTable_Platform'
      )
    );

  }

?>
```

Deleting a View

Should you no longer require a view, consider removing it from the database for organizational reasons. To do so, use the `DROP VIEW` statement:

```
mysql>DROP VIEW latest_sales_ranks;
```

Reviewing View Creation Syntax

You'll often want to make modifications to a view over its lifetime. For instance, when I first created the `view_latest_sales_ranks` view, I neglected to limit the results to the top 100 games, resulting in a list of the top 369 games being generated. But recalling the view's lengthy SQL statement isn't easy, so how can you easily retrieve the current syntax for modification? The `SHOW CREATE VIEW` statement solves this dilemma nicely:

```
mysql>SHOW CREATE VIEW latest_sales_ranks\G
View: latest_sales_ranks
Create View: CREATE ALGORITHM=UNDEFINED DEFINER=`root`@`localhost`
SQL SECURITY DEFINER VIEW `latest_sales_ranks` AS
select max(`ranks`.`id`) AS `id`,`games`.`title` AS `title`,
`games`.`asin` AS `asin`,`games`.`platform_id` AS `platform_id`,
`ranks`.`rank` AS `rank` from (`games` join `ranks`
on((`games`.`id` = `ranks`.`game_id`)))
group by `ranks`.`game_id` order by `ranks`.`rank`
character_set_client: latin1
collation_connection: latin1_swedish_ci
```

We're particularly interested in the three lines beginning with `` `latest_sales_ranks` ``, as this signifies the start of the query. It looks different from the original SQL statement because MySQL

takes care to delimit all table and column names using backticks to account for special characters or reserved words. You can however reuse this syntax when modifying the query so copy those lines to your clipboard. Next, remove the query using `DROP VIEW`:

```
mysql>DROP VIEW latest_sales_ranks;
```

Now recreate the view using `CREATE VIEW`, pasting in the syntax but modifying the syntax by adding LIMIT 100 to the end of the query:

```
mysql>CREATE VIEW latest_sales_ranks `latest_sales_ranks`
AS select max(`ranks`.`id`) AS `id`,`games`.`title` AS `title`,
`games`.`asin` AS `asin`,`games`.`platform_id` AS `platform_id`,
`ranks`.`rank` AS `rank` from (`games` join `ranks`
on((`games`.`id` = `ranks`.`game_id`))) group by `ranks`.`game_id`
order by `ranks`.`rank` ASC LIMIT 100;
```

Paginating Results with Zend_Paginator

For reasons of performance and organization, you're going to want to spread returned database results across several pages if a lengthy number are returned. However, doing so manually can be a tedious chore, requiring you to track the number of results per page, the page number, and the query results current offset. Recognizing this importance of such a feature, the Zend Framework developers created the Zend_Paginator component, giving developers an easy way to paginate result sets without having to deal with all of the gory details otherwise involved in a custom implementation.

The Zend_Paginator component is quite adept, capable of paginating not only arrays, but also database results. It will also autonomously manage the number of results returned per page and the number of pages comprising the result set. In fact, Zend_Paginator will even create a formatted page navigator which you can insert at an appropriate location within the results page.

In this section I'll show you how to paginate a large set of video games across multiple pages.

Create the Pagination Query

Next you'll want to add the pagination feature to your website. I find the Zend_Paginator component appealing because it can be easily integrated into an existing query (which was presumably previously returning all results). All you need to do is instantiate a new instance of the Zend_Paginator class, passing the query to the object, and Zend_Paginator will do the rest. The following script demonstrates this feature:

```
01 function getGamesByPlatform($id, $page=1, $order="title")
02 {
```

```
03    $query = $this->select();
04    $query->where('platform_id = ?', $id);
05    $query->order($order);
06
07    $paginator = new Zend_Paginator(new Zend_Paginator_Adapter_DbTableSelect($query));
08    $paginator->setItemCountPerPage($paginationCount);
09    $paginator->setCurrentPageNumber($page);
10    return $paginator;
11 }
```

Let's break down this method:

- Lines 03-05 create the query whose results will be paginated. Because the method's purpose is to retrieve a set of video games identified according to a specific platform (Xbox 360 or Playstation 3 for instance), the query accepts a platform ID (`$id`) as a parameter. Further, should the developer wish to order the results according to a specific column, he can pass the column name along using the $order parameter.

- Line 07 creates the paginator object. When creating this object, you're going to pass along one of several available adapters. For instance, the `Zend_Paginator_Adapter_Array()` tells the Paginator we'll be paginating an array. In this example, we use `Zend_Paginator_Adapter_DbTableSelect()`, because we're paginating results which have been returned as instances of `Zend_Db_Table_Rowset_Abstract`. When using `Zend_Paginator_Adapter_DbTableSelect()`, you'll pass in the query.

- Line 08 determines the number of results which should be returned per page.

- Line 09 sets the current page number. This will of course adjust according to the page currently being viewed by the user. In a moment I'll show you how to detect the current page number.

- Line 10 returns the paginated result set, adjusted according to the number of results per page, and the offset according to our current page.

Using the Pagination Query

When using Zend_Paginator, each page of returned results will be displayed using the same controller and view. Zend_Paginator knows which page to return thanks to a page parameter which is passed along via the URL. For instance, the URL representing the first page of results would look like this:

```
http://gamenomad.com/games/platform/id/xbox360
```

The URL representing the fourth page of results would typically look like this:

```
http://gamenomad.com/games/platform/id/xbox360/page/4
```

Although I'll formally introduce this matter of retrieving URL parameters in the next chapter, there's nothing wrong with letting the cat out of the bag now, so to speak. The Zend Framework looks at URLs using the following pattern:

```
http://www.example.com/:controller/:action/:key/:value/:key/:value/.../:key/value
```

This means following the controller and action you can attach parameter keys and corresponding values, subsequently retrieving these values according to their key names within the action. So for instance, in the previous URL the keys are id and page, and their corresponding values are `xbox360` and `4`, respectively. You can retrieve these values within your controller action using the following commands:

```
$platform = $this->_request->getParam('id');
$page = $this->_request->getParam('page');
```

What's more, using a feature known as custom routes, you can tell the framework to recognize URL parameters merely according to their location, thereby negating the need to even preface each value with a key name. For instance, if you head over to GameNomad you'll see the platform-specific game listings actually use URLs like this:

```
http://gamenomad.com/games/platform/xbox360/4
```

Although not required knowledge to make the most of the Zend Framework, creating custom routes is extremely easy to do, and once you figure them out you'll wonder how you ever got along without them. Head over to http://framework.zend.com/manual/en/zend.controller.router.html to learn more about them.

With the platform and page number identified, all that's left to do is call the `Game` model's `getGamesByPlatform()` method to paginate the results:

```
$game = new Default_Game_Model();
$this->view->platformGames = $game->getGamesByPlatform($platform, $page);
```

Within the view, you can iterate over the `$this->platformGames` just as you would anywhere else:

```
<?php if (count($this->platformGames) > 0) { ?>
  <?php foreach ($this->platformGames AS $game) { ?>
    echo "{$game->name} <br />";
  <?php } ?>
<?php } ?>
```

Adding the Pagination Links

The user will need an easy and intuitive way to navigate from one page of results to the next. This list of linked page numbers is typically placed at the bottom of each page of output. The Zend_Paginator component can take care of the list generation for you, all you need to do is pass in the returned result set (in this case, `$this->platformGames`), the type of pagination control you'd like to use (in this case, Sliding), and the view helper used to stylize the page numbers:

```
<?= $this->paginationControl($this->platformGames,
  'Sliding', 'my_pagination.phtml'); ?>
```

The Sliding control will keep the current page number in the middle of page range. Several other control types exist, including `All`, `Elastic`, and `Jumping`. Try experimenting with each to determine which one you prefer. The view helper works like any other, although several special properties are made available to it, including the total number of pages contained within the results (`$this->pageCount`), the next page number (`$this->next`), the previous page (`$this->previous`), and several others. Personally I prefer to use one which is almost identical to that found in the Zend Framework documentation, which I'll reproduce here:

```
<?php if ($this->pageCount): ?>
<div class="paginationControl">

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

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

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

```
</div>
<?php endif; ?>
```

Of course, to take full advantage of the stylization opportunities presented by a pagination control such as this, you'll need to define CSS elements for the `paginationControl` and `disabled` classes.

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Define object-relational mapping (ORM) and talk about why its an advantageous approach to programmatically interacting with a database.
- Given a model named `Application_Model_DbTable_Game`, what will Zend_Db assume to be the name of the associated database table? How can you override this default assumption?
- What are the names and purposes of the native Zend_Db methods used to navigate model associations?

Chapter 7. Chapter 7. Integrating Doctrine 2

The Zend_Db component (introduced in Chapter 6) does a pretty good job of abstracting away many of the tedious SQL operations which tend to clutter up a typical PHP-driven website. Implementing two powerful data access patterns, namely Table Data Gateway and Row Data Gateway, Zend_Db users have the luxury of interacting with the database using a convenient object-oriented interface.

And if that summed up the challenges when integrating a database into an application, we'd be sitting pretty. But the Zend_Db component isn't so much a definitive solution as it is a starting point, and the Zend Framework documentation is quite clear on this matter, even going so far as to provide a tutorial which explains how to create Data Mappers which transfer data between the domain objects and relational database. While there's no doubt Zend_Db provides a solid starting foundation, I wonder how many users have the patience to implement a complete data management solution capable of meeting their application's complex domain requirements. I sure don't.

Always preferring the path of least resistance, I've been closely monitoring efforts to integrate Doctrine (http://www.doctrine-project.org/) into the Zend Framework. Although integrating Doctrine 1.X was a fairly painful process, it has become much less so with the Doctrine 2 release. Apparently the Zend Framework developers agree that Doctrine is a preferred data persistence solution, as Zend Framework 2 is slated to include support for Doctrine 2. In the meantime, no official documentation exists for Doctrine 2 integration, therefore rather than guide you through a lengthy and time-consuming configuration process which is certain to change I've instead opted to introduce you to Doctrine 2 using a Doctrine 2-enabled Zend Framework project which is included in the book's code download. This project is found in the directory `z2d2`. You'll need to update the `application.ini` file to define your database connection parameters and a few related paths but otherwise you should be able to begin experimenting with the integration simply by associating the project with a virtual host as you would any other Zend Framework-driven website. If you can't bear to go without knowing exactly every gory integration-related detail, see the project's README file. I'll use this project's code as the basis for introducing key Doctrine 2 features, highlighting those which I've grown to find particularly indispensable.

Caution

Doctrine 2 requires PHP 5.3, meaning you won't be able to use it until you've upgraded to at least PHP 5.3.0. PHP 5.3 supports several compelling new features such as namespaces,

so if you haven't already upgraded I recommend doing so even if you wind up not using Doctrine 2.

Introducing Doctrine

The Doctrine website defines the project as an "object-relational mapper for PHP which sits on top of a powerful database abstraction layer". This strikes me as a rather modest description, as Doctrine's programmatic interface is nothing short of incredible, supporting the ability to almost transparently marry your domain models with Doctrine's data mappers, as demonstrated in this example which adds a new record to the database:

```
$em = $this->_helper->EntityManager();

$account = new \Entities\Account;

$account->setUsername('wjgilmore');
$account->setEmail('wj@wjgilmore.com');
$account->setPassword('jason');
$account->setZip('43201');
$em->persist($account);
$em->flush();
```

Doctrine can also traverse and manipulate even the most complex schema relations using remarkably little code. Consider this example, which adds the game "Super Mario Brothers" to a user's video game library:

```
$em = $this->_helper->EntityManager();

$account = $em->getRepository('Entities\Account')
              ->findOneByUsername('wjgilmore');

$game = $em->getRepository('Entities\Game')
           ->findOneByName('Super Mario Brothers');

$account->getGames()->add($game);

$em->persist($account);
$em->flush();
```

Later in this chapter I'll provide several examples demonstrating its relationship mapping capabilities.

Incidentally, the `findOneByUsername()` method used in the above example is another great Doctrine feature, known as a *magic finder*. Doctrine dynamically makes similar methods available for all of

your table columns. For instance, if a table includes a `publication_date` column, a finder method named `findByPublicationDate()` will automatically be made available to you!

Iterating over an account's game library is incredibly easy. Just iterate over the results returned by `$account->getGames()` method like any other object array:

```
foreach ($user->getGames() as $game)
{
  echo "{$game->title}<br />";
}
```

Doctrine's capabilities extend far beyond its programmatic interface. You can use it's CLI (command-line interface) to generate and update schemas, and can use YAML, XML or (my favorite) DocBlock annotations to define column names, data types, and even table associations. I'll talk about these powerful features in the section "Building Persistent Classes".

Note

Doctrine 2 is a hefty bit of software, so although this chapter provides you with enough information to get you started, it doesn't even scratch the surface in terms of Doctrine's capabilities. My primary goal is to provide you with enough information to get really excited about the prospects of using Doctrine within your applications. Of course, I also recommend reviewing the GameNomad code, as Doctrine 2 is used throughout.

Introducing the z2d2 Project

Figuring out how to integrate Doctrine 2 into the Zend Framework was a pretty annoying and time-consuming process, one which involved perusing countless blog posts, browsing GitHub code, and combing over the Doctrine and Zend Framework documentation. The end result is however a successful implementation, one which I've subsequently successfully integrated into the example GameNomad website. However, because the GameNomad website is fairly complicated I've opted to stray from the GameNomad theme and instead focus on a project which is much smaller in scope yet nonetheless manages to incorporate several crucial Doctrine 2 features. I've dubbed this project `z2d2`, and it's available as part of your code download, located in the `z2d2` directory.

The project incorporates fundamental Doctrine features, including DocBlock annotations, use of the Doctrine CLI, magic finders, basic CRUD features, and relationship management. I'll use the code found in this project as the basis for instruction throughout the remainder of this chapter, so if you haven't already downloaded the companion code, go ahead and do so now.

Key Configuration Files and Parameters

As I mentioned at the beginning of this chapter, the Doctrine integration process is a fairly lengthy process and one which will certainly change with the eventual Zend Framework 2 release. However so as not to entirely leave you in the dark I'd like to at least provide an overview of the sample project's files and configuration settings which you'll need to understand in order to integrate Doctrine 2 into your own Zend Framework projects:

- The `application/configs/application.ini` file contains nine configuration parameters which Doctrine uses to connect to the database and determine where the class entities and proxies are located.
- The `library/Doctrine` directory contains three directories: `Common`, `DBAL`, and `ORM`. These three directories contain the object relational mapper, database abstraction layer, and other code responsible for Doctrine's operation.
- The `library/WJG/Resource/Entitymanager.php` file contains the resource plugin which defines the entity manager used by Doctrine to interact with the database.
- The `application/Bootstrap.php` file contains a method named `_initDoctrine()` which is responsible for making the class entities and repositories available to the Zend Framework application.
- The `library/WJG/Controller/Action/Helper/EntityManager.php` file is an action helper which is referenced within the controllers instead of the lengthy call which would otherwise have to be made in order to retrieve a reference to entity manager resource plugin.
- The `application/scripts/doctrine.php` file initializes the Doctrine CLI, and bootstraps the Zend Framework application resources, including the entity manager resource plugin. The CLI is run by executing the `doctrine` script, also found in `application/scripts`.
- The `application/models/Entities` directory contains the class entities. I'll talk more about the purpose of entities in a later section.
- The `application/models/Repositories` directory contains the class repositories. I'll talk more about the role of repositories in a later section.
- The `application/models/Proxies` directory contains the proxy objects. Doctrine generates proxy classes by default, however the documentation strongly encourages you to disable autogeneration, which you can do in the `application/config.ini` file.

Building Persistent Classes

In my opinion Doctrine's most compelling feature is its ability to make PHP classes persistent simply by adding *DocBlock annotations* to the class, meaning that merely adding those annotations will empower Doctrine to associate CRUD features with the class. An added bonus of these annotations is the ability to generate and maintain table schemas based on the annotation declarations.

These annotations are added to your model in a very unobtrusive way, placed within PHP comments spread throughout the class file. The below listing presents a simplified version of the `Account` entity found in `application/models/Entities/Account.php`, complete with the annotations. An explanation of key lines follows the listing.

```
01 <?php
02
03 namespace Entities;
04
05 /**
06  * @Entity @Table(name="games")
07  */
08 class Game
09 {
10   /**
11    * @Id @Column(type="integer")
12    * @GeneratedValue(strategy="AUTO")
13    */
14   private $id;
15
16   /** @Column(type="string", length=255) */
17   private $name;
18
19   /** @Column(type="string", length=255) */
20   private $publisher;
21
22   /** @Column(type="decimal",scale=2, precision=5) */
23   private $price;
24
25   public function getId()
26   {
27     return $this->id;
28   }
29
30   public function getName()
31   {
32     return $this->name;
33   }
34
```

```
35   public function setName($name)
36   {
37     $this->name = $name;
38   }
39
40  ...
41
42  public function setPassword($password)
43  {
44    $this->password = md5($password);
45  }
46
47   ...
48
49   public function getPrice()
50   {
51     return $this->price;
52   }
53
54   public function setPrice($price)
55   {
56     $this->price = $price;
57   }
58
59 }
```

Let's review the code:

- Line 03 declares this class to be part of the namespace `Entities`. Doctrine refers to persistable classes as `entities`, which are defined as objects with identity. Therefore for organizational purposes I've placed these persistable classes in a the directory `application/models/Entities`, and use PHP 5.3's namespacing feature within the controllers to reference the class, which is much more convenient than using the underscore-based approach embraced by the Zend Framework (which is unavoidable since namespaces are a PHP 5.3-specific feature). Therefore while it's not a requirement in terms of making a class persistable, I nonetheless suggest doing it for organizational purposes.
- Line 06 declares the class to be an entity (done using the `@Entity` annotation). Doctrine will by default map the class to a database table of the same name as the class, however if you prefer to use a different name then you can override the default using the `@Table` annotation.
- Lines 11-14 define an automatically incrementing integer-based primary key named `id`.
- Line 16 defines a column named `name` using type `varchar` of length 255. Doctrine will by default define this column as `NOT NULL`.

- Line 22 defines a column named `price` using type `decimal` of scale `2` and precision `5`.
- Lines 25-57 define the getters and setters (accessors and mutators) used to interact with this object. You are free to modify these methods however necessary. For instance, check out the project's `Account` model, which encrypts the supplied password using PHP's `md5()` function.

Note

DocBlock annotations are only one of several supported solutions for building database schemas. Other schema definition options are available, including using YAML- and XML-based formats. See the Doctrine 2 documentation for more details.

Generating and Updating the Schema

With the entity defined, you can generate the associated table schema using the following command:

```
$ cd application
$ ./scripts/doctrine orm:schema-tool:create
ATTENTION: This operation should not be executed in an production enviroment.

Creating database schema...
Database schema created successfully!
```

If you make changes to the entity, you can update the schema using the following command:

$ ./scripts/doctrine orm:schema-tool:update --force

It goes without saying that this feature is intended for use during the development process, and should not be using this command in a production environment. Alternatively, you can pass this command the `--dump-sql` to obtain a list of SQL statements which can subsequently be executed on the production server. Or better, consider using a schema management solution such as Liquibase (http://www.liquibase.org).

Finally, you can drop all tables using the following command:

```
$ ./scripts/doctrine orm:schema-tool:drop --force
Dropping database schema...
Database schema dropped successfully!
```

With your entities defined and schemas generated, move on to the next section where you'll learn how to query and manipulate the database tables via the entities.

Querying and Manipulating Your Data

One of Doctrine's most compelling features is its ability to map table schemas to an object-oriented interface. Not only can you use the interface to conveniently carry out the usual CRUD (create, retrieve, update, delete) tasks, but Doctrine will also make your life even easier by providing a number of so-called "magic finders" which allow you to explicitly identify the argument you're searching for as part of the method name. In this section I'll show you how to use Doctrine to retrieve and manipulate data.

Inserting, Updating, and Deleting Records

Whether its creating user accounts, updating blog entries, or deleting comment spam, you're guaranteed to spend a great deal of time developing features which insert, modify, and delete database records. In this section I'll show you how to use Doctrine's native capabilities to perform all three tasks.

Inserting Records

Unless you've already gone ahead and manually inserted records into the tables created in the previous section, the `z2d2` database is currently empty, so let's begin by adding a new record:

```
01 $em = $this->_helper->EntityManager();
02
03 $account = new \Entities\Account;
04
05 $account->setUsername('wjgilmore');
06 $account->setEmail('example@wjgilmore.com');
07 $account->setPassword('jason');
08 $account->setZip('43201');
09 $em->persist($account);
10 $em->flush();
```

Let's review this example:

- Line 01 retrieves an instance of the Doctrine entity manager. The Doctrine documentation defines the entity manager "as the central access point to ORM functionality", and it will play a central role in all of your Doctrine-related operations.

- Line 03 creates a new instance of the `Account` entity, using the namespacing syntax made available with PHP 5.3.

- Lines 05-08 set the object's fields. The beauty of this approach is that we have encapsulated domain-specific behaviors within the class, such as hashing the password using PHP's `md5()` function. See the `Account` entity file to understand how this is accomplished.

- Line 09 uses the entity manager's `persist()` method that you intend to make this data persistent. Note that this does *not* write the changes to the database! This is the job of the `flush()` method found on line 10. The `flush()` method will write all changes which have been identified by the `persist()` method back to the database.

Note

It's possible to fully decouple the entity manager from the application controllers by creating a *service layer*, however I've concluded that for the purposes of this exercise it would perhaps be overkill as it would likely only serve to confuse those readers who are being introduced to this topic for the first time. In the coming weeks I'll release a second version of `z2d2` which implements a service layer, should you want to know more about how such a feature might be accomplished.

Modifying Records

Modifying a record couldn't be easier; just retrieve it from the database, use the entity setters to change the attributes, and then save the record using the `persist()` / `flush()` methods demonstrated in the previous example. I'm getting ahead of myself due to necessarily needing to retrieve a record in order to modify it, however the method name used to retrieve the record is quite self-explanatory:

```
$accounts = $em->getRepository('Entities\Account')
              ->findOneByUsername('wjgilmore');
$account->setZip('20171');
$em->persist($account);
$em->flush();
```

Deleting Records

To delete a record, you'll pass the entity object to the entity manager's `remove()` method:

```
$accounts = $em->getRepository('Entities\Account')
              ->findOneByUsername('wjgilmore');
$em->remove($account);
$em->flush();
```

Finding Records

Let's start with Doctrine's most basic finder functionality, beginning by finding a game according to its primary key. You'll query entities via their *repository*, which is the mediator situated between the domain model and data mapping layer. Doctrine provides this repository functionality for you, although as you'll learn later in this chapter it's possible to create your own entity repositories which allow you to better manage custom queries related to the entity. For now though let's just use the default repository, passing it the entity we'd like to query. We can use method chaining to conveniently call the default repository's `find()` method, as demonstrated here:

```
01 $em = $this->_helper->EntityManager();
02
03 $account = $em->getRepository('Entities\Account')->find(1);
04
05 echo $account->getUsername();
```

With the record retrieved, you're free to use the accessor methods defined in the entity, as line 05 demonstrates.

To retrieve a record which matches a specific criteria, such as one which has its `username` set to `wjgilmore`, you can pass the column name and value into the `findOneBy()` method via an array, as demonstrated here:

```
$accounts = $em->getRepository('Entities\Account')
              ->findOneBy(array('username' => 'wjgilmore'));
```

Magic Finders

I find the syntax used in the previous example to be rather tedious, and so prefer to use the many magic finders Doctrine makes available to you. For instance, you can use the following magic finder to retrieve the very same record as that found using the above example:

```
$account = $em->getRepository('Entities\Account')
              ->findOneByUsername('wjgilmore');
```

Magic finders are available for retrieving records based on all of the columns defined in your table. For instance, you can use the `findByZip()` method to find all accounts associated with the zip code `43201`:

```
$accounts = $em->getRepository('Entities\Account')
              ->findByZip('43201');
```

Because results are returned as arrays of objects, you can easily iterate over the results:

```
foreach ($accounts AS $account)
{
  echo "{$account->getUsername()}<br />";
}
```

As you'll learn in the later section "Defining Repositories", it's even possible to create your own so-called "magic finders" by associating custom repositories with your entities. In fact, it's almost a certainty that you'll want to do so, because while the default magic finders are indeed useful for certain situations, you'll find that they tend to fall short when you want to search on multiple conditions or order results.

Retrieving All Rows

To retrieve all of the rows in a table, you'll use the `findAll()` method:

```
$accounts = $em->getRepository('Entities\Account')->findAll();

foreach ($accounts AS $account)
{
  echo "{$account->getUsername()}<br />";
}
```

Introducing DQL

Very often you'll want to query your models in ways far more exotic than what has been illustrated so far in this section. Fortunately, Doctrine provides a powerful query syntax known as the *Doctrine Query Language*, or DQL, which you can use construct queries capable of parsing every imaginable aspect of your object model. While it's possible to manually write queries, Doctrine also provides an API called QueryBuilder which can greatly improve the readability of even the most complex queries. For instance, the following example queries the `Account` model for all accounts associated with the zip code `20171`, and ordering those results according to the `username` column:

```
$qb = $em->createQueryBuilder();
$qb->add('select', 'a')
   ->add('from', 'Entities\Account a')
   ->add('where', 'a.zip = :zip')
   ->add('orderBy', 'a.username ASC')
   ->setParameter('zip', '20171');

$query = $qb->getQuery();

$accounts = $query->getResult();
```

```
foreach ($accounts AS $account)
{
  echo "{$account->getUsername()}<br />";
}
```

It's even possible to execute native queries and map those results to objects using a new Doctrine 2 feature known as *Native Queries*. See the Doctrine manual for more information.

Logically you're not going to want to embed DQL into your controllers, however the domain model isn't an ideal location either. The proper location is within methods defined within custom entity repositories. I'll show you how this is done in the later section "Defining Repositories".

Managing Entity Associations

All of the examples provided thus far are useful for becoming familiar with Doctrine syntax, however even a relatively simple real-world application is going to require significantly more involved queries. In many cases the queries will be more involved because the application will involve multiple domain models which are interrelated.

Unless you're a particularly experienced SQL wrangler, you're probably well aware of just how difficult it can be to both build and mine these associations. For instance just the three tables (`accounts`, `games`, `accounts_games`) which you generated earlier in this chapter pose some significant challenges in the sense that you'll need to create queries which can determine which games are associated with a particular account, and also which accounts are associated with a particular game. You'll also need to create features for associating and disassociating games with accounts. If you're new to managing these sorts of associations, it can be very easy to devise incredibly awkward solutions to manage these sort of relations.

Doctrine makes managing even complex associations laughably easy, allowing you to for instance retrieve the games associated with a particular account using intuitive object-oriented syntax:

```
$account = $em->getRepository('Entities\Account')
              ->findOneByUsername('wjgilmore');

$games = $account->getGames();

printf("%s owns the following games:<br />", $account->getUsername());

foreach ($games AS $game)
{
  printf("%s<br />", $game->getName());
}
```

Adding games to an account's library is similarly easy. Just associate the game with the account by passing the game object into the account's `add()` method, as demonstrated here:

```
$em = $this->_helper->EntityManager();

$account = $em->getRepository('Entities\Account')
              ->findOneByUsername('wjgilmore');

$game = $em->getRepository('Entities\Game')
           ->findOneByName('Super Mario Brothers');

$account->getGames()->add($game);

$em->persist($account);
$em->flush();
```

To remove a game from an account's library, use the `removeElement()` method:

```
$account = $em->getRepository('Entities\Account')
              ->findOneByUsername('wjgilmore');

$game = $em->getRepository('Entities\Game')->find(1);

$account->getGames()->removeElement($game);

$em->persist($account);
$em->flush();
```

Configuring Associations

In order to take advantage of these fantastic features you'll need to define the nature of the associations within your entities. Doctrine supports a variety of associations, including one-to-one, one-to-many, many-to-one, and many-to-many. In this section I'll show you how to configure a many-to-many association, which is also referred to as a has-and-belongs-to-many relationship. For instance, the book's theme project is based in large part around providing registered users with the ability to build video game libraries. Therefore, an account can have many games, and a game can be owned by multiple users. This relationship would be represented like so:

```
CREATE TABLE accounts (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) NOT NULL,
  email VARCHAR(255) NOT NULL
);

CREATE TABLE games (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
```

```
  name VARCHAR(255) NOT NULL,
  publisher VARCHAR(255) NOT NULL
);

CREATE TABLE accounts_games (
  account_id INTEGER UNSIGNED NOT NULL,
  game_id INTEGER UNSIGNED NOT NULL,
);

ALTER TABLE accounts_games ADD FOREIGN KEY (account_id)
  REFERENCES accounts(id);

ALTER TABLE accounts_games ADD FOREIGN KEY (game_id)
  REFERENCES games(id);
```

Because the idea is to associate a *collection* of games with an account, you'll need to use Doctrine's `Doctrine/Common/Collections/Collection` interface. Incidentally this section is referring to the same code found in the z2d2 project `Account` entity so I suggest opening that file and follow along. We'll want to use the `ArrayCollection` class, so reference it at the top of your entity like this:

```
use Doctrine\Common\Collections\ArrayCollection;
```

Next you'll need to define the class attribute which will contain the collection, and with it the nature of the relationship it has with another entity. This is easily the most difficult step, however the Doctrine manual provides quite a few examples and if you rigorously model your code after that accompanying these examples then you'll be fine. For instance, we want the `Account` entity to manage a collection of games, and so the `ManyToMany` annotation will look like this:

```
/**
 * @ManyToMany(targetEntity="Game", inversedBy="accounts")
 * @JoinTable(name="accounts_games",
 *   joinColumns={@JoinColumn(name="account_id",
 *                referencedColumnName="id")},
 *   inverseJoinColumns={@JoinColumn(name="game_id",
 *                       referencedColumnName="id")}
 * )
 */

private $games;
```

With the relationship defined, you'll want to initialize the `$games` attribute, done within a class constructor:

```
public function __construct()
{
  $this->games = new ArrayCollection();
```

```
}
```

Finally, you'll want to define convenience methods for adding and retrieving games:

```
public function addGame(Game $game)
{
  $game->addAccount($this);
  $this->games[] = $game;
}


public function getGames()
{
  return $this->games;
}
```

As you can see in the `addGame()` method, we are updating *both* sides of the relationship. The `Game` object's `addAccount()` method does not come out of thin air however; you'll define that in the `Game` entity.

Defining the Game Entity Relationship

The `Game` entity's relationship with the `Account` entity must also be defined. Because we want to be able to treat a game's associated accounts as a collection, you'll again reference `ArrayCollection` class at the top of your entity just as you did with the `Account` entity:

```
use Doctrine\Common\Collections\ArrayCollection;
```

The inverse side of this relationship is much easier to define:

```
/**
 * @ManyToMany(targetEntity="Account", mappedBy="games")
 */
private $accounts;
```

Next, initialize the `$accounts` attribute in your constructor:

```
public function __construct()
{
  $this->accounts =  new ArrayCollection();
}
```

Finally, you'll define the `addAccount()` and `getAccounts()` methods

```
public function addAccount(Account $account)
{
```

```
  $this->accounts[] = $account;
}

public function getAccounts()
{
  return $this->accounts;
}
```

With the association definition in place, you can begin creating and retrieving associations using the very same code as that presented at the beginning of this section! Don't forget to regenerate the schema however, because in doing so Doctrine will automatically create the `accounts_games` table used to store the relations.

Defining Repositories

No doubt that Doctrine's default magic finders provide a great way to begin querying your database, however you'll quickly find that many of your queries require a level of sophistication which exceed the the magic finders' capabilities. DQL is the logical alternative, however embedding DQL into your controllers isn't desirable, nor is polluting your domain model with SQL-specific behaviors. Instead, you can create custom entity repositories where you can define your own custom magic finders!

To tell Doctrine you'd like to use a custom entity repository, modify the entity's `@Entity` annotation to identify the repository location and name, as demonstrated here:

```
/**
 * @Entity (repositoryClass="Repositories\Account")
 * @Table(name="accounts")
...
```

Next you can use the Doctrine CLI to generate the repositories:

```
$ ./scripts/doctrine orm:generate-repositories \
  /var/www/dev.wjgames.com/application/models
Processing repository "Repositories\Account"
Processing repository "Repositories\Game"

Repository classes generated to "/var/www/dev.wjgames.com/application/models"
```

With the repository created, you can set about creating your own finders. For instance, suppose you wanted to create a finder which retrieved a list of accounts created in the last 24 hours. The method might look like this:

```
public function findNewestAccounts() {
```

```
  $now = new \DateTime("now");
  $oneDayAgo = $now->sub(new \DateInterval('P1D'))
                   ->format('Y-m-d h:i:s');

  $qb = $this->_em->createQueryBuilder();

  $qb->select('a.username')
     ->from('Entities\Account', 'a')
     ->where('a.created >= :date')
     ->setParameter('date', $oneDayAgo);

  return $qb->getQuery()->getResult();
}
```

Once added to the `Account` repository, you'll be able to call this finder from within your controllers just like any other:

```
$em = $this->_helper->EntityManager();

$accounts = $em->getRepository('Entities\Account')
               ->findNewestAccounts();
```

Testing Your Work

Automated testing of your persistent classes is a great way to ensure that your website is able to access them and that you are able to properly query and manipulate the underlying database via the Doctrine API. In this section I'll demonstrate a few basic tests. Remember that you'll need to configure your testing environment before you can begin taking advantage of these tests. The configuration process is discussed in great detail in Chapter 11.

Testing Class Instantiation

Use the following test to ensure that your persistent classes can be properly instantiated:

```
public function testCanInstantiateAccount()
{
  $this->assertInstanceOf('\Entities\Account', new \Entities\Account);
}
```

Testing Record Addition and Retrieval

The following test will ensure that a new user can be added to the database via the `Account` entity and later retrieved using the `findOneByUsername()` magic finder.

```
public function testCanSaveAndRetrieveUser()
{

  $account = new \Entities\Account;
  $account->setUsername('wjgilmore-test');
  $account->setEmail('example@wjgilmore.com');
  $account->setPassword('jason');
  $account->setZip('43201');
  $this->em->persist($account);
  $this->em->flush();

  $account = $this->em->getRepository('Entities\Account')
                ->findOneByUsername('wjgilmore-test');

  $this->assertEquals('wjgilmore-test', $account->getUsername());


}
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Talk about the advantages Doctrine provides to developers.
- Talk about the different formats Doctrine supports for creating persistent objects.
- What are DocBlock annotations?
- What is DQL and why is it useful?
- What is QueryBuilder and why is it useful?
- Why is it a good idea to create a repository should your query requirements exceed the capabilities provided by Doctrine's magic finders?

Chapter 8. Managing User Accounts

GameNomad is perhaps most succinctly defined as a social network for video game enthusiasts. After all, lacking the ability to keep tabs on friends' libraries and learn more about local video games available for trade or sale in your area, there would be little reason. These sorts of features will depend upon the ability of a user to create and maintain an account profile. This account profile will describe that user as relevant to GameNomad's operation, including information such as his residential zip code, and will also serve as the foundation from which other key relationships to games and friends will be made.

In order to give the user the ability to create and maintain an account profile, you'll need to create a host of associated features, such as account registration, login, logout, and password recovery. Of course, user management isn't only central to social network-oriented websites; whether you're building a new e-commerce website or a corporate intranet, the success of your project will hinge upon the provision of these features. Thankfully, the Zend Framework offers a robust component called Zend_Auth which contributes greatly to your ability to create many of these features. In this chapter I'll introduce you to this component, showing you how to create features capable of carrying out all of these tasks.

Creating the Accounts Database Table

When creating a new model I always like to begin by designing and creating the underlying database table, because doing so formally defines much of the data which will be visible and managed from within the website. With the schema defined, it's a natural next step to create the associated model and the associated attributes and behaviors which will represent the table. So let's begin by creating the `accounts` table.

```
CREATE TABLE accounts (
  id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(255) UNIQUE NOT NULL,
  email VARCHAR(255) UNIQUE NOT NULL,
  password CHAR(32) NOT NULL,
  zip VARCHAR(10) NOT NULL,
  confirmed BOOLEAN NOT NULL DEFAULT FALSE,
  recovery CHAR(32) NULL DEFAULT '',
  created DATETIME NOT NULL,
  updated DATETIME NOT NULL
);
```

Let's review the purpose of each column:

- The `id` column is the table's primary key. Although we'll generally retrieve records using the account username or e-mail address, the `id` column nonetheless serves an important identifying role because this value will serve as a foreign key within other tables.
- The `username` column stores the account's unique username which identifies the account owner to his friends and other users.
- The `email` column stores the account's email address. The e-mail address is used to confirm a newly created account, for logging the user into the website, as the vehicle for recovering lost passwords, and for general GameNomad-related communication.
- The `password` column stores the account's password. This is defined as a 32 character `CHAR` because for security purposes the account password will be encrypted using the MD5 hashing algorithm. Any string encrypted using MD5 is always stored using 32 characters, and so we define the password column to specifically fit this parameter.
- The `zip` column stores the user's zip code. Having a general idea of the user's location is crucial to the GameNomad experience because it gives users the opportunity to learn more about games which are available for borrowing, trading, or sale in their area.
- The `confirmed` column is used to determine whether the account's e-mail address has been confirmed. Confirming the existence and accessibility of a newly created account's e-mail address is important because the e-mail address will serve as the vehicle for recovering lost passwords and for occasional GameNomad-related correspondence.
- The `recovery` column stores a random string which will form part of the *one-time URLs* used to confirm accounts and recover passwords. You'll learn more about the role of these one-time URLs later in the chapter.
- The `created` column stores the date and time marking the account's creation. This could serve as a useful data point for determining the trending frequency of account creation following a marketing campaign.
- The `updated` column stores the date and time marking the last time the user updated his account profile.

Once you've created the `accounts` table, take a moment to review the `Account` entity found in the GameNomad project source code. This entity is quite straightforward, insomuch that it doesn't contain any features which are so exotic that they warrant special mention here.

Creating New User Accounts

As you'll soon learn, the code used to manage the account login and logout process is so simple that it would seem logical to ease into this chapter by introducing these topics, however it's not practical to test those features without first having a few accounts at our disposal. So let's begin with the task of allowing visitors to create new GameNomad accounts. Begin by creating the `Account` controller, which will house all of the actions associated with account management:

```
%>zf create controller Account
```

Next create the `register` action which will house the account registration logic:

```
%>zf create action register Account
```

With the `Account` controller and `register` action created, you'll typically follow the same sequence of steps whenever an HTML form is being incorporated into a Zend Framework application. First you'll create the registration form model (`FormRegister.php` in the code download), and then create the view used to render the model (`_form_register.phtml` in the download), and finally write the logic used to process the `register` action. Because the process of creating and configuring form models and their corresponding views was covered in great detail in Chapter 4, I'm not going to rehash their implementation here and will instead refer you to the code download. Instead, let's focus on the third piece of this triumvirate: the `register` action. The `register` action as used in GameNomad is presented next, followed by some commentary.

```
01 public function registerAction()
02 {
03
04   // Instantiate the registration form model
05   $form = new Application_Model_FormRegister();
06
07   // Has the form been submitted?
08   if ($this->getRequest()->isPost()) {
09
10     // If the form data is valid, process it
11     if ($form->isValid($this->_request->getPost())) {
12
13       // Does account associated with username exist?
14       $account = $this->em->getRepository('Entities\Account')
15                       ->findOneByUsernameOrEmail(
16                           $form->getValue('username'),
17                           $form->getValue('email')
18                         );
19
20       if (! $account)
```

```
21      {
22
23        $account = new \Entities\Account;
24
25        // Assign the account attributes
26        $account->setUsername($form->getValue('username'));
27        $account->setEmail($form->getValue('email'));
28        $account->setPassword($form->getValue('password'));
29        $account->setZip($form->getValue('zip'));
30
31        $account->setConfirmed(0);
32
33        // Set the confirmation key
34        $account->setRecovery($this->_helper->generateID(32));
35
36        try {
37
38          // Save the account to the database
39          $this->em->persist($account);
40          $this->em->flush();
41
42          // Create a new mail object
43          $mail = new Zend_Mail();
44
45          // Set the e-mail from address, to address, and subject
46          $mail->setFrom(
47            Zend_Registry::get('config')->email->support
48          );
49          $mail->addTo(
50            $account->getEmail(), "{$account->getUsername()}"
51          );
52          $mail->setSubject('GameNomad.com: Confirm Your Account');
53
54          // Retrieve the e-mail message text
55          include "_email_confirm_email_address.phtml";
56
57          // Set the e-mail message text
58          $mail->setBodyText($email);
59
60          // Send the e-mail
61          $mail->send();
62
63          // Set the flash message
64          $this->_helper->flashMessenger->addMessage(
65            Zend_Registry::get('config')->messages->register->successful
66          );
67
68          // Redirect the user to the home page
```

```
69            $this->_helper->redirector('login', 'account');
70
71          } catch(Exception $e) {
72            $this->view->errors = array(
73              array("There was a problem creating your account.")
74            );
75          }
76
77        } else {
78
79          $this->view->errors = array(
80            array("The username or e-mail address already exists.")
81          );
82
83        }
84
85      } else {
86        $this->view->errors = $form->getErrors();
87      }
88
89    }
90
91    $this->view->form = $form;
92
93 }
```

Let's review this code:

- Line 05 instantiates the `FormRegister` model, which defines the form fields and validation procedures.
- Line 08 determines if the form has been submitted, and if so, Line 11 determines if the submitted form information passes the validation constraints defined in the `FormRegister` model. If the form data does not validate, The errors are passed to the `$form` view scope variable (Line 86) and the form displayed anew (Line 91).
- If the provided form data is validated, Line 23 instantiates a new `Account` entity object, and the object is populated with the form data (Lines 26-34). Note in particular how the password is set just like the other fields despite the previously discussed requirement that the password be encrypted within the database. This is accomplished by overriding the password mutator within the `Account` entity.
- Line 34 creates the user's unique recovery key by generating a random 32 character string. The `$this->_helper->generateID(32)` call is not native to the Zend Framework but rather is a

custom action helper which I've created as a convenience, since the need to generate unique strings arises several times throughout the GameNomad website. You can find this action helper in the `library/WJG/Controller/Action/Helper/` directory.

- Line 39-40 saves the object to the database.
- If the save is successful, lines 43-61 send an account confirmation e-mail to the user using the Zend Framework's Zend_Mail component. I'll talk more about this process in the section "Sending E-mail Through the Zend Framework".
- After sending the e-mail, a flash message is prepared (lines 64-66)and the user is redirected to the login page (line 68). Although you could certainly embed the notification message directly within the flash messenger helper's `addMessage()` method, I prefer to manage all of the messages together within the configuration file `application.ini`).

Sending E-mail Through the Zend Framework

Because the e-mail messages can be quite lengthy particularly if HTML formatting is used, I prefer to manage these messages within their own file. In the `register` action presented above, the confirmation e-mail is stored within a file named `_email_confirm_email_address.phtml`. Because you'll typically want to dynamically update this e-mail with information such as the user's name, not to mention need to pass this message to the `setBodyText()` method `setBodyHtml()` if you're sending an HTML-formatted message), I place the message within a variable named `$email`, using PHP's HEREDOC statement. For instance here is what `_email_confirm_email_address.phtml` looks like:

```
<?php

$email = <<< email
Dear {$account->getUsername()},

Your GameNomad account has been created! To complete registration,
click on the below link to confirm your e-mail address.

http://www.gamenomad.com/account/confirm/key/{$account->getRecovery()}

Once confirmed, you'll be able to access exclusive GameNomad features!

Thank you!
The GameNomad Team

Questions? Contact us at support@gamenomad.com
http://www.gamenomad.com/

email;
```

```
?>
```

For organizational purposes, I store these e-mail message files within `application/views/scripts`. However, chances are this particular directory doesn't reside on PHP's `include_path`, so you'll need to add it if you'd like to follow this convention. Rather than muddle up the configuration directive within `php.ini` I prefer to add it to the `set_include_path()` function call within the front controller `public/index.php`):

```
set_include_path(implode(PATH_SEPARATOR, array(
    realpath(APPLICATION_PATH . '/../library'),
    realpath(APPLICATION_PATH . '/../application/views/scripts'),
    get_include_path(),
)));
```

Configuring Zend_Mail to Use SMTP

Zend_Mail will by default rely upon the server's Sendmail daemon to send e-mail, which is installed and configured on most Unix-based systems by default. However, if you're running Windows or would otherwise like to use SMTP to send e-mail you'll need to configure Zend_Mail so it can authenticate and connect to the SMTP server.

Because you might send e-mail from any number of actions spread throughout the site, you'll want this configuration to be global. I do so by adding a method to the `Bootstrap.php` file which executes with each request. In a high-traffic environment you'll probably want to devise a more efficient strategy but for most developers this approach will work just fine. Within this method (which I call `_initEmail`) you'll pass the SMTP server's address, port, type of protocol used if the connection is secure, and information about the account used to send the e-mail, including the account username and password. I store all of this information within the `application.ini` file for easy maintenance. For instance, the following snippet demonstrates how you would define these parameters to send e-mail through a Gmail account:

```
email.server   = "smtp.gmail.com"
email.port     = 587
email.username = "example@gmail.com"
email.password = "secret"
email.protocol = "tls"
```

The `_initEmail()` method will retrieve these parameters, pass them to the `Zend_Mail_Transport_Smtp` constructor, along with the SMTP server address, and then pass the newly created `Zend_Mail_Transport_Smtp` object to Zend_Mail's `setDefaultTransport()` method. The entire `_initEmail()` method is presented here:

```
protected function _initEmail()
{

  $emailConfig = array(
    'auth'=> 'login',
    'username'  => Zend_Registry::get('config')->email->username,
    'password'  => Zend_Registry::get('config')->email->password,
    'ssl'       => Zend_Registry::get('config')->email->protocol,
    'port'      => Zend_Registry::get('config')->email->port
  );

 $mailTransport = new Zend_Mail_Transport_Smtp(
    Zend_Registry::get('config')->email->server, $emailConfig);

 Zend_Mail::setDefaultTransport($mailTransport);

}
```

With `_initEmail()` in place, you can go about sending e-mail anywhere within your application!

Confirming the Account

After the account has been successfully created, a confirmation e-mail will be generated and sent to the account's e-mail address. This e-mail contains a link known as a "one-time URL" which uniquely identifies the account by passing the value stored in the account record's `recovery` column. The URL is generated by inserting the account's randomly generated recovery key into the e-mail body stored within `_email_confirm_email_address.phtml`. The particular line within this file which creates the URL looks like this:

```
http://www.gamenomad.com/account/confirm/key/{$account->recovery}
```

When the user clicks this URL he will be transported to GameNomad's account confirmation page, hosted within the `Account` controller's `confirm` action. The action code is presented next, followed by a breakdown of relevant lines.

```
01 public function confirmAction()
02 {
03
04   $key = $this->_request->getParam('key');
05
06   // Key should not be blank
07   if ($key != "")
08   {
09
```

```
10      $em = $this->getInvokeArg('bootstrap')
11        ->getResource('entityManager');
12
13      $account = $em->getRepository('Entities\Account')
14        ->findOneByRecovery($this->_request->getParam('key'));
15
16      // Was the account found?
17      if ($account) {
18
19        // Account found, confirm and reset recovery attribute
20        $account->setConfirmed(1);
21        $account->setRecovery("");
22
23        // Save the account to the database
24        $em->persist($account);
25        $em->flush();
26
27        // Set the flash message and redirect the user to the login page
28        $this->_helper->flashMessenger->addMessage(
29          Zend_Registry::get('config')->messages
30            ->register->confirm->successful
31        );
32        $this->_helper->redirector('login', 'account');
33
34      } else {
35
36        // Set flash message and redirect user to the login page
37        $this->_helper->flashMessenger->addMessage(
38          Zend_Registry::get('config')->messages
39            ->register->confirm->failed
40        );
41        $this->_helper->redirector('login', 'account');
42
43      }
44
45    }
46
47 }
```

Let's review several relevant lines of the `confirm` action:

- Line 13-14 retrieves the account record associated with the recovery key passed via the URL.
- If the key is found (line 17), the account is confirmed, the recovery key is deleted, and the changes are saved to the database (lines 20-25)

- Once the updated account information has been saved back to the database, a message is assigned to the flash messenger and the user is redirected to the `Account` controller's `login` action (lines 28-32).

- If the recovery key is not found in the database, presumably because the user had previously confirmed his account and is for some reason trying to confirm it anew, an error message is assigned to the flash messenger and the user is redirected to the login page (lines 37-41).

Creating the User Login Feature

With the user's account created and confirmed, he can login to the site in order to begin taking advantage of GameNomad's special features. Like account registration, the account login feature is typically implemented using a form model (`application/models/FormLogin.php`), associated view script (`application/views/scripts/_form_login.html`), and a controller action which you'll find in the `Account` controller's `login` action. Just as was the case with the earlier section covering registration, I'll forego discussion of the form model and instead focus on the `login` action. As always you can review the form model and its associated parts by perusing the relevant files within the code download. The `login` action is presented next, followed by a review of relevant lines.

```
01 public function loginAction()
02 {
03
04   $form = new Application_Model_FormLogin();
05
06   // Has the login form been posted?
07   if ($this->getRequest()->isPost()) {
08
09     // If the submitted data is valid, attempt to authenticate the user
10     if ($form->isValid($this->_request->getPost())) {
11
12       // Did the user successfully login?
13       if ($this->_authenticate($this->_request->getPost())) {
14
15         $account = $this->em->getRepository('Entities\Account')
16                         ->findOneByEmail($form->getValue('email'));
17
18         // Save the account to the database
19         $this->em->persist($account);
20         $this->em->flush();
21
22         // Generate the flash message and redirect the user
23         $this->_helper->flashMessenger->addMessage(
24           Zend_Registry::get('config')->messages->login->successful
25         );
```

```
26
27          return $this->_helper->redirector('index', 'index');
28
29        } else {
30          $this->view->errors["form"] = array(
31            Zend_Registry::get('config')->messages->login->failed
32          );
33        }
34
35      } else {
36        $this->view->errors = $form->getErrors();
37      }
38
39    }
40
41    $this->view->form = $form;
42
43 }
```

Let's review the relevant lines of this snippet:

- Line 04 instantiates a new instance of the `FormLogin` model, which is passed to the view on line 34.
- If the form has been submitted back to the action (line 07), and the form data has properly validated (line 10), the action will next attempt to authenticate the user (line 13) by comparing the provided e-mail address and password with what's on record in the database. I like to maintain the authentication-specific code within its own protected method `_authenticate()`), which we'll review in just a moment.
- If authentication is successful, lines 15-16 will retrieve the account record using the provided e-mail address. Finally, a notification message is added to the flash messenger and the user is redirected to GameNomad's home page.
- If authentication fails, an error message is added to the global errors array (lines 30-32) and the login form is displayed anew.

As I mentioned, `_authenticate()` is a protected method which encapsulates the authentication-specific code and establishes a new user session if authentication is successful. You could just as easily embed this logic within your `login` action however I prefer my approach as it results in somewhat more succinct code. The `_authenticate()` method is presented next, followed by a review of relevant lines:

```
01 protected function _authenticate($data)
02 {
```

```
03
04  $db = Zend_Db_Table::getDefaultAdapter();
05  $authAdapter = new Zend_Auth_Adapter_DbTable($db);
06
07  $authAdapter->setTableName('accounts');
08  $authAdapter->setIdentityColumn('email');
09  $authAdapter->setCredentialColumn('password');
10  $authAdapter->setCredentialTreatment('MD5(?) and confirmed = 1');
11
12  $authAdapter->setIdentity($data['email']);
13  $authAdapter->setCredential($data['pswd']);
14
15  $auth = Zend_Auth::getInstance();
16  $result = $auth->authenticate($authAdapter);
17
18  if ($result->isValid())
19  {
20
21    if ($data['public'] == "1") {
22      Zend_Session::rememberMe(1209600);
23    } else {
24      Zend_Session::forgetMe();
25    }
26
27    return TRUE;
28
29  } else {
30
31    return FALSE;
32
33  }
34
35 }
```

Let's review the code:

- The Zend_Auth component authenticates an account by defining the account data source and within that source, the associated account credentials. Several sources are supported, including any database supported by the Zend Framework, LDAP, Open ID. Because GameNomad's account data is stored within the MySQL database's `accounts` table, the `_authenticate()` method uses Zend_Auth's `Zend_Auth_Adapter_DbTable()` method (line 05) to pass in the default database adapter handle (see Chapter 6 for more about this topic), and then uses the `setTableName()` method (line 07) to define the `accounts` table as the account data repository. Unfortunately at the time of this writing there is not a Doctrine-specific Zend_Auth adapter, and so you are forced to define two sets of database connection credentials within the `application.ini` file in order to

take advantage of Zend_Auth in this manner, however it is a small price to pay in return for the conveniences otherwise offered by this component.

- Lines 08 - 09 associate the `accounts` table's `email` and `password` columns as those used to establish an account's credentials. These are the two items of information a user is expected to pass along when prompted to login.
- Line 10 uses the `setCredentialTreatment()` method to determine *how* the password should be passed into the query. Because the password is encrypted within the `accounts` table using the MD5 algorithm, we need to make sure that the provided password is similarly encrypted in order to determine whether a match exists. Additionally, because the user must confirm his account before being allowed to login, we also check whether the table's `confirmed` column has been set to `1`.
- Lines 12 and 13 define the account identifier (the e-mail address) and credential (the password) used in the authentication attempt. These values are passed into the `_authenticate()` method, and originate as $_POST variables passed in via the login form.
- Line 15 instantiates a new instance of Zend_Auth, and passes in the authentication configuration data into the object using the `authenticate()` method.
- Line 18 determines whether the provided authentication identifier and credential exists as a pair within the database. If so, the user has successfully authenticated and we next determine whether the user has specified whether he would like to remain logged-in on his computer for two weeks, as determined by whether the check box on the login form was selected. If so, the cookie's expiration date will be set for two weeks from the present (the session cookie is used by Zend_Auth for all subsequent requests to determine whether the user is logged into the website). Otherwise, the cookie's expiration date will be set in such a way that the cookie will expire once the user closes the browser.
- Finally, a value of either TRUE or FALSE will be returned to the `login` action, indicating whether the authentication attempt was successful or has failed, respectively.

Determining Whether the User Session is Valid

After determining that the user has successfully authenticated, Zend_Auth will place a cookie on the user's computer which can subsequently be used to determine whether the user session is still valid. You can use Zend_Auth's `hasIdentity()` method to verify session validity. If valid, you can use the `getIdentity()` method to retrieve the account's identity (which in the case of GameNomad is the e-mail address).

```
$auth = Zend_Auth::getInstance();
```

```
if ($auth->hasIdentity()) {

  $identity = $auth->getIdentity();

  if (isset($identity)) {
    printf("Welcome back, %s", $identity);
  }

}
```

However, you're likely going to want to determine whether a valid account session exists at any given point within the website, meaning you'll need to execute the above code with every page request. My suggested solution is to insert this logic into a custom action helper and then call this action helper from within the application bootstrap (meaning the action helper will be called every time the application executes). Because this action helper can be used to initialize other useful global behaviors and other attributes, I've called it `Initializer.php` and for organizational purposes have placed it within `/library/WJG/Controller/Action/Helper/`. The authentication-relevant part of the `Initializer` action helper is presented next, followed by a discussion of the relevant lines.

```
01 $auth = Zend_Auth::getInstance();
02
03 if ($auth->hasIdentity()) {
04
05   $identity = $auth->getIdentity();
06
07   if (isset($identity)) {
08
09     $em = $this->getActionController()
10                ->getInvokeArg('bootstrap')
11                ->getResource('entityManager');
12
13     // Retrieve information about the logged-in user
14     $account = $em->getRepository('Entities\Account')
15                   ->findOneByEmail($identity);
16
17     Zend_Layout::getMvcInstance()->getView()->account = $account;
18
19   }
20
21 }
```

Let's review the code:

- Line 02 retrieves a static instance of the Zend_Auth object

- Line 03 determines whether the user is currently logged in. If so, line 05 retrieves the identity of the currently logged-in user as specified by his username.
- Line 09 retrieves the entity manager, which is needed on lines 14-15 in order to retrieve information about the logged-in user.
- Line 17 passes the retrieved account object into the application's view scope using a little-known feature of the Zend Framework which allows you to inject values into the view via the Zend_Layout component's `getView()` method.

With the `Initializer` custom action helper defined, you'll next need to add a method to the bootstrap `/application/Bootstrap.php`) which will result in `Initializer` being executed each time the application initializes. The following example method defines the custom action helper path using the Zend_Controller_Action_HelperBroker's `addPath()` method, and then executes the action using the Zend_Controller_Action_HelperBroker's `addHelper()` method:

```
protected function _initGlobalVars()
{

  Zend_Controller_Action_HelperBroker::addPath(
                 APPLICATION_PATH.'/../library/WJG/Controller/Action/Helper'
                );

  $initializer = Zend_Controller_Action_HelperBroker::addHelper(
                 new WJG_Controller_Action_Helper_Initializer()
                );

}
```

Because the account object is injected into the view scope, you can determine whether a valid session exists within both controllers and views by referencing the `$this->view->account` and `$this->account` variables, respectively. For instance, the following code might be used to determine whether a valid session exists. If so, a custom welcome message can be provided, otherwise registration and login links can be presented.

```
<?php if (! $this->account) { ?>
  <p>
    <a href="/account/login">Login to your account</a> |
    <a href="/account/register">Register</a>
  </p>
<?php } else { ?>
  <p>
    Welcome back, <a href="/account/"><?= $this->account->username; ?></a>
    &amp;middot;
```

```
    <a href="/account/logout">Logout</a>
  </p>
<?php } ?>
```

A GameNomad screenshot using similar functionality to determine session validity is presented in Figure 8.1.

Figure 8.1. Greeting an authenticated user

Creating the User Logout Feature

Particularly if the user is interacting with your website via a publicly accessible computer he will want to be confident that his session is terminated before walking away. Fortunately, logging the user out is easily accomplished using Zend_Auth's `clearIdentity()` method, as demonstrated here:

```
public function logoutAction()
{
 Zend_Auth::getInstance()->clearIdentity();
  $this->_helper->flashMessenger->addMessage('You are logged out of your account');
  $this->_helper->redirector('index', 'index');
}
```

Creating an Automated Password Recovery Feature

With everything else you need to accomplish on any given day, the last thing you'll want to deal with is responding to requests to reset an account password. Fortunately, creating an automated password recovery feature is quite easy. Like the account confirmation feature introduced earlier in this chapter, the password recovery feature will depend upon the use of a one-time URL sent via e-

mail which the user will click in order to confirm his identity. Once the user clicks this URL, the user's account will be updated with a new random password, and that random password will be e-mailed to the user. Once the user logs into the website, he can change the password as desired.

The user will initiate the password recovery process by presumably clicking on a link located somewhere within the login screen. In the case of GameNomad he'll be transported to `/account/lost`, and prompted to provide his e-mail address (see Figure 8.2). If the e-mail address is associated with a registered user, then a recovery key is generated and and a one-time URL is e-mailed to the user.

Figure 8.2. Recovering a lost password

The `lost` action used to generate and send the recovery key to the provided e-mail address is presented next. Frankly there's nothing in this action which you haven't already seen several times, so I'll forego the usual summary.

```
01 public function lostAction()
02 {
03
04   $form = new Application_Model_FormLost();
05
06   if ($this->getRequest()->isPost()) {
07
08     // If form is valid, make sure e-mail address is associated
09     // with an account
10     if ($form->isValid($this->_request->getPost())) {
11
12       $account = $this->em->getRepository('Entities\Account')
13                 ->findOneByEmail($form->getValue('email'));
14
```

```
15        // If account is found, generate recovery key and mail it to
16        // the user
17        if ($account)
18        {
19
20          // Generate a random password
21          $account->setRecovery($this->_helper->generateID(32));
22
23          $this->em->persist($account);
24          $this->em->flush();
25
26          // Create a new mail object
27          $mail = new Zend_Mail();
28
29          // Set the e-mail from address, to address, and subject
30          $mail->setFrom(Zend_Registry::get('config')->email->support);
31          $mail->addTo($form->getValue('email'));
32          $mail->setSubject("GameNomad: Generate a new password");
33
34          // Retrieve the e-mail message text
35          include "_email_lost_password.phtml";
36
37          // Set the e-mail message text
38          $mail->setBodyText($email);
39
40          // Send the e-mail
41          $mail->send();
42
43          $this->_helper->flashMessenger
44            ->addMessage('Check your e-mail for further instructions');
45          $this->_helper->redirector('login', 'account');
46
47        }
48
49      } else {
50        $this->view->errors = $form->getErrors();
51      }
52
53    }
54
55    $this->view->form = $form;
56
57  }
```

The e-mail message sent to the user is found within the file `_email_lost_password.phtml` (and included into the `lost` action on line 34). When sent to the user the e-mail looks similar to that found in Figure 8.3.

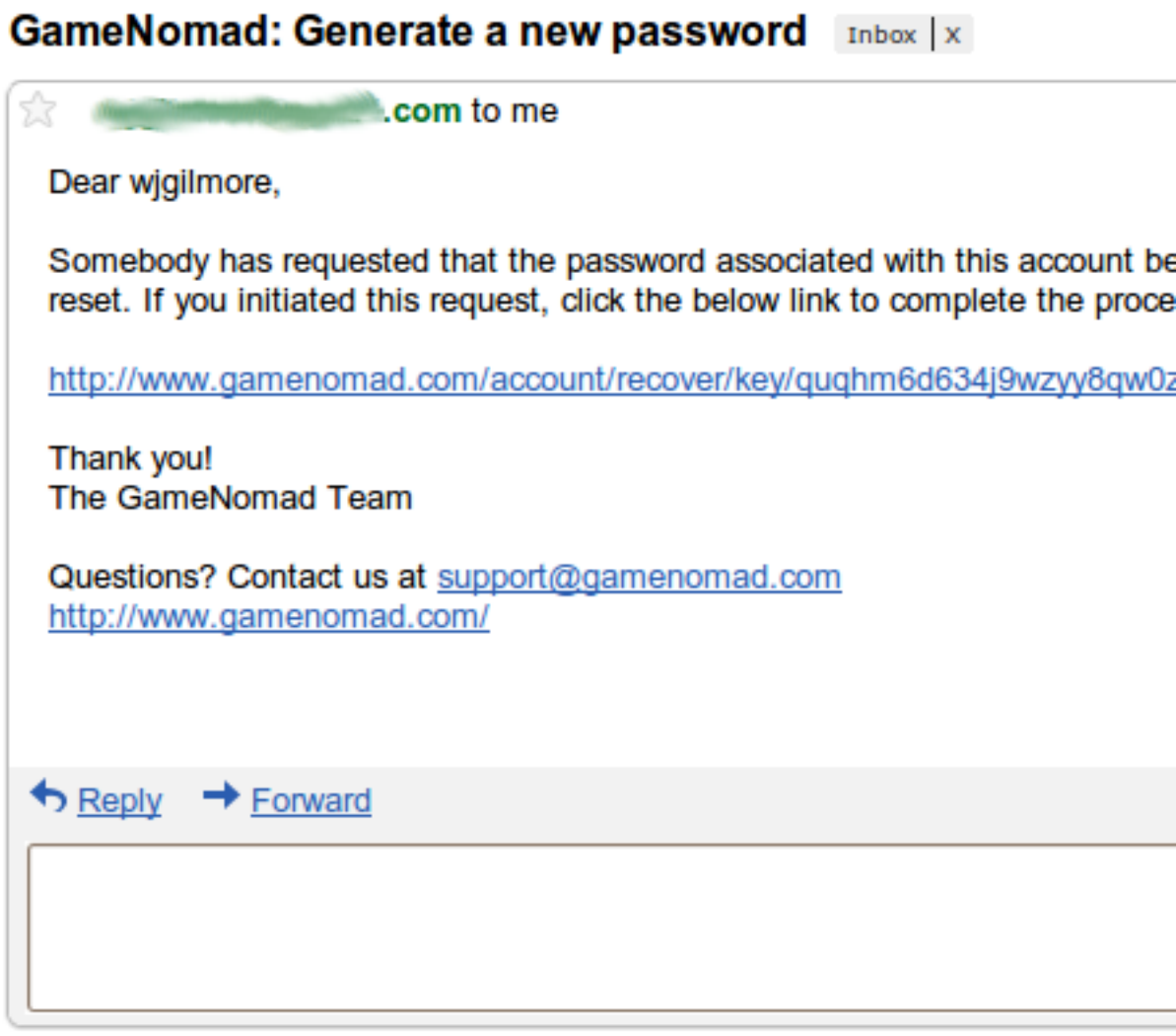


Figure 8.3. The password recovery e-mail

Once the user clicks on the one-time URL he is transported back to the GameNomad website, specifically to the `Account` controller's `recover` action. This action will retrieve the account

associated with the recovery key passed along as part of the one-time URL. If an account is found, a random eight-character password will be generated and sent to the e-mail address associated with the account. The `recover` action code is presented next. As was the case with the `lost` action, there's nothing new worth discussing in the `recover` action, so I'll just provide the code for your perusal:

```
01 public function recoverAction()
02 {
03
04   $key = $this->_request->getParam('key');
05
06   if ($key != "")
07   {
08
09     $account = $this->em->getRepository('Entities\Account')
10               ->findOneByRecovery($key);
11
12     // If account is found, generate recovery key and mail it to
13     // the user
14     if ($account)
15     {
16
17       // Generate a random password
18       $password = $this->_helper->generateID(8);
19       $account->setPassword($password);
20
21       // Erase the recovery key
22       $account->setRecovery("");
23
24       // Save the account
25       $this->em->persist($account);
26       $this->em->flush();
27
28       // Create a new mail object
29       $mail = new Zend_Mail();
30
31       // Set the e-mail from address, to address, and subject
32       $mail->setFrom(Zend_Registry::get('config')->email->support);
33       $mail->addTo($account->getEmail());
34       $mail->setSubject("GameNomad: Your password has been reset");
35
36       // Retrieve the e-mail message text
37       include "_email_recover_password.phtml";
38
39       // Set the e-mail message text
40       $mail->setBodyText($email);
41
42       // Send the e-mail
```

```
43        $mail->send();
44
45        $this->_helper->flashMessenger->addMessage(
46          Zend_Registry::get('config')->messages
47            ->account->password->reset
48        );
49        $this->_helper->redirector('login', 'account');
50
51      }
52
53    }
54
55    // Either a blank key or non-existent key was provided
56    $this->_helper->flashMessenger->addMessage(
57      Zend_Registry::get('config')
58        ->messages->account->password->nokey
59    );
60    $this->_helper->redirector('login', 'account');
61
62 }
```

Testing Your Work

While a user may forgive the occasionally misaligned graphic or other minor error, broken account management features are sure to be wildly frustrating and perhaps grounds for checking out a competing website. Therefore given the mission-critical importance of the features introduced in this chapter, you're going to want to put them through a rigorous testing procedure to make sure everything is working properly. In this section I'll guide you through several of the most important tests.

Making Sure the Login Form Exists

Because it's not possible for the user to login if the login form is inexplicably missing, consider running a simple sanity check to confirm the login form is indeed being rendered within the login view. You can use the `assertQueryCount()` method to confirm that a particular element and associated DIV ID exist within the rendered page, as demonstrated here:

```
public function testLoginActionContainsLoginForm()
{
  $this->dispatch('/account/login');
  $this->assertQueryCount('form#login', 1);
  $this->assertQueryCount('input[name~="email"]', 1);
  $this->assertQueryCount('input[name~="password"]', 1);
  $this->assertQueryCount('input[name~="submit"]', 1);
```

```
}
```

Testing the Login Process

Logically you'll want to make sure your login form is operating flawlessly, as there are few issues more frustrating to users than the inability to access their account due to no fault of their own. Thankfully it's really easy to determine whether the login form has successfully authenticated a user, because the action will *only* redirect the user to the home page if the credentials are deemed valid. The following test will POST a set of valid credentials to the `Account` controller's `login` action. Because they are valid, we will assert that the redirection has indeed occurred (using the `assertRedirectTo()` method).

```
public function testValidLoginRedirectsToHomePage()
{

  $this->request->setMethod('POST')
       ->setPost(array(
           'email' => 'wj@wjgilmore.com',
           'pswd'  => 'secret',
           'public' => 0
         ));

  $this->dispatch('/account/login');

  $this->assertController('account');
  $this->assertAction('login');

  $this->assertRedirectTo('/account/friends');

}
```

Because chances are you're going to want to test parts of the application which are only available to authenticated users, you can create a private method within your test controller which can be executed as desired within other tests, thereby consolidating the login-specific task. For instance, here's what my login-specific method looks like:

```
private function _loginValidUser()
{

  $this->request->setMethod('POST')
       ->setPost(array(
       'email'  => 'wj@wjgilmore.com',
       'pswd'   => 'secret',
       'public' => 0
  ));
```

```
  $this->dispatch('/account/login');

  $this->assertRedirectTo('/account/friends');
  $this->assertTrue(Zend_Auth::getInstance()->hasIdentity());

}
```

With this method in place, I can call it anywhere within the test suite as needed, as demonstrated in the next test.

Ensuring an Authenticated User Can Access a Restricted Page

Pages such as the logout page should only be accessible to authenticated users. Because such access control is required throughout many parts of GameNomad, I've created a custom action helper called `LoginRequired` which checks for a valid session. If not valid session exists, the user is redirected to the login page. This action helper appears within the very first line of any restricted action. Of course, you will want to make sure such helpers are indeed granting authenticated users access to the restricted page, and so the following test will ensure an authenticated user can access the `logout` action. Notice how I am using the previously created `_loginValidUser()` method to handle the authentication process.

```
public function testLogoutPageAvailableToLoggedInUser()
{

  $this->_loginValidUser();

  $this->dispatch('/account/logout');

  $this->assertController('account');
  $this->assertAction('logout');

  $this->assertNotRedirectTo('/account/login');

}
```

You'll likewise want to verify that unauthenticated users cannot access restricted pages, however at the time of this writing the Zend_Test component does not play well with redirectors used within action helpers.

Testing the Account Registration Procedure

GameNomad requires the user to provide remarkably few items of information compared to many registration procedures, asking only for a username, zip code, e-mail address, and password.

Nonetheless, repeatedly manually entering this data in order to thoroughly test the registration form is an impractical use of time, and so you can instead create a test which can verify that the form is properly receiving valid registration data and adding it to the database. Of course, this is only part of the registration process, because the user also needs to confirm his e-mail address by clicking on a one-time URL before he can login to the GameNomad website. I'll talk more about the matter of model manipulation in Chapter 11. For the moment let's focus on making sure the form is working properly.

We know that the action should redirect the user to the login page if registration is successful, and so can create a test which determines whether the redirection occurs following registration:

```
public function testUsersCanRegisterWhenUsingValidData()
{

  $this->request->setMethod('POST')
       ->setPost(array(
           'username'         => 'jasong123',
           'zip_code'         => '43215',
           'email'            => 'jason1@wjgilmore.com',
           'password'         => 'secret',
           'confirm_pswd'     => 'secret',
         ));

  $this->dispatch('/account/register');

  $this->assertRedirectTo('/account/login');

}
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Explain how Zend_Auth knows which table and columns should be used when authenticating a user against a database.
- At a minimum, what are the five features you'll need to implement in order to offer basic user account management capabilities?
- Talk about the important role played by the `account` table's `recovery` column within several features described within this chapter.

Chapter 9. Creating Rich User Interfaces with JavaScript and Ajax

There's no use hiding it; I hate JavaScript. In years past, its frightful syntax and awkward debugging requirements had brought me to the sheer edge of insanity on more than one occasion. I'm not alone; the language is widely acknowledged for its ability to cause even the most even-tempered programmer to spew profanity. To put the scope of the frustration brought about by this language another way, consider the impressive promotion of non-violent protest espoused by the likes of John Lennon. I speculate this penchant for pacifism was at least in part attributed to his unfamiliarity with JavaScript.

Yet today there is really no way to call yourself a modern web developer and avoid the language. In fact, while you might over the course of various projects alternate between several web frameworks such as the Zend Framework, Grails (http://www.grails.org/) and Rails (http://www.rubyonrails.org/), JavaScript will likely be the common thread shared by all projects. This is because JavaScript is the special sauce behind the techniques used to create highly interactive web pages collectively known as *Ajax*.

Ajax makes it possible to build websites which behave in a manner similar to desktop applications, which offer a far more powerful and diverse array of user interface features such as data grids, autocomplete, and interactive graphs. Indeed, users of popular services such as Gmail, Flickr, and Facebook have quickly grown accustomed to these cutting-edge features. In order to stay competitive, you'll want to integrate similar features into your website so as to help attract and maintain an audience who has come to consider rich interactivity the norm rather than a novelty.

This puts us in a bit of a quandary: coding in JavaScript can be a drag, but it's become an unavoidable part of modern web development. Fortunately, many other programmers have come to the same conclusion, and so have put a great deal of work into building several powerful JavaScript frameworks which go a long way towards streamlining JavaScript's scary syntax.

In this chapter I'll introduce you JavaScript and the Ajax development paradigm, focusing on the popular jQuery JavaScript framework (http://www.jquery.com). jQuery happens to be so easy to use that it almost makes JavaScript development fun!

Introducing JavaScript

Because JavaScript is interpreted and executed by the browser, you'll either embed it directly into the web page, or manage it within a separate file in a manner similar to that typically done with CSS. The latter of these two approaches is recommended. The following example demonstrates how an external JavaScript file can be referenced:

```
01 <html>
02   <head>
03     <script type="text/javascript" src="/javascript/myjavascript.js"></script>
04   </head>
05   <body>
06     ...
07   </body>
08 </html>
```

In the context of the Zend Framework, the `javascript` directory referenced in line 03 would reside within the `/public/javascript/` directory, so if this directory doesn't exist, go ahead and create it now. Next create the `myjavascript.js` file, and place it in the directory. Within that file, add just a single line:

```
alert("I love video games!");
```

Load the page into the browser, and you'll see an alert box appear atop the browser window, as shown in Figure 9.1.

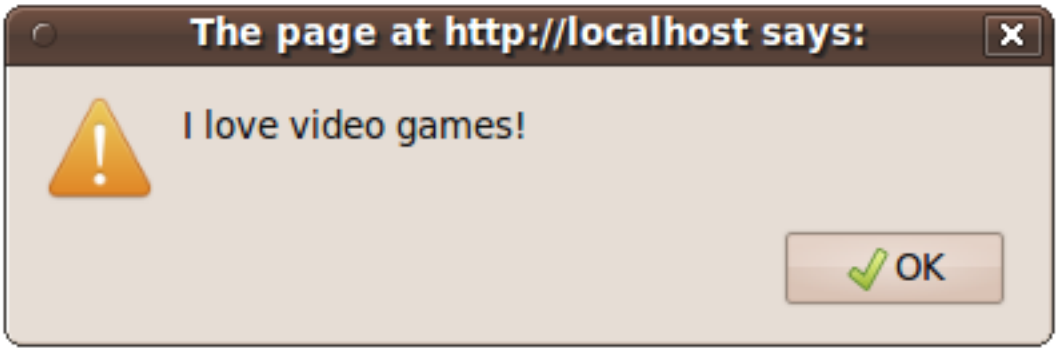


Figure 9.1. Creating a JavaScript alert window

While the approach of referencing an external script is recommended, for testing purposes you might occasionally prefer to just directly embed the JavaScript into the HTML like so:

```
<html>
  <head>
    <script type="text/javascript">
      alert("I love video games!");
    </script>
  </head>
  <body>
    ...
  </body>
</html>
```

Syntax Fundamentals

JavaScript sports countless features, and any attempt to cover even the fundamentals within a single chapter, let alone a single section, would be quite unrealistic. In fact I have cut out a great deal of original draft material in an attempt to provide you with only what's necessary to meet this chapter's ultimate goal, which is to teach you just enough JavaScript to take advantage of jQuery and Ajax within the context of the Zend Framework. From there, consider continuing your learning through the numerous online JavaScript tutorials, or by picking up one of the books mentioned in the below note.

Note

Although many great JavaScript books have been published over the years, I've long considered "Beginning JavaScript, Third Edition", co-authored by Paul Wilton and Jeremy McPeak, to be particularly indispensable.

Creating and Using Variables

Like PHP, you'll want to regularly create and reference variables within JavaScript. You can formally define variables by declaring them using the `var` statement, like so:

```
var message;
```

JavaScript is a case-sensitive language, meaning that `message` and `Message` are treated as two separate variables. You can also assign the newly declared variable a default value at creation time, like so:

```
var message = 'I love video games!';
```

Alternatively, JavaScript will automatically declare variables simply by the act of assigning a value to one, like so:

```
message = 'I love video games!';
```

I suggest using the former approach, declaring your variables at the top of the script when possible, perhaps accompanied by a JavaScript comment, which looks like this:

```
// Declare the default message
var message = 'I love video games!';
```

One aspect of JavaScript variable declarations which seems to confound so many developers is scope. However the confusion is easily clarified: whenever a variable is declared outside of a function (JavaScript functions are introduced in the next section), it is declared in the *global* scope. This is in contrast to JavaScript's behavior when declaring variables within a function. When declaring a variable within a function, it has local scope when declared using the `var` keyword; otherwise, it has global scope. It is very important that you memorize this simple point of differentiation, because it causes no end of confusion for those who neglect to heed this advice.

Creating Functions

Like PHP it's possible to create custom JavaScript functions which can accept parameters and return results. For instance, let's create a reusable function which displays the alert box presented in previous examples:

```
01 <html>
02  <head>
03    <script type="text/javascript">
04
05     // Displays a message via an alert box
06     function message()
07     {
08      // Declare the default message
09      var message = "I love video games!";
10
11      // Present the alert box
12      alert(message);
13     }
14   </script>
15  </head>
16  <body>
17   ...
18  </body>
19 </html>
```

As you can see, the function's declaration and enclosure look very similar to standard PHP syntax. Of course, like PHP the `message()` function won't execute until you call it, so insert the following line after line 13:

```
message();
```

Reloading the page will produce the same result shown in Figure 9.1.

You can pass input parameters into a JavaScript function just as you do with PHP; when defining the function just specify the name of the variable as it will be used within the function body. For instance, let's modify the `message()` method to pass along a revised statement:

```
01 // Displays a message via an alert box
02 function message(user, hobby)
03 {
04   // Present the alert box
05   alert(user + " is the " + hobby + " player of the year!");
06 }
```

You can then pass along a user's name and their favorite pastime to create a custom message:

```
message("Jason", "Euchre");
```

Reloading the browser window produces an alert box identical to that shown in Figure 9.2.

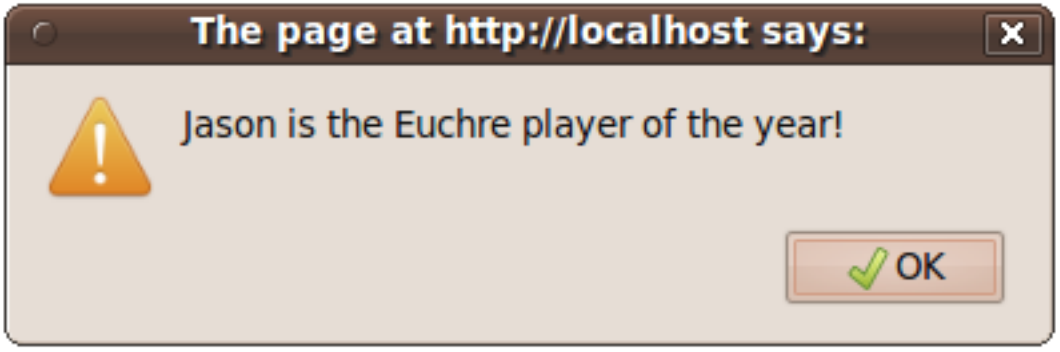


Figure 9.2. Using a custom function

Tip

Like PHP, JavaScript comes with quite a few built-in functions. You can peruse a directory of these functions here: http://www.javascriptkit.com/jsref/.

Working with Events

Much of your time working with JavaScript will be spent figuring out how to make it do something in reaction to a user action, for instance validating a form when a user presses a submit button. In fact, you can instruct JavaScript to do something only after the page has completely loaded by embedding the `onload()` event handler into the page. For instance, you can direct our custom `message()` function to execute after the page is loaded by modifying the `body` element:

```
<html>
  <head>
    <script type="text/javascript">

      // Displays a message via an alert box
      function message()
      {
        // Declare the default message
        var message = 'I love video games!';

        // Present the alert box
        alert(message);
      }
    </script>
  </head>
  <body onload="message()">
    ...
  </body>
</html>
```

Reload this example, and you'll see the alert window appear. The difference is that the window appears *only* after all of the page elements have completely loaded into the browser window. This is an important concept because you'll often write JavaScript code which is intended to interact with specific page elements such as a `div` element assigned the ID `game`. If the JavaScript happens to execute before this element has been loaded into the browser, then the desired functionality is sure not to occur. Therefore you'll find this event-based approach to ensuring the JavaScript executes only after the desired page elements are available to be quite common.

So how do you cause JavaScript to execute based on some other user action, such as clicking a submit button? In addition to `onload()`, JavaScript supports numerous other event handlers such as

`onclick()`, which will cause a JavaScript function to execute when an element attached to the event handler is clicked. Add the following code within the `body` tag (and remove the `onload()` function from the `body` element) for an example:

```
<input type="submit" name="submit" value="Click Me!" onclick="message();">
```

The button and window which pops up once the button is clicked is shown in Figure 9.3.

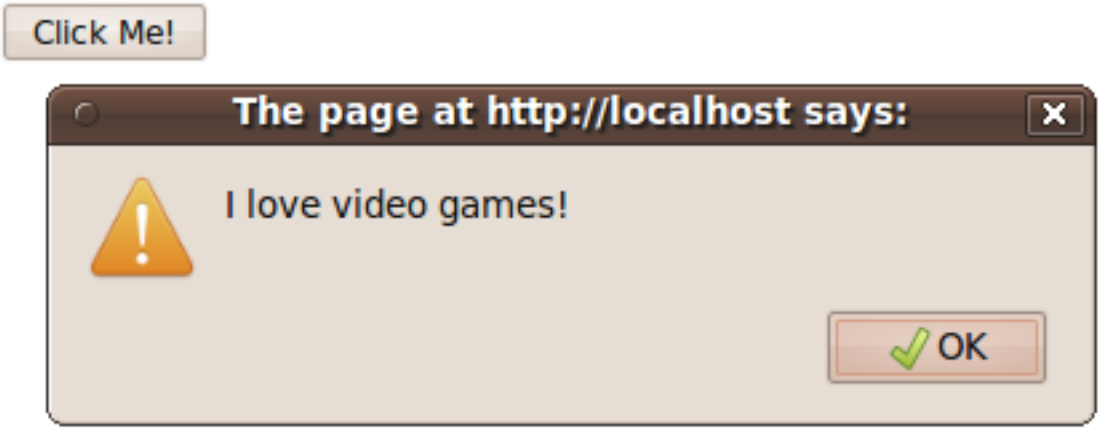


Figure 9.3. Executing an action based on some user event

The same behavior is repeated when using a simple hyperlink, an image, or almost any other element for that matter. For instance, try adding the following two lines to the page and clicking on the corresponding elements:

```
<a href="#" onclick="message();">Click me right now!</a><br />
<h1 onclick="message();">I'm not a link but click me anyway!</h1>
```

See Table 9-1 for a list of other useful JavaScript handlers. Try swapping out the `onclick` handler used in the previous examples with handlers found in this table to watch their behavior in action.

Table 9.1. Useful JavaScript Event Handlers

Event Handler	Description
`onblur`	Executes when focus is removed from a select, text, or textarea form field.

Event Handler	Description
`onchange`	Executes when the text in an input form field is changed.
`onclick`	Executes when the element is clicked upon.
`onfocus`	Executes when the element is placed into focus (typically an input form field).
`onload`	Executes when the element is loaded
`onmouseover`	Executes when the mouse pointer is moved over an element.
`onmouseout`	Executes when the mouse pointer is moved away from an element.
`onselect`	Executes when text within a text or textarea form field is selected.
`onsubmit`	Executes when a form is submitted.
`onunload`	Executes when the user navigates away or closes the page.

Forms Validation

Let's consider one more example involving an HTML form. Suppose you wanted to ensure the user doesn't leave any fields empty when posting a video game review to your website. According to what's available in Table 9-1, it sounds like the onsubmit event handler will do the trick nicely. But first we have to create the JavaScript function to ensure the form field isn't blank upon submission:

```
function isNotEmpty(formfield)
{
  if (formfield == "")
  {
    return false;
  } else {
    return true;
  }
}
```

Nothing much to review here; the `isNotEmpty()` function operates on the premise that if the formfield parameter is blank, `FALSE` is returned, otherwise `TRUE` is returned.

From here, you can reuse this function as many times as you please by referencing it within another function, which we'll call `validate()`:

```
01 function validate()
02 {
03   // Retrieve the form's title field
04   title = document.getElementById("title").value;
```

```
05
06   // Retrieve the form's review field
07   review = document.getElementById("review").value;
08
09   // Verify neither field is empty
10   if (isNotEmpty(title) && isNotEmpty(review))
11   {
12     return true;
13   } else {
14     alert("All form fields are required.");
14     return false;
15   }
16 }
```

As this is the most complex example presented thus far, let's break down the code:

- Lines 04 and 06 use something called the Document Object Model (DOM) to retrieve the values of the elements identified by the title and review identifiers. The DOM is a very powerful tool, and one I'll introduce in detail in the next section.

- Line 10 uses the custom `isNotEmpty()` function to examine the contents of the title and review variables. If both variables are indeed not empty, true is returned which will cause the form's designated action to be requested. Otherwise an error message is displayed and `FALSE` is returned, causing the form submission process to halt.

Finally, construct the HTML form, attaching the `onsubmit` event handler to the `form` element:

```
<form action="/reviews/post" method="POST" onsubmit="return validate();">
  <p>
    <label name="title">Please title your review:</label><br />
    <input type="text" id="title" name="title" value="" size="50" />
  </p>
  <p>
    <label name="review">Enter your review below</label><br />
    <textarea name="review" id="review" rows="10" cols="35"></textarea>
  </p>
  <p>
    <input type="submit" name="submit" value="Post review" />
  </p>
</form>
```

Should the user neglect to enter one or both of the form fields, output similar to that shown in Figure 9.4 will be presented.

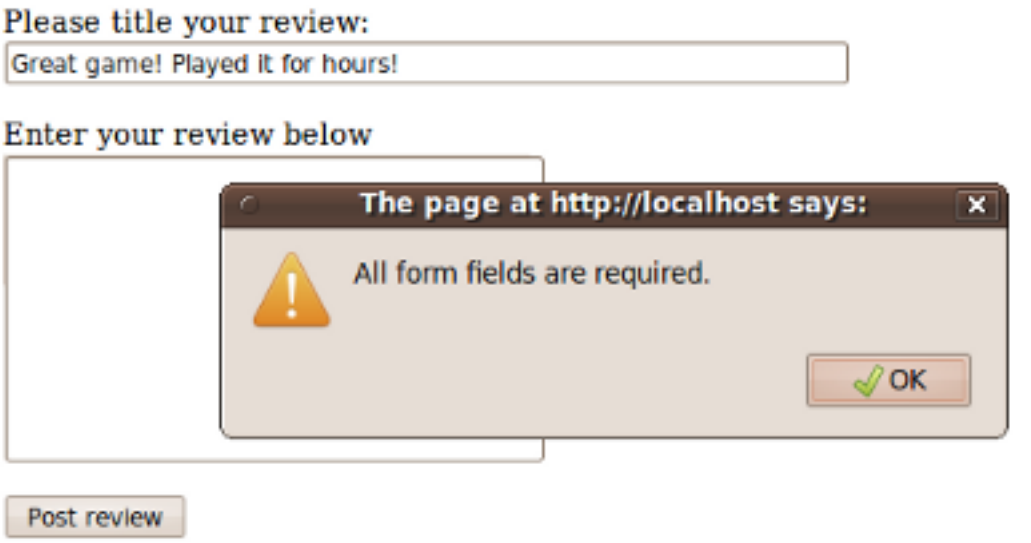


Figure 9.4. Validating form fields with JavaScript

The use of the Document Object Model (DOM) to easily retrieve parts of an HTML document, as well as user input, is a crucial part of today's JavaScript-driven features. In the next section I'll formally introduce this feature.

Introducing the Document Object Model

Relying upon an event handler to display an alert window can be useful, however events can do so much more. Most notably, we can use them in conjunction with a programming interface known as the *Document Object Model* (DOM) to manipulate the HTML document in interesting ways. The DOM is a standard specification built into all modern browsers which makes it trivial for you to reference a very specific part of a web page, such as the `title` tag, an `input` tag with an id of email, or all `ul` tags. You can also refer to properties such as `innerHTML` to retrieve and replace the contents of a particular tag. Further, it's possible to perform all manner of analytical and manipulative tasks, such as determining the number of `li` entries residing within a `ul` enclosure.

JavaScript provides an easy interface for interacting with the DOM, done by using a series of built-in methods and properties. For instance, suppose you wanted to retrieve the contents of a `p` tag (known as an element in DOM parlance) having an id of `message`. The element and surrounding HTML might look something like this:

```
01 ...
02 <p id="message">Your profile has been loaded.</p>
03 <h1 id="gamertag">wjgilmore</h1>
04 Location: <b id="city">Columbus</b>, <b id="state">Ohio</b>
05 ...
```

To retrieve the text found within the `p` element (line 02), you would use the following JavaScript command:

```
<script type="text/javascript">
  message = document.getElementById("message").innerHTML;
</script>
```

You can prove the text was indeed retrieved by passing the message variable into an alert box in a line that follows:

```
alert("Message retrieved: " + message);
```

Adding the `alert()` function produces the alert box containing the message "Your profile has been loaded.".

Retrieving the text is interesting, but changing the text would be even more so. Using the DOM and JavaScript, doing so is amazingly easy. Just retrieve the element ID and assign new text to the `innerHTML` property!

```
document.getElementById("message").innerHTML = "Your profile has been updated!";
```

Simply adding this to the embedded code doesn't make sense, because doing so will change the text from the original to the updated version before you really have a chance to see the behavior in action. Therefore let's tie this to an event by way of creating a new function:

```
function changetext()
{
  document.getElementById("message").innerHTML = "Your profile has been updated!";
}
```

Next, within the HTML body just tie the function to an `onclick` event handler as done earlier:

```
<a href="#" onclick="changetext();">Click here to change the text</a>
```

Everything you've learned so far lays the foundation for integrating Ajax-oriented features into your website. However, because your success building Ajax-driven features is going to rest heavily upon your ability to write clean and coherent JavaScript, in the next section I'll introduce you to the jQuery library, which we'll subsequently use to create these great features.

Introducing jQuery

In recent years, many ambitious efforts have been undertaken to create solutions which abstracted many of the tedious, repetitive, and difficult tasks faced by developers seeking to integrate JavaScript-driven features into their websites. By taking advantage of these JavaScript libraries, the most popular of which are open source and therefore freely available to all users, developers are able to write JavaScript not only faster, but more efficiently and with less errors than ever before. Furthermore, because many of these libraries are extendable, other enterprising developers are able to contribute their own extensions back to the community, greatly increasing library capabilities.

JavaScript libraries also deal with another significant obstacle that beginning web developers tend to overlook: the matter of cross-browser compatibility. Although significant improvements have been made in recent years to ensure uniform behavior within all browsers, a great deal of pain remains when it comes to writing cross-browser JavaScript code that is perfectly compatible in all environments. Most JavaScript libraries remove, or at least greatly reduce, this pain by providing you with a single interface for implementing a feature which the library will then adjust according to the type of browser being used by the end user.

One of the most popular such libraries is jQuery (http://www.jquery.com/). Created in early 2006 by John Resig (http://www.ejohn.org/), a seemingly tireless JavaScript guru who among other things is a JavaScript Tool Developer for the Mozilla Corporation (the company behind the Firefox browser), jQuery has fast become one of the web development world's most exciting technologies. With thousands of websites already using the library, and embraced by companies such as Microsoft and Nokia, chances are you've already marveled at its impressive features more than once.

Caution

Don't think of jQuery or any other JavaScript library as a panacea for learning JavaScript; rather it complements and extends the language in an effort to make you a more efficient JavaScript developer. Ultimately, gaining a sound understanding of the JavaScript language will serve to make you a better jQuery developer, so be sure to continue brushing up on your JavaScript skills as time allows.

Installing jQuery

jQuery is self-contained within a single JavaScript file. While you could download it directly from the jQuery website, there's a far more efficient way to add the library to your site. Google hosts all released versions of the library on their lightning-fast servers, and because many sites link to

Google's hosted version, chances are the user already has a copy cached within his browser. To include the library within your site, add the following lines within the `head` enclosure:

```
<script src="http://www.google.com/jsapi"></script>
<script type="text/javascript" >
  google.load("jquery", "1");
</script>
```

In this example, the `1` parameter tells Google to serve the most recent stable 1.X version available. If you need the highest release in the 1.3 branch, pass along `1.3`. If you desire a specific version, such as `1.4.4`, pass that specific version number.

If you would like to peruse the source code, you can download the latest release from the jQuery website. There you'll find a "minified" and an uncompressed version of the latest release. You should download the uncompressed version because in the minified version all code formatting has been eliminated, producing a smaller file size and therefore improved loading performance.

Managing Event Loading

Because much of your time spent working with jQuery will involve manipulating the HTML DOM (the DOM comprises all of the various page elements which you may want to select, hide, toggle, modify, animate, or otherwise manipulate), you'll want to make sure the jQuery JavaScript doesn't execute until the entire page has loaded to the browser window. Therefore you'll want to encapsulate your jQuery code within the `google.setOnLoadCallback()` method, like this:

```
<script type="text/javascript" >
  google.setOnLoadCallback(function() {
    alert("jQuery is cool.");
  });
</script>
```

Add the `setOnLoadCallback()` method to your newly jQuery-enabled web page, and you'll see the alert box presented in Figure 9.5.

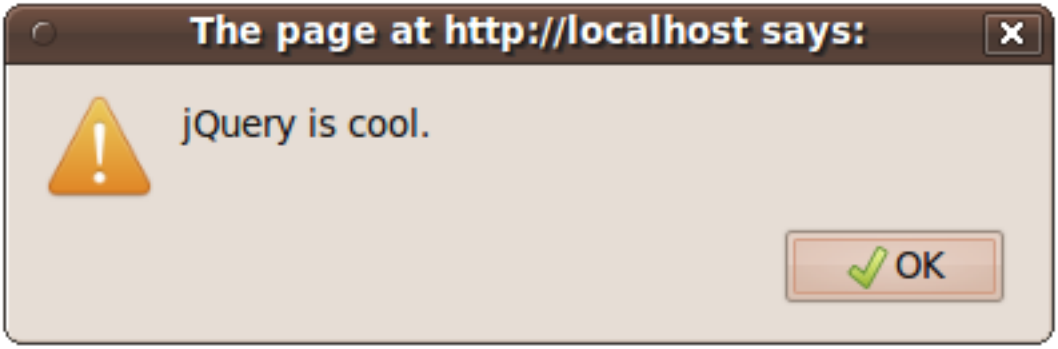


Figure 9.5. Triggering an alert box after the DOM has loaded

If you're not loading the jQuery library from Google's CDN, the loading event syntax will look like this:

```
$(document).ready(function() {
  alert("jQuery is cool.");
});
```

You can use this syntax when the jQuery library is being served from your server, however when using jQuery in conjunction with Google's content distribution mechanism you'll need to use the former syntax.

DOM Manipulation

One of the most common tasks you'll want to carry out with jQuery is DOM manipulation. Thankfully, jQuery supports an extremely powerful and flexible selector engine for parsing the page DOM in a variety of ways. In this section I'll introduce you to this feature's many facets.

Retrieving an Object By ID

You'll recall from earlier in this chapter that JavaScript can retrieve a DOM object by its ID using the `getElementById()` method. Because this is such a common task, jQuery offers a shortcut for calling this method, known as the dollar sign function. Thus, the following two calls are identical in purpose:

```
var title = document.getElementById("title");
var title = $("#title");
```

In each case, title would be assigned the object identified by a DIV such as this:

```
<p id="title">The Hunt for Red October, by Tom Clancy</p>
```

Keep in mind that in both cases `title` is assigned an *object*, and not the element contents. For instance, you can use the object's `text()` method to retrieve the element contents:

```
alert(title.text());
```

To retrieve the element content length, reference the `length` attribute like this:

```
alert(title.text().length);
```

Several other properties and methods exist, including several which allow you to traverse an element's siblings, children, and parents. Consult the jQuery documentation for all of the details.

Retrieving Objects by Class

To retrieve all objects assigned to a particular class, use the same syntax as that used to retrieve an element by its ID but with a period preceding the class name rather than a hash mark:

```
var titles = document.getElementById(".title");
var titles = $("title");
```

For instance, given the following HTML, titles would be assigned an array of three objects:

```
<p class="title">The Hunt for Red October, by Tom Clancy</p>
<p class="title">On Her Majesty's Secret Service, by Ian Fleming</p>
<p class="title">A Spy in the Ointment, by Donald Westlake</p>
```

To prove that `titles` is indeed an array containing three objects, you can iterate over the array and retrieve the text found within each object using the following snippet:

```
$.each(titles, function(index)
{
  alert($(this).text());
});
```

jQuery's dollar sign syntax can also be used to retrieve HTML elements. For instance, you can use this call to retrieve the all `h1` elements on the page:

```
var headers = $("h1");
```

Retrieving and Changing Object Text

As was informally demonstrated in several preceding examples, to retrieve the text you'll need to call the object's `.text()` method. The following example demonstrates how this is accomplished:

```
<script type="text/javascript" >
  alert($("#title").text());
</script>
<body>
  <p id="title">The Hunt for Red October, by Tom Clancy</p>
</body>
```

To change the text, all you need to do is pass text into the `.text()` method. For instance, the following example will swap out Tom Clancy's book with a book by Donald Westlake:

```
<script type="text/javascript">
  google.setOnLoadCallback(function() {
    $("#title").text("A Spy in the Ointment, by Donald Westlake");
  });
</script>
<body>
  <p id="title">The Hunt for Red October, by Tom Clancy</p>
</body>
```

Working with Object HTML

The `text()` method behaves perhaps a bit unexpectedly when an object's text includes HTML tags. Consider the following HTML snippet:

```
<p id="title"><i>The Hunt for Red October</i>, by Tom Clancy</p>
```

If you were to retrieve the title ID's text using the `text()` method, the following string would be returned:

```
The Hunt for Red October, by Tom Clancy
```

So what happened? The `text()` method will strip out any HTML tags found in the text, which might be perfectly acceptable depending upon what you want to do with the text. However, if you'd also like the HTML, use the `html()` method instead:

```
var title = $("#title").html();
```

The same concept applies when adding or replacing text. If the new text includes HTML, and you attempt to insert it using `text()`, the HTML will be encoded and output as text on the page. However,

if you use `html()` when inserting HTML-enhanced text, the tags will be rendered by the browser as expected.

Determining Whether a DIV Exists

Because jQuery or a server-side language such as PHP could dynamically create DOM elements based on some predefined criteria, you'll often need to first verify an element's existence before interacting with it. However, you can't just check for existence, because jQuery will always return `TRUE` even if the DIV does not exist:

```
if ($("#title")) {
  alert("The title div exists!");
}
```

However, an easy way to verify existence is to use one of the members exposed to each available DIV, for instance `length`:

```
if ( $("#title").length > 0 ) {
  alert("The title div exists!");
}
```

Removing an Element

To remove an element from the page, use the `remove()` method. For instance, the following example will remove the element identified by the news ID from the page:

```
$("title").remove();
...
<div id="title"><i>The Day of the Jackal</i>, by Frederick Forsyth</div>
```

Retrieving a Child

Many of the previous elements in this section referenced a book title and its author, with some of the examples delimiting the book title with italics tags (`i`):

```
<div id="title"><i>The Day of the Jackal</i>, by Frederick Forsyth</div>
```

What if you wanted to retrieve the book title, but not the author? You can use jQuery's child selector syntax to retrieve the value of a nested element:

```
var title = $("#title > i").text();
```

Similar features exist for retrieving an element's siblings and parents. See the jQuery documentation for more details.

Event Handling with jQuery

Of course, all of the interesting jQuery features we've introduced so far aren't going to happen in a vacuum. Typically DOM manipulation tasks such as those described above are going to occur in response to some sort of user- or server-initiated event. In this section you'll learn how to tie jQuery tasks to a variety of events. In fact, you've already been introduced to one such event, namely the Google Ajax API's `setOnLoadCallback()` method. This code contained within it executes once the method confirms that the page has successfully loaded.

jQuery can respond to many different types of events, such as a user-initiated mouse click, double-click, or mouseover. As you'll see later in this chapter, it can also monitor for changes to web forms, such as when the user begins to insert text into a text field, changes a select box, or presses the submit button.

Creating Your First Event Handler

Earlier in this chapter I talked about JavaScript's predefined event handlers, including mouse click `onclick`, mouse over `mouseover`, and form submission `onsubmit`). jQuery works in the same way, although its terminology occasionally strays from that used within standard JavaScript. Table 9-2 introduces jQuery's event types.

Table 9.2. jQuery's supported event types

Event Handler	Description
`blur`	Executes when focus is removed from a select, text, or textarea form field.
`change`	Executes when the value of an event changes.
`click`	Executes when an element is clicked.
`dblclick`	Executes when an element is double-clicked.
`error`	Executes when an element is not loaded correctly.
`focus`	Executes when an element gains focus.
`keydown`	Executes when the user first presses a key on the keyboard.
`keypress`	Executes when the user presses any key on the keyboard.
`keyup`	Executes when the user releases a key on the keyboard.
`load`	Executes when an element and its contents have been loaded.

Event Handler	Description
mousedown	Executes when the mouse button is clicked atop an element.
mouseenter	Executes when the mouse pointer enters the element.
mouseleave	Executes when the mouse pointer leaves the element.
mousemove	Executes when the mouse pointer moves while inside an element
mouseout	Executes when the mouse pointer leaves the element.
mouseover	Executes when the mouse pointer enters the element.
mouseup	Executes when the mouse button is released while atop an element.
resize	Executes when the size of the browser window changes.
scroll	Executes when the user scrolls to a different place within the element.
select	Executes when the user selects text residing inside an element.
unload	Executes when the user navigates away from the page.

jQuery actually supports numerous approaches to tying an event to the DOM, however the easiest involves using an *anonymous function*. In doing so, we'll bind the page element to one of the events listed in Table 9-2, defining the function which will execute when the event occurs. The following example will toggle the CSS class of the paragraph assigned the ID title:

```
01 <style type="text/css">
02   .clicked { background: #CCC; padding: 2px;}
03 </style>
04
05 <script type="text/javascript" >
06
07   google.load("jquery", "1");
08
09     google.setOnLoadCallback(function() {
10        $("#title").bind("click", function(e){
11        $("#title").toggleClass("clicked");
12      });
13   });
14
15 </script>
16
17 </head>
18 <body>
```

```
19   <p id="title">The Hunt for Red October, by Tom Clancy</p>
20 </body>
```

Let's review the code:

- Lines 01-03 define the style which will be toggled each time the user clicks on the paragraph.
- Lines 10-12 define the event handler, binding an anonymous function to the element ID title. Each time this element ID is clicked, the CSS class clicked will be toggled.
- Line 19 defines the paragraph assigned the element ID title.

Try executing this script to watch the CSS class change each time you click on the paragraph. Then try swapping out the click event with some of the others defined in Table 9-2.

Introducing Ajax

You might be wondering why I chose to name this section title "Introducing Ajax". After all, haven't we been doing Ajax programming in many of the prior examples? Actually, what we've been doing is fancy JavaScript programming involving HTML, CSS and the DOM. As defined by the originator of the term Ajax Jesse James Garrett, several other requisite technologies are needed to complete the picture, including notably XML (or similarly globally understood format) and the XMLHttpRequest object. With the additional technologies thrown into the mix, we're able to harness the true power of this programming technique, which involves being able to communicate with the Web server in order to retrieve and even update data found or submitted through the existing Web page, without having to reload the entire page!

By now you've seen the power of Ajax in action countless times using popular websites such as Facebook, Gmail, and Yahoo!, so I don't think I need to belabor the advantages of this feature. At the same time, it's doubtful an in-depth discussion regarding how all of these technologies work together is even practical, particularly because it's possible to take advantage of them without having to understand the gory details, much in the same way we can use many Zend Framework components without being privy to the underlying mechanics.

Passing Messages Using JSON

Ajax-driven features are the product of interactions occurring between the client and server, which immediately raises a question. If the client-side language is JavaScript and the server-side language is PHP, how is data passed from one side to the other in a format both languages can understand? There are actually several possible solutions, however JSON (JavaScript Object Notation) has emerged as the most commonly used format.

JSON is an open standard used to format data which is subsequently serialized and transmitted over a network. Unlike many XML dialects is actually quite readable, although of course it is ultimately intended for consumption by programming languages. For instance, the following presents a JSON snippet which represents an object describing a video game:

```
{
  "asin": "B002I0K780",
  "name": "LittleBigPlanet 2",
  "rel": "January 18, 2011",
  "price":"59.99"
}
```

Both jQuery and PHP offer easy ways to both write and read JSON, meaning you'll be able to pass messages between the client and server without having to worry about the complexities of JSON message formatting and parsing. You'll see how easy and frankly transparent it is to both construct and parse these messages in the example that follows.

Validating Account Usernames

Any social networking website requires users to be uniquely identifiable, logically because users need to be certain of their friends' identities before potentially sharing personal information. GameNomad uses a pretty simplistic solution for ensuring users are uniquely identifiable, done by requiring users to choose a unique username when creating a new account.

Such a constraint can be a source of frustration for users who complete the registration form only to be greeted with an error indicating that the desired username has already been taken. On a popular website it's entirely likely that a user could submit the form several times before happening to choose an unused username, no doubt causing some frustration and possibly causing the user to give up altogether. Many websites alleviate the frustration by providing users with real-time feedback regarding whether the desired username is available, done by using Ajax to compare the provided username with those already found in the database, and updating the page asynchronously with some indication of whether the username is available.

In order to verify the availability of a provided username in real-time an event-handler must be associated with the registration form's `username` field. The `username` field looks like this:

```
<input type="text" name="username" id="username" value="" size="35">
```

Because we want validation to occur the moment the user enters the desired username into this field, a `blur` event is attached to the `username` field. The `blur` event handler will execute as soon as focus is taken *away* from the associated DOM element. Therefore when the user completes the `username` field and either tabs or moves the mouse to the next field, the handler will execute.

This handler is presented below, followed by some commentary:

```
01 $('#username').bind('blur', function (e) {
02
03   $.getJSON('/ws/username',
04     {username: $('#username').val()},
05     function(data) {
06       if (data == "TRUE") {
07         $("#available").text("This username is available!");
08       } else {
09         $("#available").text("This username is not available!");
10       }
11     }
12   );
13
14 });
```

Let's review the code:

- Line 01 defines the handler, associating a `blur` handler with the DOM element identified by `username`.
- Line 03 specifies that a GET request will be sent to `/ws/username` (The `ws` controller's `username` action), and that JSON-formatted data is expected in return.
- Line 04 sends a GET parameter named `username` to `/ws/username`. This parameter is assigned the value of whatever is found in the `username` field.
- Lines 05-11 define the anonymous function which executes when the response is returned to the handler. If the response is `TRUE`, the username is available and the user will be notified accordingly (by updating a DIV associated with the ID `available`). Otherwise, the username has already been taken and the user will be warned.

Next let's examine the `ws` controller (`ws` is just a convenient abbreviation for web services) and the `username` action used to verify the username's existence. This controller is presented next, followed by some commentary:

```
01 <?php
02
03 class WsController extends Zend_Controller_Action
04 {
05
06   public function init()
07   {
```

```
08     $this->em = $this->_helper->EntityManager();
09     $this->_helper->layout()->disableLayout();
10     Zend_Controller_Front::getInstance()
11       ->setParam('noViewRenderer', true);
12   }
13
14   public function usernameAction()
15   {
16
17     // Retrieve the provided username
18     $username = $this->_request->getParam('username');
19
20     // Does an account associated with username already exist?
21     $account = $this->em->getRepository('Entities\Account')
22                     ->findOneByUsername($username);
23
24     // If $account is null, the username is available
25     if (is_null($account))
26     {
27       echo json_encode("TRUE");
28     } else {
29       echo json_encode("FALSE");
30     }
31
32   }
33
34 }
```

Let's review the code:

- Lines 06-12 define the controller's `init` method. In this method we'll disable both the layout and view renderer, because the controller should not render anything other than the returned JSON-formatted data.

- Lines 14-32 define the `username` action. This is pretty standard stuff by this point in the book, involving using Doctrine to determine whether the provided username already exists. If it doesn't, `TRUE` is returned to the caller, otherwise `FALSE` is returned.

Keep in mind that you shouldn't rely solely upon JavaScript-based features for important validation tasks such as verifying username availability; a malicious user could disable JavaScript and wreak a bit of havoc by introducing duplicate usernames into the system. To be safe, you should also carry out similar validation procedures on the server-side.

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Why should you link to the jQuery library via Google's content distribution network rather than store a version locally?
- What role does jQuery's `$.getJSON` method play in creating the Ajax-driven feature discussed earlier in this chapter?

Chapter 10. Integrating Web Services

Data is the lifeblood of today's economy, with companies like Google, Microsoft, and Amazon.com spending billions of dollars amassing, organizing, parsing, and analyzing astoundingly large sums of information about their products, services, users, and the world at large. So it may seem counterintuitive that all of the aforementioned companies and many others make this data available for others for free.

By exposing this data through an application programming interface (API) known as a *web service*, their goal is to provide savvy developers with a practical way to present this data to others. Additional exposure will hopefully lead to increased interest in that company's offerings, with increased revenues to follow. A great example of this strategy is evident in Amazon.com's Product Advertising API. Via the Product Advertising API, Amazon exposes information about almost every product in what is undoubtedly the largest shopping catalog on the planet, including the product title, manufacturer, price, description, images, sales rank, and much more. You're free to use this information to create new and interesting online services, provided you follow the API's terms of service, which among other requirements demands that any product information retrieved from the API is linked back to the product's primary Amazon product description page.

Other web services such as the Google Maps API and Twitter API, expose both data and useful features which allow you to interact with the service itself. For instance, the Google Maps API provides you with not only the ability to render a map centered over any pair of coordinates, but also the opportunity to plot markers, routes, and create other interesting location-based services. Likewise, the Twitter API not only gives you the ability to search the ever-growing mountain of tweets, but also the ability to update your own account with new updates.

The Zend Framework offers a particularly powerful set of web services-related components which connect to popular APIs including those offered by Amazon.com, eBay, Flickr, Google, Microsoft, Twitter, Yahoo, and others. In this chapter I'll introduce you to Zend_Service_Amazon (the gateway to the Product Advertising API), a Zend Framework component which figures prominently into GameNomad, and also show you how easy it is to integrate the Google Maps API into your Zend Framework application despite the current lack of a Zend Framework Google Maps API component.

Introducing Amazon.com's Product Advertising API

Having launched the Associates program back in 1996, before much of the world had even heard of the Internet, Amazon.com founder Jeff Bezos and his colleagues clearly had a prescient understanding of the power of hyperlinking. By providing motivated third-parties, known as associates, with an easy way to promote Amazon products on their own websites and earn a percentage of any sales occurring as a result of their efforts, Amazon figured they could continue to grow market share in the fledgling e-commerce market. Some 13 years later, the Amazon Associates program is an online juggernaut, with everybody from large corporations to occasional bloggers using the program to enhance the bottom line.

With over a decade of experience under their belts, Amazon has had plenty of time and opportunity to nurture their Associates program. Early on in the program's lifetime, associates' options were limited to the creation of banners and other basic links, however over time the program capabilities grew, with today's associates given a wealth of tools for linking to Amazon products using a variety of links, banners, widgets, search engines. Users are also provided with powerful sales analysis tools which help them gauge the efficacy of their efforts.

Along the way, Amazon.com unveiled the shining gem of the associates program: the *Amazon Product Advertising API* (formerly known as the *Amazon Associates Web Service*). This service made Amazon's enormous product catalog available via an API, giving developers the ability to retrieve and manipulate this data in new and creative ways. Via this API developers have access to all of the data they could conceivably need to build a fascinating new solution for perusing Amazon products, including product titles, ASINs (Amazon's internal version of the UPC code, known as the *Amazon Standard Identification Number*), product release dates, prices, manufacturer names, Amazon sales ranks, customer and editorial reviews, product relations (products identified as being similar to one another), images, and much more!

But before you can begin taking advantage of this fantastic service, you'll need an Amazon customer account. I'll presume like the rest of the world you already have one but if not head over to Amazon.com and create that now. Additionally, you'll probably want to create an Amazon Associates account so you can potentially earn additional revenue when linking products back to their Amazon.com product page.

Joining the Amazon Associates Program

Joining the Amazon Associates Program is free, and only requires you to complete a short registration form in which you'll provide your payment and contact information, website name, URL and description, in addition to declare agreement to the Amazon Associates operating agreement. To

register for the program, head over to https://affiliate-program.amazon.com/ and click on the `Join now for FREE!` button to start the process.

You'll be prompted to provide information about the website you intend on using to advertise Amazon products. After providing this information and agreeing to the Amazon Associates Operating Agreement, a unique Associates ID will be generated. As you'll soon learn, you'll attach this associate ID to the product URLs so Amazon knows to what account they should credit the potential purchase. At this point you'll also be prompted to identify how you'd like to be paid, either by direct deposit, check, or Amazon gift card.

Creating Your First Product Link

Although the point of this section is to introduce the Amazon Product Advertising API, it's worth taking a moment to understand how to easily create Amazon product URLs which include your affiliate ID. In order to be credited for any sales taking place as a result of using your affiliate links, you'll need to properly include your Associate ID within the product link. When using Amazon's automated wizards for creating product links (head over to https://affiliate-program.amazon.com/ to learn more about these) you'll find these links to be extremely long and decidedly not user-friendly. However, they support a shortcut which allows you to create succinct alternative versions. For instance, the following link will point users to Amazon's product detail page for the video game Halo 3 for the Xbox 360, tying the link to GameNomad's affiliate account:

```
http://www.amazon.com/exec/obidos/ASIN/B000FRU0NU/gamenomad-20
```

As you can see, this link consists of just two pieces of dynamic information: the product's ASIN, and the associate identifier. Don't believe it's this easy? Head on over to the Amazon Associates Link Checker (http://goo.gl/jOmlP) and test it out. Enter the link into the form (you'll need to swap out my Associate ID with your own), and press the Load Link button. The page will render within an embedded frame, confirming you're linking to the appropriate product. Once rendered, click the `Check Link` button to confirm the page is linked to your associate identifier.

Of course, you'll probably want to include much more than a mere few links. Using the Amazon Product Advertising API, we can do this on a large scale. I'll devote the rest of this section to how you can use the API to quickly amass a large product database.

Creating an Amazon Product Advertising API Account

To gain access to Amazon's database and begin building your catalog, you'll need to create an API account. After completing the registration process you'll be provided with two access identifiers which you'll use to sign into the API. To obtain an account, head over to http://aws.amazon.com/

associates/ and click the `Sign Up Now` button. You'll be asked to sign in to your existing Amazon.com account, and provide contact information, your company name or website, and the website URL where you'll be invoking the service. You'll also be asked to read and agree to the AWS Customer Agreement. Please read this agreement carefully, because there are some stipulations which can most definitely affect your ability to use this information within certain applications. Once done, Amazon will confirm the creation of your account. Click on the `Manage Your Account` link to retrieve your keys. From here click on the `Access Identifiers` link. You'll be presented with two identifiers, your `Access Key ID` and your `Secret Access Key`. Copy these keys into your `application.ini` file, along with your associate ID:

```
;------------------------
; Amazon
;------------------------
amazon.product_advertising.public.key = "12345678ABCDEFGHIJK"
amazon.product_advertising.public.private.key = "KJIHGFESECRET876"
amazon.product_advertising.country = "US"
amazon.associate_id = "gamenomad-20"
```

We'll use these keys to connect to Amazon using the Zend_Service_Amazon component, introduced in the next step.

Retrieving a Single Video Game

Amazon.com has long used a custom product identification standard known as the Amazon Standard Identification Number, or ASIN. These 10-digit alphanumerical strings uniquely identify every product in the Amazon.com catalog. Of course, you need to know what the product's ASIN is in order to perform such a query, so how do you find it? The easiest way is to either locate it within the product's URL, or scroll down the product's page where it will be identified alongside other information such as the current Amazon sales rank and manufacturer name. For instance, the ASIN for Halo 3 on the Xbox 360 is `B000FRU0NU`. With that in hand, we can use the Zend_Services_Amazon component to query Amazon. Use the following code snippet to retrieve the product details associated with the ASIN `B000FRU0NU`:

```
01 $amazonPublicKey = Zend_Registry::get('config')
02                     ->amazon->product_advertising->public->key;
03 $amazonPrivateKey = Zend_Registry::get('config')
04                     ->amazon->product_advertising->private->key;
05
06 $amazonCountry = Zend_Registry::get('config')->amazon->country;
07
08 $amazon =
09   new Zend_Service_Amazon($amazonPublicKey, $amazonCountry, $amazonPrivateKey);
10
```

```
11 $item = $amazon->itemLookup('B000FRU0NU', array('ResponseGroup' => 'Medium'));
12
13 echo "Title: {$item->Title}<br />";
14 echo "Publisher: {$item->Manufacturer}<br />";
15 echo "Category: {$item->ProductGroup}";
```

Although I'd venture a guess this code is self-explanatory, let's nonetheless expand upon some of its key points:

- Lines 01-06 retrieve the assigned Product Advertising API public and private keys, and the country setting. I'm based in the United States and so have set this to "US", however if you were in the United Kingdom you'd presumably want to use the product catalog associated with http://www.amazon.co.uk and so you'll set your country code to `UK`. See the Product Advertising API manual for a complete list of available codes.

- Lines 08-09 instantiates the Zend_Service_Amazon component class, readying it for subsequent authentication and product retrieval.

- Line 11 searches the catalog for a product identified by the ASIN `B000FRU0NU`. As you'll see later in this chapter, we can also perform open-ended searches using criteria such as product title and manufacturer.

- Lines 13-15 output the returned product's title, manufacturer, and product group. You can think of the product group as an organizational attribute, like a category. Amazon has many such product groups, among them `Books`, `Video Games`, and `Sporting Goods`.

Executing this code returns the following output:

```
Title: Halo 3
Publisher: Microsoft
Category: Video Games
```

Setting the Response Group

To maximize efficiency both in terms of bandwidth usage and parsing of the returned object, Amazon empowers you to specify the degree of product detail you'd like returned. When it comes to querying for general product information, typically you'll choose from one of three levels:

- *Small*: The `Small` group (set by default) contains only the most fundamental product attributes, including the ASIN, creator (author or manufacturer, for instance), manufacturer, product group (book, video game, or sporting goods, for instance), title, and Amazon.com product URL.

- *Medium*: The `Medium` group contains everything found in the `Small` group, in addition to attributes such as the product's current price, editorial review, current sales rank, the availability of this item in terms of the number of new, used, collectible, and refurbished units made available through Amazon.com, and links to the product images.
- *Large*: The `Large` group contains everything available to the `Medium` group, in addition to data such as a list of similar products, the names of tracks if the product group is a CD, a list of product accessories if relevant, and a list of available offers (useful if a product is commonly sold by multiple vendors via Amazon.com). Hopefully it goes without saying that if you're interested in retrieving just the product's fundamental attributes such as the title and price, you should be careful to choose the more streamlined `Medium` group, as the amount of data retrieved when using the `Large` group is significantly larger than that returned by the former.

If you're interested in retrieving only a specific set of attributes, such as the image URLs or customer reviews, then consider using one of the many available specialized response groups. Among these response groups include `Images`, `SalesRank`, `CustomerReviews`, and `EditorialReview`. As an example, if you'd like to regularly keep tabs of solely a product's latest Amazon sales rank, there's logically no need to retrieve anything more than the rank. To forego retrieving superfluous data, use the `SalesRank` response group:

```
$item = $amazon->itemLookup('B000FRU0NU', array('ResponseGroup' => 'SalesRank'));
echo "The latest sales rank is: {$item->SalesRank}";
```

Tip

TIP. Determining which attributes are available to the various response groups can be a tedious affair. To help sort out the details, consider downloading the documentation from http://aws.amazon.com/associates/.

Displaying Product Images

Adding an image to your product listings can greatly improve the visual appeal of your site. If your queries are configured to return a `Medium` or `Large` response group, URLs for three different image sizes (available via the `SmallImage`, `MediumImage`, and `LargeImage` objects) are included in the response. Unless you require something else only available within the `Large` response group, use the `Medium` group in order to save bandwidth, as demonstrated here:

```
$item = $amazon->itemLookup('B000FRU0NU', array('ResponseGroup' => 'Medium'));
echo $this->view->item->SmallImage->Url;
```

Executing this code returns the following URL:

```
http://ecx.images-amazon.com/images/I/41MnjYDVLqL._SL75_.jpg
```

If you want to include the image within a view, pass the URL into an
<img></img>
tag:

```
<img src="<?= $this->item->SmallImage->Url; ?>" />
```

You might be tempted to save some bandwidth by retrieving and storing these images locally. I suggest against doing so for two reasons. First and most importantly, caching the image is not allowed according to the Product Advertising API's terms of service. Second, as the above example indicates, the image filenames are created using a random string which will ensure the outdated images aren't cached and subsequently used within a browser or proxy server should a new image be made available by Amazon. The implication of the latter constraint is that the URLs shouldn't be cached either, since they're subject to change. Of course, rather than repeatedly contact the Amazon servers every time you want to display a URL, you should cache the image URLs, however should only do so for 24 hours do to their volatile nature. The easiest way to deal with this issue is to create a daily cron job which cycles through each item and updates the URL accordingly.

Putting it All Together

Believe it or not, by now you've learned enough to create a pretty informative product interface. Let's recreate the layout shown in Figure 10.1, which makes up part of the GameNomad website.

Figure 10.1. Assembling a video game profile

Let's start by creating the action, which will contact the web service and retrieve the desired game. Assume the URL is a custom route of the format `http://www.gamenomad.com/games/B000FRU0NU`. This code contains nothing you haven't already encountered:

```
public function showAction()
{

  // Retrieve the ASIN
  $asin = $this->_request->getParam('asin');

  // Query AWS
  $amazonPublicKey = Zend_Registry::get('config')
                     ->amazon->product_advertising->public->key;
  $amazonPrivateKey = Zend_Registry::get('config')
                     ->amazon->product_advertising->private->key;

  $amazonCountry = Zend_Registry::get('config')->amazon->country;

  $amazon =
   new Zend_Service_Amazon($amazonPublicKey, $amazonCountry, $amazonPrivateKey);

  $this->view->item =
    $amazon->itemLookup('B000FRU0NU', array('ResponseGroup' => 'Medium'));

}
```

Once the query has been returned, all that's left to do is populate the data into the view, as is demonstrated here:

```
<h1><?= $this->item->Title; ?></h1>
<img src="<?= $this->item->MediumImage->url; ?>" />
<b>Publisher</b>: <?= $this->item->Manufacturer; ?><br />
<b>Release Date</b>:
  <?= $this->ReleaseDate($this->item->ReleaseDate)); ?><br />
<b>Amazon.com Price</b>: <?= $this->item->FormattedPrice; ?><br />
<b>Latest Amazon.com Sales Rank</b>:
  <?= $this->SalesRank($this->item->SalesRank); ?><br />
```

Like the controller, we're really just connecting the dots regarding what's been learned here and in other chapters. Perhaps the only worthy note is that a few custom view helpers are used in order to format the publication date and sales rank. Within these view helpers native PHP functions such as `strtotime()`, `date()` and `number_format()` are used in order to convert the returned values into more desirable formats.

Of course, because GameNomad is a live website, the Product Advertising API isn't actually queried every time a video game's profile page is retrieved. Much of this data is cached locally, and

regularly updated in accordance with the terms of service. Nonetheless, the above example nicely demonstrates how to use the web service to pull this data together.

Searching for Products

All of the examples provided so far presume you have an ASIN handy. But manually navigating the Amazon.com website to find them is a tedious process. In fact, you might not even know the product's specific title, and instead just want to retrieve all products having a particular keyword in the title, or made by a particular manufacturer.

Searching for Products by Title

What if you wanted to find products according to a particular keyword found in the product title? To do so, you'll need to identify the product category, and then specify the keyword you'd like to use as the basis for searching within that category. The following example demonstrates how to search the `VideoGames` (note the lack of spaces) category for any product having the keyword `Halo` in its title:

```
$amazon =
  new Zend_Service_Amazon($amazonPublicKey, $amazonCountry, $amazonPrivateKey);

$items = $amazon->itemSearch(array('SearchIndex' => 'VideoGames',
  'ResponseGroup' => 'Medium', 'Keywords' => 'Halo'));

foreach($items as $item) {

  echo "{$item->Title}\n";

}
```

At the time of this writing (the Amazon.com catalog is of course subject to change at any time), executing this code produced the following output:

```
Halo Reach
Halo: Combat Evolved
Halo 3: ODST
Halo, Books 1-3 (The Flood; First Strike; The Fall of Reach)
Halo 3
Halo 2
Halo: Combat Evolved
Halo Wars: Platinum Hits
Halo Reach - Legendary Edition
Halo 2
```

It's worth pointing out that the ten products found in the listing aren't all video games, as the defined category might lead you to believe. For instance, the product `Halo, Books 1-3` refers to a box set

of official novels associated with the Halo video game series. Why these sorts of inconsistencies occur isn't apparent, although one would presume it has to do with making the product more easily findable on the Amazon.com website and through other outlets.

Incidentally, `VideoGames` is just one of more than 40 categories at your disposal. Try doing searches using categories such as `Music`, `DigitalMusic`, `Watches`, `SportingGoods`, `Photo`, and `OutdoorLiving` for some idea of what's available!

Executing a Blended Search

If you were creating a website dedicated to the Halo video game series, chances are you'd want to list much more than just the games! After all, there are Halo-specific books, soundtracks, toys, action figures, and even an animated series. But not all of these items are necessarily categorized within `VideoGames`, so how can you be sure to capture them all? Amazon offers a special "catch-all" category called `Blended` which will result in a search being conducted within all of the available categories:

```
$items = $amazon->itemSearch(array('SearchIndex' => 'Blended',
  'ResponseGroup' => 'Medium', 'Keywords' => 'Halo'));
```

Performing the search anew turns up almost 50 items with Halo in the title, the vast majority of which are clearly related to the popular video game brand.

Executing Zend Framework Applications From the Command Line

In order to calculate trends such as price fluctuations or sales popularity (via the Amazon.com sales rank), you'll need to regularly retrieve and record this information. You already learned how to use the Zend_Service_Amazon component to retrieve this information, but when doing the mass price and sales rank updates using a standard action won't do for two reasons. First, as your game database continues to grow, the time required to retrieve these values for each game will logically increase, meaning you run the risk of surpassing PHP's maximum execution time setting (defined by the `max_execution_time` directive). While you could certainly change this setting, the consequences of the script still managing to surpass this limit due to an unexpectedly slow network connection or other issue before all of the updates are complete are just too severe to contemplate.

The second reason to avoid performing this sort of update via a traditional action is because you certainly don't want somebody from the outside either accidentally or maliciously accessing this action. While you could password-protect the action, are you realistically going to take the time to

supply credentials each time you want to access the action in order to initiate the update? Certainly, forgetting the password isn't going to help, and it's only a matter of time before you stop doing the updates altogether.

One easy workaround involves writing a standalone script which is executed using PHP's command-line interface (CLI). This eliminates the issues surrounding the maximum execution time setting since this setting isn't taken into account when using the CLI. Additionally, provided proper file permissions are applied you won't run the risk of another user running the script. However, you'll need to deal with the hassle of finding and using a third-party Amazon API library, not to mention violate the DRY principle by maintaining a separate set of Amazon API and database access credentials. Or will you?

Believe it or not, it's possible to create a script which plugs directly into your Zend Framework application! This script can take advantage of your `application.ini` file, all of the Zend Framework's components, and any other resources you've made available to the application. This approach gives you the best of both worlds: a script which can securely execute on a rigorous basis (using a task scheduler such as cron) using the very same configuration data and other resources made available to your web application.

Just as is the case with the web-based portion of your application, you'll need to bootstrap the Zend Framework resources to the CLI script. You'll see that this script looks suspiciously like the front controller (`/public/index.php`. Create a new file named `cli.php` and place it within your `public` directory, adding the following contents:

```
<?php

defined('APPLICATION_PATH')
    || define('APPLICATION_PATH',
              realpath(dirname(__FILE__) . '/../application'));

// Define application environment
defined('APPLICATION_ENV')
    || define('APPLICATION_ENV',
              (getenv('APPLICATION_ENV') ? getenv('APPLICATION_ENV')
                                         : 'development'));

require_once 'Zend/Application.php';
$application = new Zend_Application(
    APPLICATION_ENV,
    APPLICATION_PATH . '/configs/application.ini'
);

$application->bootstrap();
```

As you can see, this script accomplishes many of the same goals set forth within the front controller, beginning with defining the application path and application environment. Next we'll instantiate the Zend_Application class, passing the environment and location of the `application.ini`. Finally, the `bootstrap()` method is call, which loads all of the application resources.

With the CLI-specific bootstrapper in place, you can go about creating scripts which use your application configuration files and other resources. For instance, I use the following script (`/scripts/update_prices.php`) to retrieve the latest prices for all of the video games found in the `games` table:

```
01 <?php
02
03 include "../public/cli.php";
04
05 // Retrieve the database connection handle
06 $db = $application->getBootstrap()->getResource('db');
07
08 // Retrieve the Amazon web service configuration data
09 $amazonPublicKey = Zend_Registry::get('config')
10                     ->amazon->product_advertising->public->key;
11 $amazonPrivateKey = Zend_Registry::get('config')
12                     ->amazon->product_advertising->private->key;
13 $amazonCountry = Zend_Registry::get('config')->amazon->country;
14
15 // Connect to the Amazon Web service
16 $amazon = new Zend_Service_Amazon($amazonPublicKey,
17             $amazonCountry, $amazonPrivateKey);
18
19 // Retrieve all of the games stored in the GameNomad database
20 $games = $db->fetchAll('SELECT id, asin, name FROM games ORDER BY id');
21
22 // Iterate over each game, updating its price
23 foreach ($games AS $game)
24 {
25
26   try {
27
28     $item = $amazon->itemLookup($game['asin'],
29               array('ResponseGroup' => 'Medium'));
30
31     if (! is_null($item)) {
32
33       if (isset($item->FormattedPrice))
34       {
35         $price = $item->FormattedPrice;
36       } else {
37         $price = '$0.00';
38       }
```

```
39
40       $update = $db->query("UPDATE games SET price = :price WHERE id = :id",
41                            array('price' => $price, 'id' => $game['id']));
42
43     } else {
44
45       $update = $db->query("UPDATE games SET price = :price WHERE id = :id",
46                            array('price' => '$0.00', 'id' => $game['id']));
47
48     }
49
50   } catch(Exception $e) {
51     echo "Could not find {$game['asin']} in Amazon database\r\n";
52   }
53
54 }
55
56 ?>
```

Let's review the code:

- Line 03 integrates the bootstrapper into the script, making all of the application resources available for use. Incidentally, I happen to place my CLI scripts in the project's root directory within a directory named `scripts`, thus the use of this particular relative path.

- Line 06 retrieves a handle to the database connection using this little known Zend Framework feature (which I incidentally introduced in Chapter 6). We'll use this handle throughout the script to execute queries against the GameNomad database.

- Lines 08-13 retrieve the Product Advertising API public and private keys, and the country setting.

- Line 16-17 instantiate the Zend_Service_Amazon class, passing in the aforementioned configuration data.

- Line 20 uses the Zend_Db `fetchAll()` method to retrieve a list of all games found in the `games` table.

- Lines 23-54 iterate over the list of retrieved games, using each ASIN to query the Amazon web service and retrieve the latest price. Because the Amazon product database is occasionally inconsistent, you need to carefully check the value before inserting it into the database, which explains why in this example I am both ensuring the video game still exists in the database and that the price is correctly set.

The ability to create CLI scripts which execute in this fashion is truly useful, negating the need to depend upon additional third-party libraries and redundantly manage configuration data. Be sure to check out the `scripts` directory in the GameNomad code download for several examples which are regularly executed in order to keep the GameNomad data current.

Integrating the Google Maps API

Although web-based mapping services such as MapQuest (http://www.mapquest.com/) have been around for years, it wasn't until Google's release of its namesake mapping API (Application Programming Interface) that we began the love affair with location-based websites. This API provides you with not only the ability to integrate Google Maps into your website, but also to build completely new services built around Google's mapping technology. Google Maps-driven websites such as http://www.walkjogrun.net/, http://www.housingmaps.com/, and http://www.everyblock.com/ all offer glimpses into what's possible using this API and a healthy dose of imagination.

Although the Zend Framework has long bundled a component named Zend_Gdata which provides access to several Google services, including YouTube, Google Spreadsheets, and Google Calendar, at the time of this writing a component capable of interacting with the Google Maps API was still not available. However, it's nonetheless possible to create powerful mapping solutions using the Zend Framework in conjunction with the Google Maps API and the jQuery JavaScript framework's Ajax functionality. In this section I'll show you how this is accomplished. If you're new to the Google Maps API take a moment to carefully read the primer which follows, otherwise feel free to skip ahead to the section "Passing Data to the Google Maps API".

Introducing the Google Maps API

The 2005 release of the Google Maps API signaled a significant turning point in the Web's evolution, with a powerful new breed of applications known as *location-based services* emerging soon thereafter. This freely available API, which gives developers access to Google's massive spatial database and an array of features which developers can use to display maps within a website, plot markers, perform route calculations, and perform other tasks which were previously unimaginable. While competing mapping solutions exist, notably the Bing Maps (http://www.bing.com/developers) and Yahoo! Maps (http://developer.yahoo.com/maps/) APIs, the Google Maps API seems to have struck a chord with developers and is at the time of this writing the de facto mapping solution within the various programming communities.

In May, 2010 Google announced a major update to the API, commonly referred to as V3. V3 represents a significant evolutionary leap forward for the project, notably due to the streamlined

syntax which makes map creation and manipulation even easier than was possible using previous releases. Additionally V3 introduces a number of powerful new features including the ability to integrate draggable directions and the popular Street View feature.

However, one of the most welcome features new to V3 is the elimination of the previously required domain-specific API key. Google had previously required developers to register for a key which was tied to a specific domain address. While the registration process only took a moment, managing multiple domain keys was somewhat of a hassle and so removal of this requirement was welcome news.

Creating Your First Map

V3 offers a vastly streamlined API syntax, allowing you to create and manipulate a map using a few short lines of code. Let's begin with a simple example which centers a map over the Columbus, Ohio region. This map is presented in Figure 10.2.

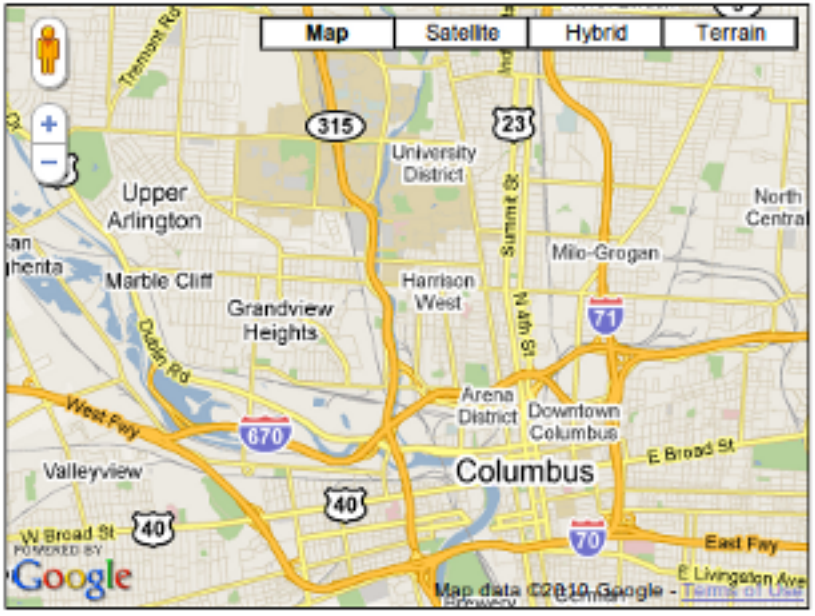


Figure 10.2. Centering a Google map over Columbus, Ohio

The code used to create this map is presented next:

```
01 <html>
02  <head>
03   <script type="text/javascript"
04    src="http://maps.google.com/maps/api/js?sensor=false">
05   </script>
06   <style type="text/css">
07     #map { border: 1px solid black; width: 400px; height: 300px; }
08   </style>
09   <script type="text/javascript">
10    function initialize() {
11     var latlng = new google.maps.LatLng(39.984577, -83.018692);
12     var options = {
13      zoom: 12,
14      center: latlng,
15      mapTypeId: google.maps.MapTypeId.ROADMAP
16     };
17     var map = new google.maps.Map(document.getElementById("map"), options);
18    }
19   </script>
20   </head>
21   <body onload="initialize()">
22     <div id="map"></div>
23   </body>
24 </html>
```

Let's review this example's key lines:

- Lines 03-05 incorporate the Google Maps API into the web page. The API is JavaScript-based, meaning you won't have to formally connect to the service like you did with the Amazon Product Advertising API. Instead, you just add a reference to the JavaScript file, and use the JavaScript methods and other syntax just as you would any other. Incidentally, the `sensor` parameter is used to tell Google whether a sensor is being used to derive the user's coordinates. Why Google requires this parameter is a mystery, since one would presume it could simply default to `false`. Nonetheless be sure to include it as Google explicitly states it to be a requirement in the documentation.

- Line 07 defines a simple CSS style for the DIV which will hold the map contents. The dimensions defined here will determine the size of the map viewport. If you were to omit dimensions, the map will consume the entire browser viewport.

- Lines 10-18 define a function named `initialize()` which will execute once the page has completely loaded (as specified by the `onload()` function call on line 21). You want to make sure the page has completely loaded before attempting to render a map, because as you'll soon see

the API requires a target DIV in order to render the map. It's possible that the JavaScript could execute before the DIV has been loaded into the browser, causing an error to occur. Keep this in mind when creating your own maps, as this oversight is a common cause for confusion!

- Line 11 creates a new object of type `LatLng`, which represents a pair of coordinates. In this example I'm passing in a set of coordinates which I know are situated atop the city of Columbus. In a later section I'll show you how to derive these coordinates given an address.

- Lines 12-16 define an *object literal* which contains several map-specific settings such as the zoom level, center point (defined by the previously created `LatLng` object), and the map type, which can be set to `ROADMAP`, `SATELLITE`, `HYBRID`, or `TERRAIN`. The `ROADMAP` type is the default setting (the same used when you go to http://maps.google.com). Try experimenting with each to get a feel for the unique environment each has to offer. Other settings exist, however the three used in this example are enough to create a basic map.

- Line 17 is responsible for creating the map based on the provided options, and inserting the map contents into the DIV identified by the `map` ID. You're free to name the DIV anything you please, however make sure the name matches that passed to the JavaScript `getElementById()` method call.

- Finally, line 22 defines the DIV where the map will be rendered. This is obviously a simple example; you can insert the DIV anywhere you please within the surrounding page contents. In fact, it's even possible to render multiple maps on the same page using multiple instances of the `Map` object.

Plotting Markers

The map presented in the previous example is interesting, however it provides the user little more than a bird's eye view of the city. Staying with the video gaming theme of the book, let's plot a few markers representing the locations of my favorite GameStop (http://www.gamestop.com) outlets, as depicted in Figure 10.3.

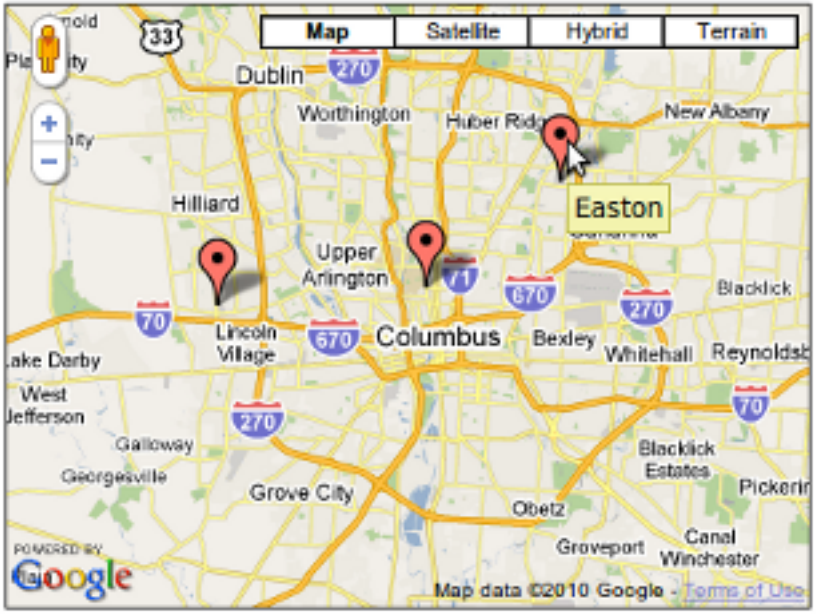


Figure 10.3. Plotting area GameStop locations

In order to stay on topic I'll presume we've already obtained the coordinates for each of the three locations placed on the map. In the next example I'll show you how to retrieve these coordinates so in the meantime let's focus specifically on the syntax used to plot the markers. For the sake of space I'll demonstrate plotting a single marker, however except for varying coordinates, marker titles, and variable names the syntax is identical:

```
...
var map = new google.maps.Map(document.getElementById("map"), options);

var campus = new google.maps.Marker({
  position: new google.maps.LatLng(39.9952654, -83.0071351),
  map: map,
  title: "GameStop Campus"
});
```

Summarizing this snippet, to plot a marker you'll create a new object of type `Marker` and pass into it an object literal consisting of the position, the map object, and the marker title (which displays when the user mouses over the marker).

Using the Geocoder

All of the examples provided thus far are based on the unlikely presumption that you already know the location coordinates. Because this is almost certainly never going to be the case, you'll need a solution for converting, or *geocoding* the location address to its corresponding latitudinal and longitudinal pair. The API is bundled with a geocoder which quite capably handles this task.

The API geocoder is bundled into a class named `geocoder`, and you'll invoke its `geocoder()` method to convert an address into its constituent coordinates, with the results passed to an anonymous function as demonstrated here:

```
01 ...
02 map = new google.maps.Map(document.getElementById("map"), options);
03
04 var address = "1611 N High St, Columbus Ohio";
05 var title = "Campus";
06
07 geocoder.geocode( {'address': address}, function(results, status) {
08
09   if (status == google.maps.GeocoderStatus.OK) {
10
11     var marker = new google.maps.Marker({
12       position: results[0].geometry.location,
13       map: map,
14       title: title
15     });
16
17   } else {
18     return FALSE;
19   }
20
21 });
22 ...
```

If you're not familiar with JavaScript's anonymous function syntax, this snippet can look a bit confusing. However if you carefully review this code you'll see that all we're doing is passing in a nameless function and body along as the `geocode()` method's second input parameter. This anonymous function accepts two parameters, `results`, which contains the geocoded coordinates if the attempt was successful, and `status`, which is useful for determining whether the attempt was successful. If successful, as defined by the `status` value `google.maps.GeocoderStatus.OK`, then the `results` object can be retrieve the coordinates `results[0].geometry.location` is a `LatLng` object containing the coordinates.

Of course, you shouldn't be repeatedly geocoding an address and plotting its coordinates. Instead, you should geocode the address once and save the coordinates to the database. I'll show you how this is done next.

Saving Geocoded Addresses

In my opinion, one of GameNomad's most interesting features is the ability to connect registered users who reside within the same geographical region. This is possible because coordinates corresponding to every user's zip code are associated with the user, and an algorithm is employed which determines which other users reside within a specified radius from the user's home zip code. These coordinates are stored in the `accounts` table's `latitude` and `longitude` columns, each of which is defined using the `double(10,6)` data type. The geocoding occurs within two areas of the GameNomad website, namely at the time of registration `/account/register`), and when the user updates his account profile `/account/profile`).

The php-google-map-api library (http://code.google.com/p/php-google-map-api/) provides a particularly easy way to convert addresses (including zip codes) into their corresponding coordinates. The php-google-map-api library offers an object-oriented server-side solution for integrating Google Maps into your website, allowing you to create and integrate maps using PHP rather than JavaScript. Although the php-google-map-api library is a very capable solution, I prefer to use the native JavaScript-based API however the php-google-map-api's geocoding feature is too convenient to ignore, allowing you to pass in a zip code and retrieve the geocoded coordinates in return, as demonstrated here:

```
$map = new GoogleMapAPI();
$coordinates = $map->getGeoCode($this->_request->getPost('zip_code'));

$latitude = $coordinates['lat'];
$longitude = $coordinates['lon'];
```

The php-google-map-api library is available for download from the aforementioned website, and consists of just two PHP files, `GoogleMap.php` and `JSMin.php`. The former file contains the `GoogleMapAPI` class which encapsulates the PHP-based interface to the Google Maps API. The latter file contains a PHP implementation of Douglas Crockford's JavaScript minifier (http://www.crockford.com/javascript/jsmin.html). If you're planning on using the library for more than geocoding then I suggest also downloading `JSMin.php` as it will boost performance by compressing the JavaScript generated by `GoogleMap.php`. Moving forward I'll presume you've only downloaded `GoogleMap.php` for the purposes of this exercise.

Place `GoogleMap.php` within your project's `library` directory or any other directory made available via PHP's `include_path` directive. Next you'll use the `require_once` statement to include the file at the top of any controller which will use the geocoding feature:

```
require_once 'GoogleMap.php';
```

All that's left to do is invoke the `GoogleMapAPI` class and call the `getGeoCode()` method to convert an address to its associated coordinates:

```
$map = new GoogleMapAPI();
$coordinates = $map->getGeoCode('43201');

$latitude = $coordinates['lat'];
$longitude = $coordinates['lon'];

printf("Latitude is %f and longitude is %f", $latitude, $longitude);
```

Executing this snippet produces the following output:

```
Latitude is 39.994879 and longitude is -82.998741
```

One great aspect of Google's geocoding feature is its ability to geocode addresses of varying degrees of specificity. It can also geocode state names (Ohio), cities and states (Columbus, Ohio), specific street names within an city (High Street, Columbus, Ohio), and specific street addresses (1611 N High Street, Columbus, Ohio 43201), among other address variations.

Finding Users within a Specified Radius

Because every user's zip code coordinates are stored in the database, it's possible to create all sorts of interesting location-based features, such as giving users the ability to review a list of all video games for sale within a certain radius (5, 10, or 15 miles away from the user's location as defined by his coordinates, for instance). Believe it or not, implementing such a feature is pretty easy, accomplished by implementing a SQL-based version of the *Haversine formula*. Although staring at the formula for too long may bring about unpleasant memories of high school geometry, the only real implementational challenge is knowing the insertion order of the variables passed into the formula.

The rather long query presented below is a slightly simplified version of the SQL implementation of the Haversine formula used on the GameNomad website. I won't pretend that I even really understand the mathematics behind the formula (nor care to understand it, for that matter), other than to say that it employs spherical trigonometry to calculate the distance between two points on the globe (or in the case of the SQL query, the distance between a user's location and all of the other users in the system).

Speaking specifically about what this query will retrieve, all games having a status of `$status` and associated with users residing within `$distance` miles of the location identified by the coordinates `$latitude` and `$longitude`

```
SELECT a.zip_code, a.latitude, a.longitude, count(g.id) as game_count,
  ( 3959 * acos( cos( radians($this->latitude) )
  * cos( radians( a.latitude ) ) *
  cos( radians( a.longitude ) - radians($longitude) ) +
  sin( radians($latitude) ) *
  sin( radians( a.latitude ) ) ) ) AS distance
  FROM accounts a
  LEFT JOIN games_to_accounts ga
  ON a.id = ga.account_id
  LEFT JOIN games g ON ga.game_id = g.id
  WHERE ga.status_id = $status
  GROUP BY a.zip_code HAVING distance < $distance
  ORDER BY distance
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Why is it a wise idea to use PHP's CLI in conjunction with scripts which could conceivably run for a significant period of time?
- What two significant improvements does the Google Maps API offer over its predecessor?

Chapter 11. Unit Testing Your Project

There are few tasks more exhausting than manually testing a website, haphazardly navigating from one link to the next and repeatedly entering countless permutations of valid and invalid information into web forms. Even thinking about these tasks is enough to wear me out. Yet leaving to chance a potentially broken user registration or worse, product purchase form is a recipe for disaster.

And so goes on the clicking, the navigating, the form filling, the checking, the double checking, the fixing, the triple checking, ad infinitum. Doesn't it seem ironic that programmers find themselves mired in such a tedious and error-prone process? Thankfully, members of our particular community tend to have little patience for inefficiency and often set out to improve inefficient processes through automation.

In fact, a great deal of work has been put into automating the software testing process, and in fact there are dozens of fantastic open source tools at your disposal which will not only dramatically reduce the time and effort you'll otherwise spend laboriously surfing your website, but also considerably reduce the amount of worry and stress you incur due to wondering whether you overlooked something!

In this chapter I'll introduce you to a popular PHP testing tool called PHPUnit (http://www.phpunit.de/) which integrates very well with the Zend Framework via the Zend_Test component. Several of the preceding chapters concluded with sections titled "Testing Your Work" which included several PHPUnit/Zend_Test-based tests intended to show you how to test your pages, page elements, forms, Doctrine entities, and other crucial components. So rather than repetitively focus on how to carry out these sorts of tests, I'll instead focus on configuration-related matters, showing you how to put all of the pieces in place in order to begin taking advantage of the tests presented in the earlier chapters.

Introducing Unit Testing

Unit testing is a software testing strategy which involves verifying that a specific portion, or *unit* of code, is working as expected. For instance, , you might want to write unit tests which answer any number of questions, including:

- Does the contact form properly validate user input?

- Is valid user registration data properly saved to the database?
- Are the finders defined in my custom entity repository retrieving the desired data?
- Does a particular page element exist?

Recognizing the importance of providing with an efficient way to integrate unit testing into the web development process, the Zend Framework developers added a component called Zend_Test early on in the project's history. Zend_Test integrates with the popular PHPUnit (http://www.phpunit.de) unit testing framework, providing an effective and convenient solution for testing your Zend Framework applications. In this section I'll show you how to install PHPUnit, and configure your Zend Framework application so you can begin writing and executing unit tests which validate the proper functioning of your website.

Readying Your Website for Unit Testing

The Zend Framework developers place a great emphasis on encouraging unit testing, going so far as to automatically create a special directory named `tests` within the project directory which is intended to house for your testing environment, and even generating test skeletons for each newly created controller. Yet a few configuration steps remain before you can begin writing and executing your unit tests. Thankfully, these steps are fairly straightforward, and in this section we'll work through each in order to configure a proper testing environment.

Installing PHPUnit

PHPUnit is available as a PEAR package, requiring you to only tell your PEAR package manager where the PHPUnit package resides by discovering its various PEAR channels, and then installing the package:

```
%>pear channel-discover pear.phpunit.de
%>pear channel-discover components.ez.no
%>pear channel-discover pear.symfony-project.com
%>pear install phpunit/PHPUnit
```

Once installed, you're ready to begin using PHPUnit! Confirm it's properly installed by opening a terminal window and viewing PHPUnit's version information:

```
%>phpunit --version
PHPUnit 3.5.3 by Sebastian Bergmann.
```

Next we'll configure your Zend Framework application so it can begin using PHPUnit for testing purposes.

Configuring PHPUnit

To begin, create a configuration file named `phpunit.xml` which serves as PHPUnit's configuration file, and place it in your project's `tests` directory. An empty `phpunit.xml` file already exists in this directory, so all you need to do is add the necessary configuration directives. A very simple (but operational) `phpunit.xml` files is presented here, followed by an overview of the key lines:

```
01 <phpunit bootstrap="./application/bootstrap.php" colors="true">
02   <testsuite name="gamenomad">
03     <directory>./</directory>
04   </testsuite>
05 </phpunit>
```

Let's review the file:

- Line 01 points PHPUnit to a bootstrap file, which will execute before any tests are run. I'll talk more about `tests/application/bootstrap.php` in a moment. Setting the `colors` attribute to `true` will cause PHPUnit to use color-based cues to indicate whether the tests had passed, with green indicating success and red indicating failure.
- Lines 02-04 tells PHPUnit to recursively scan the current directory, finding files ending in `Test.php`.

Next, we'll create the bootstrap file referenced on line 01 of the `phpunit.xml` file.

Creating the Test Bootstrap

The test bootstrap file (`tests/application/bootstrap.php`) referenced on line 01 of the `phpunit.xml` file is responsible for initializing any resources which will subsequently be used when running the tests. In the following example bootstrap file we configure a path-related constant (`APPLICATION_PATH`), and load two helper classes (`ControllerTestCase.php` and `ModelTestCase.php`) which we'll use to streamline some of the code used in controller- and model-related tests, respectively (I'll talk more about these helper classes in a moment). Like the `phpunit.xml`, a blank `bootstrap.php` file was created when your project was generated, so you'll just need to add the necessary code:

```
<?php
  define('BASE_PATH', realpath(dirname(__FILE__) . '/../../'));

  define('APPLICATION_PATH', BASE_PATH . '/application');

  require_once 'controllers/ControllerTestCase.php';
  require_once 'models/ModelTestCase.php';
```

Testing Your Controllers

When you use the ZF CLI to generate a new controller, an empty test case will automatically be created and placed in the `tests/application/controllers` directory. For instance if you create a new controller named `About`, notice how the output also indicates that a controller test file named `AboutControllerTest.php` has been created and added to the directory `tests/application/controllers/`:

```
%>zf create controller About
Creating a controller at
/var/www/dev.gamenomad.com/application/controllers/...
Creating an index action method in controller About
Creating a view script for the index action method at
/var/www/dev.gamenomad.com/application.../about/index.phtml
Creating a controller test file at
/var/www/dev.gamenomad.com/tests/...AboutControllerTest.php
Updating project profile '/var/.../.zfproject.xml'
```

Let's take a look at the `AboutControllerTest.php` code:

```
<?php

  require_once 'PHPUnit/Framework/TestCase.php';

  class AboutControllerTest extends PHPUnit_Framework_TestCase
  {

    public function setUp()
    {
        /* Setup Routine */
    }

    public function tearDown()
    {
        /* Tear Down Routine */
    }

  }
```

Each generated controller test is organized within a class which extends the Zend Framework's `TestCase` class. Within the class, you'll find two empty methods named `setUp()` and `tearDown()`. These methods are special to PHPUnit in that PHPUnit will execute the `setUp()` method prior to executing any tests found in the class (I'll talk more about this in a moment), and will execute the `tearDown()` method following completion of the tests. You'll use `setUp()` to set the application

environment up so that the tests will use to test the code, and `tearDown()` to return the environment back to its original state.

When testing Zend Framework-driven applications, the primary purpose of the `setUp()` method is to bootstrap your application environment so the tests can interact with the application code, it's a good idea to DRY up the code and create a parent test case class which readies the environment for you. You'll modify the generated test controller classes to extend *this* class, which will in turn subclass `Zend_Test_PHPUnit_ControllerTestCase`. Here's what a basic parent test controller class looks like, which I call `ControllerTestCase.php` (this file should be placed in the `tests/application/controllers` directory):

```
<?php
  require_once 'Zend/Application.php';
  require_once 'Zend/Test/PHPUnit/ControllerTestCase.php';

  abstract class ControllerTestCase
    extends Zend_Test_PHPUnit_ControllerTestCase
  {

    public function setUp()
    {

      $this->bootstrap = new Zend_Application(
        'testing',
        APPLICATION_PATH . '/configs/application.ini'
      );

      parent::setUp();

    }

    public function tearDown()
    {
      parent::tearDown();
    }

  }
```

Because we're keeping matters simple, this helper class' `setUp()` method is only responsible for creating a `Zend_Application` instance, setting `APPLICATION_ENV` to `testing` and identifying the location of the `application.ini` file, and concludes by executing the parent class' `setUp()` method. The `tearDown()` method just calls the parent class' `tearDown()` method.

Save this file as `ControllerTestCase.php` to your `/tests/application/controllers/` directory, and modify the `AboutControllerTest.php` file so it extends this class. Also, add a simple test so we can make sure everything is working properly:

```
<?php

  class AboutControllerTest extends ControllerTestCase
  {
    public function testDoesAboutIndexPageExist()
    {
        $this->dispatch('/');
        $this->assertController('about');
        $this->assertAction('index');

    }
  }
```

Save `AboutControllerTest.php`, open a terminal window, and execute the following command from within your project's `tests` directory:

```
$ phpunit
PHPUnit 3.5.3 by Sebastian Bergmann.

..

Time: 0 seconds, Memory: 8.75Mb

OK (1 tests, 2 assertions)
```

Presuming you see the same output as that shown above, congratulations you've successfully integrated PHPUnit into your Zend Framework application!

Executing a Single Controller Test Suite

Sometimes you'll want to focus on testing a specific controller and would rather not wait for all of your tests to execute. To test just one controller, pass the controller path and test file name to `phpunit`, as demonstrated here:

```
%>phpunit application/controllers/AccountControllerTest
```

Testing Your Models

While you'll want to test your models by way of your controller actions, it's also a good idea to test your models in isolation. The configuration process really isn't much different from that used to test

the controllers, the only real difference being that in order to test the Doctrine entities we need to obtain access to the `entityManager` resource. Create a helper class named `ModelTestCase.php` and place it in the `tests/application/models` directory:

```
01 <?php
02
03 class ModelTestCase extends PHPUnit_Framework_TestCase
04 {
05
06   protected $em;
07
08   public function setUp()
09   {
10
11     $application = new Zend_Application(
12       'testing',
13       APPLICATION_PATH . '/configs/application.ini'
14     );
15
16     $bootstrap = $application->bootstrap()->getBootstrap();
17
18     $this->em = $bootstrap->getResource('entityManager');
19
20     parent::setUp();
21
22   }
23
24   public function tearDown()
25   {
26     parent::tearDown();
27   }
28
29 }
```

Let's review the code:

- Line 06 defines a protected attribute named `$em` which will store an instance of the entity manager (see Chapter 7 for more about the role of Doctrine's entity manager). This attribute will be used within the tests to interact with the Doctrine entities.

- Lines 11-13 creates a `Zend_Application` instance, setting `APPLICATION_ENV` to `testing` and identifying the location of the `application.ini` file.

- Line 16 retrieves an instance of the application bootstrap, which is in turn used to access the entity manager resource (Line 18).

Save this file as `ModelTestCase.php` to your `/tests/application/models/` directory, and create a test class named `AccountEntityTest.php`, remembering to extend it with the `ModelTestCase` class. Finally, add a simple test so we can make sure everything is working properly:

```
<?php

  class AboutEntityTest extends ModelTestCase
  {

    public function testCanSaveAndRetrieveUser()
    {

      $account = new \Entities\Account;
      $account->setUsername('wjgilmore-test');
      $account->setEmail('example@wjgilmore.com');
      $account->setPassword('jason');
      $account->setZip('43201');
      $account->setConfirmed(1);
      $this->em->persist($account);
      $this->em->flush();

      $account = $this->em->getRepository('Entities\Account')
                      ->findOneByUsername('wjgilmore-test');

      $this->assertEquals('wjgilmore-test',
                           $account->getUsername());

    }

  }
```

Creating Test Reports

Viewing the `phpunit` command's terminal-based output is certainly useful, however quite a few other convenient and useful test reporting methods are also available. One of the simplest involves creating an HTML-based report (see Figure 11.1) which clearly displays passing and failing tests. The report is organized according to the test class, with failing tests clearly crossed out.

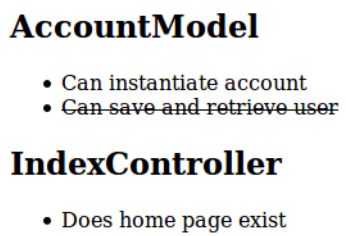


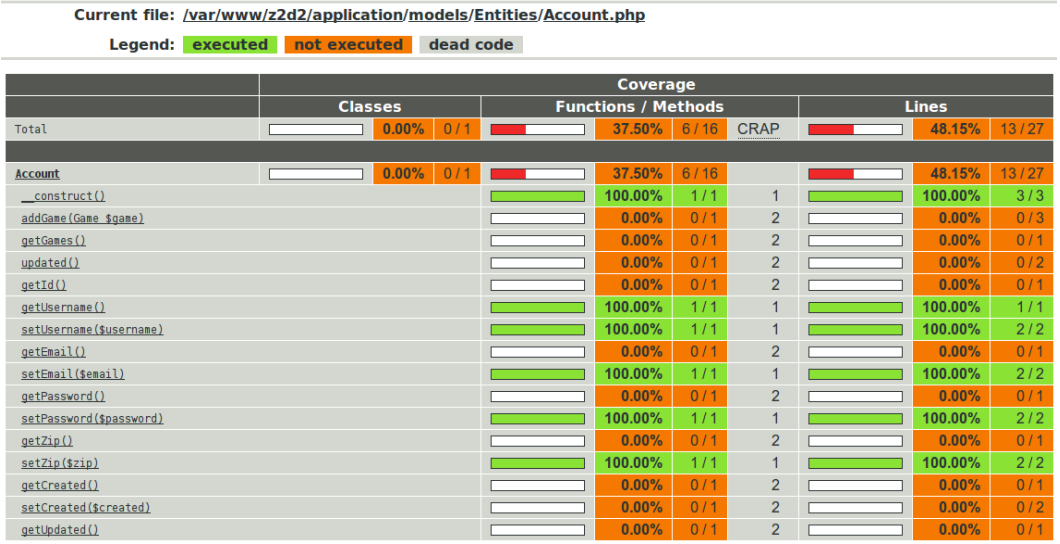
Figure 11.1. Viewing a web-based test report

To enable web-based test reports, modify your `phpunit.xml` file, adding the `logging` element as presented below. You'll also need to create the directory where you would like to store the HTML report (in the case of the below example you would create the directory `tests/log/`):

```
<programlisting>
<![CDATA[
<phpunit bootstrap="./application/bootstrap.php" colors="true">
  <testsuite name="gamenomad">
    <directory>./</directory>
  </testsuite>
  <logging>
    <log type="testdox-html"
         target="/var/www/dev.gamenomad.com/tests/log/testdox.html" />
  </logging>
</phpunit>
</programlisting>
```

Code Coverage

PHPUnit can also generate sophisticated *code coverage* reports which can prove extremely useful for determining just how much of your project has been sufficiently tested. These web-based reports allow you to drill down into your project files and visually determine specifically which lines, methods, actions, and attributes have been "touched" by a test. For instance Figure 11.2 presents an example code coverage report for an Doctrine entity named `Account`.



Current file: /var/www/z2d2/application/models/Entities/Account.php

Legend: executed not executed dead code

	Coverage								
	Classes			Functions / Methods			Lines		
Total	0.00%	0 / 1		37.50%	6 / 16	CRAP	48.15%	13 / 27	
Account	0.00%	0 / 1		37.50%	6 / 16		48.15%	13 / 27	
__construct()				100.00%	1 / 1	1	100.00%	3 / 3	
addGame(Game $game)				0.00%	0 / 1	2	0.00%	0 / 3	
getGames()				0.00%	0 / 1	2	0.00%	0 / 1	
updated()				0.00%	0 / 1	2	0.00%	0 / 2	
getId()				0.00%	0 / 1	2	0.00%	0 / 1	
getUsername()				100.00%	1 / 1	1	100.00%	1 / 1	
setUsername($username)				100.00%	1 / 1	1	100.00%	2 / 2	
getEmail()				0.00%	0 / 1	2	0.00%	0 / 1	
setEmail($email)				100.00%	1 / 1	1	100.00%	2 / 2	
getPassword()				0.00%	0 / 1	2	0.00%	0 / 1	
setPassword($password)				100.00%	1 / 1	1	100.00%	2 / 2	
getZip()				0.00%	0 / 1	2	0.00%	0 / 1	
setZip($zip)				100.00%	1 / 1	1	100.00%	2 / 2	
getCreated()				0.00%	0 / 1	2	0.00%	0 / 1	
setCreated($created)				0.00%	0 / 1	2	0.00%	0 / 2	
getUpdated()				0.00%	0 / 1	2	0.00%	0 / 1	

Figure 11.2. A Doctrine entity code coverage report

In order to enable PHPUnit's code coverage generation feature you'll need to install the Xdebug extension (http://www.xdebug.org/). Installing Xdebug is a very easy process, done by following the instructions presented here: http://www.xdebug.org/docs/install.

With Xdebug installed, you'll next need to define a `log` element of type `coverage-html` to your project's `phpunit.xml` file, as demonstrated in the below example. You'll also need to create the directory where you would like to store the code coverage reports (in the case of the below example you would create the directory `tests/log/report/`):

```
<phpunit bootstrap="./application/bootstrap.php" colors="true">

  <testsuite name="gamenomad">
    <directory>./</directory>
  </testsuite>

  <filter>
    <whitelist>
      <directory suffix=".php">../application/</directory>
      <exclude>
       <file>../application/bootstrap.php</file>
      </exclude>
```

```
    </whitelist>
  </filter>

  <logging>
    <log type="testdox-html"
         target="tests/log/testdox.html" />
    <log type="coverage-html"
         target="tests/log/report" charset="UTF-8"
         yui="true" highlight="true"
         lowUpperBound="50"
         highLowerBound="80"/>
  </logging>
</phpunit>
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Define *unit testing* and talk about the advantages it brings to your development process.
- What Zend component and open source PHP project work in conjunction with one another to bring unit testing to your Zend Framework applications?
- What is the name of the syntactical construct used to validate expected outcomes when doing unit testing?
- Why are code coverage reports useful?

Chapter 12. Deploying Your Website with Capistrano

I seriously doubt there is a developer on the planet who has not experienced at least one moment of sheer panic due to at least one botched website deployment in their career. Those who admit to having suffered through troubleshooting a sudden and mysterious issue with their production website following an update almost always attribute the problem to FTP. Perhaps a configuration file was mistakenly overwritten, you neglected to transfer all of the modified files, or you forgot to login to the production server following completion of the file transfer and adjust file permissions.

You might be surprised to learn that under no circumstances should you be using FTP to update anything but the simplest website. Unfortunately, FTP offers the illusion of efficiency, because it provides such an incredibly intuitive interface for transferring files from your laptop to your remote server. However, FTP is slow, transferring all selected files rather than only those which have been modified since the last update. It is also stupid, capable of only transferring files without any consideration of platform-specific settings (such as the `APPLICATION_ENV` setting in your Zend Framework website's `.htaccess` file). And perhaps worst of all, FTP offers no undo option; once the files have transferred, it is not possible to revert those changes should you want to return the production website to its previous state.

Fortunately, an alternative deployment solution exists called Capistrano (https://github.com/capistrano/) which resolves all of FTP's issues quite nicely. Not only can you use Capistrano to securely and efficiently deploy changes to your Zend Framework website, but it's also possible to rollback your changes should a problem arise. In this chapter I'll show you how to configure your development environment to use Capistrano, freeing you from ever having to worry again about a botched website deployment.

Configuring Your Environment

Capistrano is an open source automation tool originally written for and primarily used by the Rails community (http://rubyonrails.org/). However it is perfectly suitable for use with other languages, PHP included. But because Capistrano is written in Ruby, you'll need to install Ruby on your development machine. If you're running OS X or most versions of Linux, then Ruby is likely already installed. If you're running Windows, the easiest way to install Ruby is via the Ruby Installer for Windows (http://rubyinstaller.org/).

Once installed, you'll use the RubyGems package manager to install Capistrano and another application called Railsless Deploy which will hide many of the Rails-specific features otherwise bundled into Capistrano. Although Railsless Deploy is not strictly necessary, installing it will dramatically streamline the number of Capistrano menu options otherwise presented, all of which would be useless to you anyway because they are intended for use in conjunction with Rails projects.

RubyGems is bundled with Ruby, meaning if you've installed Ruby then RubyGems is also available. Open up a terminal window and execute the following command to install Capistrano:

```
$ gem install capistrano
```

Next, install Railsless Deploy using the following command:

```
$ gem install railsless-deploy
```

Once installed you should be able to display a list of available Capistrano commands:

```
$ cap -T
cap deploy               # Deploys your project.
cap deploy:check         # Test deployment dependencies.
cap deploy:cleanup       # Clean up old releases.
cap deploy:cold          # Deploys and starts a `cold' application.
cap deploy:pending       # Displays the commits since your last...
cap deploy:pending:diff  # Displays the `diff' since your last...
cap deploy:rollback      # Rolls back to a previous version and...
cap deploy:rollback:code # Rolls back to the previously deployed...
cap deploy:setup         # Prepares one or more servers for depl...
cap deploy:symlink       # Updates the symlink to the most recen...
cap deploy:update        # Copies your project and updates the s...
cap deploy:update_code   # Copies your project to the remote ser...
cap deploy:upload        # Copy files to the currently deployed...
cap invoke               # Invoke a single command on the remote...
cap shell                # Begin an interactive Capistrano sessi...
```

Installing a Version Control Solution

Version control is a process so central to successful software development that if you are not currently using a version control solution such as Git (http://git-scm.com/) or Subversion (http://subversion.tigris.org/) to manage your projects then consider reading the freely available chapter 1 of the book "Pro Git" (http://progit.org/book/). In short, version control brings a great many advantages to any project, including the disciplined evolution of your project's source code, changelog tracking and publication, the ability to easily revert mistakes, and experiment with new features, to name a few.

Further, maintaining a project under version control solution will greatly streamlines the deployment process because Capistrano will be able to detect and transfer only those files which have changed since the last deployment. While it's not strictly necessary for your project to be under version control in order for Capistrano to perform the deployment, I urge you to do so because like Capistrano, instituting version control will greatly reduce the possibility of unforeseen gaffes thanks to the ability to rigorously track and traverse changes to your code.

Capistrano supports quite a few different version control solutions, among them Bazaar, CVS, Git, Mercurial, and Subversion. If you haven't already settled upon a solution, I'd like to suggest Git (http://git-scm.com/), not only because it happens to be the solution I know best and can therefore answer any questions you happen to have, but also because as of the moment it is easily the most popular version control solution on the planet. Additionally, Git clients are available for all of the major platforms, Windows included. Install Git on your development machine by heading over to http://git-scm.com/download and following the instructions specific to your operating system.

Once installed you can confirm that the command-line client is working properly by executing the following command:

```
$ git --version
git version 1.6.3.3
```

Because Git associates repository changes with the user performing the commit, a useful feature when working with multiple team members, you'll need to identify yourself and your e-mail address so Git can attribute your commits accordingly:

```
$ git config --global user.name "Jason Gilmore"
$ git config --global user.email "wj@wjgilmore.com"
```

You'll only need to do this once, as Git will save this information in a configuration file associated with your system account.

To place your project under version control, enter your project's root directory and execute the following command:

```
$ git init
Initialized empty Git repository in /var/www/dev.gamenomad.com/.git/
```

Executing this command will in no way modify your project nor its files, other than to create a directory called `.git` which will host your repository changes.

Presuming your project directory contains various files and directories, you'll next want to begin tracking these files using Git. To do so, execute the `add` command:

```
$ git add .
```

The period tells Git's `add` command to recursively add everything found in the directory. You can confirm which files will be tracked by executing the `status` command. For instance, if you're tracking a project which was just created using the zf command-line tool, the `status` command will produce the following output:

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
# new file:   .zfproject.xml
# new file:   application/Bootstrap.php
# new file:   application/configs/application.ini
# new file:   application/controllers/ErrorController.php
# new file:   application/controllers/IndexController.php
# new file:   application/views/scripts/error/error.phtml
# new file:   application/views/scripts/index/index.phtml
# new file:   docs/README.txt
# new file:   public/.htaccess
# new file:   public/index.php
# new file:   tests/application/bootstrap.php
# new file:   tests/library/bootstrap.php
# new file:   tests/phpunit.xml
#
```

Finally, you'll want to *commit* these changes. Do so using the `commit` command:

```
$ git commit -m "First project commit"
[master (root-commit) 5be0656] First project commit
 10 files changed, 303 insertions(+), 0 deletions(-)
 create mode 100644 .zfproject.xml
 create mode 100644 application/Bootstrap.php
 create mode 100644 application/configs/application.ini
 create mode 100644 application/controllers/ErrorController.php
 create mode 100644 application/controllers/IndexController.php
 create mode 100644 application/views/scripts/error/error.phtml
 create mode 100644 application/views/scripts/index/index.phtml
 create mode 100644 docs/README.txt
 create mode 100644 public/.htaccess
 create mode 100644 public/index.php
 create mode 100644 tests/application/bootstrap.php
 create mode 100644 tests/library/bootstrap.php
```

```
 create mode 100644 tests/phpunit.xml
```

The `-m` option refers to the *commit message* which you'll attach to the commit by passing it enclosed in quotations as demonstrated here. Of course, you'll want these messages to clearly explain the changes you're committing to the repository, not only for the benefit of others but for yourself when you later review the commit log, which you can do by executing the `log` command:

```
$ git log
commit 5be06569a9d69214a629e888187e59023985f122
Author: Jason Gilmore <wj@wjgilmore.com>
Date:   Wed Feb 23 18:37:48 2011 -0500

  First project commit
```

Because I'll show you how to use Capistrano to only deploy the changes committed to your repository since the last deployment, you'll want to make sure you've committed your changes before beginning the deployment process, otherwise you'll be left wondering why the production server doesn't reflect your latest changes! This applies even if you aren't using Git, as the behavior is the same regardless of which version control solution you are using.

This short introduction to Git doesn't even begin to serve as an adequate tutorial, as Git is a vastly capable version control solution with hundreds of useful features. Although what you've learned so far will be suffice to follow along with the rest of the chapter, I suggest purchasing a copy of Scott Chacon's excellent book, "Pro Git", published by Apress in 2009. You can read the book online at http://progit.org/.

Configuring Public-key Authentication

The final general configuration step you'll need to take is configuring key-based authentication. Key-based authentication allows a client to securely connect to a remote server without requiring the client to provide a password, by instead relying on *public-key authentication* to verify the client's identity.

Public-key cryptography works by generating a pair of keys, one public and another private, and then transferring a copy of the *public* key to the remote server. When the client attempts to connect to the remote server, the server will *challenge* the client by asking the client to generate a unique *signature* using the private key. This signature can only be verified by the *public key*, meaning the server can use this technique to verify that the client is allowed to connect. As you might imagine, some fairly heady mathematics are involved in this process, and I'm not even going to attempt an explanation; the bottom line is that configuring public-key authentication is quite useful because it

means you don't have to be bothered with supplying a password every time you want to SSH into a remote server.

Configuring public-key authentication is also important when setting up Capistrano to automate the deployment process, because otherwise you'll have to configure Capistrano to provide a password every time you want to deploy the latest changes to your website.

Configuring Public-key Authentication on Unix/Linux

If you're running a Linux/Unix-based system, creating a public key pair is a pretty simple process. Although I won't be covering the configuration process for Windows or OSX-based systems, I nonetheless suggest carefully reading this section as it likely won't stray too far from the steps you'll need to follow. Start by executing the following command to generate your public and private key:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/wjgilmore/.ssh/id_rsa):
```

Unless you have good reasons for overriding the default key name and location, go ahead and accept the default. Next you'll be greeted with the following prompt:

```
Enter passphrase (empty for no passphrase):
```

Some tutorials promote entering an empty passphrase (password), however I discourage this because should your private key ever be stolen, the thief could use the private key to connect to any server possessing your public key. Instead, you can have your cake and eat it to by defining a passphrase and then using a service called ssh-agent to cache your passphrase, meaning you won't have to provide it each time you login to the remote server. Therefore choose a passphrase which is difficult to guess but one you won't forget.

Once you've defined and confirmed a passphrase, your public and private keys will be created. You'll next want to *securely* copy your public key to the remote server. This is probably easiest done using the `scp` utility:

```
$ scp ~/.ssh/id_rsa.pub username@remote:publickey.txt
```

You'll need to replace `username` and `remote` with the remote server's username and address, respectively. Next SSH into the server and add the key to the `authorized_keys` file:

```
$ ssh username@remote
...
$ mkdir ~/.ssh
$ chmod 700 .ssh
```

```
$ cat publickey.txt >> ~/.ssh/authorized_keys
$ rm ~/publickey.txt
$ chmod 600 ~/.ssh/*
```

You should now be able to login to the remote server, however rather than provide your account password you'll provide the passphrase defined when you created the key pair:

```
$ ssh username@remote
Enter passphrase for key '/home/wjgilmore/.ssh/id_rsa':
```

Of course, entering a passphrase each time you login defeats the purpose of using public-key authentication to forego entering a password, doesn't it? Thankfully, you can securely store this passphrase using a program called ssh-agent, which will store your passphrase and automatically supply it when the client connects to the server. Cache your passphrase by executing the following commands:

```
$ ssh-agent bash
$ ssh-add
Enter passphrase for /home/wjgilmore/.ssh/id_rsa:
Identity added: /home/wjgilmore/.ssh/id_rsa (home/wjgilmore/.ssh/id_rsa)
```

Try logging into your remote server again and this time you'll be whisked right to the remote terminal, with no need to enter your passphrase! However, in order to forego having to manually start ssh-agent every time your client boots you'll want to configure it so that it starts up automatically. If you happen to be running Ubuntu, then ssh-agent is already configured to automatically start. This may not be the case on other operating systems, however in my experience configuring ssh-agent to automatically start is a very easy process. A quick search should turn up all of the information you require.

Deploying Your Website

With these general configuration steps out of the way, it's time to ready your website for deployment. You'll only need to carry out these steps once per project, all of which are thankfully quite straightforward.

The first step involves creating a file called `Capfile` (no extension) which resides in your project's home directory. The `Capfile` is essentially Capistrano's bootstrap, responsible for loading needed resources and defining any custom deployment-related tasks. This file will also retrieve any project-specific settings, such as the location of the project repository and the name of the remote server which hosts the production website. I'll explain how to define these project-specific settings in a moment.

Capistrano will by default look for the `Capfile` in the directory where the previously discussed `cap` command is executed, and if not found will begin searching up the directory tree for the file. This is because if you are using Capistrano to deploy multiple websites, then it will make sense to define a single `Capfile` in your projects' root directory. Just to keep things simple, I suggest placing this file in your project home directory for now. Also, because we're using the Railsless Deploy gem to streamline Capistrano, our `Capfile` looks a tad different than those you'll find for the typical Rails project:

```
require 'rubygems'
require 'railsless-deploy'
load    'config/deploy.rb'
```

Notice the third line of the `Capfile` refers to a file called `deploy.rb` which resides in a directory named `config`. This file contains the aforementioned project-specific settings, including which version control solution (if any) is used to manage the project, the remote server domain, and the remote server directory to which the project will be deployed, among others. The `deploy.rb` file I use to deploy my projects is presented next, followed by a line-by-line review:

```
01 # What is the name of the local application?
02 set :application, "gamenomad.wjgilmore.com"
03
04 # What user is connecting to the remote server?
05 set :user, "wjgilmore"
06
07 # Where is the local repository?
08 set :repository,  "file:///var/www/dev.wjgames.com"
09
10 # What is the production server domain?
11 role :web, "gamenomad.wjgilmore.com"
12
13 # What remote directory hosts the production website?
14 set :deploy_to,   "/home/wjgilmorecom/gamenomad.wjgilmore.com"
15
16 # Is sudo required to manipulate files on the remote server?
17 set :use_sudo, false
18
19 # What version control solution does the project use?
20 set :scm,        :git
21 set :branch,     'master'
22
23 # How are the project files being transferred?
24 set :deploy_via, :copy
25
26 # Maintain a local repository cache. Speeds up the copy process.
27 set :copy_cache, true
28
```

```
29 # Ignore any local files?
30 set :copy_exclude, %w(.git)
31
32 # This task symlinks the proper .htaccess file to ensure the
33 # production server's APPLICATION_ENV var is set to production
34 task :create_symlinks, :roles => :web do
35   run "rm #{current_release}/public/.htaccess"
36   run "ln -s #{current_release}/production/.htaccess
37              #{current_release}/public/.htaccess"
38 end
39
40 # After deployment has successfully completed
41 # create the .htaccess symlink
42 after "deploy:finalize_update", :create_symlinks
```

Because the `deploy.rb` is almost certainly new to most readers, a line-by-line review follows:

- Line 02 assigns a name to the application. While this setting is not strictly necessary in all deployment cases, this particular deployment file requires you to define this setting because of the particular deployment approach used on Lines 24 and 27. I'll talk about this approach and why this setting is needed in a moment.

- Line 05 defines the account name used to connect to the remote server. This user should logically possess all of the rights necessary to copy and manipulate the project files on the remote server.

- Line 08 defines the location of the project repository. It's possible to define a remote repository location, for instance pointing to a GitHub-hosted project, however because I'd imagine most readers will want to deploy a project which is hosted locally, I thought it most beneficial to present an example of the syntax used to point to a locally-hosted project.

- Line 11 defines the production server address. Capistrano will use this address when attempting to connect to the remote server.

- Line 14 defines the deployment destination's *absolute path*.

- Line 17 defines whether sudo must be used by the connecting user in order to carry out the deployment. If you don't know what sudo is, then chances are high this should be set to `false`.

- Line 20 defines the version control solution used to manage your project. Defining this setting is necessary because it will determine how Capistrano goes about deploying the project. For instance if `:scm` is set to `:git` then Capistrano will use Git's `clone` command to copy the project. As mentioned earlier in this chapter Capistrano supports quite a few different version control solutions. For instance, use `:bzr` for Bazaar, `:cvs` for CVS, `:mercurial` for Mercurial, and

`:subversion` for Subversion. If your project is currently not under version control, this can be set to `:none`.

- Line 21 defines the repository branch you'd like to deploy. Repository branching is out of the scope of this chapter, however if you are using version control and don't know what a branch is, you can probably safely leave this set to `master`.

- Line 24 tells Capistrano how the files should be deployed to the remote server. The `:copy` strategy will cause Capistrano to clone the repository, archive and compress the cloned repository using the `tar` and `gzip` utilities, and then transfer the archive to the remote server using SFTP. An even more efficient strategy is `:remote_cache`, which will cause Capistrano to deploy only the latest commits rather than transfer the entire project. I suggest using `:remote_cache` if possible, however I am using `:copy` in this example due to repeated issues I've encountered using `:remote_cache`.

- Line 27 enables the `:copy_cache` option, which will greatly speed the deployment process when using the `:copy` strategy. Enabling this option will cause Capistrano to cache a copy of your project (by default in the `/tmp` directory), storing the cache in a directory of the same name as the `:application` setting. When set, at deployment time Capistrano will update this cache with the latest changes before compressing and transferring it, rather than copy the entire repository.

- Line 30 tells Capistrano to ignore certain repository files and directories when deploying the project. For instance, when using the `:copy` strategy the `.git` directory can be ignored because there is no need for the remote server to have access to the project's repository history. Because the `.git` directory can grow quite large over the course of time, excluding this directory from the transfer process can save significant time and bandwidth.

- Lines 34-38 define a Capistrano *task*, which like a programmatic function defines a grouped sequence of commands which can be executed using a named alias `:create_symlinks`). This task is responsible for setting the Zend Framework project `APPLICATION_ENV` variable to `production` by deleting the original `.htaccess` file found in the transferred project's `public` directory, and then creating a symbolic link from the `public` directory which points to a production-specific version of the `.htaccess` file residing in a directory called `production`. You'll of course need to create this directory and production-specific `.htaccess` file, however the latter task is accomplished simply by copying your existing `.htaccess` file to a newly created `production` directory and then modifying this file so that `APPLICATION_ENV` is set to `production` rather than `development`. It is this crucial step that will ensure your deployed Zend Framework application is using the appropriate set of `application.ini` configuration parameters.

- The `:create_symlinks` task defined on lines 34-38 is just a definition; it won't execute unless you tell it to do so. Execution happens on line 42, done by overriding Capistrano's

`deploy:finalize_update` task which will execute by default after the deployment process has completed.

Whew, breaking down that deployment file was a pretty laborious task. However with `deploy.rb` in place, you're ready to deploy your website!

Readying Your Remote Server

As I mentioned at the beginning of this chapter, one of the great aspects of Capistrano is the ability to rollback your deployment to the previous version should something go wrong. This is possible because (when using the copy strategy) Capistrano will store multiple versions of your website on the remote server, and *link* to the latest version via a symbolic link named `current` which resides in the the directory defined by the `:deploy_to` setting found in your `deploy.rb` file. These versions are stored in a directory called `releases`, also located in the `:deploy_to` directory. Each version is stored in a directory with a name reflecting the date and time at the time the release was deployed. For instance, a deployment which occurred on February 24, 2011 at 12:37:27 Eastern will be stored in a directory named `20110224183727` (these timestamps are stored using Greenwich Mean Time).

Additionally, Capistrano will create a directory called `shared` which also resides in the `:deploy_to` directory. This directory is useful for storing custom user avatars, cache data, and anything else you don't want overwritten when a new version of the website is deployed. You can then use Capistrano's `deploy:finalize_update` task to create symbolic links just as was done with the `.htaccess`.

Therefore given my `:deploy_to` directory is set to `/home/wjgilmore/gamenomad.wjgilmore.com`, the directory contents will look similar to this:

```
current -> /home/wjgilmore/gamenomad.wjgilmore.com/
releases/20110224184826
releases
  20110224181647/
  20110224183727/
  20110224184826/
shared
```

Note

If you start using Capistrano to deploy your Zend Framework projects, keep in mind that you'll need to change your production website's document root to `/path/to/current/public`!

Capistrano can create the `releases` and `shared` directories for you, something you'll want to do when you're ready to deploy your website for the first time. Create these directories using the `deploy:setup` command, as demonstrated here:

```
$ cap deploy:setup
```

Deploying Your Project

Now comes the fun part. To deploy your project, execute the following command:

```
$ cap deploy
```

If you've followed the instructions I've provided so far verbatim, remember that Capistrano will be deploying *your latest committed changes*. Whether you've saved the files is irrelevant, as Capistrano only cares about *committed* files.

Presuming everything is properly configured, the changes should be immediately available via your production server. If something went wrong, Capistrano will complain in the fairly verbose status messages which appear when you execute the `deploy` command. Notably you'll probably see something about rolling back the changes made during the current deployment attempt, which Capistrano will automatically do should it detect that something has gone wrong.

Rolling Back Your Project

One of Capistrano's greatest features is its ability to revert, or rollback, a deployment to the previous version should you notice something just isn't working as you expected. This is possible because as I mentioned earlier in the chapter, Capistrano stores multiple versions of the website on the production server, meaning returning to an earlier version is as simple as removing the symbolic link to the most recently deployed version and then creating a new symbolic link which points to the previous version.

To rollback your website to the previously deployed version, just use the `deploy:rollback` command:

```
$ cap deploy:rollback
```

Reviewing Commits Since Last Deploy

Particularly when you're making changes to a project which aren't outwardly obvious, it can be easy to lose track of what commits have yet to be deployed. You can review this list using the

`deploy:pending` command, which will return a list of log messages and other commit-related information associated with those commits made since the last successful deployment:

```
$ cap deploy:pending
  * executing `deploy:pending'
  * executing "cat /home/wjgilmorecom/gamenomad.wjgilmore.com
/current/REVISION"
    servers: ["gamenomad.wjgilmore.com"]
    [gamenomad.wjgilmore.com] executing command
    command finished
commit 0380f960af0db2b5d8cfb8893cb07caf038c9754
Author: Jason Gilmore <wj@wjgilmore.com>
Date:   Thu Feb 24 11:32:28 2011 -0500

  Added special offer widget to the home page.
```

Test Your Knowledge

Test your understanding of the concepts introduced in this chapter by answering the following questions. You can find the answers in the back of the book.

- Provide a few reasons why a tool such as Capistrano is superior to FTP for project deployment tasks.
- Describe in general terms what steps you'll need to take in order to ready your project for deployment using Capistrano.

Conclusion

Every so often you'll encounter a utility which can immediately improve how you go about creating and maintaining software. Capistrano is certainly one of those rare gems. Consider making your life even easier by bundling Capistrano commands into a Phing (http://phing.info) build file, creating a convenient command-line menu for carrying out repetitive tasks. I talk about this topic at great length in the presentation "Automating Deployments with Phing, Capistrano and Liquibase". You can download the presentation slides and a sample build file via my GitHub project page: http://www.github.com/wjgilmore/.

Appendix A. Test Your Knowledge Answers

This appendix contains the answers to the end-of-chapter questions located the section "Test Your Knowledge".

Chapter 1

Identify and describe the three tiers which comprise the MVC architecture.

The MVC architecture consists of three tiers, including the model, view, and controller. The model is responsible for managing the application's data and behavior. The view is responsible for rendering the model in a format best-suited for the client interface, such as web page. The controller is responsible for responding to user input and interacting with the model to complete the desired task.

How does the concept of "convention over configuration" reduce the number of development decisions you need to make?

Convention over configuration is an approach to software design which attempts to reduce the number of tedious implementational decisions a developer needs to make by assigning default solutions to these decisions. How to best go about managing configuration data, validate forms data, and structure page templates are all examples of decisions already made for you when using a web framework which embraces this notion of convention over configuration.

Name two ways the Zend Framework helps you keep your code DRY.

Although the Zend Framework reduces code redundancy in a wide variety of ways, two ways specifically mentioned in Chapter 1 include the ability to create and execute *action helpers* and *view helpers*.

Chapter 2

What command-line tool is used to generate a Zend Framework project structure?

The command-line tool commonly used to generate a Zend Framework project and its constituent parts is known as *zf*.

What file should you never remove from the project directory, because it will result in the aforementioned tool not working properly?

The zf command-line tool uses a file named `.zfproject.xml` to keep track of the project structure. Removing or modifying this file will almost certainly cause zf to behave erratically.

What is a virtual host and why does using virtual hosts make your job as a developer easier?

A virtual host is a convenient solution for serving multiple websites from one web server. Using virtual hosts on your development machine is particularly convenient because you can simultaneously work on multiple projects without having to reconfigure or restart the web server.

What two files are found in the `public` directory when a new project is generated? What are the roles of these files? What other types of files should you place in this directory?

A Zend Framework project's `public` directory contains an `.htaccess` and `index.php` file. The `.htaccess` file is responsible for rewriting all incoming requests to the `index.php` file, which serves as the application's front controller. You'll also place CSS and JavaScript files in this directory, in addition to your website images.

Chapter 3

The Zend Framework's convenient layout feature is not enabled by default. What ZF CLI command should you use to enable this feature?

Execute the command `zf enable layout` to enable your project's layout file. This file is stored by default in the directory `application/layouts/scripts` and is named `layout.phtml`.

From which directory does the Zend Framework expect to find your website CSS, images, and JavaScript?

The CSS, images, and JavaScript files should be placed in the `public` directory.

What is the name of the Zend Framework feature which can help to reduce the amount of PHP code otherwise found in your website views?

View helpers are useful for not only reducing the amount of PHP code found in a view, but also for helping to DRY up your code by abstracting layout-related logic which might be reused within multiple areas of your website.

Which Zend Framework class must you extend in order to create a custom view helper? Where should your custom view helpers be stored?

Custom view helpers should extend the `Zend_View_Helper_Abstract` class. They should be placed in the project's `application/views/helpers` directory.

Name two reasons why the Zend Framework's URL view helper is preferable over manually creating hyperlinks?

The native URL view helper is convenient because it can dynamically adapt the URL in accordance with any changes made to the website structure. Additionally, URL helpers can refer to a custom route definition rather than explicitly naming a controller or action.

Chapter 4

Which Zend Framework component is primarily responsible for simplifying the accessibility of project configuration data from a central location?

The Zend_Config component is the primary vehicle used for accessing a Zend Framework project's configuration data.

What is the name and location of the default configuration file used to store the configuration data?

Although it's possible to store a Zend Framework project's configuration data using a variety of formats, the default solution involves using an INI-formatted file named `application.ini` stored in the directory `application/configs`.

Describe how the configuration data organized such that it is possible to define stage-specific parameters.

The `application.ini` file is broken into multiple sections, with each section representative of a lifecycle stage. These sections are identified by the headers `[production]`, `[staging:production]`, `[testing:production]`, and `[development:production]`. The latter three stages inherit from the `production` stage, as is indicative by the syntax used to denote the section start.

What is the easiest way to change your application's lifecycle stage setting?

Although the Zend Framework's `APPLICATION_ENV` can be set in a variety of ways, the most common approach involves setting it in the `.htaccess` file.

Chapter 5

Name two reasons why the Zend_Form component is preferable to creating and processing forms manually.

Although one could cite dozens of reasons why the Zend_Form component is preferable to creating and processing forms manually, two particularly prominent reasons include the ability to encapsulate a form's components and behavior within a model, and the ease in which form fields can be validated and repopulated.

How does the Flash Messenger feature streamline a user's website interaction?

The Flash Messenger is useful because the developer can create a message which should be presented to the user following the completion of a specific task, such as successfully registering or logging in, and then present this message on a subsequent page, thereby streamlining the number of interactions a user needs to make in order to navigate the website.

What is the role of PHPUnit's data provider feature?

PHPUnit's data provider feature is useful for vetting various facets of a particular website feature by repeatedly executing a test and passing in a different set of test data with each iteration.

Chapter 6

Define object-relational mapping (ORM) and talk about why its an advantageous approach to programmatically interacting with a database.

Object-relational mapping provides a solution for interacting with a database in an object-oriented fashion, thereby reducing and possibly entirely eliminating the need to inconveniently mingle incompatible data structures.

Given a model named `Application_Model_DbTable_Game`, what will Zend_Db assume to be the name of the associated database table? How can you override this default assumption?

Given a model named `Application_Model_DbTable_Game`, the Zend_Db component will assume the associated database table is named `game`. You can override this assumption by defining a protected property named `name` in your model.

What are the names and purposes of the native Zend_Db methods used to navigate model associations?

The `findParentRow()` method is used by a child to retrieve information about its parent row. The `findDependentRowset()` method is used by a parent to retrieve information about its children rows.

Chapter 7

Talk about the advantages Doctrine provides to developers.

Doctrine offers quite a few powerful advantages, including notably a mature object-relational mapper, a database abstraction layer, and a powerful command-line tool. Its possible to identify standard PHP classes as persistent through a simple configuration process, thereby greatly reducing the amount of code a developer would otherwise have to write to implement persistence features.

Talk about the different formats Doctrine supports for creating persistent objects.

Doctrine can marry PHP objects and the underlying database using *DocBlock annotations*, `XML`, and `YAML`. DocBlock annotations are the author's preferred approach, although any of the three will work just fine.

What are DocBlock annotations?

DocBlock annotations allow developers to define a standard PHP class as persistent by embedding metadata within PHP comment blocks. These annotations are used to define the mapping between a class' properties and the associated SQL type, specify primary keys, and define associations.

What is DQL and why is it useful?

The *Doctrine Query Language* (DQL) is a query language useful for querying and interacting with your project's object model. DQL is useful because it allows the developer to mine data which continuing to think in terms of objects rather than in SQL.

What is QueryBuilder and why is it useful?

The QueryBuilder is an API which provides developers with a means for rigorously constructing a DQL query.

Why is it a good idea to create a custom repository should your query requirements exceed the capabilities provided by Doctrine's magic finders?

Custom repositories provide a convenient way to encapsulate your custom data-access features in a specific location rather than polluting the domain entities.

Chapter 8

Explain how Zend_Auth knows which table and columns should be used when authenticating a user against a database.

The Zend_Auth component includes an adapter named `Zend_Auth_Adapter_DbTable` which supports methods used to map the adapter to a specific database table, and identify the table

columns used store the account username and password. These methods include `setTableName()`, `setIdentityColumn()`, and `setCredentialColumn()`, respectively.

At a minimum, what are the five features you'll need to implement in order to offer basic user account management capabilities?

The five fundamental account management features include account registration, account login, account logout, password update, and password recovery.

Talk about the important role played by the `account` table's `recovery` column within several features described within this chapter.

The `account` table's `recovery` column is used for unique identifiers which serve to create a *one-time URL*. A one-time URL is sent to a newly registered user's e-mail address. When clicked, the unique identifier will be compared against the database in order to confirm the account.

Chapter 9

Why should you link to the jQuery library via Google's content distribution network rather than store a version locally?

Linking to the jQuery library via Google's content distribution network will greatly increase the likelihood that the library is already cached within a user's browser, thereby reducing the amount of bandwidth required to serve your website.

What role does jQuery's `$.getJSON` method play in creating the Ajax-driven feature discussed earlier in this chapter?

The `$.getJSON` method is useful for asynchronously communicating with a remote endpoint via an HTTP GET request.

Chapter 10

Why is it a wise idea to use PHP's CLI in conjunction with scripts which could conceivably run for a significant period of time?

When using PHP scripts to execute batch processes, using the command-line interface (CLI) is a wise idea because CLI-based PHP scripts do not take PHP's `max_execution_time` setting into account. Additionally, CLI-based scripts can be automatically executed using a scheduling daemon such as CRON.

What two significant improvements does the Google Maps API V3 offer over its predecessor?

The Google Maps API V3 offers several improvements over its predecessor, however two of the most notable changes include the removal of the API key requirement and the streamlined syntax which greatly reduces the amount of code otherwise required to implement key features such as map customization and marker display.

Chapter 11

Define unit testing and talk about the advantages it brings to your development process.

Unit testing provides developers with a tool for formally and rigorously determining whether specific parts of an application's source code behave as expected. Using an automated unit testing tool is advantageous because a suite of tests can be created, managed and organized in an efficient manner.

What Zend component and open source PHP project work in conjunction with one another to bring unit testing to your Zend Framework applications?

PHPUnit and the Zend Framework's Zend_Test component work together to bring unit testing features to your Zend Framework projects.

What is the name of the syntactical construct used to validate expected outcomes when doing unit testing?

Expected outcomes are confirmed using *assertions*.

Why are code coverage reports useful?

Code coverage reports are useful because they provide developers with a valuable tool for determining how many lines of a project have been "touched" by unit tests.

Chapter 12

Provide a few reasons why a tool such as Capistrano is superior to FTP for project deployment tasks.

Capistrano is superior to FTP because Capistrano is faster, more secure, and able to automatically revert a website to its last known good state should a problem occur during the deployment process. Additionally, the developer can manually revert a website to its previous state should he detect a problem following deployment.

Describe in general terms what steps you'll need to take in order to ready your project for deployment using Capistrano.

After installing Capistrano (and optionally Railsless-deploy), and configuring shared-key authentication, you'll want to create a Capfile and a deployment file (which contains the project's deployment configuration settings). Before deploying your project you'll want to ready the deployment server by executing Capistrano's `deploy:setup` command. Finally, when ready to deploy you'll execute the `deploy` command.