THE EXPERT'S VOICE® IN WEB DEVELOPMENT

SECOND EDITION

# PHP for Absolute Beginners

*EVERYTHING YOU NEED TO KNOW TO GET STARTED IN PHP*

Thomas Blom Hansen and Jason Lengstorf

Apress®

*For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.*


Apress®

# Contents at a Glance

# Introduction

Modern web development relies on the successful integration of several technologies. Content is mostly formatted as HTML. With server-side technologies, you can create highly dynamic web applications. PHP is the single most used server-side scripting language for delivering browser-based web applications. PHP is the backbone of online giants such as Facebook, Flickr, and Yahoo.

There are other server-side languages available for web application development, but PHP is the workhorse of the Internet. For an absolute beginner, it should be comforting to know that PHP is a relatively easy language to learn. You can do many things with a little PHP. Also, there is a thriving, friendly community supporting PHP. It will be easy to get help with your own PHP projects.

## Who Should Read This Book

This book is intended for those who know some HTML and CSS. It is for those who are ready to take their web developer skills to the next level. You will learn to generate HTML and CSS dynamically, using PHP and MySQL. You will learn the difference between client-side and server-side scripting through hands-on experience with PHP/MySQL and JavaScript code projects. Emphasis will be on getting up and running with PHP, but you will also get to use some MySQL and some JavaScript in your projects. By the end of the book, you will have created a number of PHP-driven projects, including the following:

- A personal portfolio site with dynamic navigation
- A dynamic image gallery where users can upload images through an HTML form
- A personal blogging system, complete with a login and an administration module

In the process, you will become acquainted with such topics as object-oriented programming, design patterns, progressive enhancement, and database design. You will not get to learn everything there is to know about PHP, but you will be off to a good start.

## How to Read This Book

This book is divided into two main parts. Part I will quickly get you started writing PHP for small, dynamic projects. You will be introduced to a relatively small subset of PHP–just enough for you to develop entry-level web applications. Part I will also teach you the basic vocabulary of PHP.

Part II is a long hands-on project. You will be guided through the development of the aforementioned personal blogging system, starting from scratch. Part II will show you how to use your PHP vocabulary to design dynamic, database-driven web applications.

PART I



# PHP/MySQL Basics

You will learn how to set up a PHP/MySQL development environment, the basics of PHP and how to connect PHP to a MySQL database.

CHAPTER 1

# Setting Up a PHP Development Environment

Getting a working development environment put together can be intimidating, especially for the absolute beginner. To follow along with the project in this book, you'll need to have access to a working installation of Apache, PHP, and MySQL, preferably on your local machine. It's always desirable to test locally, both for speed and security. Doing this both shelters your work-in-progress from the open Internet and decreases the amount of time spent uploading files to an FTP server and waiting for pages to reload.

## Why You Need Apache, MySQL, and PHP

PHP is a powerful scripting language that can be run by itself in the command line of any computer with PHP installed. However, PHP alone isn't sufficient for building dynamic web sites. To use PHP on a web site, you need a server that can process PHP scripts. Apache is a free web server that, once installed on a computer, allows developers to test PHP scripts locally; this makes it an invaluable piece of your local development environment.

Additionally, web sites developed with PHP often rely on information stored in a database, so it can be modified quickly and easily. This is a significant difference between a PHP site and an HTML site. This is where a relational database management system such as MySQL comes into play. This book's examples rely on MySQL. I chose this database because PHP provides native support for it, and because MySQL is a free, open source project.

---

**Note** An open source project is available for free to end users and ships with the code required to create that software. Users are free to inspect, modify, and improve the code, albeit with certain conditions attached. The Open Source Initiative lists ten key provisions that define open source software. You can view this list at `www.opensource.org/docs/osd`.

---

PHP is a general-purpose scripting language that was originally conceived by Rasmus Lerdorf in 1995. Lerdorf created PHP to satisfy the need for an easy way to process data when creating pages for the World Wide Web.

---

**Note** PHP was born out of Rasmus Lerdorf's desire to create a script that would keep track of how many visits his online résumé received. Due to the wild popularity of the script he created, Lerdorf continued developing the language. Over time, other developers joined him in creating the software. Today, PHP is one of the most popular scripting languages in use on the Internet.

---

PHP originally stood for *Personal Home Page* and was released as a free, open source project. Over time, the language was reworked to meet the needs of its users. In 1997, PHP was renamedPHP: *Hypertext Preprocessor*, as it is known currently. At the time I write this, PHP 5.5.7 is the current stable version. Older versions of PHP are still in use on many servers.

## How PHP Works

PHP is generally used as a server-side scripting language; it is especially well-suited for creating dynamic web pages. The scripting language features integrated support for interfacing with databases, such as MySQL, which makes it a prime candidate for building all manner of web applications, from simple personal web sites to complex enterprise-level applications.

HTML is parsed by a browser when a page loads. Browsers cannot process PHP at all. PHP is processed by the machine that serves the document (this machine is referred to as a server). All PHP code in the document is processed by the server before the document is sent to the visitor's browser. Because PHP is processed by a server, it is a *server-side scripting language*.

With PHP, you can create *dynamic* web pages—web pages that can change according to conditions. For example: When I log in to my Facebook account, I see my content. When you log in to your Facebook account, you see your content. We would be loading the same resource (`www.facebook.com`), but we would be served different content *dynamically*. This would be impossible with HTML web documents, because they are *static*, meaning they can't change. Every user would see exactly the same HTML page. The rest of this book explores some of the things you can achieve with dynamic web pages.

PHP is an interpreted language, which is another great advantage for PHP programmers. Many programming languages require that you compile files into machine code before they can be run, which is a time-consuming process. Bypassing the need to compile means you're able to edit and test code much more quickly.

Because PHP is a server-side language, running PHP scripts requires a server. To develop PHP projects on your local machine means installing a server on your local machine. The examples in this book rely on the Apache Web Server to deliver your web pages.

## Apache and What It Does

Apache is the most popular web server software on the Web; it hosts nearly half of all web sites that exist today. Apache is an open source project that runs on virtually all available operating systems. Apache is a community-driven project, with many developers contributing to its progress. Apache's open source roots also means that the software is available free of charge, which probably contributes heavily to Apache's overwhelming popularity relative to its competitors, including Microsoft's IIS and Google's GWS, among others.

On the Apache HTTP Server Project web site (`http://httpd.apache.org`), Apache HTTP Server is described as "an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows NT. The goal of this project is to provide a secure, efficient, and extensible server that provides HTTP services in sync with the current HTTP standards."

As with all web servers, Apache accepts an HTTP request and serves an HTTP response. The World Wide Web is founded on web servers, and every web site you visit demonstrates the functionality of web servers. I've already mentioned that while HTML can be processed by a web browser, server-side scripting languages such as PHP have to be handled by a web server. Due to its overwhelming popularity, Apache is used for testing purposes throughout this book.

## Storing Info with MySQL

MySQL is a relational database management system (RDBMS). Essentially, this means that MySQL allows users to store information in a table-based structure, using rows and columns to organize different pieces of data. There are many other relational database management systems. The examples in this book rely on MySQL to store the information you'll use in your PHP scripts, from blog entries to administrator information. This approach has great advantages, which we will explore in detail.

■ **Note** *Blog* is short for *weblog*, which is an online journal produced by an individual or a business.

# Installing PHP, Apache, and MySQL

One of the biggest hurdles for new programmers is starting. Before you can write your first line of PHP, you must download Apache and PHP, and usually MySQL, and then fight through installation instructions that are full of technical jargon you might not understand yet. This experience can leave many developers feeling unsure of themselves, doubting whether they've installed the required software correctly.

In my own case, this hurdle kept me from learning programming for months, even though I desperately wanted to move beyond plain ole HTML. I unsuccessfully attempted to install PHP on my local machine not once, but three different times before I was able to run my first PHP command successfully.

Fortunately, the development community has responded to the frustration of beginning developers with several options that take all the pain out of setting up your development environment, whether you create applications for Windows, Mac, or Linux machines. These options include all-in-one solutions for setting up Apache, MySQL, and PHP installations.

The most common all-in-one solution is a program called XAMPP (`www.apachefriends.org/en/xampp.html`), which rolls Apache, MySQL, PHP, and a few other useful tools together into one easy installer. XAMPP is free and available for Windows, Mac, and Linux. This book assumes that you will use it as your development environment.

■ **Note** Most Linux distributions ship with one flavor or another of the LAMP stack (Linux-specific software that functions similarly to XAMPP) bundled in by default. Certain versions of Mac OS X also have PHP and Apache installed by default.

## Installing XAMPP

Enough background. You're now ready to install XAMPP on your development machine. This process should take about five minutes and is completely painless.

### Step 1: Download XAMPP

Your first task is to obtain a copy of the XAMPP software. Head over to the XAMPP site (`www.apachefriends.org/en/xampp.html`) and download the latest version (1.8.3 at publication time).

### Step 2: Open the Installer and Follow the Instructions

After downloading XAMPP, find the newly downloaded installer and run it. You should be greeted with a screen similar to the one shown in Figure 1-1.

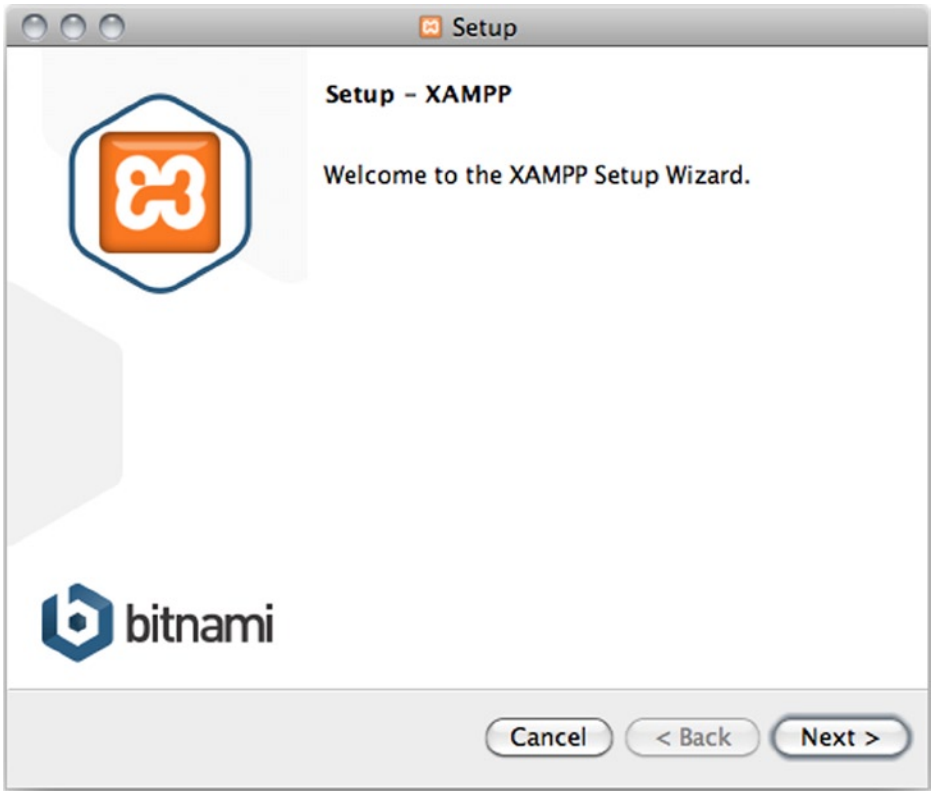


***Figure 1-1.*** *The introductory screen for the XAMPP installer on Mac OS X*

---

■ **Note** All screenshots used in this book were taken on a computer running Mac OS X 10.6.8. Your installation might differ slightly, if you use a different operating system. XAMPP for Windows offers additional options, such as the ability to install Apache, MySQL, and Filezilla (an FTP server) as services. This is unnecessary and will consume computer resources, even when they are not being used, so it's probably best to leave these services off. Additionally, Windows users should keep the `c:\xampp` install directory for the sake of following this book's examples more easily.

---

Click the Next button to move to the next screen (see Figure 1-2), where you can choose which components to install. Just go with the default selection. The XAMPP installer will guide you through the installation process. Figures 1-3 through 1-5 show the remaining steps.

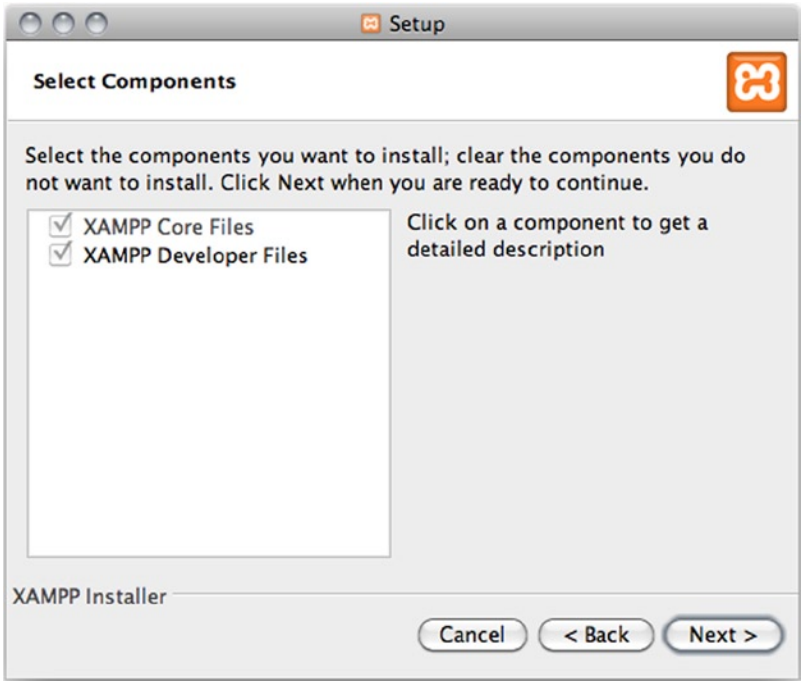


***Figure 1-2.*** *Select components to install*

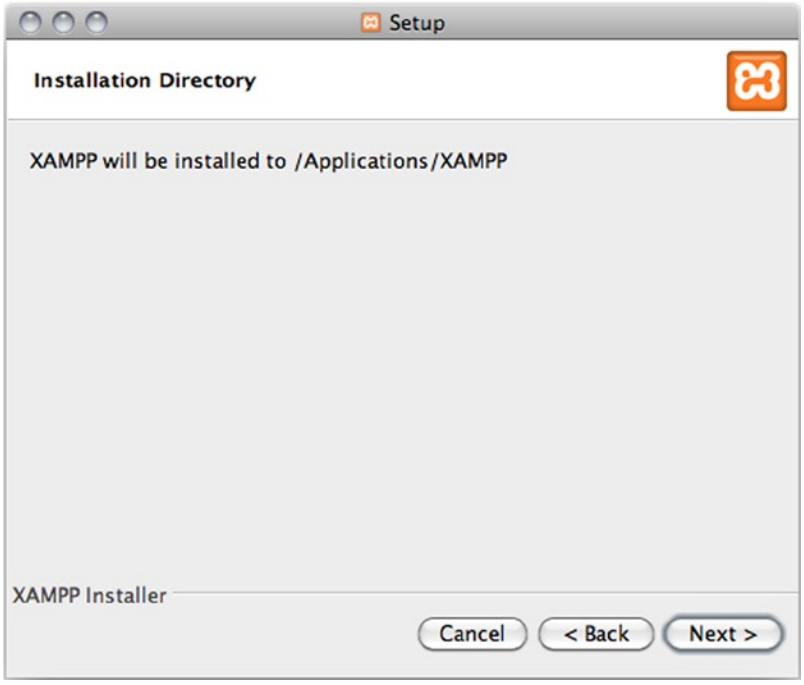


***Figure 1-3.*** *XAMPP installation directory*


***Figure 1-4.*** *You don't have to learn more about BitNami at this point*

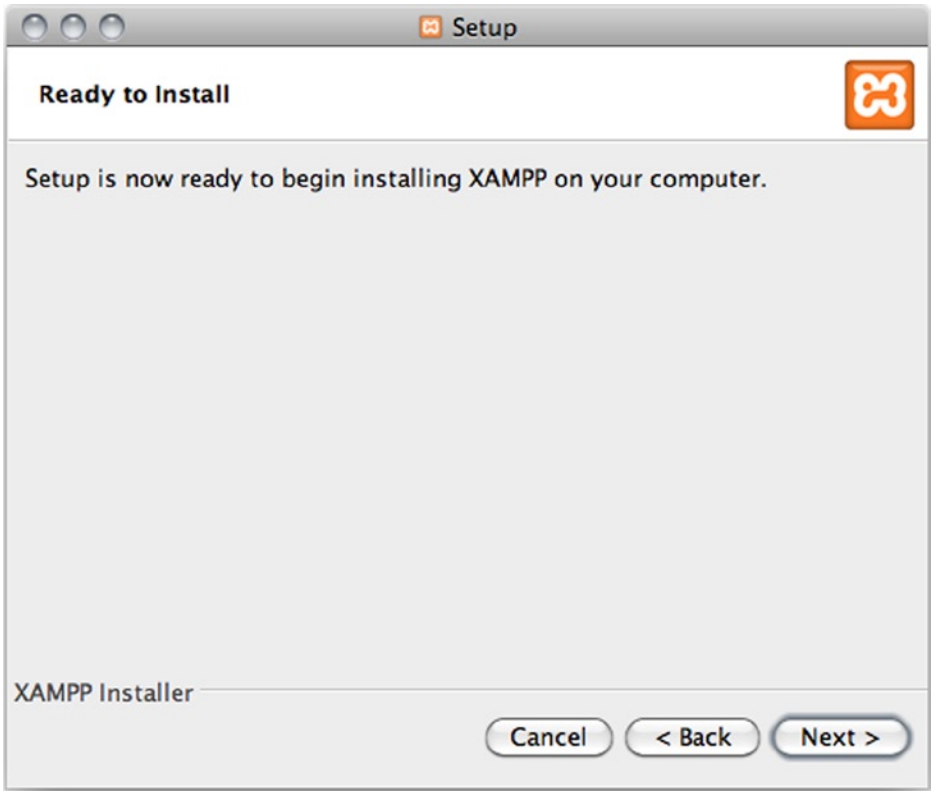


***Figure 1-5.*** *When you're ready to install, click Next*

Installation requires a minute or two to complete, whereupon the installer displays the final screen (see Figure 1-6), which confirms that the installation was successful.

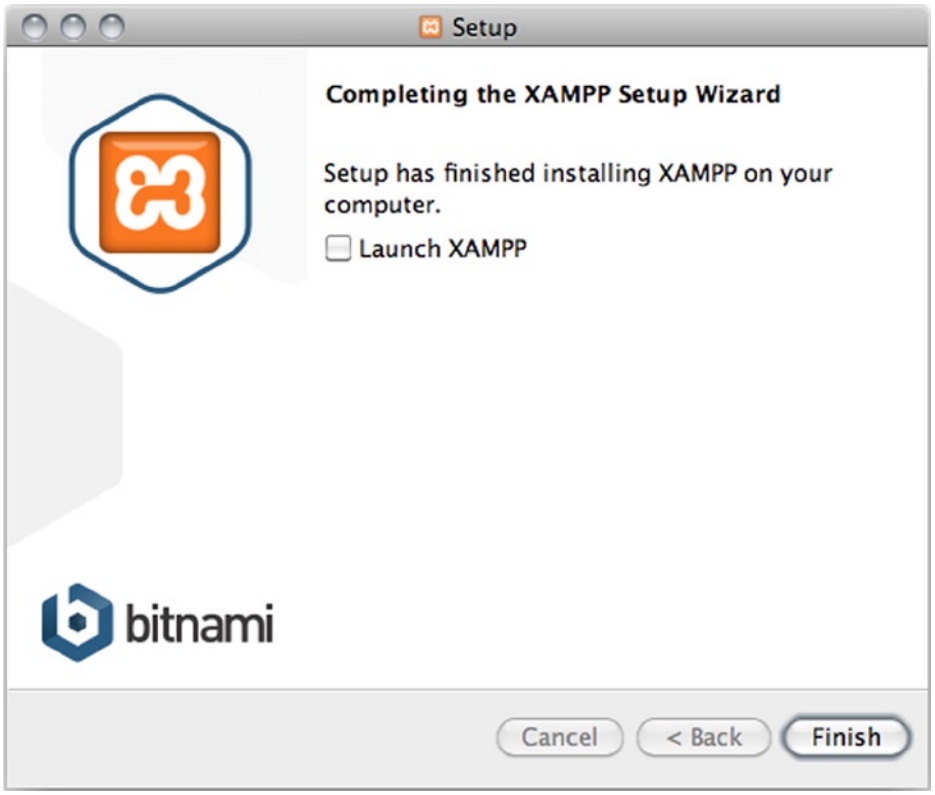


***Figure 1-6.*** *Installation is complete*

## Step 3: Test XAMPP to Ensure Proper Installation

So far, you've used the XAMPP wizard to install Apache, PHP, and MySQL. The next step is to activate Apache, so you can write some PHP.

### Open the XAMPP Control Panel

You can activate the just-installed applications by navigating to the newly installed XAMPP folder and opening the XAMPP manager (see Figure 1-7).

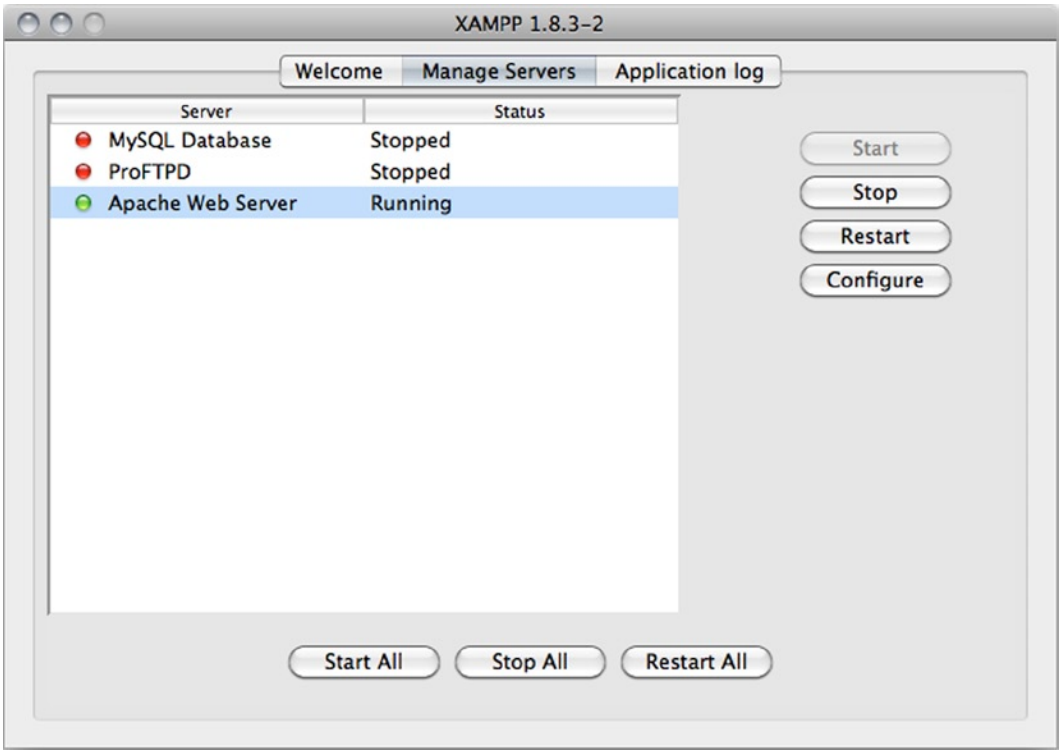


***Figure 1-7.*** *The XAMPP manager shows that the local Apache Web Server is running*

■ **Note** When opening the XAMPP control panel, you may be prompted for your password. This has no effect on the services themselves and should not affect the projects covered in this book.

Activating Apache, PHP, and MySQL on your development machine is as simple as clicking the Start button next to Apache in the XAMPP manager. You might be prompted to confirm that the server is allowed to run on your computer, and you might be required to enter your system password. After you do this, the Status should indicate that Apache is running, as shown in Figure 1-7.

■ **Note** There is an FTP (file transfer protocol) option available in XAMPP. FTP provides a method for moving files between networks. The examples in this book don't require this option, so there is no need to activate it in the XAMPP control panel. The first few chapters don't even require a MySQL database.

## What If Apache Isn't Running?

Sometimes, XAMPP Apache Server doesn't run, even if you try to start it. The most common problem is that it conflicts with some other service using the same port on your computer. Check if you have Skype or Messenger or some similar networking service running. Shut Skype completely down, and if you're lucky, your Apache can run.

If it still doesn't run, you could turn to the Internet for help. The XAMPP online community is extremely helpful, and most installation issues have been addressed in the Apache Friends forum at `https://community.apachefriends.org/f/viewforum.php?f=34`. You could also turn to search or ask at `http://stackoverflow.com/`.

## Verify That Apache and PHP Are Running

It's a simple matter to check whether Apache is running properly on your development machine. Simply open a browser and go to the following address: `http://localhost`. If everything has gone correctly, you'll be redirected to `http://localhost/xampp/splash.php` (see Figure 1-8).


***Figure 1-8.*** *Check in your browser that your Apache Web Server is running*

If this screen loads, you've installed Apache and PHP on your development machine successfully! The address, `http://localhost`, is an alias for the current computer you're working on. When using XAMPP, navigating to `http://localhost` in a browser tells the server to open the root web directory. This is the `htdocs` folder contained in the XAMPP install directory. Another way to use your server to access the root web directory on your local machine is to navigate to the IP address—a numerical identifier assigned to any device connected to a computer network—that serves as the "home" address for all HTTP servers: `http://127.0.0.1`.

# Choosing a PHP Editor

Your development machine is now running all the necessary programs for programming with PHP. The next step is to decide how you're going to write your scripts. PHP scripts are text-based, so you have myriad options, ranging from the simple `Notepad.exe` and text-edit programs to highly specialized integrated development environments (IDEs).

Most experienced PHP developers use an IDE, because they offer many benefits. Many beginners have some difficulties using an IDE, perhaps because IDEs have so many features that beginners are simply left confused.

You can probably write PHP code using whichever program you have used for writing HTML and CSS. There are some features you should expect from a good editor.

- *Syntax highlighting*: This is the ability to recognize certain words in a programming language, such as variables, control structures, and various other special text. This special text is highlighted or otherwise differentiated to make scanning your code much easier.
- *Built-in function references*: When you enter the name of a function or an object method, this feature displays available parameters, as well as the file that declares the function, a short description of what the function does, and a more in-depth breakdown of parameters and return values. This feature proves invaluable when dealing with large libraries, and it can save you trips to the PHP manual to check the order of parameters or acceptable arguments for a function.
- *Auto-complete features*: This feature adds available PHP keywords to a drop-down list, allowing you to select the intended keyword from the list quickly and easily, saving you the effort of remembering and typing it out every time. When it comes to productivity, every second counts, and this feature is a great way to contribute to saved time.
- *Code folding*: This feature lets you collapse snippets of code, making your workspace clutter-free and your code easy to navigate.
- *Auto-indent*: This automatically indents the code you write in a consistent manner. Such indented code is vastly easier to read for human readers, because indentation indicates relationships between code blocks.
- *Built-in ftp*: You need ftp to upload your PHP files to an online web server when you want to publish your project on the World Wide Web. You can use a stand-alone ftp program, but if it is built into your IDE, you can upload an entire project with a single click.

You have many good IDEs and editors to choose from. NetBeans and Eclipse PDT are both excellent, free IDEs. Try one or both, if you want to get used to the tools professional developers often gravitate to. Beginners may find it easier to start with a simpler editor. I really like Komodo Edit: It is as easy to use as any editor, and it provides most of the features just listed out of the box, including excellent auto-complete for PHP and many other languages. Following are the download links for the three PHP editors just mentioned:

- Get NetBeans from `https://netbeans.org/downloads/`.
- Get Eclipse PDT from `http://projects.eclipse.org/projects/tools.pdt`.
- Get Komodo Edit from `www.activestate.com/komodo-edit/downloads`.

I will use Komodo Edit for the examples in this book. You should have no difficulties following the examples with any other editor. If you decide to use an IDE, you will have to consult online documentation to learn how to set up a new project in your chosen IDE.

# Creating Your First PHP File

With everything set up and running as it should, it is time to take the plunge and write your first PHP script. As a server-side scripting language, PHP requires a web server such as Apache to run. You have just installed Apache on your local computer, so your system is ready.

Apache will interpret any PHP files saved inside a folder called `htdocs`. You can find it inside your XAMPP installation in `XAMPP/xamppfiles/htdocs`.

You'll be making many PHP files soon, so it is a good idea to keep them organized. Create a new folder inside `htdocs` and call it `ch1`.

Now open Komodo Edit, or whichever editor or IDE you have decided to use. From Komodo Edit, you can select File ➤ New ➤ New File from the main menu. In the new file you write the following:

```
<?php

echo "Hello from PHP";
```

In Komodo Edit, select File ➤ Save to see the file saving dialog box (Figure 1-9). Save the new file as `test.php`, in `htdocs/ch1`; set format to All Files; and then click Save.

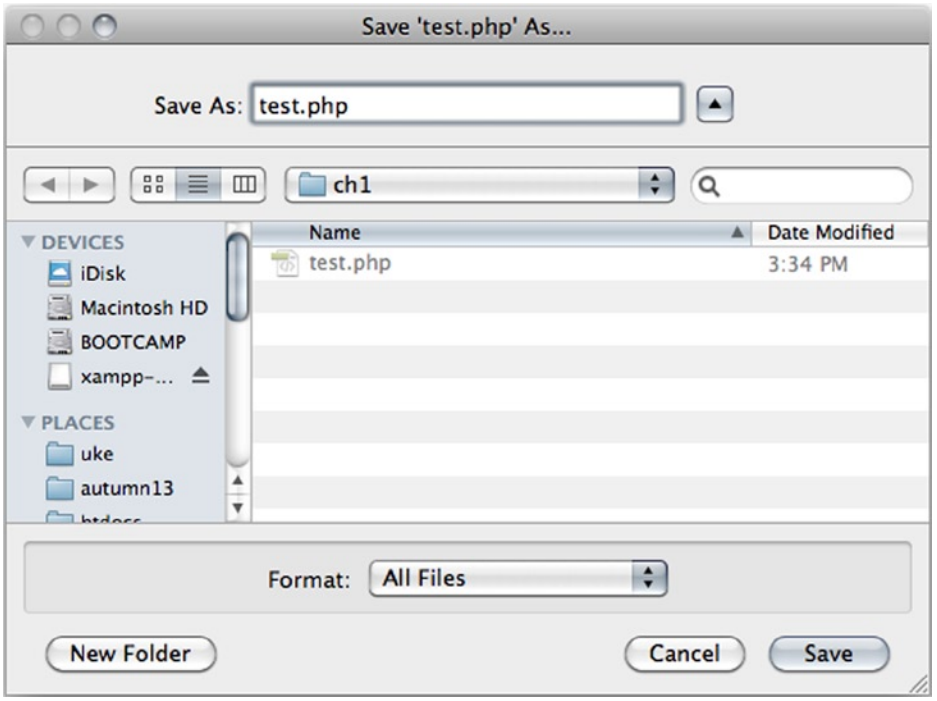


***Figure 1-9.*** *The Save As dialog box from Komodo Edit*

# Running Your First PHP Script

The next step is to get Apache to process your PHP script. That happens automatically, if you request the script through a browser. So, open a web browser and navigate to `http://localhost/ch1/test.php` and marvel at the PHP-generated output you should see in your browser (Figure 1-10). You have successfully created and executed your first PHP script!

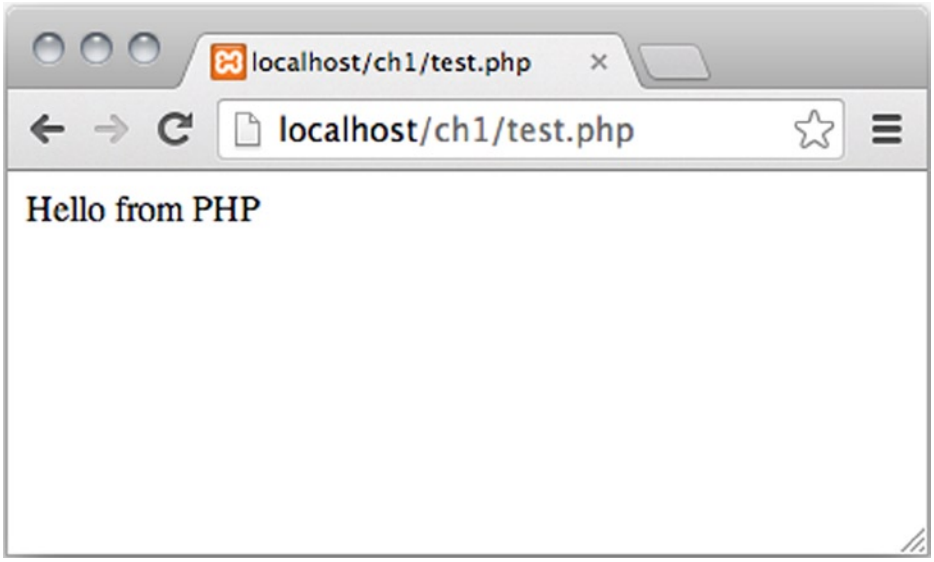


***Figure 1-10.*** *Seeing the output from* `test.php` *in the Chrome web browser*

# Summary

In this chapter, you learned a little bit about PHP, MySQL, and Apache. You found out what they are and what role they play in the development of dynamic web sites. You also learned a quick and easy way to install a fully functional development environment on your local computer, by installing XAMPP and Komodo Edit.

In the next chapter, you'll learn a small but potent subset of PHP, including variables, objects, and some native language constructs and statements. Nearly everything you learn will be tested in your new development environment, so keep XAMPP's Apache Server open and running.

CHAPTER 2

# Understanding PHP: Language Basics

So far, you've bypassed the old, cumbersome method of creating a development environment, and you're now ready to start writing code.

But where do you start? In this chapter, I'll cover the steps you need to follow to start using PHP in the creation of powerful, dynamic web applications. You'll also begin to develop the basic skills you need to create your blog. In addition, you'll learn how to accomplish several tasks, including how to do the following:

- Embed PHP in web pages
- Send data as output to the browser
- Add comments in your code
- Use variables
- Work with PHP errors
- Create an HTML5 template
- Use objects
- Concatenate strings
- Access URL variables with the `$_GET` superglobal
- Declare a class definition
- Embed dynamic CSS

By the end of this chapter, you will have seen some basic PHP that will allow you to create, store, manipulate, and output data. You will have used those skills to develop a bare-bones version of a personal portfolio web site.

---

**Note** This chapter discusses basic aspects of the PHP language, but not in complete detail. For clarification, more examples, or for concept reinforcement, visit the PHP manual at `www.php.net/manual/en/` and search the function in the field where it says "search for _____ in the function list." Alternatively, you can access information about many PHP functions by navigating to `http://php.net/function_name`. Don't forget to read the comments, because many of your fellow programmers offer insight, tips, and even additional functions in their commentary.

---

# Embedding PHP Scripts

In Chapter 1, when I talked about Apache and web servers in general, I mentioned how a server will process PHP in a file before sending that file to the browser. But you might be curious as to how the server knows where to look for PHP.

By default, servers look for PHP only in files that end with the `.php` extension. But a `.php` file can contain elements that aren't part of your PHP script, and searching the entire file for potential scripts is confusing and resource-intensive. To solve this issue, all PHP scripts need to be contained with PHP delimiters. To begin a PHP script, you include the opening delimiter `<?php` and start coding. To finish, you simply add `?>` to the end of the script. Anything outside of these delimiters will be treated as HTML or plain text.

You can see this in action. Start by creating a new folder `ch2` in `/xampp/htdocs/`. Next, create a new file, `test.php`, with Komodo Edit. Write the following code:

```
<p>Static Text</p>
<?php
echo "<p>This text was generated by PHP!</p>";
?>
<p>This text was not.</p>
```

Save the file, navigate to `http://localhost/ch2/test.php` in your browser, and you should see the following output in your browser:

```
Static Text
This text was generated by PHP!
This text was not.
```

As you can see, the text inside the PHP delimiters , was handled as a script, but the text outside was rendered as regular HTML. There is no limit to how many blocks of PHP you can include in a page, so the following snippet is completely valid:

```
<?php
echo "<p>This is some text.</p>";
?>
<p>Some of this text is static, <?php echo "but this sure isn't!"; ?></p>
<?php echo "<p>"; ?>
This text is enclosed in paragraph tags that were generated by PHP.
<?php echo "</p>"; ?>
```

The preceding code snippet outputs the following to the browser:

```
This is some text.
Some of this text is static, but this sure isn't!
This text is enclosed in paragraph tags that were generated by PHP.
```

If you write a PHP script that holds nothing but PHP, you don't have to end the PHP delimiter. You only have to mark the ending of a PHP code block, if you are going to write something that is *not* PHP in the file.

## Using echo

Take an extra look at the use of `echo` in the preceeding code examples. PHP's `echo` is a so-called *language construct*—the basic syntactic units PHP is made of. The `echo` statement is probably the most common approach for outputting text from PHP to the browser. That is all `echo` does. It sends output to the browser.

Notice that the output strings are delimited with double quotes in the preceding code example. The initial double quote indicates the beginning of a string of characters. The second double quote marks the end of the string to output. In PHP, you must delimit any strings you are using in your code. The string delimiters tell PHP when a string of characters begin and end, something PHP needs to know in order to process your code.

---

■ **Note** *String* is a geeky word for "text." Because computers are not human, they don't really see texts, much less words. They see *strings* of characters.

---

## What Is a Variable?

A variable is a keyword or phrase that acts as an identifier for a value stored in a system's memory. This is useful, because it allows us to write programs that will perform a set of actions on a variable value, which means you can change the output of the program simply by changing the variable, rather than changing the program itself.

### Storing Values in a Variable

It is quite straightforward to store a value in a variable. In one single line, you can declare a new variable and assign a value to it:

```
<?php
$myName = "Thomas";
$friendsName = "Brennan";
echo "<p>I am $myName and I have a friend called $friendsName.</p>";
```

If you type the preceding lines into your `test.php` file and load it in your browser, you should see an output such as the following:

```
I am Thomas and I have a friend called Brennan.
```

Perhaps you will notice that the preceding code holds nothing but PHP. Consequently, there is no need to mark the end of the PHP block with a PHP delimiter. You can add `?>` at the end, if you like; it'll make no difference.

### A Variable Is a Placeholder

Variables are used extensively in programming. It is a basic concept you must come to understand. There is an important lesson to be learned from the example preceding. When you read the PHP code, you see variable names:

```
echo "<p>I am $myName and I have a friend called $friendsName.</p>";
```

You can see the output from PHP in the browser. You can see that PHP replaces the variable names with string values. For example, when you see `$myName`, PHP sees *Thomas*. When you see `$friendsName`, PHP sees *Brennan*.

A variable is a placeholder for a specific value. PHP doesn't even notice the variable; it sees the value stored inside. Metaphorically, you could understand a variable as a container—a cup, for example. I have a cup right next to my computer, and I can put all sorts of things inside it: coffee, a pencil, or some loose change. PHP variables are like that. PHP sees what is contained, not the container.

---

■ **Note** In technical terms, PHP variables are *passed by value*, as opposed to *passed by reference*.

---

## Valid PHP Variable Names

In PHP, all variables must begin with a dollar sign character ($). There are some further restrictions on valid variable names, but if you simply use alphabetical characters only, you will encounter no problems with invalid variable names. So, avoid whitespace characters, numbers, and special characters such as !"#€%&/.

---

■ **Note** You can actually use numbers in variable names but not in initial positions. So, `$1a` is an invalid variable name, whereas `$a1` is perfectly valid.

---

# Displaying PHP Errors

On your journey toward learning PHP, you are bound to produce some errors. It is easy to think that you have done something bad when you have written some erroneous PHP. In a sense, it is, of course, bad. You would probably prefer to write perfect PHP from the very start.

In another sense, errors are a very good thing. Many such errors present a learning opportunity. If you really understand the cause of an error, you are less likely to repeat it, and even if you do repeat it, you can easily correct the error if you understand it.

PHP error messages are not always displayed—it depends on how your development environment is set up. If you write the following two lines of PHP at the beginning of your scripts, all error messages will be displayed. Let's produce an error:

```
<?php
//these two lines tell PHP to show errors in the browser
error_reporting( E_ALL );
ini_set( "display_errors", 1 );

//here comes the error
echo "This string never ends;
```

Do you see the error? There is only one string delimiter. To write valid PHP, you must wrap your strings in string delimiters, for example, double quotes. In the preceding example, the end delimiter is missing, so PHP cannot see where the output ends. If you run the code, you will see an error message in your browser, as follows:

---

**Parse error:** syntax error, unexpected $end, expecting T_VARIABLE or T_DOLLAR_OPEN_CURLY_BRACES or T_CURLY_OPEN in**/Applications/XAMPP/xamppfiles/htdocs/ch2/test.php** on line **4**

---

Error messages are friendly but not always as precise as you might prefer. When PHP is unable to process your code, an error is triggered. PHP will make an educated guess about what the problem might be. In the preceding example, PHP has encountered an "unexpected end" on line 4. There is a "bug" in your script. Please debug the script by adding the missing double quote.

I recommend you make a habit of forcing error messages to display and try to read all error messages you come across. If you encounter an error message you don't understand, you can always search the Internet for an explanation. A site such as `www.stackoverflow.com` is very likely to have an explanation for your particular error message.

# Creating an HTML5 Page with PHP

PHP is a wonderful language for creating dynamic HTML pages. With a tiny bit of PHP, you can create a valid HTML5 page with variable content in memory and have PHP output the created page to the browser. Let's make a bare-bones skeleton for a personal portfolio site. Create a new PHP file called `index.php` in `XAMPP/htdocs/ch2`:

```
<?php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );

$title = "Test title";
$content = "<h1>Hello World</h1>";
$page = "
<!DOCTYPE html>
<html>
<head>
<title>$title</title>
<meta http-equiv='Content-Type' content='text/html;charset=utf-8'/>
</head>
<body>
$content
</body>
</html>";
echo $page;
```

If you save and load `http://localhost/ch2/index.php` in your browser, you should see a well-formed HTML5 page with a title and a heading. It's a good habit to inspect the source code of your PHP-generated HTML pages. Do it, and you should see that the variables have been replaced by their corresponding values by PHP. The HTML source code should look like the following:

```
<!DOCTYPE html>
<html>
<head>
<title>Test title</title>
<meta http-equiv='Content-Type' content='text/html;charset=utf-8' />
</head>
<body>
Hello World</h1>
</body>
</html>
```

## Including a Simple Page Template

Creating a valid HTML5 page with PHP is a very, very common task. You should have few problems understanding the preceding code. Let's try to create the same output in a way that's easier to reuse in other projects. If you can reuse your code in other projects, you can develop solutions faster and more efficient. Let's keep the HTML5 page template in a separate file.

Create a new folder called `templates` in your existing PHP project. Create a new PHP file called `page.php` in the templates folder, as follows:

```
<?php
return "<!DOCTYPE html>
<html>
<head>
<title>$title</title>
<meta http-equiv='Content-Type' content='text/html;charset=utf-8'/>
</head>
<body>
$content
</body>
</html>";
```

## Returning Values

The `return` statement in PHP is very useful. It simply stops execution of the script. Any value indicated immediately after the `return` statement will be returned. In the preceding example, a valid HTML5 page will be returned.

## Including the Template

To use the template from your index, you will have to load the script into PHP's memory. You can do that with another PHP statement: `include_once`. Update your `index.php` file, as follows:

```
<?php
//complete code for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
$title = "Test title";
$content = "<h1>Hello World</h1>";
//indicate the relative path to the file to include
$page = include_once "templates/page.php";
echo $page;
```

The output of the preceding code will be identical to that you had when you first created the page. There are no functional changes, but there are some aesthetic changes in code architecture. A reusable page template is now kept in a separate file. The template is included into `index.php`, when needed. We're really splitting different parts of the code into different files. The result is that more of the code becomes readily reusable in other projects. This process of separating different parts is also known as *separation of concerns*.

# Commenting Your Code

It can be very helpful for your learning process to write comments in your code. Such comments should remind you what the code does and why. Explaining the code to yourself in your own terms will speed up your learning process. Also, should you ever find yourself working with a team of developers, code comments will help you collaborate effectively, because you can write notes to your codevelopers as code comments.

## Block and Single-Line Comments

You can write comments in your code. Comments can remind you, and anybody else reading your code, what different parts of the code does. You have to clearly delimit comments, so PHP will not try to interpret comments as if they were actual production code. You should know two ways of writing code comment in PHP: block and single-line comments.

```
<?php
//this is a single-line comment
/*
This is a comment block
It may span across
several lines
*/
```

# Avoiding Naming Conflicts

You will soon find yourself writing PHP projects with hundreds of lines of code. You will need many variables, and each one must be named uniquely and meaningfully. You must avoid naming conflicts, as in the following example:

```
<?php
$title = "Welcome to my blog";
/*
hundreds lines of code later
*/
$title = "Web developer";
```

See the problem? Initially, a variable named `$title` is used to indicate the value of an HTML page's `<title>` element. Much later in the same system, a variable also named `$title` is used to store a job title. A system with such variable names is vulnerable. You are likely to see unwanted system behavior when you use that variable. A better solution would be to clearly indicate the context of the `$title`. One approach is to use an object.

```
<?php
$pageData = new StdClass();
$pageData->title = "Welcome to my blog";
/*
hundreds lines of code later
*/
$jobData = new StdClass();
$jobData->title = "Web developer";
```

You can create a new standard PHP object by using PHP's native `StdClass`. A PHP object is just like a variable in that it can store values. One normal PHP variable can store one value. One object can store as many values as you need. Each individual value can be stored as a unique object property.

In the preceding code example, you can see two different objects, each with a title property. It should be clear that the `$pageData->title` is different from `$jobData->title`, even if both properties are named `title`.

The object provides a context, and that will make it easier for you to use the right title in the right place in your code. You can use objects to organize your code into meaningful units that belong together. You could say that an object and its properties are much like a folder and the files inside.

---

■ **Note** There is more to objects than this—a lot more. Using objects in your code is a de facto standard for dealing with complexity in systems, without introducing unnecessary complexity in your code. You will learn much more about programming with objects throughout the book.

---

## The Object Operator

Objects can be used as namespaces for properties, to avoid naming conflicts, by providing a clear context. To get values from an object property, you have to specify two things: which object and which of its properties to get. To that end, you use PHP's object operator. The general syntax is like the following:

```
$objectName->propertyName;
```

PHP's object operator looks like an arrow. It indicates that you are getting a particular property inside a specific object.

## Using a StdClass Object for Page Data

Let's refactor `index.php` and the page template with an object, to prevent annoying naming conflicts. Here are some changes for `index.php`:

```
<?php
//complete code for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
$pageData = new stdClass();
$pageData->title = "New, object-oriented test title";
$pageData->content = "<h1>Hello from an object</h1>";
$page = include_once "templates/page.php";
echo $page;
```

You will also have to update `templates/page.php`, so it uses the newly created object and its properties in the right places:

```
<?php
return "<!DOCTYPE html>
<html>
<head>
<title>$pageData->title</title>
<meta http-equiv='Content-Type' content='text/html;charset=utf-8' />
</head>
<body>
$pageData->content
</body>
</html>";
```

Save the files and reload `index.php` in your browser. Expect to see the changed values in the `<title>` and `<body>` elements.

### Object Properties Are Like Variables

Plain PHP variables are simple placeholders for values. Objects are clearly more complex, in that one object can hold several values. The preceding `StdClass` object holds two separate values. Each value is stored in a unique object property. Object properties behave like PHP variables. They are simple placeholders for values. One object can have many properties. Each property can contain one value. In the preceding example, you can see that you have to specify both object and property, in order to get to the value.

PHP properties can be understood as cups. Their contained values can be understood as the coffee (or whatever) inside a cup. Metaphorically, you could see an object as a tray and its properties as a few cups standing on the tray. To get your coffee, you have to get the right cup from the right tray.

In the preceding code example, you can see that you use the `title` property of the `$pagedata` object inside the `<title>`, and you use the `content` property of the `$pageData` object inside the `<body>` element.

## Page Views

A personal portfolio site is likely to have a few different pages. Perhaps one page about your skills and educational background, and another page with links to examples of your work.

Because you're making a dynamic web site, you don't have to create two complete HTML pages. You can use your page template to display two different *page views*. A page view is something that looks like an individual page. One page view may be composed of several smaller views. You could think of a page view as a Lego house and a view as a Lego brick: the smaller parts are combined to build something bigger.

Let's keep all views in one folder. Create a new folder called `views` inside your existing project folder. Create a new file, `views/skills.php`.

```
<?php
return "<h1>Skills and educational background</h1>
<p>Read all about my skills and my formal training</p>
";
```

That is the complete file. It is a quite small view at this point. It is often a good idea to begin small, when you are developing code. Any error that might creep in will be easier to spot in fewer lines of code. You need another small view in `views/projects.php`.

```
<?php
return "<h1>Projects I have worked on</h1>
<ul>
<li>Ahem, this will soon be updated</li>
</ul>";
```

### Making a Dynamic Site Navigation

You have to show the right view at the right time. You can make a global, persistent site navigation, i.e., a navigation that will be the same on every page of the web site. Because PHP can include files, you can simply keep the code for the navigation in one file and include it in every script that needs it. An obvious advantage is that you can change the navigation in one file, and that change will be reflected on every site page, however many there are. Create a new file in `views/navigation.php`.

```
<?php
return "
<nav>
    <a href='index.php?page=skills'>My skills and background</a>
    <a href='index.php?page=projects'>Some projects</a>
</nav>
";
```

Notice that the entire navigation string is delimited with double quotes. use single quotes to delimit the `href` attribute values Because of that, you cannot use double quotes inside the navigation string. A third double quote would trigger a PHP error. So, you.

To see the navigation on the index page, you must include it from `index.php`.

```
<?php
//complete code for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
$pageData = new stdClass();
//changes begin here
$pageData->title = "Thomas Blom Hansen: Portfolio site";
$pageData->content = include_once "views/navigation.php";
//end of changes
$page = include_once "templates/page.php";
echo $page;
```

Save and run this code, and you should see a page with a navigation. Don't expect to see any of the views just yet.

## Passing Information with PHP

Passing data is what separates dynamic web pages from static ones. By customizing an experience based on the user's choices, you're able to add an entirely new level of value to a web site. You can pass information to PHP through *URL variables*. A URL variable is simply a variable declared in the URL. You can see two URL variables in the navigation. Take a close look at the `href` attributes in the navigation `<a>` elements.

```
index.php?page=skills
index.php?page=projects
```

The `href` indicates that clicking the navigation item will load `index.php` and encode a URL variable named `page`. If you click one link, the URL variable named `page` will get a value of `skills`. If you click the other link, `page` will get a value of `projects`.

PHP can access URL variables and use them, for example, to load the right page view at the right time. URL variables are the lifeblood of dynamic sites.

## Accessing URL Variables

To access URL variables, you use the `$_GET` superglobal array. Here's how you might use it in `index.php`:

```
<?php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
$pageData = new stdClass();
$pageData->title = "Thomas Blom Hansen: Portfolio site";
$pageData->content = include_once "views/navigation.php";
//changes begin here
$navigationIsClicked = isset($_GET['page']);
if ($navigationIsClicked ) {
    $fileToLoad = $_GET['page'];
    $pageData->content .= "<p>Will soon load $fileToLoad.php</p>";
}
//end of changes
$page = include_once "templates/page.php";
echo $page;
```

That was quite a mouthful! PHP can access URL variables through `$_GET`. To access the value of the URL variable named `page`, you write `$_GET['page']`. There will be a URL variable named `page` only when a user has clicked a navigation item.

## Using isset( ) to Test If a Variable Is Set

If you try to use a variable that does not exist, you will trigger a PHP error. So, before you try to access a variable, you have to be sure that the variable is set. PHP has a language construct to that end. You have already seen it in action.

```
$navigationIsClicked = isset($_GET['page']);
```

The `isset()` function will return `TRUE`, if the variable inside the parentheses is set. So, if a user has clicked a navigation item, `$navigationIsClicked` will be `TRUE`; if not, it will be `FALSE`.

If `$navigationIsClicked` is `TRUE`, then declare a PHP variable named `$fileToLoad`, to store the value of the URL variable named `page`. Next, add a string to the `$pageData->content` property, to display the value of the URL variable named `page`. Save and run the code. Once loaded in your browser, click the "My skills" navigation item. That should produce the following output in your browser:

```
Will soon load skills.php
```

If you click the other navigation item, you can see the output change. You are seeing that *output changes dynamically, according to how the user interacts* with the site.

## $_GET, a Superglobal Array

PHP can access URL variables through a so-called *superglobal array* called `$_GET`. PHP has a few other superglobal arrays for other purposes. With `$_GET`, you can access URL variables by their name. In the navigation, you have two `<a>` elements. Clicking either one will encode a unique value for a URL variable named `page`.

You can see a URL variable in the browser's address bar in Figure 2-1. Notice how the value of the URL variable page is represented in the output? To get the value of a URL variable with PHP you write

```
$_GET['the name of the url variable'];
```

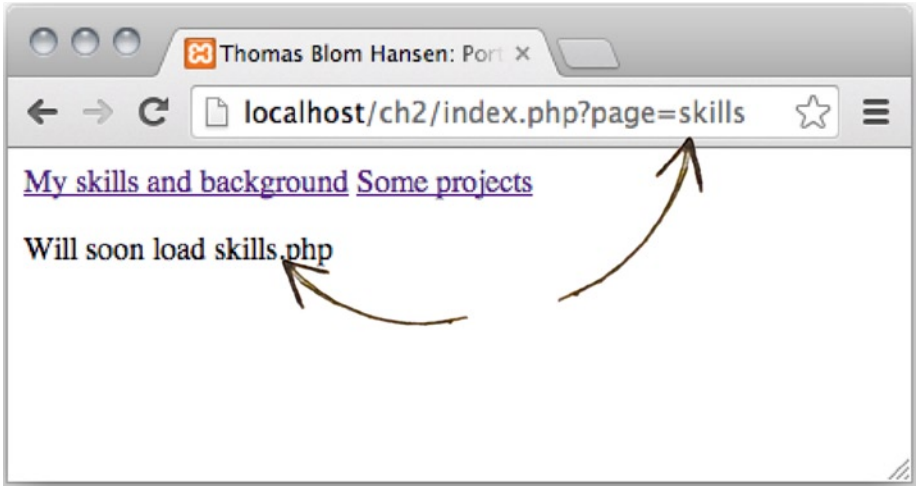


***Figure 2-1.*** *A URL variable in action*

## Including Page Views Dynamically

The dynamic site navigation is nearly complete. It works perfectly, except that page views are not loaded when navigation items are clicked. Let's change that, by updating the code in `index.php`, as follows:

```
<?php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
$pageData = new stdClass();
$pageData->title = "Thomas Blom Hansen: Portfolio site";
$pageData->content = include_once "views/navigation.php";
$navigationIsClicked = isset($_GET['page']);
if ($navigationIsClicked ) {
    $fileToLoad = $_GET['page'];
    //change one line to load page views dynamically
    $pageData->content .=include_once "views/$fileToLoad.php";
}
$page = include_once "templates/page.php";
echo $page;
```

Save the changes and reload `index.php` in your browser. You will see that the output changes when you click a navigation item. The corresponding page view will be loaded and displayed. It is your `index.php` file that dynamically changes how it looks. The URL variable named page will determine which file is loaded.

That is it! That is a basic, dynamic site with a persistent, global navigation.

## Concatenation

Did you notice the `.=` in the code above? It is PHP's *incremental concatenation operator*, and it is a little different from `=`, the basic *assignment operator*.

Here's an example to illustrate the differences:

```
<?php
$test = "<p>Hello ";
$test = "world</p>";
echo $test;
$concatenationTest = "<p>Hello ";
$concatenationTest .= "world</p>";
echo $concatenationTest;
```

The HTML source code output from the above example would clearly show the difference between assignment and incremental concatenation.

```
world</p>
<p>Hello world</p>
```

The assignment operator assigns a new string value to a variable and overwrites any previous strings in the process. So, the initial `"<p>Hello "` in the variable `$test` is overwritten.

The incremental concatenation operator merges the existing string in `$concatenationTest` with a new string. Incremental concatenation adds the new string at the end of the existing one.

## Strict Naming Convention

It is great to see your first dynamic site working, isn't it? It works, and it depends on a strict naming convention. The navigation items encode different values for a URL variable named `page`. The corresponding page view file must be named identically and be saved inside the `views` folder.

| href | URL variable | view file |
|---|---|---|
| index.php?page=**skills** | page=**skills** | views/**skills.php** |
| index.php?page=**projects** | page=**projects** | views/**projects.php** |

## Displaying a Default Page

The dynamic navigation works wonderfully, but it has one flaw: there is no default page view displayed when a user navigates to `index.php`, in which case, the URL variable named `page` does not have a value. It is easy to change in `index.php`. You simply have to change the `if` statement a tiny bit.

```
//partial code for index.php
if ($navigationIsClicked ) {
    $fileToLoad = $_GET['page'];
} else {
    $fileToLoad = "skills";
}
$pageData->content .=include_once "views/$fileToLoad.php";
```

The significant change is that `$fileToLoad` gets its value from the URL variable `page`, if that is set. If it is not set, `$fileToLoad` will have a default value of `skills`. Once `$fileToLoad` has a value, you can use it to load either the page view requested by a user or the default page view about "My skills."

# Validating Your HTML

The process of generating HTML pages is a bit abstract. It is easy to assume that everything is perfect, if the right page view is displayed at the right time. If you see the right action, your PHP scripts works perfectly. But that does not mean your HTML is perfectly valid. Dynamic web pages should conform to web standards, just as static HTML pages should. You should validate the generated HTML just as you would normally validate any other HTML.

---

■ **Note** You could load a dynamic page in your browser and view the generated HTML source code through your browser. When you see the generated HTML source code, you can select it all, copy it, and paste it into an online HTML validation service. I usually use http://validator.w3.org/#validate_by_input.

---

# Styling the Site with CSS

When the HTML of all page views validates, you can start styling your site with CSS. You do it exactly as you would normally style a static HTML site: create an external style sheet with style rules for the visual design of your site. To do that for the portfolio site, you could create a new folder, called `css`, in your project folder. Create a new file called `layout.css` in the `css` folder:

```
nav {
    background-color: #CCCCDE;
    padding-top: 10px;
}
nav a{
    display:inline-block;
    text-decoration:none;
    color: #000;
    margin-left: 10px;
}
nav a:hover{text-decoration: underline;}
```

You can change or add any style rules you prefer. The preceding `css` is just to get you started. You will probably want to style all your dynamic HTML pages, so why don't you build this functionality into the page template? You simply have to add one new placeholder for `<link>` elements pointing to external style sheets. Let's update `templates/page.php`:

```
<?php
return "<!DOCTYPE html>
<html>
<head>
<title>$pageData->title</title>
<meta http-equiv='Content-Type' content='text/html;charset=utf-8' />
$pageData->css
</head>
```

```
<body>
$pageData->content
</body>
</html>";
```

Notice that the new property is used as placeholder for `<link>` elements referencing external style sheets. To use the updated page template, you must update `index.php` and declare a value for the new property:

```
<?php
//partial code listing for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
$pageData = new stdClass();
$pageData->title = "Thomas Blom Hansen: Portfolio site";
$pageData->content = include_once "views/navigation.php";
//one line of code added here
$pageData->css = "<link href='css/layout.css' rel='stylesheet' />";
```

Save your files and load `index.php` in your browser. Expect to see your style rules take effect.

# Declaring a Page_Data Class

Sometimes, it can be quite useful to use an internal, embedded style sheet to supplement your external style sheets. You can easily update the page template with a placeholder for a `<style>` element. Update `templates/page.php`.

```
<?php
return "<!DOCTYPE html>
<html>
<head>
<title>$pageData->title</title>
<meta http-equiv='Content-Type' content='text/html;charset=utf-8' />
$pageData->css
$pageData->embeddedStyle
</head>
<body>
$pageData->content
</body>
</html>";
```

It would be equally easy to declare a property value from `index.php`, but let's do something different. The problem is that sometimes you don't need any embedded `<style>` element and sometimes you do.

Now that your template has a placeholder for embedded CSS, that property must always have a value. You don't want to waste time declaring a value for a redundant `<style>` element, so let's make a more intelligent solution. Let's take the next step toward object-oriented programming and create a custom class for page data. Create a new folder, called `classes`, in your project folder. Create a new file, called `Page_Data.class.php`, in the `classes` folder.

```
<?php
class Page_Data {
    public $title = "";
    public $content = "";
    public $css = "";
    public $embeddedStyle = "";
}
```

That's it—a custom-made class with predefined empty string values for those properties required by the page template. The keyword `class` indicates that the name following is a custom class name.

PHP class names can begin with letters or an underscore. It is conventional to begin with an uppercase letter. If the class name is a compound word, it is common to separate individual words with an underscore character and begin the next word with an uppercase letter, for example, `My_Custom_Class_Name`.

I usually keep each of my custom PHP class definitions in a separate file named like the class name. I also like to end the file name with an optional suffix of `.class.php`. So, the file for `My_Custom_Class_Name` would be called `My_Custom_Class_Name.class.php`.

Following the class name comes a set of curly braces, to delimit a code block for the class definition. Take a look at the code block for `Page_Data`. Inside the curly braces, four properties are declared, using the `public` keyword. The effect is that the `Page_Data` class will be born with four default properties, each with a default value declared.

## Classes Make Objects

You can use the new class definition from `index.php`. It will be a tiny change. Update `index.php` as follows:

```
//Partial code listing for index.php
include_once "classes/Page_Data.class.php";
$pageData = new Page_Data();
//delete or comment out the previous object
//$pageData = new stdClass();
//no changes below this point
```

Load `http://localhost/ch2/index.php` in your browser, to test your code. Your site should work exactly as before. You know you have made some changes to the code, but these changes are not visible to ordinary users. You have *refactored* the code.

To use a custom-made class, you must first include the class definition. Next, you must use the `new` keyword to create a new object with the class definition. The `Page_Data` class enables us to keep a placeholder for embedded styles in the page template and only assign an actual value to that property whenever you need a page with an embedded `<style>` element.

## Highlighting Current Navigation Item with a Dynamic Style Rule

You have a page template and a `Page_Data` object, and they are prepared to work with embedded styles. You would usually want to keep your style rules in an external style sheet. With dynamic web sites, this convention still applies, but because you can embed styles into `index.php`, you can quite easily work with dynamic styles. Most of the time, good ole external style sheets do the job just fine. But there are a few cases in which dynamic styles are quite powerful. You can use a dynamic style rule to highlight the current navigation item. It's really quite simple, once you get the idea. Update `index.php` as follows:

```
<?php
//complete code listing for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
include_once "classes/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "Thomas Blom Hansen: Portfolio site";
$pageData->content = include_once "views/navigation.php";
$pageData->css = "<link href='css/layout.css' rel='stylesheet' />";
$navigationIsClicked = isset($_GET['page']);
```

```
if ($navigationIsClicked ) {
    $fileToLoad = $_GET['page'];
} else {
    $fileToLoad = "skills";
}
$pageData->content .=include_once "views/$fileToLoad.php";

//new code below: dynamic style added below
$pageData->embeddedStyle =  "
<style>
nav a[href *= '?page=$fileToLoad']{
    padding:3px;
    background-color:white;
    border-top-left-radius:3px;
    border-top-right-radius:3px;
}
</style>";
$page = include_once "templates/page.php";
echo $page;
```

Save your work and load the index into your browser. You should see a simple, tabbed navigation with the current navigation item clearly highlighted. In the following example, I have clicked the "Some projects" navigation item. Users can clearly see which page is displayed, because of the highlighted navigation tab. You can see my example in Figure 2-2.

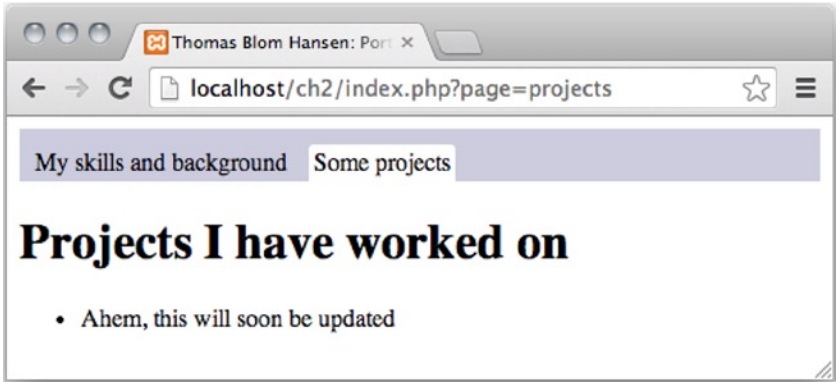


***Figure 2-2.*** *Current navigation item highlighted with a dynamic style rule*

You know some PHP, so you know all this happened because the URL variable named page has a value of `skills`. It may not be much of a visual design just yet, but it should be enough for you to apply all your CSS skills to design a beautiful user interface.

## Taking a Closer Look at the Dynamic CSS

The used CSS attribute selector is not so commonly used. Let's take a closer look at it.

```
nav a[href *= '?page=$fileToLoad']
```

First of all, notice the PHP variable `$fileToLoad`. It is a placeholder for an actual value. For example, when a user clicks the "Some projects" navigation item, `$fileToLoad` will have a value of `projects`, because the URL variable named `page` has a value of `projects`. You can see it in the address bar on the browser screenshot in the preceding figure. So when the browser interprets the CSS, it really sees the following:

```
nav a[href *= '?page=projects']
```

The selector tells the browser something such as "look for a `<nav>` element inside which you find an `<a>` element with an `href` attribute that contains the string `?page=projects`."

Pretend you are the browser. Look in your `<nav>` element. Look for an `<a>` element that has an `href` attribute that contains `?page=projects`.

You—and the browser—will find only one such `<a>` element. The browser will apply a special style rule to that `<a>` element that will essentially highlight it.

# Summary

Now you have seen how to use a little basic PHP to build a very dynamic site. Your learning process will probably benefit from a bit of experimenting at this point.

You could try to complete a personal portfolio site. Add however many page views you see fit and update your navigation accordingly.

You could try to create some more comprehensive page views. In the process, you will gradually become more comfortable with the dynamic site structure where page views are returned to be displayed on `index.php`.

You could use your existing CSS skills to develop a consistent web site design for your portfolio. It will be a very good exercise to use your existing HTML and CSS skills in this new context of dynamic sites. It will probably be relatively easy for you, because the portfolio site is quite simple. It is a good idea to get this exercise while the site you are working on is simple. The kinds of dynamic sites you develop will soon be anything but simple.

When you feel ready for it, flip the page to learn about HTML forms, PHP functions, and conditional statements, all of which you will encounter in Chapter 3.

CHAPTER 3

# Form Management

In Chapter 2, we built a dynamic, personal portfolio site. In the process, you saw how to encode URL variables with `<a>` elements and how to access such URL variables using the `$_GET` superglobal. Passing data is what separates dynamic web pages from static ones. By customizing an experience based on the user's choices, you're able to add an entirely new level of value to a web site.

Now that you have seen a little PHP and written a basic dynamic site, you're ready to go deeper into URL variables. HTML `<form>` elements are commonly used to create interfaces that allow users to interact with your dynamic site. You have to learn how to deal with such HTML forms. In this chapter, you'll learn the following:

- What HTML forms are and how to create them
- What superglobal arrays are and how to use them
- How to encode URL variables with HTML forms using the `GET` method
- How to encode URL variables with HTML forms using the `POST` method
- How to write a dynamic PHP quiz
- When to use `if-else` conditional statements
- What a named function is and how to write one
- What an American western film can teach you about clean code
- Why code really is poetry

## What Are Forms?

HTML forms allow visitors to interact with a site. Figure 3-1 shows Google's search form. When a user visits `www.google.com`, types a search term in to the text input field, and clicks Google Search, Google performs the search.

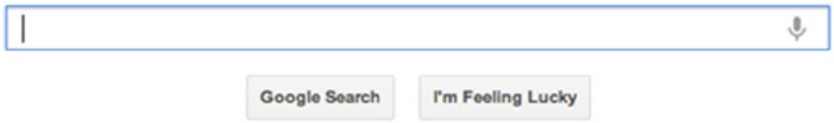


***Figure 3-1.*** *Search form from `www.google.com`*

Another kind of form you must have come across is a login form, through which registered users can log in and enter a restricted area. You may have seen such forms when you log in to your Facebook account, your bank account, or your Gmail account. The login in Figure 3-2 is from Facebook.

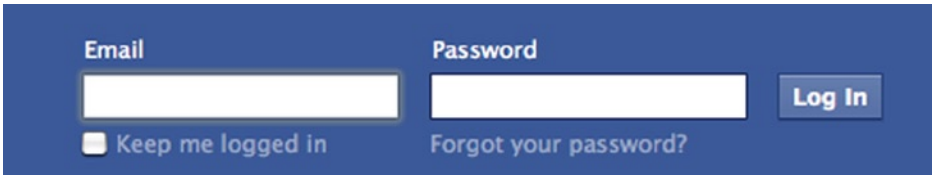


***Figure 3-2.*** *Login form from* `www.facebook.com`

A final familiar example could be the star rating system. You may have come across a star rating system if you have bought a book from an online bookstore. Figure 3-3 shows the star rating form from Amazon.

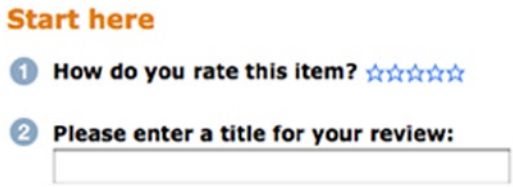


***Figure 3-3.*** *Star rating form from* `www.amazon.com`

If you are going to work with web development or web design, you will surely get to work with developing and designing usable, functional forms. Because web forms are the interface between a system and its users, developing and designing web forms is extremely important.

# Setting Up a New PHP Project

Learning requires repetition, so let's repeat some of the lessons learned in the previous chapter. Create a new project folder, called `ch3`, in the `XAMPP/htdocs` folder. Inside `ch3`, you'll need copies of the `templates` and `classes` folders, and the PHP scripts inside, from the previous project. Create an empty folder called `views`. Open Komodo Edit and create a new `index.php` file in `ch3`. Make sure to set Format to All Files when you save the file. Figure 3-4 illustrates how.

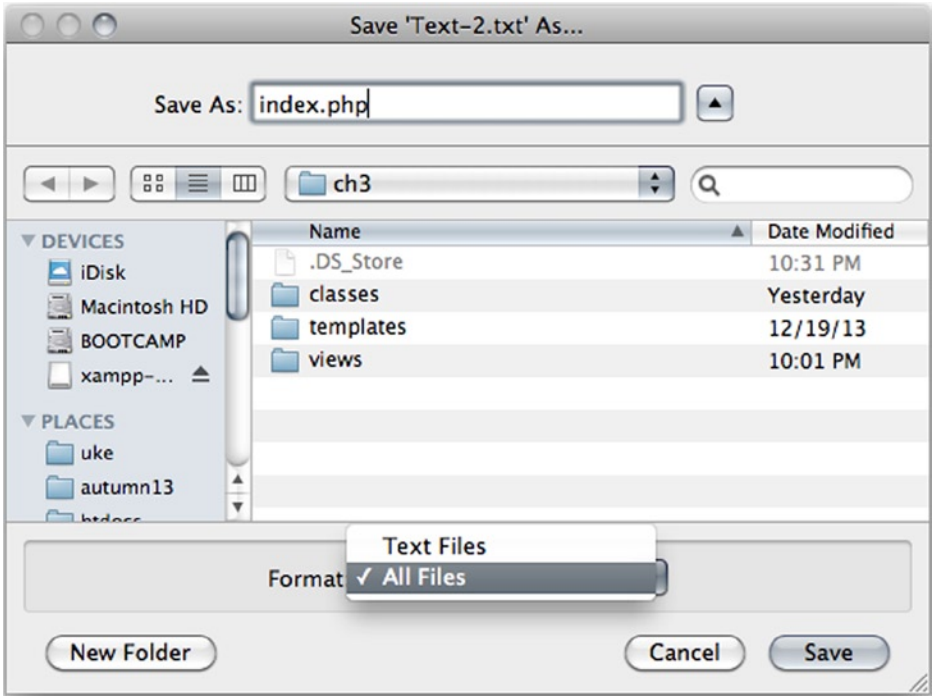


***Figure 3-4.*** *Save a new file as* `index.php` *with Komodo Edit*

You need some PHP inside `index.php` for anything much to happen. You can begin by outputting a simple HTML page. Notice that you should reuse `classes/Page_Data.class.php` and `templates/page.php`, without changing a single line of code inside either scripts. When you know PHP, you don't have to solve the same task several times. Just solve it once and code it to be reused, as in the following:

```
<?php
//code listing for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
include_once "classes/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "Building and processing HTML forms with PHP";
$pageData->content = "<nav>will soon show a navigation...</nav>";
$pageData->content .= "<div>...and a form here</div>";
$page = include_once "templates/page.php";
echo $page;
```

## Seeing for Yourself

To check if you typed everything correctly, you can save `index.php` and navigate your browser to `http://localhost/ch3/index.php`. You will probably see the expected output:

```
will soon show a navigation...
...and a form here
```

There is no Zen master to prod you with a stick, but I have some questions for you. Your answers will indicate what you have learned so far. If you're in doubt, you can consult Chapter 2 for explanations.

- What does `include_once` do?
- How can `$pageData->title` change the `<title>` of the generated HTML page?
- What does `.=`mean? What is the technical name for it?
- What happens when we `echo $page`?

## Creating a Dynamic Navigation

You will create two different forms. You will require a site navigation to navigate between those forms. Create a new file `ch3/views/navigation.php`, as follows:

```
<?php
//code listing for views/navigation.php
return "
<nav>
    <a href='index.php?page=search'>Search on bing</a>
    <a href='index.php?page=quiz'>Dynamic quiz</a>
</nav>
";
```

Just as in Chapter 2, you create a PHP script that simply returns a small snippet of HTML. In `index.php`, you will use some PHP to stitch a selection of small HTML snippets together, to generate a well-formed, dynamic HTML page. Following, `index.php` is updated to display the navigation:

```
<?php
//code listing for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
include_once "classes/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "Building and processing HTML forms with PHP";
//change this one line below only
$pageData->content = include_once "views/navigation.php";
$pageData->content .= "<div>...and a form here</div>";
$page = include_once "templates/page.php";
echo $page;
```

## Creating Page Views for the Form

You can follow the naming convention for page views from Chapter 2, because it seems to provide a solid code architecture for dynamic web sites. This way of organizing and naming page views can give you a mental framework for building dynamic sites. When you have the framework internalized, you'll know which files you need to develop the site you want to develop. You don't have to reinvent a good dynamic code architecture every time you make a new site.

The navigation described in the preceding section has links to pages called *search* and *quiz*. So, we will have to create two new PHP files in the `views` folder.

| href | url variable | view file |
|---|---|---|
| index.php?page = **search** | page = **search** | views/**search**.php |
| index.php?page = **quiz** | page= **quiz** | views/**quiz**.php |

Create the two new files with Komodo Edit, as follows:

```
<?php
//code listing for views/search.php
return "will soon show the search form";

<?php
//code listing for views/quiz.php
return "quiz will go here";
```

## Displaying Page Views on index.php

To get `index.php` to display those page views when they are requested, you have to write a few extra lines of code almost identical to those you wrote in the `index.php` for the previous project, as follows:

```
<?php
//code listing for ch3/index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
include_once "classes/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "Building and processing HTML forms with PHP";
$pageData->content = include_once "views/navigation.php";
//changes begin here
//$pageData->content .= "<div>...and a form here</div>";
$navigationIsClicked = isset($_GET['page']);
if ( $navigationIsClicked ) {
    $fileToLoad = $_GET['page'];
} else {
    $fileToLoad = "search";
}
$pageData->content .=include_once "views/$fileToLoad.php";
//no changes below
$page = include_once "templates/page.php";
echo $page;
```

The code essentially tells PHP to load any view requested by a site visitor. If no navigation item is clicked, we will display `views/search.php`. You can test that your code works by loading `http://localhost/ch3/index.php` in your browser.

## Spending Your Time Wisely: Conventions and Reuse

You have surely noticed that this dynamic site works very much like the dynamic site from Chapter 2. We have reused a few files, such as the `Page_Data` class and the page template. We couldn't reuse `index.php` or `navigation.php` exactly as they were in Chapter 2, but this project is built with the same conventions.

Reusing code is a good idea, because this allows you to develop solutions much faster. If you have scripts that work in one project, you can pretty much trust them to do the same in other projects. Hence, code reuse decreases debugging time.

There will always be parts you can't easily reuse, such as the navigation. But if you get into the habit of creating dynamic navigations in much the same way across different projects, you'll be able to develop new dynamic navigations quickly and painlessly. So, when you can't reuse code as is, perhaps you can reuse the principles underpinning the code you know works.

# A Super-Simple Search Form

HTML forms are created with `<form>` elements. There are a number of other HTML elements that are made specifically for forms. Perhaps the most essential one is the `<input>` element. Let's create an example in `views/search.php`, as follows:

```
<?php
return "
<form method='get' action='http://www.bing.com/search'>
    <input type='text' name='q' />
    <input type='submit' value='search on bing' />
</form>
";
```

## Trying Your Search Form

Save your work and point your browser to `http://localhost/ch3/index.php` to see the form. You should expect to see something like Figure 3-5.

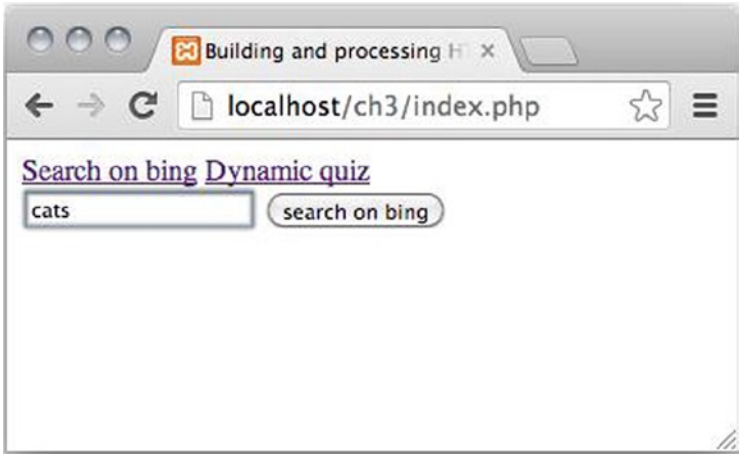


***Figure 3-5.*** *A simple search form completely unstyled*

You can type some search term in the text field and click the button. Your browser will load `bing.com`, and Bing will perform a search for whatever you typed in. I typed *cats*. When you have performed your search, take a look in your browser's address bar. You will find something like `http://www.bing.com/search?q=cats`.

---

■ **Tip** A <form> is like an <a> element; action is like the href attribute.

---

When you click the Submit button, your browser requests a new URL. This is just like clicking an `<a>` element. When you click an `<a>` element, your browser will request the resource indicated by its `href` attribute. When you submit a form, your browser will request the resource indicated by the `<form>` element's `action` attribute.

## Forms Encode URL Variables

The `<form>` element's `action` attribute is `http://www.bing.com/search`, but when you submitted the form, your browser requested `http://www.bing.com/search?q=cats`. Somehow, the form encoded a URL variable named `q` into the requested URL and set its value to `cats`. This is what forms can do: They can encode URL variables in HTTP requests.

As you saw in the previous chapter, URL variables can be accessed with PHP. URL variables are essential to dynamic web sites. Evidently, `www.bing.com` will perform a search, if a URL variable named `q` is set. Bing will search for whatever value `q` holds.

## A Name Attribute Declares the Name of a URL Variable

It is important to understand how the URL variable `q` was declared. Understand that, and you understand the essence of forms!

The URL variable `q` got its name because the `<form>` has an `<input>` element with a `name` attribute set to *q*. You can deduct that `<input>` elements with a `name` attribute will declare a URL variable if the `<input>` element is nested inside a `<form>`. This rule also applies to a small subset of other HTML elements commonly used with forms. You will see more such form-related elements in action throughout this book.

## The <input> Element and Some Common Types

Did you notice that `<input type='text' />`displays as a single-line text field and that `<input type='submit'/>` displays as a Submit button? There are many possible values for the input `type` attribute. In this book, you will see a small handful of input types. Once you can work with those, you should have no problems learning how to use the remaining input types.

---

■ **Note** HTML5 introduces quite a few new `<input>` types, not all of which are implemented in all major browsers yet. Many of these new types are immensely useful. See which browsers implement which features at http://caniuse.com/#search=input.

---

## Understanding the Method Attribute

So far, you have only seen URL variables that could be seen in the URL, in the browser's address bar. This kind of URL variable is encoded using the HTTP method `GET`. You have used such variables to create a dynamic navigation and a form that can perform a search at www.bing.com.

Any URL variable encoded with `GET` is limited to relatively few characters. The exact number varies from browser to browser, but the effective maximum seems to be about 2,000 characters. Because `GET` variables are evident from the URL, pages can be bookmarked and linked to. Therefore, `GET` variables are perfectly suited for site navigation.

# Named PHP Functions

Perhaps one the most powerful features of PHP is the ability to define and execute functions from within your code. A *function* is a named block of code you declare within your scripts that you can call at a later time. You will soon write a dynamic quiz using functions, but let's first look into the basics of named functions in PHP:

```
function functionName () {
    //function body
}
```

## The Basic Syntax for Functions

The basic format of a function requires that you first declare the function using the `function` keyword in front of the function's name. Function names can contain any alphanumeric characters and underscores, but they must not start with a number. The function name must be followed with a set of parentheses and a code block delimited by curly braces. Create a new PHP file with Komodo Edit in `ch3`. Call it `test-functions.php`. Declare a named function, as follows:

```
<?php
function p(){
    echo "<p>This paragraph came from a function</p>";
}
```

If you load `http://localhost/ch3/test-functions.php`, you will see no output. Many beginners would expect to see an output from the preceding code. But functions don't always behave as beginners assume. The code inside the function body will not be executed until the function name is explicitly *called*. You can add a function call in `test-functions.php` to execute the code, as follows:

```
<?php
//function declaration
function p(){
    echo "<p>This paragraph came from a function</p>";
}
//function call
p();
```

Run the code again, and you will see the expected output in your browser. A really interesting feature of functions is that they can be reused very easily. Simply call a function twice, and it runs twice. Let's do it.

```
<?php
//function declaration
function p(){
    echo "<p>This paragraph came from a function</p>";
}
//function calls
p();
p();
```

You can probably correctly guess that the code will output two `<p>` elements, each with the same text: `This paragraph came from a function`. What is more important is that you can see the difference between *function declarations* and *function calls*. The example has two distinct *function calls.* Because the function is called two times, it will run two times.

But this is a hideously ugly example. The function is very inflexible. It can only do one thing, i.e., output that one string. Let's make it a little smarter.

```
<?php
//function declaration
function p(){
    return "<p>This paragraph came from a function</p>";
}
//function calls
$output = p();
$output .= "<h1>Just some heading</h1>";
$output .=p();
echo $output;
```

Now this is much, much better! The significant change is that the function no longer has an `echo`. Instead, it returns a generated `<p>`. That has a consequence. In order to have the `<p>` echoed, you have to write that `echo` somewhere else. In the example, the `echo` now appears at the very end.

You may ask why that is any smarter? That's easy! Now that the function doesn't `echo`, you can manipulate the output further before you `echo` it, if you need to do that.

Actually, this is a good rule of thumb: *don't* `echo` *directly from a function*. It is much better to use a `return` statement. It is much better to have a single `echo` statement in one place in your code instead of having `echo` statements scattered all over the place.

## Using Function Arguments for Increased Flexibility

You might think that it is a little silly to have a function that always return a <p> with exactly the same content. You are quite right, of course; it is not very flexible. So, let's improve the `function p()` with a *function argument.*

```
<?php
//function declaration
function p( $content ){
    return "<p>$content</p>";
}
//function calls
$output = p( "I want this text in my first paragraph" );
$output .=p( "...and this in my second" );
echo $output;
```

Notice that I declare a variable called `$content` inside the parentheses in the function declaration. That is a function argument. The `$content` is used to store the content to be used in the returned <p> element. But how does `$content` ever get a value? That happens every time the function is called. Whatever argument is used when function `p()` is called will be stored temporarily in `$content`. Function arguments are extremely cool, because they allow you to write one function that can be reused with many different values. You will get to see many more examples of functions with parameters later in the book.

Did you consider the function name `p()`? I like my function names to be meaningful, and as this function always returns a <p> element, I thought `p()` would be a great name for it. You might consider using another name, such as `returnPTag()`. Good function names should be accurate and meaningful. Anyway, that was a little detour to explore some of the things you can do with PHP functions. Next, Let's write a dynamic quiz using functions.

# Creating a Form for the Quiz

Create a new PHP file, called `quiz-form.php`, in the `views` folder.

```
<?php
//complete code for views/quiz-form.php
return "<form method='post' action='index.php?page=quiz'>
        <p>Is it hard fun to learn PHP?</p>
        <select name='answer'>
            <option value='yes'>Yes, it is</option>
            <option value='no'>No, not really</option>
        </select>
        <input type='submit' name='quiz-submitted' value='post' />
    </form>";
```

## Showing the Quiz Form

To show the quiz form, you have to update the code in `views/quiz.php`, as follows:

```
<?php
$output = include_once "views/quiz-form.php";
return $output;
```

Save both files and point your browser to `http://localhost/ch3/index.php?page=quiz`, to see what you have created.

## Using <select> and <option>

The preceding form uses two HTML elements that may be unfamiliar to you. A `<select>` element is a good element to use when you want a user to choose between several predefined options. Options are displayed by nesting `<option>` elements. The structure is very similar to regular HTML lists such as `<ul>` and the corresponding `<li>`.

When a user selects an option, a new URL variable will be encoded into the request sent when the form is submitted. Notice that the name of the URL variable will be defined by the `name` attribute of the `<select>` element, and its value by the `value` attribute of the selected `<option>` element.

## The POST Method

Your first form used the `GET` method, but it is not the only possible HTTP method. There is another method called `POST`. The `POST` method has no defined maximum of characters—in fact, the `POST` method is not even limited to text. When using the HTTP `POST` method, it is possible to upload files through a form.

Also, HTTP `POST` variables are not directly visible in the URL. They are sent hidden from view. This makes HTTP `POST` the perfect candidate for forms that have to deal with larger amounts of content and forms with sensitive information. Because HTTP `POST` variables are not an integrated part of the URL, users cannot bookmark page views dependent on HTTP `POST` variables.

## Using the $_POST Superglobal

PHP has a native superglobal called `$_POST`. It can be used to access URL variables encoded with the `POST` method. You can use it to process the form when it is submitted. Update `views/quiz.php`, as follows:

```
<?php
//add a new variable and an if statement
$quizIsSubmitted = isset( $_POST['quiz-submitted'] );
if ( $quizIsSubmitted ){
    $answer = $_POST['answer'];
    $output = showQuizResponse( $answer );
} else {
    $output = include_once "views/quiz-form.php";
}
//keep the return statement as it was
return $output;
//declare a new function
function showQuizResponse( $answer ){
    $response = "<p>You clicked $answer</p>";
    $response .= "<p>
        <a href='index.php?page=quiz'>Try quiz again?</a>
    </p>";
    return $response;
}
```

You can load `http://localhost/ch3/index.php?page=quiz` in your browser, to see what the code does. It first checks to see if the form was submitted. Remember how the form had a Submit button?

```
<input type='submit' name='quiz-submitted' value='post' />
```

Well, if PHP can find a URL variable encoded with the POST method under the name quiz-submitted, you know that the form was submitted. If the form was submitted, you can get the selected answer using the superglobal $_POST. The answer is then passed as an argument to a new function, showQuizResponse(), which will simply return a string to indicate a user's answer and show an <a> element to restart the quiz.

The first form you made had the action attribute pointing to www.bing.com. The quiz form should reload the quiz page when the form is submitted. The URL to load the quiz is index.php?page=quiz, so the action attribute of the <form> references exactly that resource.

## $_POST Is an Array

You have read that $_GET is a superglobal array. $_POST is another superglobal array. But what is an array really? Basically, an array is a data type that can hold multiple items. Each item is stored under an index. I'd like to share an example. If you want, you can create a new PHP file and code the example, but it is not really necessary. I keep the following code in a file I called test-assoc-array.php:

```
<?php
//complete code for ch3/test-assoc-array.php
$my['name'] = "Thomas";
$my['year-of-birth'] = 1972;
$my['height'] = "193cm";

$out = "My name is " . $my['name'];
echo $out;
```

If you were to run http://localhost/ch3/test-assoc-array.php in your browser, you would see an output of "My name is Thomas." In the preceding example, $my is an array. You can see that it holds a collection of data stored in the same one variable. In order to get data from an array, you must use the right index. In the preceding example, "Thomas" is stored under the index ['name']. Arrays that store items under named indices are called associative arrays.

It can often be handy to inspect all items in an array. PHP has a function to do just that. It is called print_r(). Here's one way to use it:

```
<?php
//complete code for ch3/test-assoc-array.php
$my['name'] = "Thomas";
$my['year-of-birth'] = 1972;
$my['height'] = "193cm";

$out = "<pre>";
$out .=print_r($my, true);
$out .= "</pre>";
echo $out;
```

If you run this code, you can see every index of $my and its corresponding value. You will see something such as the following:

```
Array
(
    [name] => Thomas
    [year-of-birth] => 1972
    [height] => 193cm
)
```

You are looking at an the array as PHP sees it. You see one array with three named indices and their values. Arrays can be very helpful in your code, because they allow you to group items together. The $_GET and $_POST arrays are provided by PHP to give you access to all data encoded with the http methods GET or POST. The quiz example uses POST. I'd like you to inspect $_POST, so you can see for yourself what the form does when it is submitted. Update some code in the if-else statement in views/quiz.php, as follows:

```
<?php
//complete code for views/quiz.php
$quizIsSubmitted = isset( $_POST['quiz-submitted'] );
if ( $quizIsSubmitted ){
    $answer = $_POST['answer'];
    $output = showQuizResponse( $answer );
    //inspect the $_POST superglobal array
    $output .= "<pre>";
    $output .= print_r($_POST, true);
    $output .= "</pre>";
} else {
    $output = include_once "views/quiz-form.php";
}

return $output;

function showQuizResponse( $answer ){
    $response = "<p>You clicked $answer</p>";
    $response .= "<p>
        <a href='index.php?page=quiz'>Try quiz again?</a>
    </p>";
    return $response;
}
```

Save your work and point your browser to http://localhost/ch3/index.php?page=quiz. If you submit the quiz form, you can see all items in the $_POST array:

```
Array
(
    [answer] => yes
    [quiz-submitted] => post
)
```

You can see from the output that I selected yes before I submitted the form, and you can also see that the index quiz-submitted holds a value of post. What I'd really like you to see is that every form-related HTML element with a name attribute encodes a named index in $_POST when the form is submitted. Take a look in views/quiz-form.php. See how the Submit button encodes the index quiz-submitted, because it has a name attribute with a value of quiz-submitted. The <select> element encodes answer index, because it has a name attribute with a value of answer. PHP provides access to all encoded quiz form data through the $_POST superglobal array.

You will learn to use print_r() to debug your PHP code. This example was just an appetizer. Usually you don't want to show users the data passed to PHP in the background. You only inspect $_POST to see what PHP sees. Now that you have seen it, you can comment out the part of the code that inspects $_POST:

```
//partial code for views/quiz.php
//$output .= "<pre>";
//$output .=print_r($_POST, true);
//$output .= "</pre>";
```

## If and Else Explained

The quiz relies on form data encoded with `$_POST`. It also relies on `if-else` statements. You have already seen `if-else` statements used a few times. It is about time you get a more detailed explanation about such *conditional statements*. All conditional statements follow a certain pattern.

```
if ( Boolean expression ) {
    //code block
}
```

Inside the parentheses of any conditional statement, you must write an expression that evaluates to either `TRUE` or `FALSE`, 0 or 1. In computer science, such an expression is called a *Boolean expression*. If the expression evaluates to `TRUE`, the code block will run. If the expression evaluates to `FALSE`, the code block will not run.

Often, it comes in handy to do one thing if the expression is `TRUE` and something else if the expression is `FALSE`. That can easily be achieved with an `else` code block. The general form looks like the following:

```
if ( boolean expression ) {
    code block
} else {
    another code block
}
```

The first code block will run if the expression is `TRUE`. The second code block will run if it is `FALSE`. You can see it working in the quiz example. PHP can check if a user has submitted the form. If that happens, the PHP code will get the answer supplied by the user and generate a response. If not, PHP will simply return the HTML for displaying the quiz.

## Evaluating the Quiz Response

The quiz response right now provides no more feedback than what a parrot might supply. It simply repeats whichever answer the user selected. With PHP, you can do better. Open `views/quiz.php` in your editor and update `showQuizResponse()`, as follows:

```
function showQuizResponse( $answer ){
    //changes begin here
    $response = "<p>You clicked $answer";
    if ( $answer === "yes" ){
        $response .= " - I know exactly how you feel!";
    }
    $response .= "</p>";
    //end of changes
    $response .= "<p>
                  <a href='index.php?page=quiz'>Try quiz again?</a>
            </p>";
    return $response;
}
```

## The Identical Comparison Operator

You hadn't seen any `===` before the preceding code example. The triple equal signs make up PHP's *identical comparison operator*. It compares whether two values are identical. The identical comparison operator is commonly used to formulate a condition for an `if` statement. The condition used in the example in the preceding section really means "if the user's answer is identical to 'yes.'"

■ **Note** If you find PHP code examples elsewhere, you will often see ==. The double equal signs represent PHP's *equality comparison operator*. It is almost identical to the identical comparison operator. In most cases, you can use the two interchangeably. Learn more at http://php.net/manual/en/language.operators.comparison.php.

# Curly's Law: Do One Thing

Did you ever see the 1991 movie *City Slickers*? Yes, that feel-good western comedy featuring Billy Crystal. Jack Palance played Curly, a rugged, old cowboy who knew the secret of life and reluctantly shared it with Crystal's character, Mitch:

> *Curly*: Do you know what the secret of life is?
>
> *(holds up one finger)*
>
> *Curly*: This!
>
> *Mitch*: Your finger?
>
> *Curly*: One thing. Just one thing. You stick to that, and the rest don't mean shit.
>
> *Mitch*: But what is the "one thing"?
>
> *Curly*: (*smiles*) That's what *you* have to find out.

We can probably rest assured that Curly wasn't talking about principles of clean code. But, incidentally, he formulated a principle we can use to write clean functions. Every function should do one thing. Just one thing.

■ **Note** Jeff Atwood wrote a funny and interesting blog entry about applying Curly's law to clean code. Read it at http://blog.codinghorror.com/curlys-law-do-one-thing/.

Clean code is code that is nice to work with. If your functions do just one thing, they will be short. Short code is normally easier to read and understand than long code. If you can read and understand your code, it becomes much easier to find errors—and you *will* be making errors. Don't be surprised if you spend 50% of your development time chasing errors in your code.

In the earlier quiz example, you can see two clean functions, each doing just one thing. One function shows the quiz; the other function shows a response.

## Meaningful Names

Function and variable names are arbitrary. You can call them just about anything. In the quiz, we have the following:

```
if ( $quizIsSubmitted ){
    $answer = $_POST['answer'];
    $output = showQuizResponse( $answer );
} else {
    $output = include_once "views/quiz-form.php";
}
```

We could rename files, functions, and variables, without losing any functionality at all. We could have, for example:

```
if ( $a ){
    $c = d( $_POST['answer'] );
} else {
    $c = include_once "views/e.php";
}
```

The preceding code is bad, because the names are not expressive at all. The code would work, but it would be hard to read and understand. Reading such code requires a very attentive reader. But it is possible to write even worse code. You can use names that are misleading. The following code is still the quiz example, and it still works, but it has become really hard to read:

```
if ( $itIsLate ){
    $output = goToSleep( $_POST['answer'] );
} else {
    $output = include_once "views/coffee.php";
}
```

## Code Is Poetry

Strive for expressive, beautiful code. Strive for code that is easy-to-read. When you develop new solutions with code, you will spend a very significant part of your time reading your own code. Code is like poetry. You write it once, but read it many times. So write your code as if you were writing poetry: choose your words carefully.

Function names and variable names should be descriptive, accurate, and not excessively long. They should make your code easier to read and understand—not harder. Often, you will find that a function or a variable cannot be described accurately with a single word. I often use compound variable or function names, such as `$quizIsSubmitted`. I like to write such names using camel case: every new word is capitalized. Camel case is a quite common naming convention. I like it, because I find `$quizIsSubmitted` much easier to read than `$quizissubmitted`.

# Styling Forms

The first time you try to style a form, you may easily be a bit confused by the unfamiliar element names. But you can style forms and most related elements just as you would any other HTML elements. Usually, you can completely avoid using `id` and `class` attributes as CSS hooks. Your forms and their various attributes will give you plenty of opportunity to select just the element you're after, with CSS attribute selectors. Here's an example to get you started:

```
/*this selector will target the quiz form only*/
form[action='index.php?page=quiz']{
    position:relative;
    margin: 30px 10px;
}
/*select only <p> and <select> inside the quiz form*/
form[action='index.php?page=quiz'] p,
form[action='index.php?page=quiz'] select{
    display:inline-block;
}
```

# Exercises

Exercising what you learn is a good way to actually learn. Following are a few uncomplicated exercises that could help you internalize some of the PHP you have encountered. Some of these exercises might seem simple. You have written code to solve more complex tasks already. But copying code examples from a book is one thing. Writing your own code from scratch is something entirely different.

Take this opportunity to challenge yourself. You will probably find that you learn a lot from writing your own code to solve simple problems—perhaps at least as much as from working through the examples in this book.

First of all, you could try to create an external style sheet and link the `index.php` page to that style sheet. If you have forgotten how, consult Chapter 2 for hints.

You could also try to refine the dynamic quiz a little. How about changing `views/quiz.php` and have it output a meaningful response whenever a user selects the *no* option?

You could also write another HTML form that can calculate a person's body mass index (BMI), based on the person's height and weight. The formulae for calculating BMI follows. Your task is to create a form, on which users can input height and weight, and to write some PHP code to calculate BMI based on the input.

```
//metric
bmi = kg/ (2 * m)
//for UK and US readers
bmi = ( lb/(2 * in) ) * 703
```

Last but not least, you could try to write a form that converts money from one currency to another. If you want it to be really advanced, you could have a `<select>` element with a list of possible currencies to convert to.

# Summary

We covered a lot of ground in this chapter. You have learned how to write HTML forms. HTML forms encode URL variables when they are submitted. URL variables are passed from the browser to the web server with an HTTP request. You have learned how to process HTTP requests with URL variables encoded, using either `GET` or `POST` methods. And you have learned to organize your code with named functions. But most important, you have learned about Curly's law and how to apply it to enhance the beauty of your code.

CHAPTER 4



# Building a Dynamic Image Gallery with Image Upload

You know how to make a simple dynamic web site. You know how to write a form. You know how to access URL variables with `$_GET` or `$_POST`. I say it's time to put your new knowledge to good use. Let's build a dynamic image gallery with a form to allow users to upload new images to the gallery. In the process, you will learn quite a bit.

- Set up a dynamic site.
- Write named functions.
- Use `$_GET` and `$_POST` superglobal arrays.
- Iterate with a `while` loop.
- Use PHP's native `DirectoryIterator` class.
- Write custom object methods.
- Upload files with PHP's `$_FILES` superglobal array.
- Plan and code a class for uploading files easily.

## Setting Up a Dynamic Site

Make a new project folder, called `ch4`, for this chapter in `XAMPP/htdocs`. Copy the `templates` and `classes` folders and the PHP files inside from Chapter 3. Create new folders `css`, `views`, and `img`.

### Prerequisites: A Folder with Some Images

An image gallery should have some images. This image gallery will use JPEG images only. Prepare a small handful of JPEG images for your gallery. Save the images in the `img` folder.

We're sticking to a site architecture identical to that used in the previous chapters. This will make it easier to reuse code from previous projects. Reusing your own code will help you develop your solutions faster, which in the end, will make you a more valuable developer.

## Creating a Navigation

This site will have two main page views: one for displaying the gallery and one to show a form to allow users to upload new images. Because we know we'll need these two page views, we can prepare a site navigation with two navigation items. Create a new file in the `views` folder and call it `navigation.php`.

```
<?php
return "
<nav>
    <a href='index.php?page=gallery'>Gallery</a>
    <a href='index.php?page=upload'>Upload new image</a>
</nav>
";
```

## Creating Two Dummy Page View Files

It is always a good idea to start small when coding something new. Let's prepare two separate page views: one for the gallery and one for the upload form. Each page view will be generated and returned from separate files. So, we create two files inside the `views` folder.

```
<?php
//complete source code for views/gallery.php
return "<h1>Image Gallery</h1>";
```

```
<?php
//complete source code for views/upload.php
return "<h1>Upload new images</h1>";
```

You'll notice that there is no end delimiter for PHP blocks, i.e., there is no `?>` in the code. You may remember from Chapter 1 that there is no need to end PHP code blocks, unless you specifically want to write some static HTML in your files. As long as you're only writing PHP, you don't have to end your PHP code blocks. On the other hand, you can end your PHP blocks with `?>`, if you prefer. It'll make no difference one way or the other.

## Creating the Index File

Every site should have an index file. It will be the only file users will ever request, and as such, it is like a main entrance door to all site content. Let's create an `index.php` file and display a functional, dynamic navigation linking to two very simple page views.

```
<?php
//complete code for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
include_once "classes/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "Dynamic image gallery";
$pageData->content = include_once "views/navigation.php";
$userClicked = isset($_GET['page']);
```

```
if ( $userClicked ) {
    $fileToLoad = $_GET['page'];
} else {
    $fileToLoad = "gallery";
}
$pageData->content .=include_once "views/$fileToLoad.php";
$page = include_once "templates/page.php";
echo $page;
```

## Time to Test

All code so far has been just like that we have been working on in the first few chapters, so it should hold few surprises for you. In total, we have fewer than 20 lines of code, but this is enough to perform an initial test. When you are working with code, it is recommended that you write a little code, test it, and then write a little more.

If you test your progress often, you'll be able to identify errors in their infancy. An error is much easier to find in fewer lines of code, so let's make an effort to catch errors as early as possible.

Beginners can have a hard time knowing when to test and what to expect. Obviously, beginners who don't know much about PHP will have a hard time predicting how a piece of PHP code will behave. It is important that you learn to predict how PHP code will behave.

The best way to learn this skill is to use your imagination: Before you load `http://localhost/ch4/` into your browser, try to guess what you will see there. Try to guess how your site will behave at this point.

To perform the test, you'll have to open your XAMPP manager and start Apache. With Apache running, you are ready to load `http://localhost/ch4/` into your browser. Did everything go as you expected? I hope you can see a very basic site with a dynamic navigation displaying two links that work. If you don't, you should read slowly through your code and look for anything that doesn't fit in—misspelled variable names, for example. You can also compare your code to one of your previous projects in which you have the dynamic navigation working.

# Adding Two Style Sheets to One Page

When developing bigger sites, it is very common to use multiple style sheets. We could do it from `index.php` with code such as the following:

```
$pageData->css = "<link href='css/layout.css' rel='stylesheet' />";
$pageData->css .= "<link href='css/navi.css' rel='stylesheet' />";
```

As you may recall, the `$pageData` object is created in `index.php` before `templates/page.php` is included. The property `$pageData->css` is used in `templates/page.php`. Essentially, `templates/page.php` is like a skeleton for an HTML5 page. The `$pageData` object provides all the muscles for the skeleton, by providing the content to be used in the HTML5 page. The skeleton and the muscles are joined in `index.php`.

The code above would work just fine. But I would like to take this opportunity to introduce you to another clean-code principle: staying DRY.

## Staying DRY

All good programmers try to stay DRY. DRY is an acronym for "Don't Repeat Yourself." When you find yourself writing repetitive code, you should always stop and think: Is there a way to reorganize this code to avoid repetition? The example earlier repeated the code to generate `<link>` elements. I have emphasized the repeated code here.

```
$pageData->css = "<link href='css/layout.css' rel='stylesheet' />";
$pageData->css .= "<link href='css/navi.css' rel='stylesheet' />";
```

Repetition is ugly! There are smarter ways to work around this code problem. It just so happens that one such smarter way involves a very powerful concept: object methods. Let's implement a method for adding style sheets.

## Improving the Page_Data Class with a Method

Object methods are just like functions. The significant difference is that methods are declared in class definitions and are usually brought to work on object properties. Here's a method for the `Page_Data` class to avoid repetition when adding style sheets:

```
<?php
//complete code listing for classes/Page_Data.class.php
class Page_Data {
    public $title = "";
    public $content = "";
    public $css = "";
    public $embeddedStyle = "";

    //declare a new method
    public function addCSS( $href ){
        $this->css .= "<link href='$href' rel='stylesheet' />";
    }
}
```

To add a style sheet, you have to generate a `<link>` element with a unique `href` attribute. To add another style sheet, you must generate another `<link>` element with another unique `href` attribute. The two `<link>` elements are identical, except for the `href` attribute.

The object method `addCSS()` takes advantage of that. The method requires an *argument* representing the `href` attribute. For every time the method is called, a new `<link>` element will be created. All created `<link>` elements will be stored together in one string in the object's `$css` property.

## Is It a Function or a Method?

You can see that methods are declared with the `function` keyword. In fact, the preceding method looks just like one of the named functions you have already written. Functions and methods are nearly identical. The only syntactical difference is that methods are functions declared inside a class definition. Apart from that, there are no noticeable differences.

## What Is $this?

When I refer to my personal properties, I use the word *my*—as in my hair, my height, and my weight. The word *my* is a self-reference. When you refer to PHP object properties from inside a class definition, you must use the keyword `$this`: it is the word PHP objects use for self-reference. So, `$this->css` is the object's internal reference to its `$css` property.

In the preceding method, you can see that it is necessary to use the `$this` keyword. Without it, you couldn't refer to the `$css` property. And if you couldn't refer to the `$css` property, the object wouldn't remember the created `<link>` elements.

## Using the New Method

It will be quite simple to use the new method from `index.php`. Anywhere after a new `Page_Data` object is created and before the final `echo`, you can add two lines of code to add two style sheets to the `index.php` page.

```
$pageData->addCSS('css/layout.css');
$pageData->addCSS('css/navigation.css');
```

You'll need two separate style sheets to test whether everything works at this point. Let's create two very simple style sheets to have something to test.

```
/* code listing for css/layout.css */
h1{color:red;}

/* code listing for css/navigation.css*/
nav a{text-decoration: none; color:black}
nav a:hover{text-decoration: underline;}
```

Now save your files and load `http://localhost/ch4/index.php` in your browser. If everything works, all `<h1>` elements should be red, and navigation items should be black and not underlined until you hover them with your mouse. The design is definitely not pretty, but it demonstrates that the new object method works as intended.

## You Can Only Use Methods That Are Declared

PHP can only be used as intended. You can only use methods actually declared in the class definition. In the code example, you declared a method for adding style sheets, so you can now add style sheets. You cannot add pencils to your `$pageData` objectl.

```
//this would trigger an error
$pageData->addPencil( "2b" );
```

It may seem like a blatantly obvious observation, but it is important to realize that objects only have the behavior you give them when you declare methods in the class definition. If you try to call a method on an object that does not have that method declared in its class definition, PHP will display an error message.

# Preparing a Function for Displaying Images

Open the `views/gallery.php` file in your editor and declare a short function that simply returns an HTML string including a `<ul>` with one `<li>`.

```
<?php
//complete source code for views/gallery.php
//function call
return showImages();
//function defintion
function showImages(){
    $out = "<h1>Image Gallery</h1>";
    $out .= "<ul id='images'>";
    $out .= "<li>I will soon list all images";
    $out .= "</ul>";
    return $out;
}
```

See how the variable `$out` inside the function `showImages()` gradually gets more and more content over several lines of code, using incremental concatenation. In the end, when the HTML string is complete, the variable `$out` is returned to the caller. The caller will be the place in your code where the function `showImages()` was called, i.e., in the beginning of `views/gallery.php`.

You can see a `return` statement in the beginning of `views/gallery.php`. Once the HTML string is returned from `showImages()`,the whole thing is returned to `index.php`, because `views/gallery.php` was included from `index.php`. Essentially, the generated string is returned to `index.php`, where it will be added to the `$pageData` object, merged with the page template, and echoed to the browser.

## Iteration

I hope function, variables, and incremental concatenation are slowly beginning to make sense to you. It is time to focus on iteration: repeating stuff in code. Let's begin with `while` loops. `while` loops will repeat the same code block, as long as a condition is true. The basic syntax is

```
while ( $condition ) {
    //repeat stuff here
}
```

A `while` loop is syntactically quite similar to an `if` statement. If the condition holds `true`, the code in the subsequent code block will repeat; it will loop. Here's a simple example to illustrate the concept:

```
<?php
$number = 1;
while ( $number < 5 ) {
    echo "the while loop has concluded $number loops<br />";
    $number = $number + 1;
}
```

If you were to run this code, you would see four lines printed in the browser, as follows:

```
the while loop has concluded 1 loops
the while loop has concluded 2 loops
the while loop has concluded 3 loops
the while loop has concluded 4 loops
```

Notice that the code block is repeated four times. That is because of the condition declared inside the parentheses. It states that the code block will repeat for as long as `$number` is smaller than 5. Every time the code block runs, `$number` is increased by 1. The variable `$number` has a value of 5 when the `while` loop has repeated four times. Because 5 is not smaller than 5, the `while` loop terminates, and a fifth line is never echoed.

`while` loops are great for repeating the same operation many times. There are other kinds of loop structures in PHP. You might come across them, if you look into PHP code from other books or from the Internet. The other kinds of loops are all very similar to `while` loops, even though syntax differs.

## Using a DirectoryIterator to Find Files in a Folder

We can use a `while` loop to create `<img>` elements for every one `JPEG` file found in the `img` folder. But `while` loops can't look into folders by themselves. We can use a native PHP object designed specifically for looking in folders: it is called a `DirectoryIterator`.

*Iterate* is just a technical term for "looping," just as *directory* is a technical name for "folder." So, you can guess by its name that a `DirectoryIterator` can loop through files in folders. That's all it does, and it does it really well. Here's a general example:

```
$filesInFolder = new DirectoryIterator( $folder );
$numItemsInFolder = 0;
while ( $filesInFolder->valid() ) {
    $numItemsInFolder = $numItemsInFolder + 1;
    $filesInFolder->next();
}
echo "found $numItemsInFolder items in folder named $folder";
```

See that condition for the `while` loop? You're calling the method `valid()` of the `DirectoryIterator` object. It will return true, if the `DirectoryIterator` object currently points to a valid item in the folder. You can probably guess what the method `next()` does. It will cause the `DirectoryIterator` to point to the next item in the folder.

So, by combining a `while` loop with `$filesInFolder->valid()` and `$filesInFolder->next()`, you can build a loop that repeats a block of code as many times as there are items in a folder. The preceding code will do just that.

## Showing All Images

Let's implement a similar code block in the gallery. Here's my final version of `showImages()`:

```
//edit existing function
function showImages(){
    $out = "<h1>Image Gallery</h1>";
    $out .= "<ul id='images'>";
    $folder = "img";
    $filesInFolder = new DirectoryIterator( $folder);
    while ( $filesInFolder->valid() ) {
        $file = $filesInFolder->current();
        $filename = $file->getFilename();
        $src = "$folder/$filename";
        $fileInfo = new Finfo( FILEINFO_MIME_TYPE );
        $mimeType = $fileInfo->file( $src );

        if ( $mimeType === 'image/jpeg' ) {
            $out .= "<li><img src='$src' /></li>";
        }
        $filesInFolder->next();
    }
    $out .= "</ul>";
    return $out;
}
```

If you save and run this code, you will see that PHP generates a list of `<img>` elements showing all JPEG images found inside the folder. If you have two JPEG images in the folder, you will see those two images in your online gallery, and if you have ten JPEG images, you will see ten images. It all happens dynamically because of the function `showImages()`.

■ **Note** The used `Finfo` object is enabled by default in PHP 5.3.0, but it may be disabled in some PHP installations. Alternatively, you could use `mime_content_type( $src )` to get the mime type of a file, but you should know that `mime_content_type()` is deprecated: you cannot trust it to work in the future. You can find source code using `mime_content_type()` on the book's companion site.

With a little PHP, you can create solutions that are much more attractive to your clients. Just think about how easy it would be to keep this image gallery updated? Your client would simply have to put a few more images in the right folder, and the gallery would be updated.

# Creating a Form View

You could write some CSS to make the gallery prettier. We will get to that, but first, I want to show you how you can upload new images to the gallery through an HTML form. Let's begin by displaying a form. In a sense, the form is like the site navigation: it is a snippet of static HTML that will not need to change. Create a separate file for it in the `views` folder. Call the file `upload-form.php`, as follows:

```
<?php
return "
Upload new jpg images</h1>
<form method='post' action='index.php?page=upload' enctype='multipart/form-data' >
    <label>Find a jpg image to upload</label>
    <input type='file' name='image-data' accept='image/jpeg'/>
    <input type='submit' value='upload' name='new-image' />
 </form>";
```

Some of the preceding should look familiar. We have an HTML form with `method` and `action` attributes. But this form is a bit different from the previous forms you have written.

Did you notice the `enctype` attribute declared for the form? The default encoding used by forms will not allow file uploads. We must specifically declare that this particular form should use `multipart/form-data` as `content-type`, because this is required for uploading files through HTTP.

Another notable difference is the new input `type='file'` attribute. It will create a *file upload control* to allow users to browse their own hard drives for image files to upload. Please also notice the `accept` attribute on the same `<input>` element. It really declares that the only kind of file that can be uploaded through this form are files with a `content-type` of `image/jpeg`.

Declaring an `accept` attribute is quite helpful for end users. When it is declared, it will narrow down which files users can select through the form. Users are helped to select a file with an appropriate file type. You should know that the `accept` attribute is not supported by older browsers. So, users using an older browser will not get the added benefit offered by the `accept` attribute. But it will not harm the form's basic functionality: all users can choose a file to upload, no matter which browser is used.

■ **Note** The `accept` attribute can be used with any Internet media type. An Internet media type is a standard way of identifying a file type. See more about Internet media types at http://en.wikipedia.org/wiki/Internet_media_type.

## Showing a Form for Uploading Images

To actually display the upload form, you will have to include the snippet of HTML at the right time. You want the form to be displayed when a user clicks the "Upload new image" navigation item. So, to show the form, you must update the code in `views/upload.php`, as follows:

```
<?php
//complete source code for views/upload.php
$output = include_once "views/upload-form.php";
return $output;
```

If you save your work and load `http://localhost/ch/index.php?page=upload` in your browser, you can see what a file upload control looks like, but don't expect to be able to actually upload files just yet.

---

■ **Note** The file upload control will be rendered differently on different browsers. Normally, you would use some custom CSS to design the appearance of an HTML element, but file upload controls are hard to style. Search the Internet for a solution, if you want, and be prepared to test your design rigorously in multiple browsers and browser versions.

---

## $_FILES

When you try to upload a file through an HTML form, the file date can be accessed through a PHP superglobal array called `$_FILES`. Let's see what PHP sees, before actually uploading a file. You can use `print_r()` to inspect `$_FILES` in much the same way you used it to inspect `$_POST` in the previous chapter. Update `views/upload.php`, as follows:

```
<?php
//complete source code for views/upload.php
//$newImageSubmitted is TRUE if form was submitted, otherwise FALSE
$newImageSubmitted = isset( $_POST['new-image'] );
if ( $newImageSubmitted ) {
    //this code runs if form was submitted
    $output = upload();
} else {
    //this runs if form was NOT submitted
    $output = include_once "views/upload-form.php";
}
return $output;
//declare new function
function upload(){
    $out = "<pre>";
    $out .=print_r($_FILES, true);
    $out .= "</pre>";
    return $out;
}
```

Declare a new function in `views/upload.php` and add a conditional statement at the very top of the file. The HTML `<pre>` element will preserve text formatting, such as tabs. The native PHP function `print_r()` will output an array, so that human eyes can read it.

This is enough for you to test your upload form. Save your work and run it in your browser. Pick some `.jpg` file through the form and you should see an output such as the following.

```
Array (
     [image-data] => Array (
             [name] => alberte-lea.jpg
             [type] => image/jpeg
             [tmp_name] => /Applications/XAMPP/xamppfiles/temp/phpYPcBjK
             [error] => 0
             [size] => 119090
         )
  )
```

From that output, you can deduct quite a lot. You can see that `$_FILES` is an `array`. In the preceding example, `$_FILES` has one index: `image-data`. It is important to realize that it is called `image-data` because the file upload control element in the form has its `name` attribute set to `image-data`.

Inside `$_FILES['image-data']`, there is another array with five indices: `name`, `type`, `tmp_name`, `error`, and `size`. `name`, `type`, and `size` indices should be obvious, but the remaining may be a bit obscure when you first come across them.

## tmp_name

When a form is uploaded, its file data will be stored temporarily in the web server's memory. PHP can access the temporarily stored file data through `$_FILES['image-data']['tmp_name']`. You will have to access file data to save the temporary file permanently on the server's file system.

## error

Things can go wrong when you upload files. This image gallery runs on a local web server installed with XAMPP. Probably, the most common problem you may come across is too restrictive file permission settings. Should you encounter an upload error, you can inspect `$_FILES[ ['image-data']['error']` to get the relevant error code. You can consult the PHP manual at www.php.net. It can help you understand the meaning of any error code you come across. Later in this book, I will show you how you can deal with upload errors programmatically.

# Uploading Files with PHP

Uploading a file to a web server is simple with PHP. You simply access the temporary file data and save it permanently. In the process, you have to indicate which folder to save in and what file name to save as. There is a native PHP function to do just that.

```
move_uploaded_file( $fileData, $destination );
```

The function takes two arguments. The first, `$fileData`, should hold valid file data. The second, `$destination`, should be an existing, writable folder. The function `move_uploaded_file()`returns a Boolean. It will return `TRUE`, if the file was saved successfully, and `FALSE`, if something went wrong.

# Planning an Uploader Class

Chances are you will have to write code to upload files many times in your life as a PHP developer. It would be a good idea to write some code for uploading in such a way that you can easily reuse it in later projects. Objects are easily reused, so plan a class you can reuse to upload files through a PHP object.

## UML

I like to use simple UML class diagrams for planning classes. You can see the basic notation for an imaginary class in Figure 4-1.

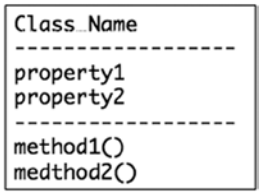


***Figure 4-1.*** *Basic UML diagram*

Notice a naming convention for class names: always begin the first name with an uppercase letter. If the class name is a compound word, separate the words with an underscore and begin the second word in uppercase.

---

■ **Note** UML is an acronym for "Unified Modeling Language." The language provides a standard syntax for documenting code. There is more to UML than just these class diagrams.

---

## Uploader Class Requirements

You know you will need to save file data received from a form. So, the class needs a property for storing the file data and a method for saving it as a file. You know that a file requires a name, so the class needs a `$filename` property. Last, you know that files have to be saved somewhere. You'll need a property to remember a `$destination` for saving the file, and you could add a method for specifying where to save. Knowing these requirements, you can begin to plan the new class definition. You can sketch a UML class diagram, as in Figure 4-2.

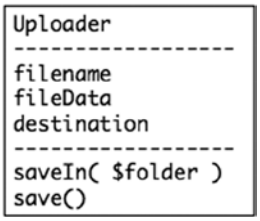


***Figure 4-2.*** *UML diagram of the* `Uploader` *class*

With a plan and a UML class diagram, it is easy to get started writing the class definition. Create a new file `classes/Uploader.class.php`, as follows:

```
<?php
class Uploader {
    private $filename;
    private $fileData;
    private $destination;

    public function saveIn( $folder ) {
        $this->destination = $folder;
    }

    public function save(){
        //no code here yet
    }
}
```

The preceding code declares a class with a class name and class code block delimited with curly braces. Inside the class, there are three properties and two methods declared. It should be quite easy to see that the preceding skeleton for a class is based on the UML class diagram.

You can see from the code that the property `destination` gets its value whenever the method `saveIn` is called. The properties `filename` and `fileData` don't have any values. You can get both `filename` and `fileData` values from the superglobal array `$_FILES`. It would be nice if they got values whenever a new `Uploader` object was created, so their values reflect whatever file you want to upload at that point.

## The Magic Method _construct()

It just so happens that you can declare a method that will run only once, whenever a new `Uploader` object is created. In object-oriented terms, such a method is called a *constructor*. In PHP syntax, it is called `__construct`. It is a so-called magic method. Please notice there are *two underscore characters* before the method name. Declare a constructor method for the `Uploader`, so `filename` and `fileData` properties can get their values from `$_FILES` whenever a new `Uploader` object is created.

```
<?php
//complete code for classes/Uploader.class.php
class Uploader {
    private $filename;
    private $fileData;
    private $destination;
```

```
    //declare a constructor method
    public function __construct( $key ) {
        $this->filename = $_FILES[$key]['name'];
        $this->fileData = $_FILES[$key]['tmp_name'];
    }

    public function saveIn( $folder ) {
        $this->destination = $folder;
    }

    public function save(){
        //no code here yet
    }
}
```

Remember how you must know the `name` attribute of the `<input type='file'>` element used to upload a file? You need the `name` attribute to access all file data in `$_FILES`. In the preceding code, the constructor method takes a `$key` as argument. The `$key` value should be identical to the `name` attribute's value. With that in place, the constructor method can access the data for the uploaded file, which only resides in the server's memory, until it is saved.

## Saving the Uploaded File

The `Uploader` class is nearly complete. You only have to complete the method for saving the new file. As you are working on a local web server, there is one common problem you are likely to come across while performing file uploads: the destination folder may not be writeable. Because you can expect that particular error, you can prepare for it in code, as follows:

```
//partial code for classes/Uploader.class.php
//edit the save method in the Uploader class
public function save(){
    $folderIsWriteAble = is_writable( $this->destination );
    if( $folderIsWriteAble ){
        $name = "$this->destination/$this->filename";
        $succes = move_uploaded_file( $this->fileData, $name );
    } else {
        trigger_error("cannot write to $this->destination");
        $succes = false;
    }
    return $succes;
}
```

Reading through the code, you can probably guess that PHP will check if the destination folder is writeable. If it is not, PHP will trigger an error message that tells you what went wrong. In other words, if you encounter this particular upload error, PHP will display an error message. Errors messages are good; they help you diagnose bugs in your code.

## Using the Uploader Class

You can put the `Uploader` class to good use and upload a file now. It doesn't take a lot of code in `views/upload.php`, because most code is written inside the `Uploader` class.

```
//partial code for views/upload.php
//edit existing function in views/upload.php
function upload(){
    include_once "classes/Uploader.class.php";

    //image-data is the name attribute used in <input type='file' />
    $uploader = new Uploader( "image-data" );
    $uploader->saveIn("img");
    $fileUploaded = $uploader->save();
    if ( $fileUploaded ) {
        $out = "new file uploaded";
    } else {
        $out = "something went wrong";
    }
    return $out;
}
```

How cool is that? You have a completely dynamic image gallery, and users can upload their own images through the web site. It is perhaps not quite `flickr.com` yet, but I hope you'll agree that you are really starting to use PHP to create something fun and useful.

## What Could Go Wrong?

The most common error I see among students learning to upload files through PHP on a local web server is that the destination folder has file permission settings that are too restrictive. Should you encounter that problem, the `Uploader` object will trigger a PHP error and let you know. The solution is simple: change the destination folder's permission settings, so that everybody has read/write access.

Another common problem occurs when PHP doesn't find any file data through `$_FILES`. This mostly happens if a wrong `$key` is supplied when a new `Uploader` object is created. You must pass one argument to the `Uploader` constructor when you create a new `Uploader` object. The argument must hold the `name` attribute of the file upload control from the HTML form. In the `<form>` used in the preceding example, you have a file upload control.

```
<input type='file' name='image-data' />
```

To upload a file received through that `<input>` element, you need an `Uploader` object that knows where to look for the bit stream. When the `Uploader` object is created, you must pass the relevant `name` attribute value as an argument. In this case, you must use the string `image-data`, as follows:

```
$uploader = new Uploader("image-data");
```

## The Single Responsibility Principle

I hope you marvel at the beauty of the `Uploader` class definition. It is planned and written with a single focus: It wants to upload files. It has three properties and two methods. The properties are all about the file to be uploaded, and the methods are about uploading the file.

The single responsibility principle is a common principle used in object-oriented programming. The single responsibility principle states that a class should be written for a single purpose. All properties and methods of the class should relate directly to that single purpose. The class should only have a single reason to change.

For example: The `Uploader` has only one reason to change. It will change if you want to use it for uploading a different file. The single responsibility principle is a beautiful ideal to strive for in code. It is really Curly's law again, only this time, applied to object-oriented programming.

---

■ **Note** You can read more about the single responsibility principle at
`http://en.wikipedia.org/wiki/Single_responsibility_principle`.

---

You already know intuitively that the single responsibility principle is a great idea. If you bought a multipurpose kitchen utility machine that could make coffee and ice cream, bake bread, and fry sausages, you would expect it to make pretty bad coffee. In fact, you could trust it to perform all of its actions pretty badly. If you want great coffee, you will buy yourself a machine designed with a single purpose in mind: a coffee machine!

If you appreciate great coffee, you can probably think of some coffee machines that don't make great coffee. You are learning PHP so that you can plan and build classes to do one thing great.

# Summary

In this chapter, you have seen how you can make a dynamic image gallery, using objects and object methods. You have seen a native PHP object and learned to use a few of its methods, but you have also declared a custom class definition with properties and methods. You have even tried to use `while` loops for repeating code automatically.

You have written two class definitions now: the `Uploader` and the `Page_Data`. I suspect you don't fully understand what classes, objects, properties, and methods are all about. You will get to work with many more classes and objects throughout this book. There are plenty of examples and explanations waiting for you in the pages to come, so hang in there. Learning takes time, and we're just getting started . . .

You have seen enough of basic PHP now. We will soon move on to explore databases and learn about the new possibilities you can implement in your projects, as you gain familiarity with database-driven, dynamic web sites. But first, we'll take a short and intense detour, involving a bit of JavaScript and client-side scripting.

CHAPTER 5

# Spicing Up Your Image Gallery with JavaScript and CSS

This chapter is completely optional reading! You can consider it an invitation to a small detour from PHP. This chapter explores a way to integrate JavaScript into your PHP projects. In the process, you will have to learn some JavaScript to develop an interactive image gallery. You may explore this detour or skip it altogether.

If you follow any blogs about web design and web development, you are bound to come across something that involves JavaScript. Perhaps you will come across something you would like to implement in your PHP projects. This is not really a book about JavaScript, but I will show you some examples of using JavaScript. My goal is to show you an approach to integrating JavaScript in your PHP projects. If you really want to learn JavaScript, you'll have to consult other resources.

---

**Note** Want to learn more about JavaScript in the browser? Consider picking up a copy of *Foundation Game Design with HTML5 and JavaScript* by Rex van der Spuy (Apress, 2012). You'll be building games for the browser using HTML5, CSS, and JavaScript. It is a fun way to learn!

---

## Client-Side vs. Server-Side Programming

PHP is a great language for web development. It is extremely popular for many good reasons. But PHP is just one server-side scripting language out of many. Server-side languages only run on your server; there is no way to execute PHP code in a browser. So far, you have written PHP code to output HTML, which is sent to a browser.

In order to change anything in the generated HTML, the browser must send an HTTP request to the server, so PHP can run and send back an HTTP response. But it takes time and bandwidth to send an HTTP request, wait while PHP runs, and finally receive an HTTP response.

In some situations, it would be better to simply run some code when your system needs it. Luckily, there is a way: It is possible to manipulate HTML programmatically in the browser, using JavaScript, which happens to be the only scripting language that runs natively in the browser. You can choose between many different languages on the server side. On the client side, i.e., in the browser, there is only one: JavaScript.

JavaScript is a wonderful language, but different versions of different browsers have implemented different parts of JavaScript in different ways. So, the JavaScript that runs beautifully in one browser might trigger embarrassing errors in another.

A common approach to dealing with these differences is to use *progressive enhancement*, which basically means that you write your code in such a way that your beautiful JavaScript only runs in browsers that fully understand it.

Progressive enhancement with JavaScript provides an optimal user experience to modern browsers with JavaScript enabled. Older browsers or browsers with JavaScript disabled still get all the content served. The extra JavaScript features should remain hidden from incapable browsers, so that JavaScript errors are avoided. The image gallery covered in this chapter uses progressive enhancement.

# Coding a Lightbox Gallery

Let's code a so-called lightbox for the image gallery, to present the images in an aesthetically pleasing way. A *lightbox* is a very common approach to displaying images. When a user clicks a small image on a web page, JavaScript will place a semitransparent overlay on top of all page content. A large version of the clicked image will be displayed on top of the overlay.

From PHP on the server side, you'll continue to serve every visiting browser a list of all JPEG images. But if a user comes along with a top-notch browser, you can serve an even better solution: small thumbnails of all images, so the user can quickly navigate the entire gallery. If a user clicks a thumbnail, you can really bring focus to that particular picture. You can hide all other content behind a semitransparent overlay and really highlight the selected picture. You can even display a bigger version of the clicked image: that is a lightbox gallery.

That is what progressive enhancement is all about: serve all content to all browsers but provide a better user experience for capable browsers. Let's get started!

## Embedding an External JavaScript File

It is possible, but not recommended, to write JavaScript code directly in your HTML. A better approach is to keep your HTML and your JavaScript decoupled. It is possible to embed several JavaScript files in one HTML file, just as you can link multiple style sheets to a single HTML file. To link a JavaScript to an HTML file, you can use a `<script>` element:

```
<script src="path/to/Javascript-file.js"></script>
```

The `src` attribute should point to an existing JavaScript file, so it is important that the path is correct. You may wonder why the `<script>` element is a container tag? It is because you can decide to write JavaScript code directly in your HTML file, inside a `<script>` element.

---

■ **Note** Using external JavaScript files and carefully avoiding any JavaScript code in HTML is also known as *unobtrusive* JavaScript.

---

You already have a PHP-driven, dynamic image gallery from Chapter 4. Adding some JavaScript to that project should give you a good idea of some of the things you can do with JavaScript. The code examples in this chapter rely on your having the PHP source code for the gallery developed in Chapter 4.

## Preparing the Page_Data Class for JavaScript Files

You can change your PHP code to prepare for one or more JavaScript files. The existing `Page_Data` class needs a property to hold one or more `<script>` elements. You can also declare a new method in the `Page_Data` class for adding a new JavaScript file. It will be very similar to the property for `<link>` elements holding style sheet references and the method for adding new style sheets. I propose you continue working with the project you started in Chapter 4, so the file to update is `ch4/classes/Page_Data.class.php`. Here's the complete code for it:

```
<?php
//complete code listing for classes/Page_Data.class.php
class Page_Data {
    public $title = "";
    public $content = "";
    public $css = "";
    public $embeddedStyle = "";
    //declare a new property for script elements
    public $scriptElements = "";

    //declare a new method for adding Javascript files
    public function addScript( $src ){
        $this->scriptElements .= "<script src='$src'></script>";
    }

    public function addCSS( $href ){
        $this->css .= "<link href='$href' rel='stylesheet' />";
    }
}
```

Notice the new public property called `$scriptElements`. It will hold however many `<script>` elements you require for the page. Also note the public function `addScript()`. See how it takes a `$src` as argument. The `$src` should hold a path to a JavaScript file. The received path will be used to create a `<script>` element. The created `<script>` element will be stored, together with any previously added `<script>` elements, by way of incremental concatenation.

## Preparing the Page Template for JavaScript Files

You have to update the page template file to accept `<script>` elements for JavaScript, just as you did when you updated the page template to accept `<link>` elements for CSS. Edit `template/page.php`:

```
<?php
//complete code listing for templates/page.php
return "<!DOCTYPE html>
<html>
<head>
<title>$pageData->title</title>
<meta http-equiv='Content-Type' content='text/html;charset=utf-8' />
    $pageData->css
    $pageData->embeddedStyle
</head>
<body>
    $pageData->content
    $pageData->scriptElements
</body>
</html>";
```

The script elements will be embedded by PHP by way of `$pageData->scriptElements`. Note that any `<script>` elements will be placed after any other content on the page. When you do that, you can be sure that all HTML elements are loaded into browser memory before your JavaScript starts executing. That is exactly what we want!

JavaScript is often used to manipulate HTML. It is necessary to have the HTML loaded into browser memory before we can manipulate it.

## Writing and Running an External JavaScript File

I like to keep my JavaScript files in a designated folder, to maintain a well-organized file structure. I propose you get used to doing the same. Create a new folder called `js`. Use your editor to create a new JavaScript file called `lightbox.js`. Save it in your `js` folder.

```
//complete code listing for js/lightbox.js
window.console.log("Hello from Javascript");
```

To run JavaScript code, you must tell the browser that there is a JavaScript to run. You can point to an external JavaScript file from `index.php`. The JavaScript you'll be writing will manipulate your HTML, so that certain attributes are changed dynamically.

You will also require an external style sheet. Following is a little code from `index.php` that shows how to point to an external style sheet and how to point to an external JavaScript. These lines of code belong in `index.php` somewhere *after* a new `$pageData` object is created and *before* the generated `$page` is echoed:

```
//partial code listing for index.php
//this line of code you already have. It creates a Page_Data object
$pageData = new Page_Data();
//new code below
//add this new line to embed an external Javascript file to your index.php
$pageData->addScript("js/lightbox.js");
//no other changes in index.php
```

You have an external JavaScript, and you have linked to it from `index.php`. Any JavaScript code you write should run perfectly now. It is quite simple to test, if you use the Google Chrome browser or another browser with a similar JavaScript console. I suggest you use Google Chrome, unless you are already used to another browser with a JavaScript console.

First, open Google Chrome. Next, open the Chrome menu, by clicking the ☰ in the top-right corner of the browser. Select Tools ➤ JavaScript Console. When the console is open, you simply load `http://localhost/ch4/index.php` in your Chrome browser. You should see a message in the console, as shown in Figure 5-1.

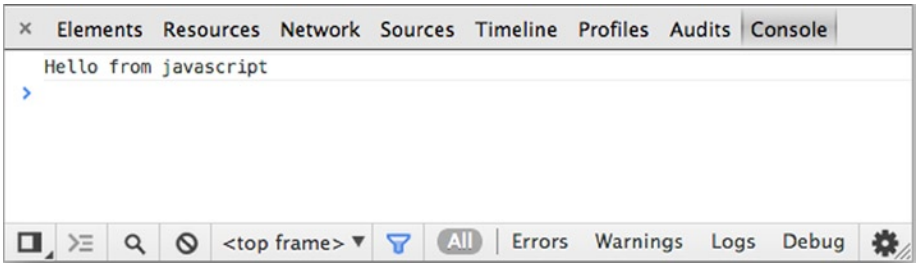


***Figure 5-1.*** *The JavaScript Console in Google Chrome*

## Using window.console.log()

As you can see, the message in the console is identical to what you wrote in the JavaScript code:

```
window.console.log("Hello from Javascript");
```

In JavaScript, `window` is an object representing an open browser window. Inside the `window` object, you can find the `console` object. The `console` object has a method `log()` that outputs messages in the JavaScript console window. Logging to the console is often used to check that some JavaScript works as intended.

In this example, you use it to check if JavaScript runs at all. If you don't see a message in your console, you know that your JavaScript does not run. In that case, you can inspect the HTML source code of `index.php`, to see if you find a `<script>` element that links to an existing JavaScript file. Perhaps there is no `<script>` element, or its `src` attribute doesn't point to your JavaScript file. Please make sure that you have completed all previous steps correctly.

From the single line of JavaScript you have written, you can infer that JavaScript is like PHP, in that it has objects and methods. You can also see that JavaScript syntax is a little different. JavaScript's object operator is a `.`, whereas PHP uses `->`. If JavaScript had exactly the same syntax as PHP, you should have written the following:

```
//If Javascript had PHP's object operator
window->console->log("hello");
```

I'd like you to notice both the functional similarities and the syntactical differences. JavaScript is very similar to PHP in many ways, but the syntax differs slightly. In some ways, JavaScript is really different from PHP, but that's another story.

## JavaScript Arrays

You have already tried to work with arrays in PHP: `$_GET` and `$_POST` are superglobal arrays. Let's take a closer look at arrays and see how they can be used in JavaScript. Open your `lightbox.js` file and write some JavaScript, as follows:

```
var pets = new Array("cat", "dog", "canary");
var firstPet = pets[0];
window.console.log( "The first pet is at index 0. It is a " +  firstPet);
```

### Using var to Declare a Variable

First you declare a variable `pets` and assign it to hold a new array object. You use the keyword `var` to declare a JavaScript variable. The new array object holds a list of three string values. That's what arrays can do: they can hold a list of items.

In order to do anything meaningful with arrays, you have to be able to get the right array item at the right time. Array items are remembered by their position in the list. The technical term for such a position is `index`. The first item in an array has an `index` of `0`, the second an `index` of `1`, and so on. The general syntax for getting an item from an array is as follows:

```
arrayName[index];
```

If you look back into the code for the pets example, you can see that the variable `firstPet` holds the item found at `index` `0` in the `pets` array. Once again, JavaScript is very similar to PHP. To get an item from a PHP array, we could use exactly the same syntax.

### Looping Through Array Items

Loops are very commonly used together with arrays. With a simple `while` loop you can iterate through all items in an array:

```
var pets = new Array("cat", "dog", "canary");
var index = 0;
while ( index < pets.length ) {
    window.console.log( pets[index] );
    index = index + 1;
}
```

Once again, you can see that JavaScript and PHP are quite similar languages: they can both work with `while` loops. The preceding loop will iterate through every item in the `pets` array and output each one item to the console.

In the first iteration, the variable `index` will hold the value 0 and, therefore, the code outputs "cat," which is the item found at `pets[0]`.

The condition for the `while` loop stipulates that the loop will continue as long as `index` is smaller than the length of the `pets` array, in other words, as long as `index` is smaller than 3.

At the end of the first iteration, the value of `index` is changed. It began with a value of 0, now it becomes 1, because `index = 0 + 1`. In the next iteration, the code will output "dog" to the console, because dog is found at `pets[1]`. The variable `index` becomes 2, and the loop continues, so "canary" appears in the console.

Now, `index` becomes 3, and so the `while` loop terminates, because 3 is not smaller than 3.

## Simple Progressive Enhancement

In this lightbox script, you want JavaScript to provide a better experience to users with new browsers. You can do that by responding to an event that is only understood by relatively new browsers. You can do that in `js/lightbox.js`, as follows:

```
//complete code for js/lightbox.js
function init(){
    window.console.log("Welcome, user with a new browser");
}
document.addEventListener("DOMContentLoaded", init, false);
```

Notice the `document` object used in the code above. The `document` is a native JavaScript object. Every HTML web page loaded in a browser has its own `document` object. You can use the `document` object to retrieve and manipulate HTML content in the loaded page.

The preceding code assigns the function `init` to be called automatically when the event `DOMContentLoaded` is dispatched. The `DOMContentLoaded` event will be dispatched when the browser has finished loading the DOM (Document Object Model). The DOM is a representation of the HTML on the page. Only new browsers dispatch the `DOMContentLoaded` event. Therefore, any JavaScript code you write or call from inside the function `init()` will only run if the user has a relatively new browser.

## Using Event Listeners

Event listeners are native to JavaScript. This is one point where JavaScript and PHP really are different, as there are no event listeners in PHP. Event listeners are used to associate an event with a function. The idea is that every time a certain event occurs, a particular function should run. The general syntax for adding an event listener is as follows:

```
object.addEventListener(event, event handler, useCapture);
```

As you can see, the `addEventListener` takes three arguments: an event, an event handler, and useCapture.

### The Event

The first argument specifies which event to listen for. Different objects can respond to different events. In the preceding example, you are listening for the document object's `DOMContentLoaded` event. The browser will dispatch the event, and the `document` object can respond to the event, but only if you specifically tell it to listen for the event.

### The Event Handler

The second argument specifies which function to run when the event is heard. The `addEventListener` registers an event-handling function to a specific event. In your example, you have registered the function `init` as event handler for the document object's `DOMContentLoaded` event.

### The Optional useCapture

The third argument of the addEventListener indicates an advanced topic known as useCapture. For most modern browsers, this is an optional argument, meaning you don't have to specify it. Most browsers will simply assume it to be `false`, if it is not specifically set to `true`. But some browsers and browser versions require this parameter to be set, so you might as well get into the habit of setting it.

As a rule of thumb, you can declare the third argument and set it to `false`. You may come across a special situation that requires you to set it to `true`, but it will not be in the code examples in this book. It is one of those JavaScript topics you can explore on your own by consulting other resources.

## Creating Markup for the Overlay and Big Image

You have established a very basic skeleton for progressive enhancement: the `init` function will only run if the browser is capable.

It's time to get started with the lightbox gallery. Begin by adding some JavaScript to create a little HTML dynamically, to provide markup structures for displaying a big image on top of a transparent overlay. You have to update the `init()` function in `js/lightbox.js`, as follows:

```
//edit existing function
function init() {
    var lightboxElements = "<div id='lightbox'>";
    lightboxElements  += "<div id='overlay' class='hidden'></div>";
    lightboxElements  += "<img class='hidden' id='big-image' />";
    lightboxElements  += "</div>";
    document.querySelector("body").innerHTML += lightboxElements;
}
```

This code will create a string of HTML elements and add them after the HTML elements already found inside the `<body>`. Note especially the `<img>` element. It is missing an `src` attribute, so it doesn't show a picture at this point. If you save your work and load `http://localhost/ch4` in your browser, you may be surprised that nothing seems to have changed, despite all your JavaScript efforts. If your JavaScript works, you should have a page with a few HTML elements added at the end. But they hold no content, so you can see nothing—yet!

You can see, in the preceding code example, that JavaScript can concatenate strings just like PHP can. Please notice that JavaScript's incremental concatenation operator is different from PHP. It is another case of same functionality, different syntax.

```
//Javascript's incremental concatenation operator
+=
//PHP's incremental concatenation operator
.=
```

## document.querySelector()

The last line in the function looks like nothing you have used so far in the book. The `querySelector()` is a method of the `document` object. It is an absolutely wonderful method, if you are comfortable with CSS selectors. With the `querySelector()`, you can select HTML elements from the loaded page using CSS selector syntax.

```
document.querySelector("body").innerHTML += lightboxElements;
```

The preceding line uses the `querySelector` to get the `<body>` element and all its content. JavaScript adds the HTML string stored in the variable `lightboxElements`, after the existing content inside `<body>`. To get access to the HTML content inside the `<body>`, you use the `innerHTML` property.

## Showing the Overlay

I suppose it is very rewarding to have created a `<div>` element with JavaScript. But it would be a lot more rewarding if you could see it working as an overlay. You can achieve that by adding a bit of CSS to your existing style sheet. I keep mine in `css/layout.css`.

```
/*declare a new style rule in css/layout.css */
div#overlay{
    position: absolute;
    width: 100%;
    height:100%;
    top:0px;
    left:0px;
    background:black;
    opacity: 0.85;
}
```

If you save that and reload `http://localhost/ch4` in your browser, you should see a semitransparent overlay covering all the content visible in the browser's viewport. If you scroll down, you can see that the overlay only covers the viewport, not the entire page content. It is as it should be. Seeing the overlay is a visual confirmation that the JavaScript code is doing something. But you only want the overlay to be displayed when a user has clicked a particular image. Also, you would want the clicked image to be displayed on top of the overlay. You still have some work to do.

## Hiding the Overlay and Resize Thumbnails

By default, you would want the images to be displayed as small thumbnails. When a thumbnail is clicked, the overlay should appear to hide the other thumbnails, and the clicked image should be displayed in almost full screen. To achieve this, you need both CSS and JavaScript. You can begin by preparing a few CSS rules, which you can use later by way of JavaScript. Here are a few more rules for your style sheet in `css/layout.css`:

```
/*hide overlay and big-image*/
div#overlay.hidden, img#big-image.hidden{ opacity: 0; left:-200%; }

/*resize images and display them as a horisontal list*/
li.lightbox img{ height: 100px; }
li.lightbox{ display: inline-block; margin: 10px; }
```

If you refresh the browser, you can see that the overlay is hidden. You can also see that thumbnails are not resized yet, despite the CSS. Don't be too surprised. It is simply because the <li> elements do not have a class attribute set to lightbox yet. You will set the class dynamically, using JavaScript. But before writing any more JavaScript, I'd like you to look into the CSS that hides the image and the overlay. You can see that both elements are styled to be completely transparent and positioned so far to the left that the elements would be invisible, even if they weren't completely transparent. Keep in mind that it is important for the overlay and the big image to each have a class attribute set to hidden. If the class is not set to hidden, both elements would show.

By default, there are no <li> elements with a class attribute set to lightbox. So, the CSS rules written above don't apply to anything at the moment. You can change that by writing some JavaScript to declare a class attribute with a value set to lightbox on all <li> elements used for your gallery images, *but only for capable browsers*. Update js/lightbox.js as follows:

```
//complete code listing for js/lightbox.js
//edit existing function
function init(){
    var lightboxElements = "<div id='lightbox'>";
    lightboxElements    += "<div id='overlay' class='hidden'></div>";
    lightboxElements    += "<img class='hidden' id='big-image' />";
    lightboxElements    += "</div>";
    document.querySelector("body").innerHTML += lightboxElements;
    //add a new function call here
    prepareThumbs();
}

//declare a new function
function toggle(){
    window.console.log("show or hide a big image");
}

//declare new function
function prepareThumbs() {
    var liElements = document.querySelectorAll("ul#images li");
    var i = 0;
    var image, li;
    //loop through all <li> elements
    while ( i < liElements.length ) {
        li = liElements[i];
        //set class='lightbox'
        li.setAttribute("class", "lightbox");
        image = li.querySelector("img");
        //register a click event handler for the <img> elements
        image.addEventListener("click", toggle, false);
        i += 1;
    }
}

document.addEventListener("DOMContentLoaded", init, false);
```

Save this code and refresh your browser. You should expect to see a horizontal list of small thumbnail images. When you click one image, you should expect to see a message in the console window saying "show or hide a big image."

That was a big chunk of code in one step. Let's look into it and figure out what's really going on there.

## Only Showing Thumbnails to Capable Browsers

The JavaScript uses progressive enhancement and effectively hides JavaScripted features from browsers that might not understand. Remember how the function `init()` only runs if the browser dispatches a `DOMContentLoaded` event? Only relatively new browsers dispatch that event. As already mentioned, it is dispatched when all HTML elements of a given page are loaded into the browser's memory.

The function `prepareThumbs()` is called from inside the function `init()`, so `prepareThumbs()` will only run in new browsers. You have effectively hidden your JavaScript from older browsers: You have progressive enhancement!

## Getting an Array of HTML Elements with querySelectorAll()

Next, take a look at how `class` attributes are added on all `<li>` elements for the gallery. The first task is to select all `<li>` elements. You've used the method `querySelectorAll()` for that task. The `querySelectorAll()` is just like `querySelector()`, except it returns not just one HTML element but *all* HTML elements that match the used CSS selector.

```
var liElements = document.querySelectorAll("ul#images li");
```

In the preceding line, the variable `liElements` will hold an array of all `<li>` elements found inside the `<ul>`, with an `id` attribute set to `images`.

---

■ **Note** You can learn more about using `querySelector()` and `querySelectorAll()` at `www.kirupa.com/html5/finding_elements_dom_using_querySelector.htm`.

---

You have already seen how to loop through an array of pets. Looping through an array of HTML elements is just like that. Inside `prepareThumbs()`, you can see a `while` loop. It will loop for as long as the variable `i` holds a value smaller than the number of `<li>` elements found. That effectively means you'll loop through every `<li>` item inside the `<ul>` element with an `id` attribute of *images*.

The variable `i` can be used as index to get one particular `<li>` element from the array of all the `<li>` elements. Each one `<li>` will be remembered in the variable `li`, and each `<li>` will get a `class` attribute of `lightbox`. There is a CSS rule for such `<li>` elements, and that is why thumbnails are displayed as a horizontal list in your browser. There is another CSS rule selecting `<img>` elements inside such `<li>` elements. The rule resizes the width of the thumbnail images to 100px.

Still inside the `while` loop, you use `querySelector()` to select the `<img>` element inside the `<li>` element. You assign an event listener to each `<img>` element. So, whenever a user clicks an `<img>`, the function `toggle()` will be called. In other words, you register an event handling function called `toggle()` to each `<img>` element.

## Showing a Big Image

Whenever a user clicks one of your `<img>` elements, the function `toggle()` will run. At this point, it doesn't do much. It simply outputs a message in the console window. You are looking for a different behavior. If a thumbnail is clicked, you want the transparent overlay to hide all thumbnails, and you want a big version of the clicked image displayed. If a big image is clicked, you want the overlay and the big image to disappear, so all thumbnails once again become clearly visible. You will need a CSS rule for styling the big image, and you will require some JavaScript for manipulating HTML class attributes. You can begin with the CSS. Add one more rule to `css/layout.css`:

```
/*partial code listing for css/layout.css*/
/*new CSS rule for showing the big-image*/
#big-image.showing{
    max-width: 80%;
    max-height:90%;
    position:absolute;
    background-color: white;
    padding: 10px;
    top:5%;
    left: 10%;
}
```

To see some action in your browser, you also have to add some code to the `toggle()` function declared in js/lightbox.js, as follows:

```
//edit existing function
function toggle( event ){
    //which image was clicked
    var clickedImage = event.target;
    var bigImage = document.querySelector("#big-image");
    var overlay = document.querySelector("#overlay");
    bigImage.src = clickedImage.src;
    //if overlay is hidden, we can assume the big image is hidden
    if ( overlay.getAttribute("class") === "hidden" ) {
        overlay.setAttribute("class", "showing");
        bigImage.setAttribute("class", "showing");
    } else {
        overlay.setAttribute("class", "hidden");
        bigImage.setAttribute("class", "hidden");
    }
}
```

You can see that JavaScript doesn't really show or hide the big image or the overlay as such. All JavaScript does is manipulate class attributes of the `#overlay` and the `#big-image`. You can check in your browser that overlay and image are in fact hidden to begin with. If you click a thumbnail, the overlay will appear on top of the thumbnails, and the big image will appear on top of the overlay.

This effect is achieved by uniting CSS and JavaScript. In CSS, you keep rules dictating how to render `#overlay` and `#big-image`. If a `class` is set to `hidden` on these elements, they will be hidden from view. If a `class hidden` is not set, the elements will be displayed. JavaScript manipulates the `class` attribute dynamically. CSS declares how to render `#big-image` and `#overlay`, depending on the current value of the `class` attribute.

JavaScript has a quite simple job: it is only responsible for setting `class` attributes on the overlay and the big image. If the `class` attribute is currently set to `hidden`, it will be changed to `showing`. Otherwise, if the class is not set to `hidden`, it will be set to `hidden`.

Let's inspect the code inside `toggle` in greater detail to understand it—and JavaScript—better.

## Using the MouseEvent Object

The first thing to note is the `event` argument added to the `toggle()` function. The function `toggle()` is called, because it is registered as event handler for click events on the `<img>` elements. It is called from an event listener. When that happens, an `Event` object is passed along when the event is triggered. A click is triggered when a user clicks on a mouse button and, consequently, the `Event` object sent along is a `MouseEvent` object.

Event objects have quite a few very useful properties that you can use in code. The `MouseEvent` object has a `target` property, which holds a reference to the clicked HTML element.

---

■ **Note** You can see the other available properties in your console window, if you add the following line of code inside the toggle() function: `window.console.log(event);`.

---

You use the `MouseEvent.target` property to get the clicked `<img>` element. You use that to replace the `src` attribute of the big image with the `src` attribute of the clicked image. Essentially, you use it to display a big version of the clicked thumbnail, seen in the following code:

```
bigImage.src = clickedImage.src;
```

## Toggling

To toggle means to change between two states. You toggle a light switch when you turn on the light, and you toggle the same light switch to turn off the same light. In this piece of JavaScript, you want to toggle the overlay and the big image.

If the `class` attribute of the overlay element is set to `hidden`, you want to hide the overlay and the big image. If the `class` attribute of the overlay is set to `showing`, you want the overlay and big image to show. If you look in the `toggle` function, you can see the same idea expressed in code.

```
if ( overlay.getAttribute("class") === "hidden" ) {
    //code to show overlay and image
} else {
    //code to hide overlay and image
}
```

## Manipulating Attributes

To read the value of the `class` attribute, you use the `getAttribute()` method. It is a standard JavaScript method found on all HTML objects. The `getAttribute()` method can be used to read the value of any attribute. The general syntax is as follows:

```
element.getAttribute( whichAttribute );
```

The `getAttribute()` method will return the value of the requested attribute found in the specified element. There is a similar method, called `setAttribute()`, for changing attribute values. The general syntax is the following:

```
element.setAttribute( whichAttribute, newValue );
```

The `setAttribute()` method can set a new value for a specified attribute on a particular HTML element. In the preceding `toggle()` function, you use it to change the attribute values of the overlay and the big image.

## Hiding the Big Image

At this moment, you can click a thumbnail to have the overlay and big image displayed. This is great. But you cannot hide the overlay or the big image again, which is not so great. To enable hiding, you simply have to register the `toggle()` function as the event handler to be triggered when the big image is clicked. It can be done with the following (boldfaced) two extra lines of code in the `init()` function:

```
//edit existing function
function init(){
    var lightboxElements = "<div id='lightbox'>";
    lightboxElements    += "<div id='overlay' class='hidden'></div>";
    lightboxElements    += "<img class='hidden' id='big-image' />";
    lightboxElements    += "</div>";
    document.querySelector("body").innerHTML += lightboxElements;

    //new code: register toggle as event handler
    var bigImage = document.querySelector("#big-image")
    bigImage.addEventListener("click",toggle, false);
    //end of changes
    prepareThumbs();
}
```

Test it for yourself. At this point, you should be able to click a thumbnail to show the big image on top of the overlay. If you click the big image, both the big image and overlay will be hidden again, thus revealing the thumbnails underneath.

## Using a CSS Animation

Wouldn't it be nice if the semitransparent overlay would fade in to hide thumbnails? It could be a final touch to make the lightbox gallery even nicer. You can create a CSS animation by adding the following (boldfaced) single line of CSS in `css/layout.css`:

```
#overlay{
    position: absolute;
    width: 100%;
    height:100%;
    top:0px;
    background:black;
    opacity: 0.85;
    left:0px;
    /*this is the animation to fade the overlay in gradually over 1 second*/
    transition: opacity 1s ease-in;
}
```

## Coding Challenge

The lightbox gallery is complete now. You can easily find many more examples of JavaScript-powered image galleries, if you browse the Internet. Perhaps you will come across a behavior you would like to implement in your gallery.

A very common feature involves clicking the overlay when the big image is displayed. Most galleries will toggle when the overlay is clicked. It is not incredibly hard to achieve, so perhaps this is a task you can tackle on your own. You'll simply want to register `toggle` as event handler for click events detected on the overlay. This approach will work, and it will trigger a JavaScript error. You can see the error message in your console. An extra coding challenge could be for you to figure out what the error message means and how you can change your code to avoid the error.

You can find a tutorial for another JavaScript gallery at `www.webmonkey.com/2010/02/make_a_javascript_slideshow/` with Next and Previous buttons. Perhaps you can figure out how to implement such buttons in your lightbox gallery. It could be an interesting learning experience and also a nice addition for your lightbox gallery.

# Summary

You have covered a lot of ground in relatively few pages. The primary goal was to provide an approach for you to integrate JavaScript solutions in your PHP projects. In the process, you worked through a relatively simple lightbox image gallery.

You have seen that the PHP and JavaScript languages are similar in many ways. Often, it is only syntax that differs, and sometimes not even that. This is great news for you. Once you have learned PHP, you can learn JavaScript relatively easily.

On the other hand, you have also seen that there are notable differences between JavaScript and PHP. To be really proficient in both languages, you'll eventually want to pay close attention to these differences.

Perhaps the most significant difference to note is that JavaScript is a *client-side scripting language,* whereas PHP is a *server-side scripting language*. Your JavaScript code runs in your users' browsers. PHP only runs on your server, so browsers will never see your PHP. Browsers will only be served the result created by PHP.

---

■ **Note** Actually, JavaScript *can* run on servers. Search the Internet for `node.js` to learn more. Also, PHP can run without a web server. But JavaScript is mostly used client-side, and PHP is mostly used server-side.

---

So far, you have seen that when PHP has done its thing, it will create an output. The output is often an HTML file, which is sent to the browser. JavaScript can manipulate the HTML in the browser, without any need for contacting the server. You will probably want to learn more JavaScript eventually, but that is beyond the scope for this book.

CHAPTER 6

# Working with Databases

Modern web sites are incredibly powerful, and much of this power derives from their ability to store information. Storing information allows developers to create highly customizable interactions between their software and its users, ranging from entry-based blogs and commenting systems to high-powered banking applications that handle sensitive transactions securely.

This chapter covers the basics of MySQL, a powerful, open source database. I also demonstrate an object-oriented approach to using MySQL in your PHP projects. Subjects covered include the following:

- The basics of MySQL data storage
- Manipulating data in MySQL tables
- Database table structure
- Using PHP for interacting with MySQL databases
- Organizing PHP scripts with a model-view-controller approach
- Why coding is like playing the blues

There is a lot to learn in this chapter, and some of it might make your head spin, initially. But rest assured that all topics covered will be repeated and elaborated upon in subsequent chapters. You will be presented with ample opportunities to learn.

## The Basics of MySQL Data Storage

MySQL is a relational database management system that lets you store data in multiple tables. Each table contains a set of named columns, and each row consists of a data entry into the table. Tables will often contain information about other table entries. That way, one fact can be stored in one table yet be used in other tables. For example, take a look at how you might store information about musical artists (see Tables 6-1 and 6-2).

***Table 6-1.*** *The Artist Table*

| artist_id | artist_name |
|---|---|
| 1 | Bon Iver |
| 2 | Feist |

***Table 6-2.*** *The Album Table*

| album_id | artist_id | album_name |
|---|---|---|
| 1 | 1 | For Emma, Forever Ago |
| 2 | 1 | Blood Bank - EP3 |
| 3 | 2 | Let It Die |
| 4 | 2 | The Reminder |

The first table, artist, includes two columns. The first column, `artist_id`, stores a unique numerical identifier for each artist. The second column, `artist_name`, stores the artist's name.

The second table, album, stores a unique identifier for each album in the `album_id` column and the album name in—you guessed it—the `album_name` column. The album table includes a third column, `artist_id`, that relates the artist and album tables. This column stores the unique artist identifier that corresponds to the artist who recorded the album.

At first glance, it might seem a silly way of storing data. Why keep an abstract, incomprehensible number, instead of simply writing the artist name for every album? Table 6-3 imagines you did that.

***Table 6-3.*** *The Badly Designed Album Table*

| album_id | artist | album_name |
|---|---|---|
| 1 | Bon Iver | For Emma, Forever Ago |
| 2 | Bon Iver | Blood Bank - EP3 |
| 3 | Feist | Let It Die |
| 4 | fiest | The Reminder |

Please note the spelling error for `album_id 4`. Because the artist name is spelled out separately for every album, it is possible to store different names for the same artist. In a tiny table with four entries, like the one preceding, it is easy to spot and correct errors. But tables are rarely that small in the real world. Imagine you were building a database for a music store. You would have to keep track of thousands of albums.

If you were lucky, you would catch the error—and then you would have to go through every album by Feist to check whether the artist name was spelled correctly. It is possible, but it would be an insane waste of time.

Do the same thought experiment with the other table structure (listed in Table 6-2). If you misspelled *Feist* as *fiest*, you would be more likely to catch the error, because every album by *Feist* would be listed under *fiest*. Also, correcting the error would not have you trudging through thousands of entries. You would simply go to the single place where the artist name is declared and write *Feist* instead of *fiest*, and every album would now be listed correctly.

By designing tables that store one piece of data once, and only once, you design a robust database with data integrity. Joe Celko, a prominent figure in the SQL community, has aptly coined the slogan "one simple fact, in one place, one time." Memorize that slogan, and let your database tables follow this one rule.

# Manipulating Data with SQL

You can manipulate the data in a MySQL table via the Structured Query Language (SQL). SQL is a small language, and most of it is very easy to read and understand. In this section, you will learn the SQL statements that perform the following actions:

- Create a database
- Create a table in the database
- Insert data into the table
- Retrieve data from the table
- Update data in the table

You'll test these commands using the phpMyAdmin control panel provided by XAMPP. To use XAMPP, you must start it first. Open the XAMPP control panel (see Figure 6-1) and start your MySQL Database and your Apache Web Server.

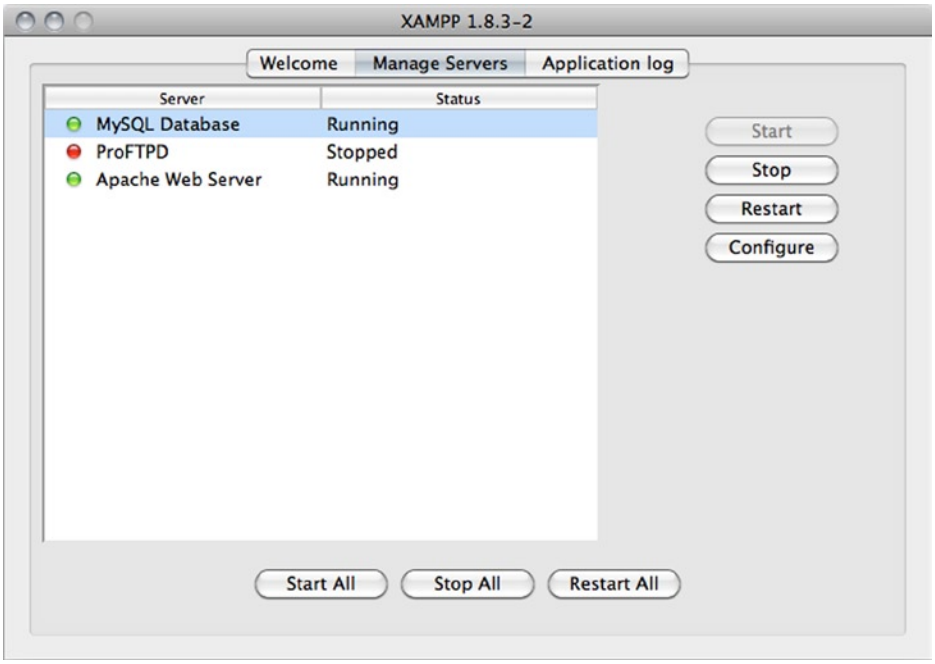


***Figure 6-1.*** *The XAMPP control panel*

With MySQL and Apache running, you can open a browser and navigate to `http://localhost/phpMyAdmin` to access the phpMyAdmin control panel (Figure 6-2).

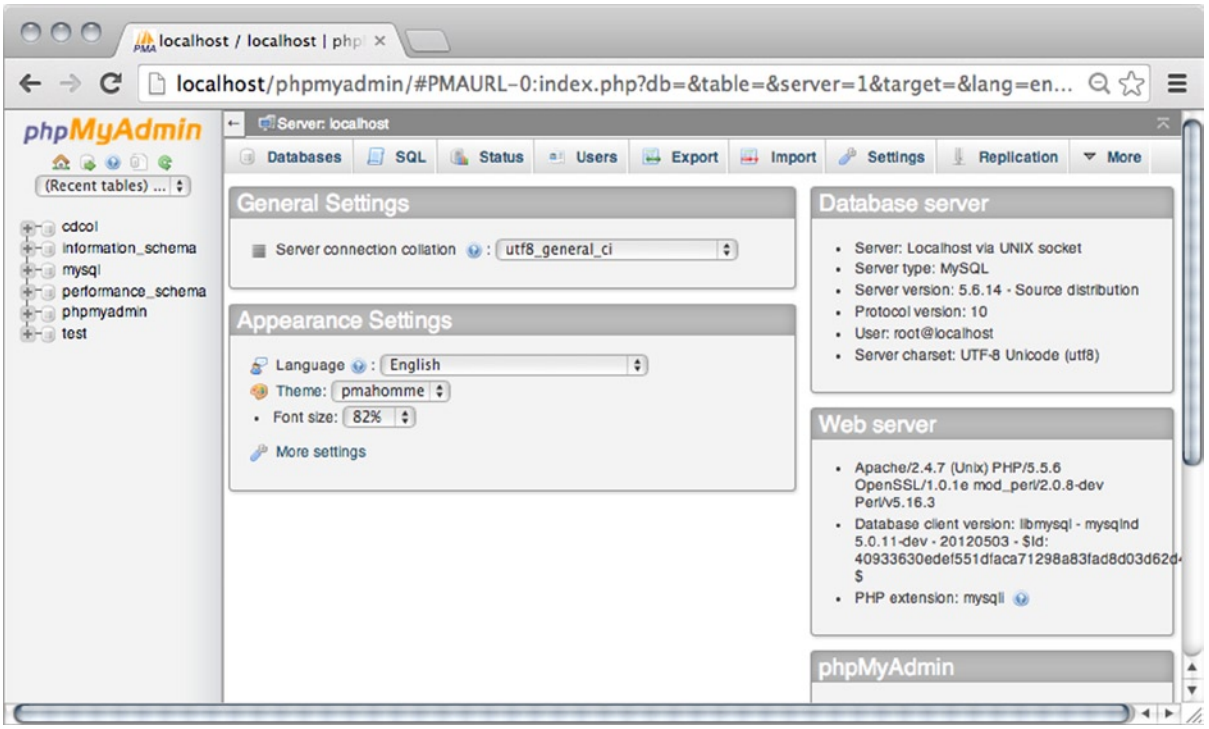


***Figure 6-2.*** *The phpMyAdmin control panel*

# Developing a Database for the Poll

The best way to get a feel for database-driven web pages is to create one for testing. Over the next pages, you will create a database-driven site poll. It is a simple example of database-driven development, but it is quite sufficient to demonstrate the essential principles. The simplest possible site poll will present one question to which site visitors can reply *yes* or *no*. All replies from users will be displayed, so every site user can see how other site visitors replied.

As simple as the example is, it will sum up everything you have seen in the book so far, and it will require you to learn how to integrate database-driven data in your PHP projects.

It is a perfect project for learning *because* it is so simple. It will require relatively few lines of code, which means you can focus on the principles involved, rather than drown in verbose syntax. It will be perfect preparation for the personal blog system you'll get started on in the next chapter.

The site poll relies on a database table to store the poll question and the poll replies. PHP will have to connect to MySQL and retrieve the relevant data, so it can be displayed in a browser as HTML. From PHP, you will also output an HTML form allowing site visitors to interact with the site poll. Whenever a visitor submits the form, PHP should get the submitted answer and update the MySQL database table accordingly. Begin by creating a database with a table and some poll data.

## Creating a Database Using CREATE

SQL uses the word *CREATE* to indicate that a table or database is being created. After you start the CREATE clause, you must indicate whether you're creating a database or a table. In your case, you use the keyword *DATABASE* to indicate that you are, in fact, creating a database. Finally, you have to indicate the name to use for the new database.

MySQL was originally developed in Sweden. So, the default character set used in MySQL is Swedish. Maybe you don't want to use Swedish in your solutions. I like to use utf-8 in my solutions. It is easy to create a database that uses utf-8; you simply have to indicate utf-8 as the character set to use. Your complete command should look like the following:

```
CREATE DATABASE playground CHARSET utf8
```

To execute an SQL statement, you must select the SQL tab in the phpMyAdmin control panel (Figure 6-3). This should bring up a text field that you can use to enter SQL statements. To actually execute the SQL, you have to click the Go button beneath the text field.

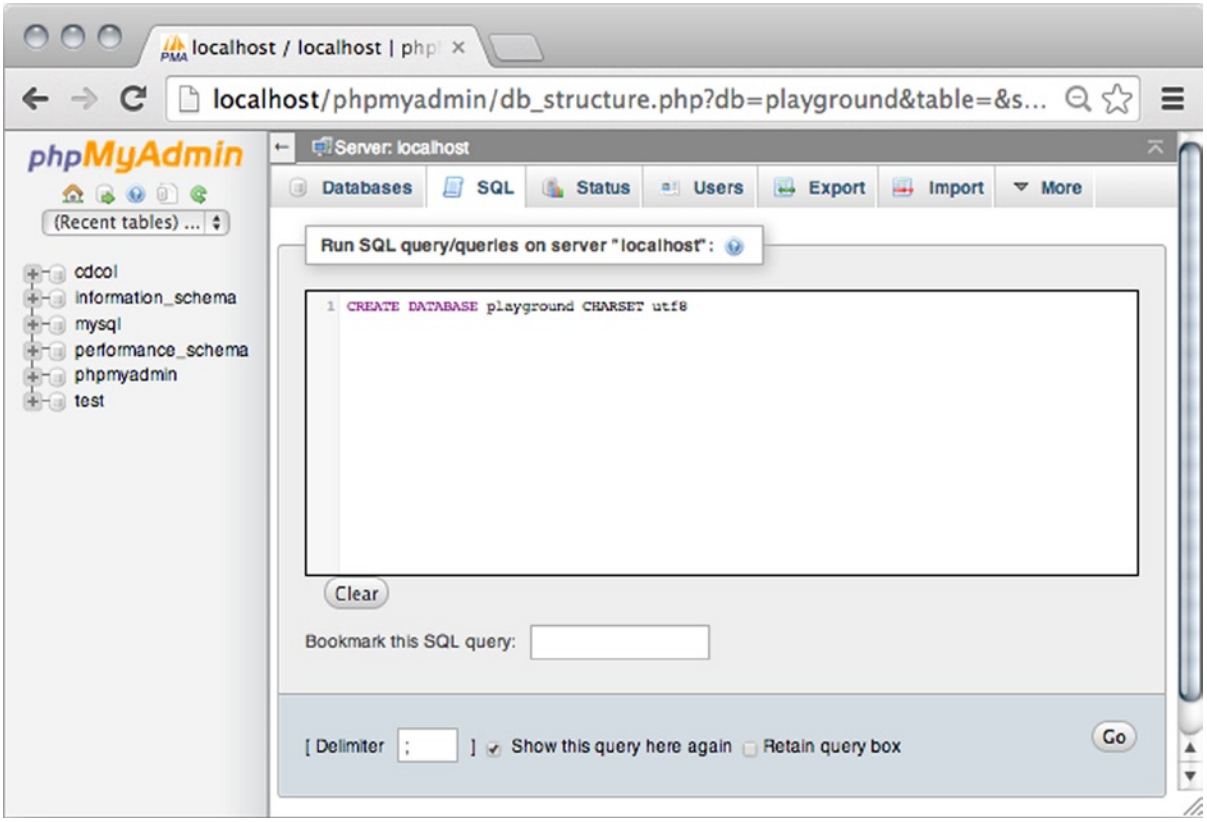


***Figure 6-3.*** *The SQL tab in phpMyAdmin*

## The CREATE TABLE Statement

MySQL stores data in tables. Naturally, the first thing you have to do to start working with MySQL is to create your first table. You have to know a little more SQL to do that. Luckily, SQL syntax is very simple to read and understand. The general syntax for creating a table is as follows:

```
CREATE TABLE table_name (
    column_name datatype [any constraints or default values],
    column_name datatype [any constraints or default values]
)
```

As you can see, an SQL CREATE statement must declare a table name. It should also declare names and data type of every table column or attribute. The SQL statement may declare constraints or default values for the created attributes.

Access the playground database by clicking its name in the left column of the phpMyAdmin control panel. Click the SQL tab at the top of the screen, and you're ready to create your first table. Here's the SQL you require:

```
CREATE TABLE poll (
    poll_id INT NOT NULL AUTO_INCREMENT,
    poll_question TEXT,
    yes INT DEFAULT 0,
    no INT DEFAULT 0,
    PRIMARY KEY (poll_id)
)
```

Once you have entered the SQL into phpMyAdmin's SQL tab, you can click Go to execute the SQL (Figure 6-4). This will create the new table.

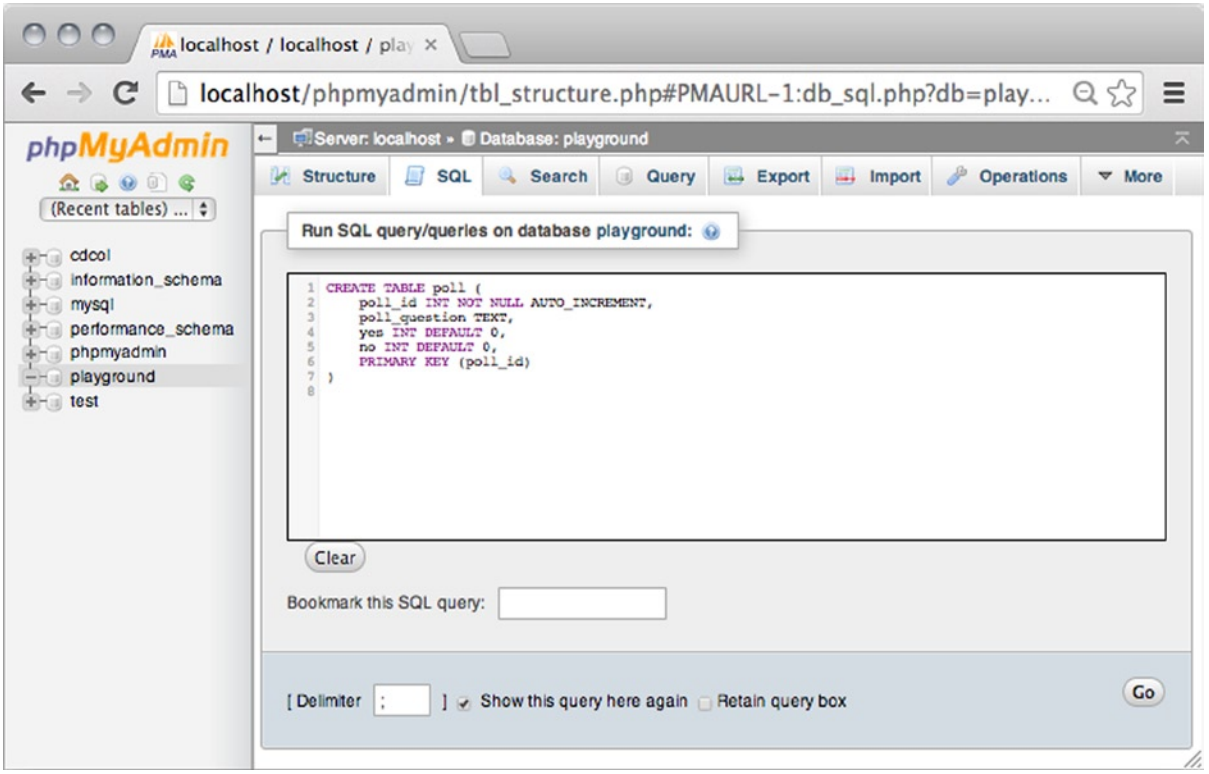


***Figure 6-4.*** *Create a new table in the playground database*

You can explore the new table you have just created by selecting the poll table from the panel at the left side of phpMyAdmin. Next, you select the Structure tab, which you can find next to the SQL tab (Figure 6-5).

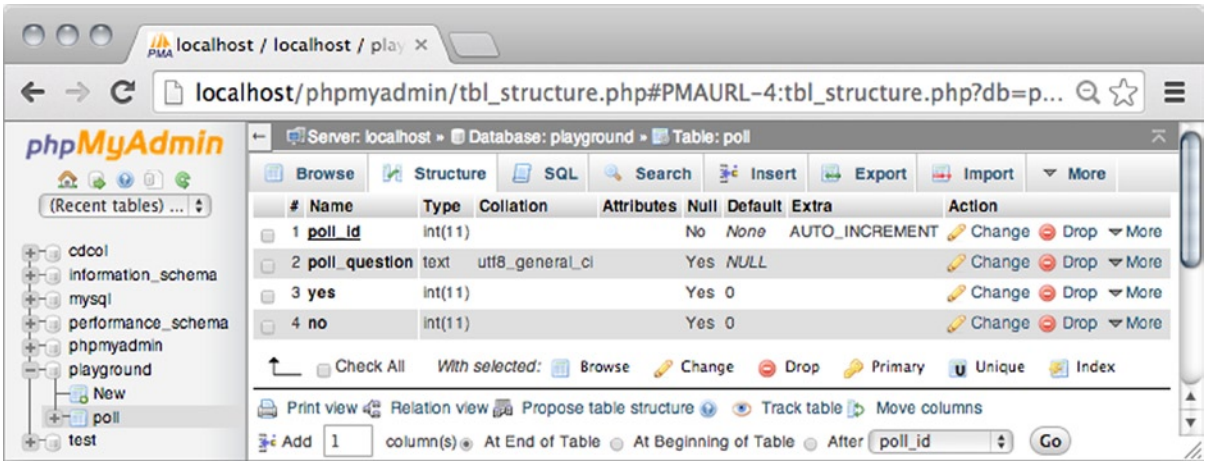


***Figure 6-5.*** *Poll table structure*

There is a lot of information for you to think about here. You can see that the poll table has four attributes, or columns: `poll_id`, `poll_question`, `yes`, and `no`.

You can see that each attribute has a *type*. The fields of a table can only hold data of the correct type. For example, you can only store *integers* in `poll_id`, `yes`, and `no`. You can see that in the preceding figure, because the type is `int(11)`. You can feel free to disregard the number 11. It is there because 11 happens to be the default display width of MySQL integers.

---

■ **Note** An integer is a non-decimal number, i.e., what you might call a whole number.

---

You can also see that you can only store text in `poll_question`. Looking more into `poll_question`, you can also see that the *collation* is utf8.

---

■ **Note** A collation is a set of rules that specify which characters in the character set come first. It is obvious that *a* comes before *b*, but how about the character *7*? Should that come before or after alphabetical characters? And what about special characters such as *#"#€%&*? A collation explicitly states how characters should be ordered.

---

Finally, you can see that the `yes` and `no` attributes are created with a default value of 0. None of the other attributes has default values.

## Understanding PRIMARY KEY

You can see that the `poll_id` attribute is underlined. It is a visual indication that `poll_id` is set to be the primary key of poll entities. When an attribute is declared as primary key, it must hold a unique value. So, however many rows of data the poll table will eventually contain, there can be no two identical `poll_id` values.

Imagine you have a row of data with a `poll_id` of 1. If you were to try to insert another row of data also with a `poll_id` of 1, MySQL would reject the new row and give an error message. A primary key is used to unambiguously identify one row of data. You can actually create tables in MySQL without a primary key, but such tables are special cases.

Most of the time, you will want to create tables with a primary key, because data isn't really useful if you can't identify entries uniquely.

You can see that the poll table is created in such a way that a primary key `poll_id` must have a value. The `poll_id` attribute is declared as NOT NULL, meaning that a null value will not be accepted for `poll_id`. The `poll_id` attribute must invariably hold an `INTeger` value. The `poll_id` attribute cannot be left empty or undeclared.

## Understanding AUTO_INCREMENT

The poll table has an auto-incrementing primary key. It is a simple but powerful idea: the first row of data in the poll table will get a `poll_id` of 1. The next row will automatically get a `poll_id` of 2. The next row will get a `poll_id` of 3, and so forth. The value of `poll_id` will automatically increment.

MySQL will keep track of the values that have already been used as `poll_id`. That way, every new row of data in `poll` will get a unique `poll_id`. In a sense, an auto-incrementing primary key is quite similar to the social security numbers used in many countries across the world to uniquely identify one citizen: it is an arbitrary number used to uniquely identify one *thing*.

## The INSERT Statement

With your table created, you're ready to start storing data. Every new entry into the poll table will be stored as a separate row. For the sake of simplicity, you can start with inserting a single row of data. Here's an SQL statement to that end:

```
INSERT INTO poll (
    poll_question
) VALUES (
    "Is it hard fun to learn PHP?"
)
```

This SQL statement will insert a new row of data into the table, called `poll`. It will declare a value for the `poll_question` column, or attribute. More specifically, the `poll_question` column will get a value of *Is PHP hard fun?* Remember how the poll table has a total of four attributes or columns? The remaining columns, `poll_id`, `yes`, and `no` will simply be created with default values. So `poll_id` will get a value of 1, while `yes` and `no` will both get a value of 0.

To have your MySQL program execute the SQL statement, you must first select the playground database in the phpMyAdmin control panel. Next, you click the SQL tab and enter the preceding SQL statement. Finally, you click Go, to actually execute the entered SQL statement.

I assume you can deduct some of the general syntax for `INSERT` statements. I'd like you to learn the following general syntax, so you can start to formulate your own `INSERT` statements soon:

```
INSERT INTO table_name (
    column_name, other_column_name
) VALUES (
    [data for column], [data for other column]
)
```

When you write an `INSERT` statement, you must first indicate which table you want to insert data into. Then, you indicate which columns of that table you will be inserting data into. If there are more columns in the table, they will get default values.

Once you have indicated the table and columns to which you will be adding insertions, you must list the actual data to insert. If you indicate one column in your `INSERT` statement, you must list one value. If you indicate two columns, you must list two values. In other words, the number of columns must match the number of values indicated in your `INSERT` statement.

## The SELECT Statement

Once you have inserted a row of data into the poll table, you would probably like to see the new row. You probably want some visual confirmation that the row was in fact inserted, so you know you have a database table with a bit of data stored inside. To retrieve data from your database tables, you will have to use an SQL `SELECT` statement. The general syntax for `SELECT` statements is quite simple.

```
SELECT column_name, column_name FROM table_name
```

The main keyword to notice is `SELECT`. It is used to retrieve data specified properties `FROM` a specified table in a database. A `SELECT` statement always returns a temporary table populated with any retrieved data. The temporary table will have exactly the properties indicated immediately after the `SELECT` keyword. You could retrieve data from the poll table with the following SQL statement:

```
SELECT poll_id, poll_question, yes, no FROM poll
```

Please go to the SQL tab in your phpMyAdmin control panel, enter the above `SELECT` statement, and see the returned table (see Figure 6-6).

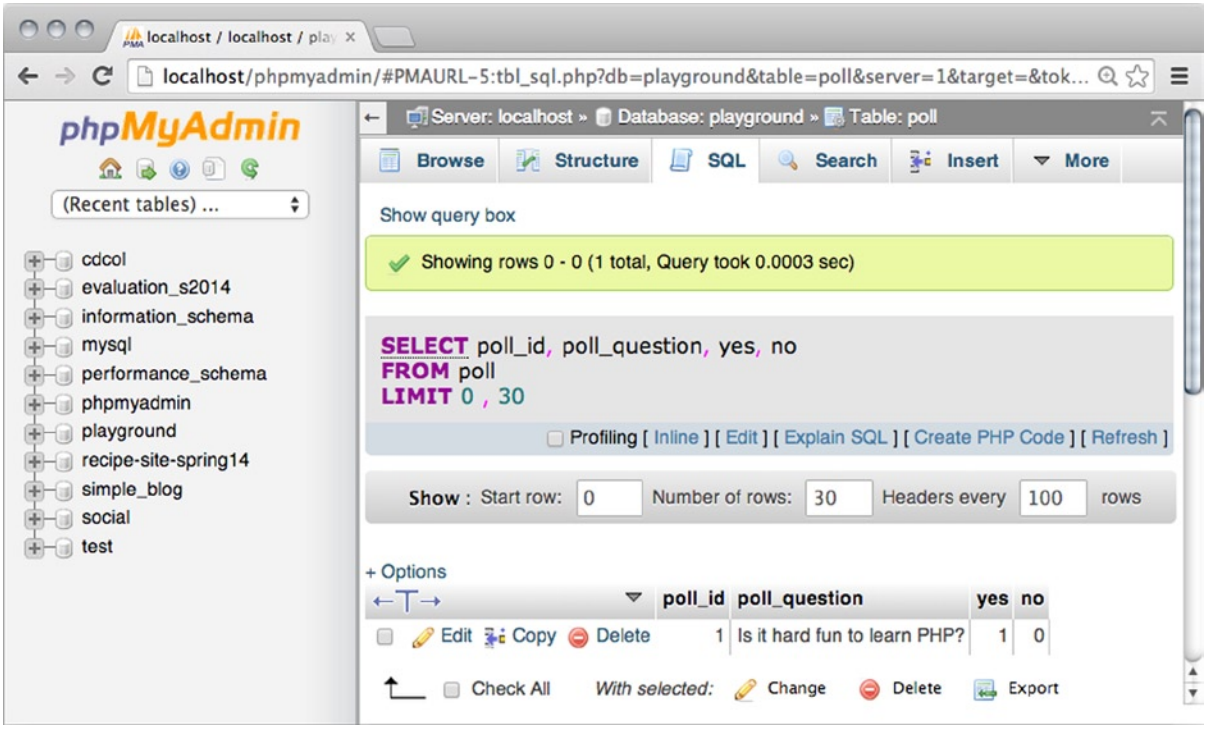


***Figure 6-6.*** *Poll table with one row inserted*

You can see in Figure 6-6 that the `SELECT` statement returns a temporary, unnamed table with four columns, one for each column indicated in the `SELECT` statement. You can see that there is one row of data in the table. It has a `poll_id` of 1 and a `poll_question`. The `yes` and `no` columns are 1 and 0, respectively.

It is not much to look at in its present state, but perhaps you can appreciate that this is all the data you will require to have a site poll displayed on your web site. Your web site would display the poll question. Site visitors could post their responses through an HTML form. Possible options would be *yes* or *no*. All responses from site visitors would be stored in the yes or no fields. So, with a tiny bit of math, you could calculate the relative responses and display a message such as the following: *79% of all site visitors think PHP is hard and fun to learn*.

## The UPDATE Statement

As you can probably work out, you will have to change the `yes` or `no` values in the poll table every time a site visitor submits a response. You must know one more SQL statement to do that. You can pretend a site user just agreed that PHP is hard to learn. You would need an SQL statement to increase the stored value for the `yes` property with a value of 1, as follows:

```
UPDATE poll SET yes = yes + 1
WHERE poll_id = 1
```

If you want, you can run the `UPDATE` statement by entering it into phpMyAdmin's SQL tab and clicking Go. If you do so, you can see that the *yes* property of the first row of data in `poll` gets a value of 1. If you run the same SQL statement again, `yes` will get a value of 2.

Note how the `WHERE` clause limits which rows will be affected by the update. Only the row with a `poll_id` of 1 will be affected. Any other rows in the table will not be updated, because of the `WHERE` clause.

An `UPDATE` statement without a `WHERE` clause would update the `yes` attribute of all rows in the poll table. In your case, there is just one row, so the `WHERE` clause isn't absolutely necessary. But most tables you will work with will have much more than just one row, so it's a good habit to explicitly indicate which row to update.

In the preceding `WHERE` clause, you can be certain that only one row will be updated, because the `WHERE` clause identifies a row of data by its primary key. You can always trust a primary key to uniquely identify a single row (unless your tables are really badly designed).

# Coding a Database-Driven Site Poll

Let's code a database-driven site poll in the interest of learning how to work with MySQL databases from PHP. Let's use the playground database and the poll table for data storage. You will learn to use a so-called PDO object to connect your PHP application to a MySQL database. Stepping up from basic, dynamic PHP sites to database-driven, dynamic PHP sites has some consequences.

Obviously, you will have to connect to a database from PHP, and your PHP scripts will have to communicate with database tables, to get the content you need for your site. PHP is a very forgiving language, and you can approach this task in many ways. But some of these ways are more scalable than others. Some ways that seem easy at first can transform your code into a completely disorganized, tangled, spaghetti mess, once you start to tackle larger projects, such as a blogging system. Let's take a tried-and-tested approach to code architecture that can be scaled to accommodate complex projects, even if this site poll is a simple project.

## Separating Concerns with MVC

The model-view-controller (MVC) design pattern is a common approach to organizing scripts consistently. Using a consistent approach to organizing your scripts can help you develop and debug faster and more efficiently.

Learning to understand the basic principles behind MVC can also prepare you for learning an MVC framework. Eventually, you are likely to come across CodeIgnitor, cakePHP, yii, or some other PHP MVC framework. Such frameworks will aid you in designing and developing more complex web applications.

At its most basic, MVC separates coding concerns into three categories: models, views and controllers. A *model* is a piece of code that represents data. Your models should also hold most logic involved in the system you're building. A *view* is a piece of code that shows information visually. The information to be displayed by the view is received from a model. A *controller* is a piece of code that retrieves input from users and sends commands to relevant model(s). In short, *MVC separates user interactions from visual representation from system logic and data*.

---

■ **Note** You can read much more about MVC at `http://en.wikipedia.org/wiki/Model-view-controller`.

---

You have already seen examples of separating model, view, and controller. Remember how you made a template for HTML pages? You have worked with a view that holds a bare-bones HTML page skeleton. You can find it in the gallery you started building in Chapter 4. The view is in `ch4/templates/page.php`.

In the same project, you created a model related to that view: the `ch4/classes/Page_Data.class.php`, which declares a number of methods and properties related to the content of your HTML pages.

The model and view were hooked up through a controller. In `ch4/index.php`, you assigned values to the model and made the model available to the view, so a well-formed HTML5 page with content could be created and displayed in browsers. So, `index.php` was your controller.

In this book, I aim to use a simple implementation of MVC. Most other MVC implementations you will come across are likely to be much more elaborate. You can easily find many MVC examples that are not meant for beginning programmers. Once you understand the basic MVC principles and have gained some experience working with those principles in a simple context, you will find it much easier to understand the more elaborate implementations.

## Planning the PHP Scripts

Let's keep the poll simple. Create an `index.php` to output a valid HTML5 page that will show the poll. The index will be a *front controller*.

A front controller is a design pattern very often seen in MVC web applications. A front controller is a single "entrance door" to a web application. You have used a front controller already in the projects you have made so far. Remember how `index.php` has been the only script loaded directly in your browser? That's the front controller idea for you.

---

■ **Note** The front controller design pattern is well-documented online. You could start your own research at `http://en.wikipedia.org/wiki/Front_Controller_pattern`.

---

As in the previous projects, `index.php` will output a valid HTML5 page, and it will load the poll controller. The poll controller should return the `poll` as HTML, so it can be displayed on `index.php`. Note how every one view has its own model and its own controller (Figure 6-7).

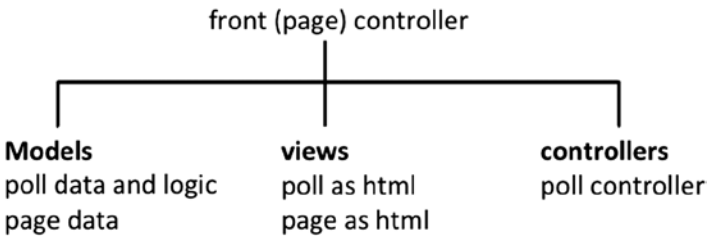


***Figure 6-7.*** *Distribution of responsibilities*

See how there is a poll model, the poll controller, and the poll view. Those three should work together to display a functional poll. You can also see that the page has its own model, view, and controller. The front controller is the page controller.

## Creating the Poll Project

You can create a site structure to mimic code responsibilities. Create a new folder in `XAMPP/htdocs`. Call the new folder `poll`. Inside the poll folder, you can create three other folders: `models`, `views`, and `controllers` (Figure 6-8).

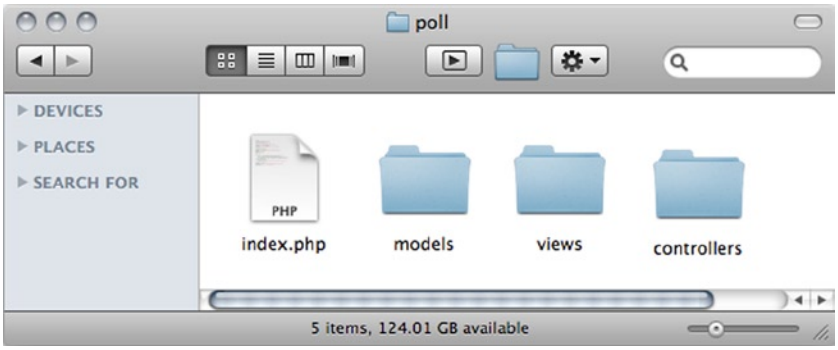


***Figure 6-8.*** *Folder structure for the poll project*

You can copy the `ch4/templates/page.php` file from the gallery project. Save a copy of `page.php` as `poll/views/page.php`.

Likewise, copy `ch4/classes/Page_Data.class.php` from the gallery project and save a copy in `poll/models/Page_Data.class.php`.

Now it is time to create `poll/index.php` and write a little code to check that everything is working nicely together so far:

```
<?php
//complete code for htdocs/poll/index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
//load model
include_once "models/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "PHP/MySQL site poll example";
$pageData->content .= "<h1>Everything works so far!</h1>";
//load view so model data will be merged with the page template
$page = include_once "views/page.php";
//output generated page
echo $page;
```

There should be no surprises for you in this code. It is nearly identical to the code you have used in the other projects in this book. Only folder names have changed!

I would like to think that your perspective has changed too. You are now looking at this code with a model-view-controller perspective. You can test the code by pointing your browser to `http://localhost/poll/index.php`.

You can see how the created HTML page is a combination of a *view* merged with a *model*. You can see how the *front controller* hooks up the model and the view and outputs a well-formed HTML5 page to the browser for the user to see.

## Making a Poll Controller

With a nearly blank page created with its own model and view, and a front controller set up, you can prepare a file for displaying your poll example in the browser. Sticking to the MVC approach, you will eventually need a poll model, a poll view, and a poll controller. The smallest possible step is to create a bare-bones poll controller and load that from the front controller, from `index.php`. Create a new file, `poll.php`, in the `controllers` folder:

```
<?php
//complete code listing for controllers/poll.php
return "Poll will show here soon";
```

Next, you should load the poll controller from `index.php`. You should load the controller somewhere *after* the `$pageData` object was created and *before* the page template is included, as follows:

```
//partial code listing for index.php
//comment out or delete this line
$pageData->content .= "<h1>Everything works so far!</h1>";

//new line of code to load poll controller
$pageData->content = include_once "controllers/poll.php";
//no changes below

$page = include_once "views/page.php";
echo $page;
```

If you save your files and load `http://localhost/poll/index.php` in your browser, you should see *Poll will show here*. If you don't, something went wrong when you typed in the code.

## Making a Poll Model

With a preliminary poll controller in place, you can go on to develop a preliminary poll model. Make a poll class definition with just one method. Create a new file in `models/Poll.class.php`, as follows:

```
 <?php
//complete code for models/Poll.class.php
//beginning of class definition
class Poll {

    public function getPollData() {
        $pollData = new stdClass();
        $pollData->poll_question = "just testing...";
        $pollData->yes = 0;
        $pollData->no = 0;
        return $pollData;
    }
}
//end of class definition
```

Notice how the keyword `class` is used to declare a class name. This class is called `Poll`, and the code inside the class definition defines a blueprint for all `Poll` objects. The `Poll` class has just one method. It will create a hard-coded `StdClass` object called `$pollData` and return it to the caller.

See how the `$pollData` object has properties for `poll_question`, `yes`, and `no`. The `$pollData` object *represents* all the content required to show a poll. In other words, the `$pollData` *models* poll data.

## Making a Poll View

A data object is not much to look at. You can create a simple poll view, so you can get a poll to look at. Create a new file in `views/poll-html.php`, as follows:

```
<?php
//complete code for views/poll-html.php
return "
<aside id='poll'>
    <h1>Poll results</h1>
    <ul>
        <li>$pollData->yes said yes</li>
        <li>$pollData->no said no</li>
    </ul>
</aside>
";
```

## Hooking Up Poll View with Poll Model

With a preliminary poll model and poll view created, you can open the poll controller to hook up model and view and, finally, show something in the browser. Open `controllers/poll.php` in your editor and make the following necessary changes:

```
<?php
//complete code listing for controllers/poll.php
include_once "models/Poll.class.php";
$poll = new Poll();
$pollData = $poll->getPollData();
$pollView = include_once "views/poll-html.php";
return $pollView;
```

That's it! You have an MVC poll. If you save the files and load `http://localhost/poll/index.php` in your browser, you should see a well-formed HTML5 page with a simple `<ul>` element displaying some of your preliminary, hard-coded poll data. You can see what it should look like in Figure 6-9.

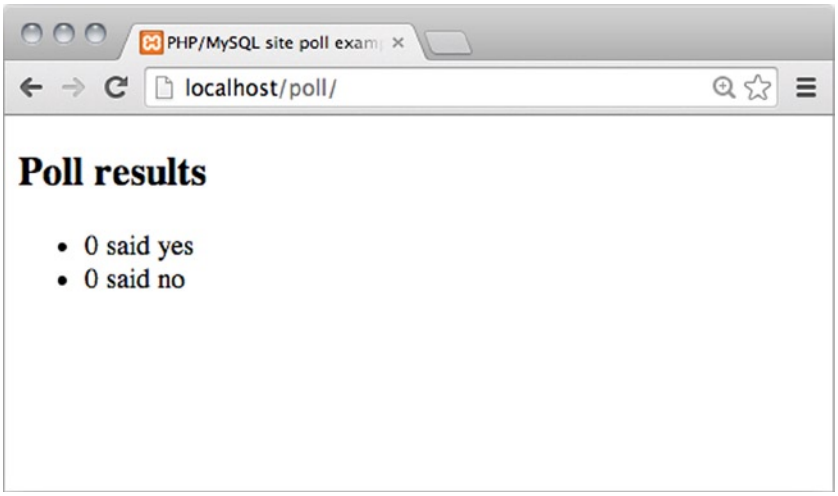


***Figure 6-9.*** *The initial poll, seen in Chrome*

Perhaps you're dying to ask a question such as *Why should I create three different files to show a simple* `<ul>` *element?* It would be a completely justified question to ask. If all you wanted was to show a `<ul>` element with a few hard-coded values, the MVC approach would be complete overkill. The best approach would probably be to write a short HTML file by hand.

The point here is to introduce the MVC design pattern with a very simple example, so there is no overly complex code hiding the basic principles of MVC from your eyes. With the MVC approach, you are perfectly set up for creating a database-driven web application. The MVC architecture is hardly necessary for something this simple, but it can really solve some challenges you will come across with projects of greater complexity, such as the blogging system you'll start making in the next chapter.

MVC encapsulates views from models from controllers. That means you can change a view without changing anything else. Imagine you didn't want a `<ul>` element for your poll. You could simply change the HTML tags used in `views/poll-html.php` and trust the rest of your code to run correctly. You can easily change a view without changing anything else.

Similarly, you could change the content and still trust your code to run as expected. It would be a simple task to set the `no` property to 9. You would only need to change a tiny bit of code in `models/Poll.class.php`. Your poll application is built with self-contained, loosely coupled elements.

## Coding Is Like Playing the Blues

Some creatively inclined readers might object to rigorous organization of code. You may feel that coding, especially if you try to implement a standardized approach such as MVC, is a prison for your creative sensibilities. You might conclude that it leaves no room for creativity.

I see where such objections could come from, but I strongly disagree. Coding is really a lot like playing the blues, and it is every bit as creative and calls as much for individual expressions of creativity.

Yes, implementing MVC requires that you separate your code into three categories. Yes, MVC will force you to write your code in specific, well-defined places. When you are learning, such restrictions can feel like a prison. But it is just like learning to play the blues.

Blues isn't *any* kind of music: Blues is blues! To get that blues sound, you can't just play any note on your instrument. In much blues music, you have three chords, and all blues improvisation stems from a pentatonic scale—giving the musician only five notes to choose from. Perhaps the strict limitations are not counterproductive for creative expression. Perhaps blues musicians are so good at creatively expressing themselves *because* blues is limited to three chords and five notes.

To be a great blues musician, you have to know the rules of blues music intimately. And *then* you start to bend them. You start to add transitional chords to lead the music between those three basic blues chords. You begin to bend those five notes of the pentatonic scale. You start to express yourself creatively within the constraints of blues.

Coding is exactly like that. There are rigorous, restricting rules, and there is much room for individual creative expression. In the process of learning, you will gradually find your own way, but it takes as much effort to become a great coder as it does to become a great musician. So start practicing!

# Connecting to MySQL from PHP

Your MVC architecture will make it a fairly straightforward task to make a database connection and use database-driven data for the poll. Once you have established such a connection, you can retrieve data from your database and publish it as HTML using PHP. That is the essence of database-driven web sites.

There are several ways of connecting PHP to MySQL. You might come across a few different approaches, if you look for PHP code examples on the Internet or in other books. You are quite likely to come across the outdated `MYSQL()` and the updated `MYSQLI()`.

## PHP Data Objects (PDO)

In this book you will exclusively use PHP data objects (PDO). It is a very safe and efficient way of connecting to a database from PHP. PDO supports multiple databases and provides a uniform set of methods for handling most database interactions. This is a great advantage for applications that have to support multiple database types, such as PostgreSQL, Firebird, or Oracle.

With PDO, changing from one database type to another generally requires that you rewrite only a very small amount of code and continue with business as usual.

A potential downside of using PDO is that it relies on the OOP features of PHP5, which means that servers running PHP4 won't be able to run scripts using PDO. This is not much of an issue anymore, as few servers lack access to PHP5; however, it's still something you need to be aware of.

## Opening a Connection

It is time to connect to your database. For the sake of simplicity, I propose that you write the code for connecting in `index.php`. When the database connection is available in the front controller, it will be very easy to pass it on to any other pieces of code that will need it.

A default XAMPP installation has a default username *root* and no password. You created a database called *playground* for this learning exercise. So you can connect to a MySQL database running on your localhost, using these credentials.

Your XAMPP may use different credentials. You will have to use valid credentials. You can create a new database connection by adding a few lines of code in `index.php`, as follows:

```
<?php
//complete code for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
include_once "models/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "PHP/MySQL site poll example";
//new code starts here
//database credentials
$dbInfo = "mysql:host=localhost;dbname=playground";
$dbUser = "root";
```

```
$dbPassword = "";
try {
    //try to create a database connection with a PDO object
    $db = new PDO( $dbInfo, $dbUser, $dbPassword );
    $db->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
    $pageData->content = "<h1>We're connected</h1>";
}catch ( Exception $e ) {
    $pageData->content = "<h1>Connection failed!</h1><p>$e</p>";
}
//comment out loading poll controller
//$pageData->content = include_once "controllers/poll.php";
//end of code changes
$page = include_once "views/page.php";
echo $page;
```

The preceding code creates a PDO object and stores it in the `$db` variable. By default, PDO will hide any error messages. You want to see error messages to learn. The preceding code will display any PDO-related errors as so-called exceptions.

---

■ **Note** There are other possible settings you might use for creating a PDO connection to a database. You can consult `www.php.net/manual/en/book.pdo.php` for complete and detailed coverage.

---

Save the changes in `index.php` and load `http://localhost/poll/index.php` in your browser. If you did everything absolutely correctly, you should see an output in your browser, confirming that you connected successfully. Notice that the poll controller is no longer loaded. We'll load it again shortly. This code only tests if database connection was successfully established.

## Using a try-catch Statement

Many things can go wrong when you try to connect to a database. Perhaps your XAMPP MySQL Server isn't running, or you supplied invalid credentials, such as a misspelled username or a misspelled database name. If your code attempts to connect to a database and fails, your entire script fails. This is because PDO will throw a so-called exception.

Exceptions are interesting, because your code can continue execution if you handle exceptions. That is what a `try-catch` statement can do. It will try to run a block of code that might cause an exception. If an exception is thrown, it will be caught and dealt with, so the remaining script can continue. One advantage is that you can formulate meaningful error messages for users. The general syntax for `try-catch` statements is as follows:

```
try{
    //try something that can fail
} catch ( Exception $e ) {
    //whoops! It did fail
}
```

You can see a `try-catch` statement in action, if you change one line of code in `index.php`. You could try to connect to your database with wrong credentials, as follows:

```
//partial code from index.php
//$dbUser = "root";
//use an invalid database user name for testing purposes
$dbUser = "bla bla";
```

Save and run `index.php`, and you will see the catch block doing its thing. It will handle the thrown exception and output an error message.

Please change back to valid database credentials. By now, you should have a connection to your database. Change your code to load the poll controller again:

```
//partial code listing for index.php
try {
    $db = new PDO( $dbInfo, $dbUser, $dbPassword );
    $db->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
    //delete or comment out the connection message
    //$pageData->content = "<h1>We're connected</h1>";
}catch ( Exception $e ) {
    $pageData->content = "<h1>Connection failed!</h1><p>$e</p>";
}
//loading poll controller again
$pageData->content = include_once "controllers/poll.php";
```

Your code is set up. You will soon be using database-driven content for the poll. But first, a little detour into an important topic in object-oriented programming: constructors.

## Using Constructor Arguments

When programming with an object, you will often have to set a few initial values in new objects created. In PHP, you can do this by using a constructor method. A *constructor* is a method that runs only once during an object's lifetime, when the object is first created.

It is an important topic to know about, and it is a powerful tool for your object-oriented toolbox. You can ease into it and begin with a simple example of a class definition without a constructor. I have created a PHP script in `poll/test.php`. You don't have to; you can simply read the following code example:

```
<?php
class Greeter {
    private $greeting = "Hello";
    private $subject = "World";

    public function greet(){
        return "$this->greeting $this->subject";
    }

}
$greeter = new Greeter();
echo $greeter->greet();
```

If you were to run `http://localhost/poll/test.php`, you would see the following expected output:

```
Hello World
```

You can see how the properties `$greeting` and `$subject` are used to return a string from the method, `greet()`. You can also observe how the `$this` keyword is used to get access to object properties from inside the class definition.

Imagine you wanted a `Greeter` that could be used to say things apart from "Hello." You would have to be able to change the `$greeting` property before calling `greet()`. One approach would be to use an argument when creating a `Greeter` object, as follows:

```
<?php
class Greeter {
    private $greeting = "Hello";
    private $subject = "World";

    //notice the two underscore characters in __construct()
    //declare a constructor method with an argument
    public function __construct( $g ) {
        $this->greeting = $g;
    }

    public function greet(){
        return "$this->greeting $this->subject";
    }

}
//call constructor with an argument
$greeter = new Greeter( "Good Morning" );
echo $greeter->greet();
```

If you were to reload `http://localhost/poll/test.php` in your browser, you would see the following:

```
Good Morning World
```

A constructor method is a method that runs when a new object is created. The process of creating a new object is often called *instantiation* in technical terminology. To declare a constructor for a class, you declare a new method that must be called `__construct()`. Note that there are two underscore characters.

If a class has a constructor method, the constructor method will run as a new object is *instantiated*. So, writing `new Greeter()` will call the constructor of `Greeter`. In the preceding test example, the constructor function takes an argument, so a value must be sent along when it is called. The received value is stored in a predefined property `$greeting`. That way, the received value will be remembered as long as the object lives.

---

■ **Note** It can be tricky to understand arguments. You could reread the explanation about function arguments in Chapter 3, or you could search the Internet, perhaps for "understanding PHP functions with arguments" or "learn PHP methods and arguments." You are bound to find many explanations using a wide variety of metaphors and code examples. I hope you will find an explanation that does the trick for you.

---

## Sharing the Database Connection with the Poll Model

You have already created a database connection, a PDO object, in `index.php`. You can use the idea of constructors with arguments to share a database connection with the poll model. Update your code in `models/Poll.class.php`, as follows:

```
<?php
//complete code listing for  models/Poll.class.php
class Poll {
    //new code: declare a new property
    private $db;

    //new code: declare a constructor
    //method requires a database connection as argument
    public function __construct( $dbConnection ){
        //store the received conection in the $this->db property
        $this->db = $dbConnection;
    }

    //no code changes below - keep method getPollData() as it is
    public function getPollData(){
        $pollData = new stdClass();
        $pollData->poll_question = "just testing...";
        $pollData->yes = 0;
        $pollData->no = 0;
        return $pollData;
    }
}
```

The preceding changes have prepared the `Poll` class. It can now receive a PDO object. So now, you have to call the `Poll` class's constructor and pass a PDO object as an argument. Take a pause to reflect. Where in your code would you load your poll model?

From your poll controller, from `poll/controllers/poll.php`, as follows:

```
<?php
//complete code listing for controllers/poll.php
include_once "models/Poll.class.php";
//Only change here: pass PDO object as argument
$poll = new Poll( $db );
$pollData = $poll->getPollData();
$pollView = include_once "views/poll-html.php";
return $pollView;
```

Now, you have passed the database connection to the poll model, to the new `Poll` object just created. You can save and test your work now. You should expect to see exactly the same poll as before. Your code shares a database connection, but it doesn't use it for anything yet.

Perhaps you are wondering how the PHP code in `controllers/poll.php` can know the variable `$db`. After all, `$db` was declared in `index.php`, so how is it possible to use it in another file? It's a good question, and the answer isn't obvious at all—until you understand it.

The variable `$db` is declared in `index.php`. It is available in `controllers/poll.php` because `controllers/poll.php` is included from `index.php`. Including a file is a lot like copying all the code from one file and pasting it into another file. As a consequence, all variables declared in the including file can be available in the included file and vice versa.

## Retrieving Data with a PDOStatement

You have established a connection from PHP to MySQL running on your local XAMPP. The PDO object is explicitly connected to the playground database you have already created with SQL through the phpMyAdmin control panel.

Inside the playground database, you have a `poll` table with a `poll_question` and values for its `yes` and `no` attributes. You are ready to retrieve data from the database, so you can use it in PHP. It will require a couple of steps in the PHP code.

Code dealing with the data for your poll belongs in the poll model. So, you should open `models/Poll.class.php`. You can change the existing method `getPollData`. At this point, the method returns hard-coded poll data. You want it to return database-driven poll data.

First, you must write an SQL statement and pass that to MySQL by using PDO. PDO can tell MySQL that it should execute the SQL statement and return a result: the poll data. Here's how to express that in PHP:

```
//partial code listing for models/Poll.class.php
//udate existing method
public function getPollData () {
    //the actual SQL statement
    $sql = "SELECT poll_question, yes, no FROM poll WHERE poll_id = 1";
    //Use the PDO connection to create a PDOStatement object
    $statement = $this->db->prepare($sql);
    //tell MySQL to execute the statement
    $statement->execute();
    //retrieve the first row of data from the table
    $pollData = $statement->fetchObject();
    //make poll data available to the caller
    return $pollData;
}
```

The preceding code queries the database and retrieves the first row of data from the poll table. You already have a poll view and a poll controller that hooks up the poll model with the poll view.

Save your work and load `http://localhost/poll/index.php` in your browser, to see it in action. You should see data from the poll table represented and displayed as HTML. Just take a minute to marvel at what you see: a database-driven web page!

You might also appreciate the clean separation between HTML and data. If you don't like the HTML elements I have used in the example, you can easily change them to something else. Changing the HTML elements will not influence the PHP script in any way.

Similarly, you will soon see that the data can change without your having to change the HTML elements. The content of the HTML elements will be updated dynamically, according to whatever values are retrieved from the database.

There is one dependency, though, and it is important you see it. In `views/poll-html.php`, you need a `$pollData` object, and it must have properties for *yes* and *no*. So, in `views/poll.php`, you must create such an object, or the poll will fail. The name and properties of that one object are critical.

## PDO and PDOStatement Objects

I'd like to point out that the SQL string should be converted to a `PDOStatement` object before PDO can pass the SQL to MySQL, where it will then be executed. PDO has a method called `prepare()`, which converts a simple SQL string to a `PDOStatement` object.

PDOStatement objects have another method, called execute(), for getting MySQL to execute the SQL. PDOStatements also have a method, called fetchObject(), to retrieve one row of data from the queried database table.

---

■ **Note** You can consult the official documentation for PDOStatement objects at www.php.net/manual/en/class.pdostatement.php.

---

## The PDOStatement's fetchObject() Method

The fetchObject() method returns an StdClass object that represents one row of data from the queried table. In the preceding code, you have an SQL statement that selects poll_question, yes, and no from the poll table.

Consequently, the returned StdClass object will be created automatically with properties for poll_question, yes, and no. You can access data from the poll table through the StdClass object properties. In the following, you can see for yourself how these properties are used in views/poll-html.php to display data retrieved from the poll table on a web page:

```
<?php
//complete code for views/poll-html.php
//$pollData->no holds the current value of the no attribute
//$pollData->yes holds the current value of the yes attribute
return "
<aside id='poll'>
    <h1>Poll results</h1>
    <ul>
        <li>$pollData->yes said yes</li>
        <li>$pollData->no said no</li>
    </ul>
</aside>
";
```

You have already used an StdClass for $pollData. Initially, you hard-coded properties and values for a poll. The significant difference is that the fetchObject() will automatically create a new StdClass object and return it. Property names of the created StdClass object are identical to the table column names.

For example, $pollData has poll_question, yes, and no properties, because the SQL SELECT statement created a temporary table with poll_question, yes, and no columns. The temporary table was returned from MySQL to PHP. PDO converted the received data to an StdClass object, because you used the fetchObject() method.

## Showing a Poll Form

It's just brilliant that you have database-driven content displayed, but your example isn't much of a site poll yet. Site visitors should be allowed to submit their opinions and, thus, contribute to the poll results. Somehow, you have to provide a graphical user interface to site visitors: an HTML form is the obvious choice. Take a few seconds to think. Where would the HTML for such a form belong? In the model, the view, or the controller? Come up with your best answer before you continue reading...

An HTML form is something users *see*, so it is a *view*. Update the poll view code in `views/poll-html.php`, as follows:

```
<?php
//complete code listing for views/poll-html.php
//new code below
$dataFound = isset( $pollData );
if( $dataFound === false ){
    trigger_error( 'views/poll-html.php needs an $pollData object' );
}

return "
<aside id='poll'>
    <form method='post' action='index.php'>
            <p>$pollData->poll_question</p>
            <select name='user-input'>
                <option value='yes'>Yes, it is!</option>
                <option value='no'>No, not really!</option>
            </select>
            <input type='submit' value='post' />
    </form>
    <h1>Poll results</h1>
    <ul>
        <li>$pollData->yes said yes</li>
        <li>$pollData->no said no</li>
    </ul>
</aside>
";
```

As you can probably figure out, this will show an HTML form very much like the one you created for the dynamic quiz in Chapter 3. You can refresh your browser to see what it looks like.

## Triggering a Custom Error Message

Did you notice those initial lines of code to trigger an error message? Well, you have established that the `$pollData` object is critical for the scripts to collaborate meaningfully about outputting a poll.

If you—or a fellow developer working on the same project—forget to create a `$pollData` object or perhaps misspell it (e.g., `$pillData`), the site poll will not behave as it should. You can test if a `$postData` object is available. If it is not, you can trigger a custom error message. It is really a kind of self-help: should you or another developer incidentally make this mistake, you can trust the system to output a meaningful error message, so the error can be corrected easily and speedily.

When you are developing new projects on your own, you are likely to spend a significant part of your time fixing errors. With custom error messages, you can speed up your development time by testing for errors you predict are likely to occur.

# Updating a Database Table According to Form Input

There is one final step required to complete the site poll example. You should retrieve any user input submitted through the form and update the poll table with the received input. If a site visitor submits a *no*, you should increment the value of the `no` attribute in the poll database table.

It will be much like what you did with the dynamic quiz: you need to detect if the form was submitted. If it was, you can retrieve the submitted input and update the poll table accordingly. You can see there are two concerns here: first, detect input; next, update database.

Updating the database is a task for a *model*. Dealing with user interactions is job for a *controller*. You can start by updating the poll model class with a method for updating the database table. Update `models/Poll.class.php`.

```
<?php
//declare a new method for the Poll class in models/Poll.class.php
//NB. Declare the method inside the Poll class code block
public function updatePoll ( $input ) {
    if ( $input === "yes" ) {
        $updateSQL = "UPDATE poll SET yes = yes+1 WHERE poll_id = 1";
    } else if ( $input === "no" ) {
        $updateSQL = "UPDATE poll SET no = no+1 WHERE poll_id = 1";
    }
    $updateStatement = $this->db->prepare($updateSQL);
    $updateStatement->execute();
}
//no other code changes in Poll class
```

The new method, `updatePoll()`, should be fairly straightforward to understand. If the user submitted a *yes*, you create an SQL string that can update the `yes` attribute in the poll table. If the user submitted a *no*, you create a different SQL string to update the `no` attribute. Notice that no matter if a *yes* or a *no* was submitted, the created SQL string will be stored in a variable called `$updateSQL`.

Once the `$updateSQL` string is created, you use the PDO method `prepare()` to convert the SQL string to a `PDOStatement` object, which is stored in the variable `$updateStatement`. Through the `PDOStatement`, you can `execute()` the query to actually update the poll table.

The `updatePoll()` method will not do anything until it is called. Calling model methods is a job for a controller...

## Responding to User Input

Controllers are responsible for dealing with user interactions. So, the `poll controller` is the right place to intercept user input from the poll form. There are a couple of things to add to the script.

You have to check if the form was submitted at all. If the form was submitted, you should grab the submitted input received through the form. The received input should be passed on to the model, as you call the method to `updatePoll()`. Here's how to formulate that with PHP in `controllers/poll.php`:

```
<?php
//complete code listing for controllers/poll.php
include_once "models/Poll.class.php";
$poll = new Poll( $db );
//check if form was submitted
$isPollSubmitted = isset( $_POST['user-input'] );
//if it was just submitted...
if ( $isPollSubmitted ) {
    //get input received from form
    $input = $_POST['user-input'];

    //...update model
    $poll->updatePoll( $input );
}
```

```
//no changes here
$pollData = $poll->getPollData();
$pollAsHTML = include_once "views/poll-html.php";
return $pollAsHTML;
```

If you enter this code, you will have a fully functional site poll, allowing site visitors to see what other site visitors thought about your poll question. Any site visitor can contribute an opinion through the form. The input will be saved in the database and displayed on `index.php`.

You can see that the preceding script checks if the poll form was submitted. It happens near the top of the script. The code looks for a URL variable named `user-input`. Such a URL variable will be declared when the poll form is submitted, because the poll form has a `<select name='user-input'>` element. The poll form uses the `POST` method, so your code should look for user input in the `$_POST` superglobal, as follows:

```
//check if form was submitted
$isPollSubmitted = isset( $_POST['user-input'] );
if ( $isPollSubmitted ) {
    $input = $_POST['user-input'];
    $poll->updatePoll( $input );
}
```

If the URL variable is set, you know that the `poll` form was submitted. In that case, you can retrieve the site visitor's input. Next, the code calls the `updatePoll()` method in the `Poll` object and passes along the user input as an argument.

# Summary

You have the full picture now—you are developing database-driven web applications with object-oriented PHP! There are probably certain parts of the picture that are still a bit unclear. Some of that will change as you work through the rest of this book. But the real change, the really significant learning, takes place when you start to create your own projects.

When you began reading this book, you probably knew a bit about static web sites, i.e., web sites created with handwritten HTML. With static HTML, you could have written an article about how hard/easy you think PHP is. Users would be able to read your opinion—the communication is one-directional from you to your users.

In the first few chapters, you learned about making dynamic solutions with PHP. You created the dynamic quiz that allowed your users to interact with your content. Because users might have different responses to the quiz question, different users would see different content. Your quiz is an engaging way of communicating your opinion about learning PHP. Users can read if their opinion is the same as yours, and you could write the quiz so it provides humorous responses—the communication is still one-directional, from you to your users.

Now, you are starting to learn about database-driven web sites. When you use a database for content, new possibilities open up. Users can contribute new content to a web site you authored. The new content submitted by users can be published, so that all site visitors can see what the other site visitors contributed. You have made a page that facilitates communication between site visitors. Notice how the communication is now multidirectional, and you are no longer the sole author of the content. These are some of the new possibilities that open up to you, as you progress from static to dynamic to database-driven solutions.

In this chapter, you've learned the basics of SQL statements, as well as how to interact with a database from your PHP scripts. In the next chapters, you'll learn how to build a blog with a basic entry manager that will allow you to create, modify, and delete entries, as well as display them on a public page.

PART II



# A Blogging System

You will learn to design and develop a personal blogging system from scratch, using PHP/MySQL. In the process, you will learn about clean code architecture and design patterns.

CHAPTER 7



# Building the Entry Manager

At this point, you know enough to start building your personal blogging system. The rest of this book covers developing and improving a personal blog. This chapter walks you through how to build the backbone of your blogging application: the blog entry manager. The pieces you'll build include the following:

- A view, which is an HTML form to accept entry input
- A controller, to handle input from the form
- A model, to save and retrieve the entry in the database

By the end of this chapter, you will have a basic entry manager for your personal blog system. In the process of building the entry manager, you will get to revisit previously covered topics, such as basic database design, SQL, organizing your PHP with MVC, and connecting to a database.

## Creating the blog_entry Database Table

One of the most important steps with any new application is the planning of the tables that will hold data. This has a huge impact on the ease of scaling the application later. *Scaling* is the expansion of an application to handle more information and/or users, and it can be a tremendous pain, if you don't look ahead when starting a new project. At first, your blog needs to store several types of entry information to function, including the following:

- Unique ID
- Entry title
- Entry text
- Date created

Using a unique ID for each entry in the entry table enables you to access the information contained with just a number. This is extremely helpful for data access, especially if the data set changes in the future (if you add an "imageURL" column to the table, for example).

The first step is to determine the fields you will require for the entries table. Your table must define what type of information is stored in each column, so let's take a quick look at the information each column has to store.

- `entry_id`: A unique number identifying the entry. This will be a positive integer, and it makes sense for this number to increment automatically, because that ensures the number is unique. Because each entry has a unique `entry_id`, you can use the number to identify one entry. The `entry_id` will be the primary key for the table.
- `title`: An alphanumeric string that should be relatively short. You'll limit the string to 150 characters.

- `entry_text`: An alphanumeric string of indeterminate length. You won't limit the length of this field (within reason).
- `date_created`: The timestamp generated automatically at the original creation date of the entry. You'll use this to sort your entries chronologically, as well as for letting your users know when an entry was posted originally.

Now it's time to create the database. Open your XAMPP control panel to start MySQL and Apache. Point your browser to `http://localhost/phpmyadmin/` and select the SQL tab to create a database called `simple_blog`. Here's the SQL to do it:

```
CREATE DATABASE simple_blog CHARSET utf8
```

The next step is to write the code that creates your `entries` table. Make sure to select the `simple_blog` database from the menu at the left side of the phpMyAdmin control panel. Next, bring up the SQL tab and enter the following SQL statement:

```
CREATE TABLE blog_entry (
    entry_id INT NOT NULL AUTO_INCREMENT,
    title VARCHAR( 150 ),
    entry_text TEXT,
    date_created TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY ( entry_id )
)
```

You can see that the statement to create the `blog_entry` table is similar to the `CREATE` statement you used to create the poll table. You can also see there are some differences.

The `title` attribute has a new data type: `VARCHAR(150)`. As a consequence, any title must contain `CHAR`acters of `VAR`iable length. Blog entry titles can be a string of characters between 0 and 150 characters long. If you were to insert a title that was 151 characters long, only the first 150 characters would be saved in the `blog_entry` table. That's what `VARCHAR(150)` does.

The `date_created` attribute is also declared with a new data type: `TIMESTAMP`. A `TIMESTAMP` holds rather precise information about a moment in time. It stores *year*, *month*, *day*, *hour*, *minute,* and *second* as YYYY-MM-DD HH:MM:SS

You have already seen how MySQL table attributes can be created with a default value. Here it is again for the `date_created` attribute. When a new entry is inserted for the first time, MySQL will automatically store the current `TIMESTAMP` based on the server's clock.

# Planning the PHP Scripts

You have created a database for your blog. A logical next step would be to create a blog entry editor with PHP, so that you can create new blog entries. The blog entry editor is only meant for you as blog author. Ordinary site visitors should not be able to create new entries. Normal site visitors should simply see your blog entries, without being able to edit existing entries or create new ones.

One approach to such a task is to create two main site entrances: `index.php` for regular visitors and `admin.php` for your eyes only. In MVC terminology, `index.php` and `admin.php` will both be *front controllers*. Later in this book, I show you how to restrict access to `admin.php`, with a login.

The admin page should be able to list all blog entries, and it should give you access to an entry editor, so that you can create new entries and edit or delete existing entries. You will require separate views: one for listing all entries and one for showing the editor.

The script `admin.php` should output an HTML5 page. It will be your front controller, and it will decide whether to show the editor or list all entries. Figure 7-1 uses the MVC idea to develop a schematic overview of the blog administration module.

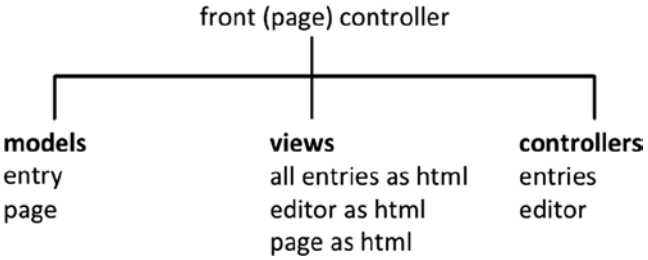


***Figure 7-1.*** *Distribution of responsibilities*

Note how every view has a corresponding controller. Note, too, that the entry model is used by entries to display all entries and reused by the editor to save a new entry. Let's just go over the separation of concerns proposed by the MVC approach (see Figure 7-2). A view is something that can be seen by users. A model holds content. The controller is responsible for hooking up the right view with the right model and returning the resulting output to users. The controller is also responsible for responding to user input; often this means updating the model.

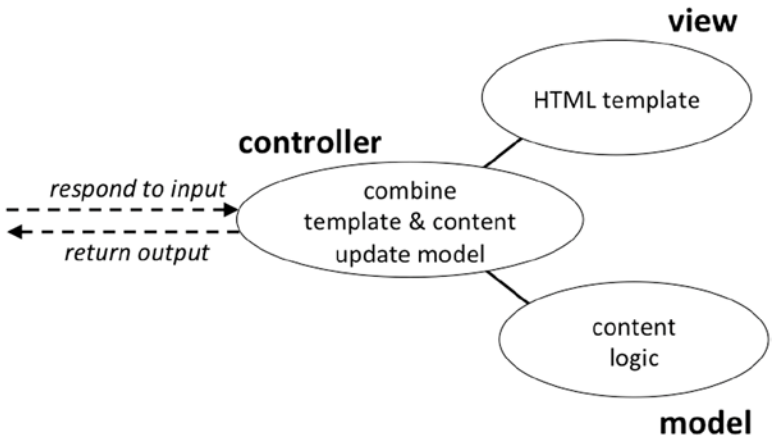


***Figure 7-2.*** *Model-view-controller*

# Creating the Blog Site

Create a new folder in `XAMPP/htdocs`. Call the new folder `blog`. Inside the `blog` folder, you can create four other folders: `models`, `views`, `controllers`, and `css`.

You can copy the `ch4/templates/page.php` file from the gallery project. Save a copy of `page.php` in `blog/views/page.php`. Likewise, copy `ch4/classes/Page_Data.class.php` from the gallery project and save a copy in `blog/models/Page_Data.class.php`. Create a blank style sheet in `css/blog.css`.

Now it is time to create `blog/admin.php` and write a little code to check that everything is working nicely together so far:

```
<?php
//complete code for blog/admin.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );

include_once "models/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "PHP/MySQL blog demo";
$pageData->addCSS("css/blog.css");
$pageData->content = "<h1>YES!</h1>";

$page = include_once "views/page.php";
echo $page;
```

Save your files and load `http://localhost/blog/admin.php` in your browser. If everything works as expected, you should get a well-formed HTML5 with a `<title>` of *PHP/MySQL blog demo* and an `<h1>` element happily exclaiming *YES*! Seeing YES in the browser is visual confirmation that your project is set up correctly.

## Creating the Entry Manager Navigation

Your entry manager should have two base views: one to list all entries and one to show an entry editor. Let's make a navigation for the entry manager. You can expect this project to contain many PHP files before you're done. I recommend you create some folders to keep scripts related to the administration module grouped together. Create a folder called `admin` in the existing `views` folder. Create a new file in `views/admin/admin-navigation.php`:

```
<?php
//complete code for views/admin/admin-navigation.php

return "
<nav id='admin-navigation'>
    <a href='admin.php?page=entries'>All entries</a>
    <a href='admin.php?page=editor'>Editor</a>
</nav>";
```

You can see that the entry manager navigation is very similar to the navigation you made for the dynamic gallery in Chapters 4 and 5, or for the dynamic portfolio site from Chapter 2, for that matter. You should make a mental note about the URL variable `page` that gets encoded whenever a user clicks a navigation item.

The admin navigation is a *static view*, meaning there is no dynamic or database-driven information in the script. You don't need a model for the navigation, because all content is hard-coded into the view as it is. You do need a controller to load the navigation whenever it should be loaded. It will be quite simple, because the navigation should be loaded and displayed all the time. You can control that from `admin.php`:

```
//partial code listing for admin.php
$pageData->addCSS("css/blog.css");
//code changes below here
//comment out or delete the YES
//$pageData->content = "<h1>YES!</h1>";
//load navigation
```

```
$pageData->content = include_once "views/admin/admin-navigation.php";
//no changes below
$page = include_once "views/page.php";
echo $page;
```

Save your work and reload `http://localhost/blog/admin.php` in your browser. You should see the navigation displayed in the top of your browser window. Clicking a navigation item will not have any immediately visible effect; the navigation is just a view. Clicking any navigation item will encode a URL variable named `page`. You can see it if you look in the browser's address bar. You'll want to create controller code to respond to user interactions, such as a click.

## Loading Admin Module Controllers

You can use `admin.php` to control what to do when a navigation item is clicked. That is the main concern for a front controller. The front controller should load any controller(s) associated with the navigation item a user clicked. There are two links in your navigation, so you will need two controllers.

To see any visible changes in the browser when you click a navigation item, you have to create two preliminary controllers. Create a new folder called `admin` in the `controllers` folder and create a new file for controlling the entry editor view, as follows:

```
<?php
//complete source code for controllers/admin/editor.php
return "<h1>editor controller loaded!</h1>";
```

As you can see, the editor controller will not be doing a whole lot to begin with. Initially, you just want to check that you hook up the right files. With that in place, you can develop code of greater complexity. Create another file for controlling the view that will eventually list all entries, as follows:

```
<?php
//complete source code for controllers/admin/entries.php

return "<h1>entries controller loaded!</h1>";
```

You can load these controllers from `admin.php`. You have to check whether a navigation item was clicked. If it was, you should load the corresponding controller. Insert these lines of code *after* the `$pageData` object is created and *before* the `echo`:

```
<?php
//complete code for blog/admin.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );

include_once "models/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "PHP/MySQL blog demo";
$pageData->addCSS("css/blog.css");
$pageData->content = include_once "views/admin/admin-navigation.php";
```

```
//new code begins here
$navigationIsClicked = isset( $_GET['page'] );
if ( $navigationIsClicked ) {
    //prepare to load corresponding controller
    $contrl = $_GET['page'];
} else {
    //prepare to load default controller
    $contrl = "entries";
}
//load the controller
$pageData->content .=include_once "controllers/admin/$contrl.php";
//end of new code

$page = include_once "views/page.php";
echo $page;
```

Remember the file names `entries.php` and `editor.php`. These names are essential. They must be matched with corresponding values of the URL variable `page` declared when a user clicks a navigation item. Let's take a closer look at the `href` values used in the navigation:

```
<a href='admin.php?page=entries'>All entries</a>
<a href='admin.php?page=editor'>Editor</a>
```

When a user clicks the `All entries` item, the URL variable `page` gets a value of `entries`. In `admin.php`, the URL variable `page` is fetched using `$_GET`. The string value `entries` is stored inside a variable called `$contrl` and subsequently used to include `controllers/admin/$contrl.php`, which will really translate to including `controllers/admin/entries.php`, because the variable `$contrl` holds the value `entries`.

If the `Editor` item is clicked, `controllers/admin/editor.php` will be included.

There should be no surprises for you in the preceding code. Still, a sanity check is advisable. Save your files and reload `http://localhost/blog/admin.php` in your browser. By default, you should see the message returned from the entries controller. If you click the navigation item for the editor, you should see the returned message from the editor controller.

## Creating the Entry Input Form

Now that you have a dynamic navigation, you might as well use it for something meaningful. You could show an HTML form for the editor. Eventually, it should be possible to use the editor to create new blog entries. So, the editor form should have fields for creating an entry title and an entry article. There should also be a button for saving a new entry. While you're at it, you might as well create a button for deleting an entry. Create a new file `views/admin/editor-html.php`, as follows:

```
<?php
//complete source code for views/admin/editor-html.php
return "
<form method='post' action='admin.php?page=editor' id='editor'>
    <fieldset>
        <legend>New Entry Submission</legend>
        <label>Title</label>
        <input type='text' name='title' maxlength='150' />
```

```
        <label>Entry</label>
        <textarea name='entry'></textarea>

        <fieldset id='editor-buttons'>
            <input type='submit' name='action' value='save' />
            <input type='submit' name='action' value='delete' />
        </fieldset>
    </fieldset>
</form>
";
```

Most of the preceding code should be familiar ground, even if you see a few elements you might not have come across before. You can see two `<fieldset>` elements. They are used to group related form fields together. The main `<fieldset>` has a `<legend>` element. A `<legend>` is like a heading for a `<fieldset>` element.

The `<input>` element for the entry title has a `maxlength` attribute set to 150. You can probably guess that the displayed text field will only accept 150 characters. That's perfect, because your entry table in the database accepts a maximum of 150 characters for new `title` attributes.

The `maxlength` attribute enhances form usability, in that it becomes harder for users to create an invalid title through the form. The `maxlength` attribute performs client-side validation and will only allow submission of valid titles. One thing to keep in mind is the fact that client-side validation is great for enhancing usability. It does not improve security, because you should expect a malicious user to be able to override client-side validation.

With a new editor view created, you have to update the controller so it shows the view. The controller for the editor can be found in `controller/admin/editor.php`. Change the code so it is like the following:

```
<?php
//complete source code for controllers/admin/editor.php
$editorOutput = include_once "views/admin/editor-html.php";
return $editorOutput;
```

These few lines of code should hook up the view and display it when the controller is loaded from `admin.php`. You can see it for yourself by reloading `http://localhost/blog/admin.php?page=editor` in your browser. You should expect to see the form.

## Styling the Editor

You'll probably agree that the unstyled entry editor form is hideous to look at. A little CSS can take you a long way toward improved aesthetics. You will most likely want to work more on the visual design of the editor. Here's a little CSS to get you started:

```
/* code listing for blog/css/blog.css */
form#editor{
    width: 300px;
    margin:0px;
    padding:0px;
}

form#editor label, form#editor input[type='text']{
    display:block;
}
```

```
form#editor #editor-buttons{
    border:none;
    text-align:right;
}

form#editor textarea, form#editor input[type='text']{
    width:90%;
    margin-bottom:2em;
}

form#editor textarea{
    height:10em;
}
```

As you may or may not recall, my code in `admin.php` expects to find an external style sheet in `css/blog.css`. If your `admin.php` is like mine, you'll want to have your style sheet in that location. Obviously, you could create your style sheet in a different folder, under a different name. Remember to update `admin.php`, so that it points to your style sheet. You can see the resulting editor design in Figure 7-3.


***Figure 7-3.*** *The editor displayed in Google Chrome*

I like to keep my HTML clean and free of `id` and `class` attributes, and I like to hand-code my CSS. I need to be able to hook up style rules with the right HTML elements. I've found that I can often get by, using contextual selectors combined with attribute selectors.

If you prefer to use a CSS framework, you will probably take a completely opposite approach. You will write very little CSS by hand and make extensive use of the `class` and `id` attributes used by your favorite CSS framework. Do as you see fit!

## Connecting to the Database

You've completed the basic editor view. Soon, you should be able to insert new blog entries into your `blog_entry` database table through the editor form. To do that, you will need a database connection from your PHP application to your MySQL database.

You can take the same approach you used for the poll: use PDO for making a connection and make a connection in the front controller to share it with subsequently loaded controllers.

You can create a PDO object in `admin.php` and share it with all controllers you'll be loading. You should write the following code in `admin.php` somewhere *after* you create `$pageData` and *before* the `echo`:

```
//partial code listing for admin.php

//new code starts here
$dbInfo = "mysql:host=localhost;dbname=simple_blog";
$dbUser = "root";
$dbPassword = "";
$db = new PDO( $dbInfo, $dbUser, $dbPassword );
$db->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
//end of new code - no changes below

$navigationIsClicked = isset( $_GET['page'] );
if ($navigationIsClicked ) {
    $contrl = $_GET['page'];
} else {
    $contrl = "entries";
}
$pageData->content .=include_once "controllers/admin/$contrl.php";
$page = include_once "views/page.php";
echo $page;
```

Be careful to enter correct database credentials and to indicate that you want to connect to the `simple_blog` database. You can test if the connection works by reloading your browser. If the connection fails, you should see an error message. Consequently, if you see no error message, it means you have successfully connected to a database.

## Using Design Patterns

A design pattern is a general, best-practice solution to a common task. Some design patterns are defined rather comprehensively. As your experience grows, you'll come across more design patterns. As an absolute beginner, you don't need a comprehensive treatment of design patterns—that would likely be more confusing than helpful.

This book uses simple implementations of a few design patterns. You've already seen a simple implementation of the front controller design pattern and an equally simple implementation of MVC. You can see it is possible to combine several design patterns in the same project.

---

■ **Note** You can read about design patterns at http://en.wikipedia.org/wiki/Software_design_pattern.

---

Many design patterns tend to be quite tricky for beginners to understand. But there are design patterns for many, common coding problems. This book uses a few design patterns in simple implementations. You should know which design patterns you're using in your code and that there are other possibilities. You can consider such knowledge a road map for future development.

■ **Note** I can highly recommend Matt Zandstra's book *PHP Objects, Patterns, and Practice*, Fourth Edition (Apress, 2013). Find it at www.apress.com/9781430260318. It is not really a book for beginners, but you can keep it in mind for future reference.

## The Table Data Gateway Design Pattern

Your code will have to communicate extensively with database tables. There are a number of approaches to managing such communication consistently. One approach is to implement a table data gateway design pattern.

The *table data gateway* design pattern is not the most commonly seen approach to dealing with database access in larger MVC frameworks. Many popular PHP MVC frameworks use an *active record* pattern. I propose that you use a table data gateway, because it is a relatively simple design pattern to understand. The table data gateway pattern specifies that you create one (PHP) class for every one table in your database. The idea is that all communication between your system and that one table happens through one such object. The table data gateway encapsulates data communication between your system and the specific database table.

That means that all SQL statements needed for this communication will be declared in the table data gateway class definition. This has a couple of advantages: One is that you know where to write your SQL statements. Consequently, you also know where to find your SQL statements related to the particular database table.

Database design and development is a profession in and of itself. If you as a PHP developer keep all your SQL encapsulated in relatively few class definitions, then any database expert on your team would only have to work with these few classes. That is a huge advantage over having your SQL statements scattered all through your code base.

## Writing the Entry_Table Class

Initially, you'll need to be able to insert new blog entries received from the editor form. You can create a new class definition for a table data gateway for the `blog_entry` table. It will be used to communicate with your database.

The new class will need a PDO object to communicate with the database. You can use the idea you already saw in the previous chapter. Create a class with a constructor method and pass a PDO object as an argument. Take a pause to reflect on the class you're about to write. Should it be a model, a view, or a controller? The answer should be obvious: a script responsible for communicating with your database should be a model!

Create a new class definition in the `models` folder, in a new file called `Blog_Entry_Table.class.php`. Here's the code for it:

```
<?php
//complete code listing for models/Blog_Entry_Table.class.php

class Blog_Entry_Table {

    private $db;

    //notice there are two underscore characters in __construct
    public function __construct ( $db ) {
        $this->db = $db;
    }

    public function saveEntry ( $title, $entry ) {
        $entrySQL = "INSERT INTO blog_entry ( title, entry_text )
                     VALUES ( '$title', '$entry' )";
        $entryStatement = $this->db->prepare( $entrySQL );
```

```
        try{
            $entryStatement->execute();
        } catch (Exception $e){
            $msg = "<p>You tried to run this sql: $entrySQL<p>
                    <p>Exception: $e</p>";
            trigger_error($msg);
        }
    }
}
```

The `Blog_Entry_Table` will not do anything until it is used from another script. I'd like to take a look at this code before using it. As you can see, the `Blog_Entry_Table` has one property `db` and two methods: the `constructor` and `saveEntry()`. This constructor takes a PDO object as argument. The received PDO object will be stored in the `db` property. This way, all methods of `Blog_Entry_Table` will have access to the PDO object and, through that, access to the `simple_blog` database.

In object-oriented terminology, `Blog_Entry_Table` and PDO are now associated through a `has-a` relationship. The `Blog_Entry_Table has-a` PDO object.

At this early stage of development, you'll only be saving new entries. So, the `Blog_Entry_Table` class has just one method besides the constructor. The `saveEntry()` method takes two arguments: the `title` and the `blog_entry` to save in the database.

The variable `$entrySQL` holds an SQL string to insert a new `blog_entry` using the received `title` and `blog_entry`. The `$entryStatement` is a `PDOStatement` object, which you can then `try()` to `execute()` to actually insert a new `blog_entry`.

If this operation fails, it will throw an `Exception` object, which your code catches. In case an exception occurs, your code will trigger an error showing the SQL string that caused the exception and a more detailed look at the exception.

## Processing Form Input and Saving the Entry

With the `Blog_Entry_Table` class created, you can continue development. A logical next step could be to process input received from the editor form and use a `Blog_Entry_Table` object to save a new blog entry in the database. Which is responsible for dealing with user interactions? Models, views, or controllers? The MVC design pattern specifies that user interactions should be dealt with in the relevant controller. In this case, the relevant controller would be `controllers/admin/editor.php`:

```
<?php
//complete code for controllers/admin/editor.php

//include class definition and create an object
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );

//was editor form submitted?
$editorSubmitted = isset( $_POST['action'] );
if ( $editorSubmitted ) {
    $buttonClicked = $_POST['action'];
    //was "save" button clicked
    $insertNewEntry = ( $buttonClicked === 'save' );
```

```
    if ( $insertNewEntry ) {
        //get title and entry data from editor form
        $title = $_POST['title'];
        $entry = $_POST['entry'];
        //save the new entry
        $entryTable->saveEntry( $title, $entry );
    }
}

//load relevant view
$editorOutput = include_once "views/admin/editor-html.php";
return $editorOutput;
```

Save and test your editor. It should be able to insert new blog entries. Try to create a few new blog entries through your editor form. Once you have done that, you can browse the content of your `blog_entry` table using the phpMyAdmin control panel. You can find the created blog entries in your `blog_entry` table. Your entry editor works!

## Which Button Was Clicked?

There are two buttons in the entry editor form. Eventually, you'll want your script to respond differently, depending on which button was clicked. So, your code must know which button was clicked. Let's take a look at the HTML that produces the buttons:

```
//partial source code for views/admin/editor-html.php
<fieldset id='editor-buttons'>
    <input type='submit' name='action' value='save' />
    <input type='submit' name='action' value='delete' />
</fieldset>
```

See how the two different buttons have the same `name` attribute. The only difference between buttons are their `value` attributes. You can use that knowledge to identify which button was clicked. Actually, you already did it in `controllers/admin/editor.php`:

```
//one line of code from controllers/admin/editor.php
//don't make any changes
$buttonClicked = $_POST['action'];
```

The variable `$buttonClicked` will hold the value of the clicked button. So, if a user clicked the button to save, `$buttonClicked` will hold the corresponding value `'save'`. If `$buttonClicked` has a value of `save`, you know the user is trying to insert a new entry. Take a look at the following line. What will the variable `$insertNewEntry` hold?

```
//one line of code from controllers/admin/editor.php
//don't make any changes
$insertNewEntry = ( $buttonClicked === 'save' );
```

Inside the parentheses, the variable `$buttonClicked` is compared to the string `'save'`. If `$buttonClicked` is identical to `'save'`, `$insertNewEntry` will have a value of `true`. If not, it will have a value of `false`. In the subsequent code, you can see that a new `blog_entry` is inserted if `$insertNewEntry` is true. So much of your code will only run if a user clicked the Save button.

## Security Alert: SQL Injection Attacks

The `saveEntry()` method illustrates a nifty coding practice. It uses form input data to generate an SQL string dynamically. It's a wonderful trick to know, because it's the key to insert input received from a user into a database table. But it's also a gaping security hole. SQL injection attacks are possibly the single most common attack on database-driven web applications. Your entry editor is wide open for such attacks now.

---

■ **Note** Learn more about SQL injections at http://en.wikipedia.org/wiki/SQL_injection.

---

Basically, SQL injection attacks exploit the fact that text entered through a form is used to generate SQL statements dynamically. Anything a user enters through the form will be used in SQL:

```
//the code that makes you vulnerable
$entrySQL = "INSERT INTO blog_entry ( title, entry_text )
             VALUES ( '$title', '$entry' )";
```

Look at the preceding code. The variables `$title` and `$entry` are simple placeholders for text received from a form. Whatever was entered through the form will become part of the SQL string. So, a malicious user with a solid understanding of SQL can enter evil SQL through the form and thus gain direct access to your database tables. Any data you keep in your database may be exposed. An attacker might submit the form with a string somewhat like the following entered into the entry field:

```
attack'); DROP TABLE blog_entry;--
```

With that input, `$entrySQL` would become

```
$entrySQL = "INSERT INTO blog_entry ( title, entry_text )
             VALUES ( '$title', ' attack'); DROP TABLE blog_entry;--' )";
```

You can see that the `$entrySQL` now holds two SQL statements: one to insert a `blog_entry` and another one that will drop the entire `blog_entry` table. It should be obvious that you don't want users to be able to drop tables in your system! You should know that the particular attack described previously will not actually work. I don't want to teach you how to perform SQL injection attacks, but I would like you to know about the vulnerability.

## Usability Alert: Blog Entries with Quote Characters

The entry editor also has a major usability flaw. (As if it weren't enough that your editor is vulnerable for the most common type of hacker attack!) You can see the problem for yourself, if you enter the following text through the editor's entry field:

```
Brennan's child said: "I want some breakfast!"
```

PDO throws an exception. The culprit is the single-quote character:

```
//the code that makes your form submission break
$entrySQL = "INSERT INTO blog_entry ( title, entry_text )
             VALUES ( '$title', '$entry' )";
```

Because the variables `$title` and `$entry` are wrapped in single-quote characters, your entry editor form cannot deal with any content that contains single-quote characters. So, the sentence you typed causes an exception.

With a minor change in code, you could wrap `$title` and `$entry` in double quotes instead, but then your entry editor wouldn't be able to handle any entry with a double-quote character. So, the sentence would still cause an exception.

## Solution: Prepared Statements

It should be evident that the code where the SQL string is generated dynamically gives you a lot of grief. It leaves you vulnerable to SQL injection attacks, and because of it, your blog entries must be written without single- or double-quote characters. Graciously, PDO supports a feature that will fix both problems at once. PDO supports *prepared statements*.

---

■ **Note** Here's a good tutorial for further information about PDO and prepared statements:
http://net.tutsplus.com/tutorials/php/why-you-should-be-using-phps-pdo-for-database-access.

---

A *prepared statement* is an SQL statement *prepared* for dynamic content. Instead of dropping your PHP variables directly in the SQL statement, you can declare placeholders in the SQL and subsequently replace those placeholders with actual values. If you use PDO prepared statements, you will be safe from SQL injection attacks and from any trouble with special characters such as single and double quotes. Here's how to do it in `models/Blog_Entry_Table.class.php`:

```
//partial code for models/Blog_Entry_Table.class.php
//edit existing method
public function saveEntry ( $title, $entry ) {
    //notice placeholders in SQL string. ? is a placeholder
    //notice the order of attributes: first title, next entry_text
    $entrySQL = "INSERT INTO blog_entry ( title, entry_text )
                 VALUES ( ?, ?)";
    $entryStatement = $this->db->prepare( $entrySQL );
    //create an array with dynamic data
    //Order is important: $title must come first, $entry second
    $formData = array( $title, $entry );
    try{
        //pass $formData as argument to execute
        $entryStatement->execute( $formData );
    } catch (Exception $e){
        $msg = "<p>You tried to run this sql: $entrySQL<p>
                <p>Exception: $e</p>";
        trigger_error($msg);
    }
}
```

There are three equally important changes, as follows:

1. characters are used as placeholders in the SQL string.
2. An array is created with the dynamic data. The order of items must match the order used in the SQL string.
3. Pass the array with dynamic data as an argument to the `execute()` method.

You can test for yourself. Insert an entry with single and double quotes. It will be harder for you to verify that your editor is no longer vulnerable to SQL injection attacks. You would have to be able to perform an SQL attack to be able to verify that the editor is now safe. I guess you'll have to take my word for it.

# Summary

This was a relatively short chapter, and there were only a few new coding principles demonstrated. But you've taken the first steps toward a personal blogging system and developed a supercool entry editor.

In the process, you have had an opportunity to become more familiar with almost everything you have learned in the previous chapters. You have

- Created a MySQL database with a table
- Inserted data into a table through a web form
- Created an MVC web application structure
- Used the front controller design pattern
- Used the table data gateway design pattern

On top of all that, you've learned about SQL injection attacks and learned how to protect your PHP projects using PDO's prepared statements.

CHAPTER 8

# Showing Blog Entries

The entry editor is well on its way. You can use it to create new blog entries, which will be saved in the database. You are probably slowly developing some understanding of the model-view-controller (MVC) paradigm, but surely, you need some more practice to become comfortable using it.

In this chapter, you will learn how to display all blog entries on your `index.php` page. In the process, you will revisit much of what you have seen or learned so far. I will also introduce the following new topics:

- Making a new front controller
- Updating your table data gateway: adding a method to your `Blog_Entry_Table` class
- Iterating through a data set from a database table
- Creating a model, view, and controller for showing blog entries

## Creating a Public Blog Front Page

Later in this book, you will learn how to hide your administration module behind a login. You wouldn't want everybody to be able to write new blog entries on your blog, would you? When you get around to it, `admin.php` will be reserved for authorized users only. The public face of your blog will be `index.php`. Create a public face. Create a new file `index.php`, as follows:

```
<?php
//complete code for index.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );
include_once "models/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "PHP/MySQL blog demo example";
$pageData->addCSS("css/blog.css");

$dbInfo = "mysql:host=localhost;dbname=simple_blog";
$dbUser = "root";
$dbPassword = "";
$db = new PDO( $dbInfo, $dbUser, $dbPassword );
$db->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );

$pageData->content .= "<h1>All is good</h1>";
$page = include_once "views/page.php";
echo $page;
```

Save the `index.php` file. Remember that MySQL and Apache should run before you try to load anything through your localhost. You can start MySQL and Apache through the XAMPP control panel. Once MySQL and Apache are running, you can load `http://localhost/blog/index.php` in your browser and check for yourself that everything works so far. You should expect to see "All is good" in your browser. If your database credentials are invalid, you will see an exception displayed in your browser.

## Creating a Blog Controller

By default, you want blog entries to be displayed in an index page. You can start with the smallest possible step and create a super-simple blog controller. Create a new file `blog.php` in the `controllers` folder:

```
<?php
//complete code for controllers/blog.php
return "<h1>blog entries coming soon!</h1>";
```

With a preliminary blog controller ready, you can load it from `index.php` to test that the controller is loaded. Update `index.php`:

```
//partial code for index.php

//changes begin here
//comment out initial test message
//$pageData->content .= "<h1>All is good</h1>";
//include blog controller
$pageData->content .=include_once "controllers/blog.php";
//no changes below here

$page = include_once "views/page.php";
echo $page;
```

Save the blog controller and your `index.php`. Reload `http://localhost/blog/index.php` in your browser. You should see an output like this:

```
 blog entries coming soon!
```

When you see that output in your browser, you know that your `index.php` (your front controller) loads your blog controller.

## Getting Data for All Blog Entries

You want to be able to show a list of all blog entries found in the database. How about showing the `title`, the first 150 characters of the `entry_text`, and a *Read more* link for each of the blog entries?

Because you're determined to stick to clean code with an MVC approach, you already know that you'll need a model, a view, and a controller. You already have a bare-bones blog controller. You'll gradually improve the existing blog controller. You also have a `Blog_Entry_Table` class that provides a single point of access to the `blog_entry` database table. That will be your model, though it needs to be updated a bit to give you what you want. You don't have a view for listing all entries yet.

Begin by getting the right data from the `blog_entry` table. Declare a new method in `models/Blog_Entry_Table.class.php`, as follows:

```
//partial code for models/Blog_Entry_Table.class.php

//declare a new method inside the Blog_Entry_Table class
public function getAllEntries () {
    $sql = "SELECT entry_id, title,
            SUBSTRING(entry_text, 1, 150) AS intro
            FROM blog_entry";
    $statement = $this->db->prepare( $sql );
    try {
        $statement->execute();
    } catch ( Exception $e ) {
        $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                <p>Exception: $e</p>";
        trigger_error($exceptionMessage);
    }
    return $statement;
}
```

---

■ **Caution** Be careful where you write this code. You can't just place it anywhere in the `Blog_Entry_Table.class.php` file. You should write it between the two curly braces that delimit the class definition—inside the class definition.

---

Perhaps you're wondering what it means to write the new code *inside the class definition*? In the following code example, you can see two methods declared inside one class definition:

```
//class definition starts here
class Something {

    //all methods must be declared inside the class definition
    public function someMethod () {
        //
    }

    public function someOtherMethod () {
        //
    }

} //class definition ends here
```

The `getAllEntries()` method will return a `PDOStatement` object, through which you can get access to all blog entries, one at a time. The SQL statement involves some SQL you haven't seen before, so it might be a bit tricky to understand initially. Let's go through it step by step.

## Using an SQL SUBSTRING Clause

The first thing to notice is that the SELECT statement selects three columns—entry_id, title, and entry_text—from the blog_entry table. But you're not selecting everything from the entry_text column. You're only selecting the first 150 characters. You do that with a SUBSTRING() function. The general syntax for the MySQL SUBSTRING() function is as follows:

```
SUBSTRING( string, start position, length )
```

The SUBSTRING() returns a part of a string: a substring. The string argument indicates which string to return a part of. The start position argument indicates at which position to start the substring. The length argument specifies how long a substring to return. In the SELECT statement, you're really selecting the first 150 characters from the string found in the entry_text field.

## Using an SQL Alias

With SQL, you can use an alias to rename a table or a column. In the SQL used to select all blog entries, you have renamed the substring to intro. It's not strictly necessary to rename the column. Without the AS clause, you would still have a SELECT statement that returns data from three columns. The returned data set might be something like the following table:

| entry_id | title | SUBSTRING(entry_text, 1, 150) |
|---|---|---|
| 1 | My first entry | Blah… |
| 2 | My second entry | Blah, blah… |

I think you will agree that the third column has a strange name. Strange names in code are bad, because the code becomes harder to read and understand. By renaming the returned column using the AS keyword, you get a table that is easier to read and understand, as follows:

| entry_id | title | intro |
|---|---|---|
| 1 | My first entry | Blah… |
| 2 | My second entry | Blah, blah… |

## Testing Your Model

It is important to look for small steps when developing something with code. If you write only a little code between tests, it will be easier to spot coding errors early. Do a little test to see if the getAllEntries() method returns the right data. The Blog_Entry_Table object will be used from the blog controller, because the blog controller will load the blog model and the blog view, in the process of generating an output for the browser. So, the blog controller is the logical place to write a test. Here's some code to test getAllEntries() from controllers/blog.php:

```
<?php
//complete code for controllers/blog.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );
```

```
//$entries is the PDOStatement returned from getAllEntries
$entries = $entryTable->getAllEntries();
//fetch data from the first row as a StdClass object
$oneEntry = $entries->fetchObject();
//print the object
$test = print_r( $oneEntry, true );

//return the printed object to index to see it in browser
return "<pre>$test</pre>";
```

## Using print_r() for Inspecting an Object

The preceding code uses a PHP function `print_r()` to inspect an object. You have already used `print_r()` to inspect two superglobal arrays, `$_POST` and `$_FILES`. You can use the same function to inspect objects. If you save `controllers/blog.php` and load `http://localhost/blog/index.php` in your browser, you should get an output a bit like the following, except you probably have something else saved in your `title` and `intro`:

```
stdClass Object (
     [entry_id] => 1
     [title] => Testing title
     [intro] => test
)
```

Looking closely at the output, you can see that the printed `$oneEntry` is an `StdClass` object with three properties: `entry_id`, `title`, and `intro`. The bigger question is whether that's what you want. Was the test successful or not?

The test was successful! The `$oneEntry` variable should hold an `StdClass` object with exactly those three properties. You're looking at a PHP object representing one row of data; you're looking at your blog entry model. The `$oneEntry` variable should represent one row of data received from MySQL through PDO. The three properties are created by your `SELECT` statement from the `getAllEntries()` method:

```
//the SQL statement used in Blog_Entry_Table->getAllEntries()
SELECT entry_id, title,
SUBSTRING(entry_text, 1, 150) AS intro
FROM blog_entry
```

All SQL `SELECT` statements will return a new, temporary table with the columns specified in the `SELECT` clause. The returned table will be populated with data found in the table specified in the `FROM` clause.

The `getAllEntries()` method will query your database and return a `PDOStatement` object. Every time you call the `PDOStatement`'s `fetchObject()` method, the `PDOStatement` will return an `StdClass` object representing one row of data.

Even absolute beginners can easily understand that developing in small steps is sensible. It's easy to see that an error is easier to spot in fewer lines of code. It's also easy to understand that if a piece of code has an undetected error, subsequent code might end up with more errors, because any subsequent code has to collaborate with the code with the original error.

The result is code with multiple errors in multiple places. The result is code errors that hide underlying code errors. The result is difficult debugging! This is how a coding error becomes a bug: when you write your code around a tiny, undetected error.

The only countermeasure is to write code in small steps and test your code between every step. The problem is that it can be difficult for beginners to test whether their code works. Beginners don't always realize exactly what their code is doing or what it should be doing. Beginners often experiment with their code without really knowing what to expect—that's what it's like to be a beginner! Your ability to predict PHP behavior and formulate small tests will only improve through experience and reflection.

■ **Note** There is an approach to development called *test-driven* development. It's based on writing formal tests for every unit of code and test units individually. It's hardly a topic for beginners, but it could be useful for you to know that test-driven development exists.

I'll continue to show you many examples of how you can create small, informal tests and how you can write as little code as possible between steps.

## Preparing a View for All Blog Entries

With a model prepared, loaded, and tested from your controller, it's time to create a view. This view is supposed to list all blog entries. The number of blog entries is likely to change, so it would be a bad idea to create a view for a specific number of entries.

The view should automatically change to accommodate however many—or few—blog entries are found in the database. You need to iterate over blog entries with a loop. Create a new file, `views/list-entries-html.php`, and have it loop through entries, as follows:

```
<?php
//complete code for views/list-entries-html.php
$entriesFound = isset( $entries );
if ( $entriesFound === false ) {
    trigger_error( 'views/list-entries-html.php needs $entries' );
}

//create a <ul> element
$entriesHTML = "<ul id='blog-entries'>";

//loop through all $entries from the database
//remember each one row temporarily as $entry
//$entry will be a StdClass object with entry_id, title and intro
while ( $entry = $entries->fetchObject() ) {
    $href  = "index.php?page=blog&amp;id=$entry->entry_id";
    //create an <li> for each of the entries
    $entriesHTML .= "<li>
        <h2>$entry->title</h2>
        <div>$entry->intro
            <p><a href='$href'>Read more</a></p>
        </div>
    </li>";
}
//end the <ul>
$entriesHTML .= "</ul>";
return $entriesHTML;
```

Just take a minute to look at that wonderful `while` loop before actually using it in your application. It will dynamically create `<li>` elements for however many rows are found in the `blog_entry` database table. If there is one blog entry in your database, there will be one `<li>`. If there are ten blog entries in your database, there will be ten `<li>` elements.

Notice the comment about which properties an `$entry` should have. It is easy to forget which properties should be available, so it can be helpful to write a comment about it.

## Hooking Up View and Model

The final step is to hook up your view with the data from your model. It doesn't take much code, but it will produce a radically different output in your browser. Update your code in controllers/blog.php:

```
<?php
//complete code for controllers/blog.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );
$entries = $entryTable->getAllEntries();


//code changes start here
//test completed - delete of comment out test code
//$oneEntry = $entries->fetchObject();
//$test = print_r($entryTable, true );

//load the view
$blogOutput = include_once "views/list-entries-html.php";
return $blogOutput;
```

That's it! Load `http://localhost/blog/index.php` in your browser, and you should see a `<ul>` with separate `<li>` elements for each blog entry. You could visit `http://localhost/blog/admin.php?page=editor`, to create a new entry, and then reload `http://localhost/blog/index.php` in your browser, to see the newly created blog entry listed. Your blogging system is really starting to look like a proper blog.

It's very tempting to click *Read more*, isn't it? Don't you just want to click it and read a blog entry? Well, clicking *Read more* at this point will not have a great impact. You haven't written the code to display all content of individual blog entries, so nothing will change when you click.

## Responding to User Requests

You want to be able to show all content for one blog entry when a user clicks *Read more*. You will have to select individual blog entries by their `entry_id`, by the primary key. It is already available in the code; you just need to find it.

Which part of your code should respond when a user clicks *Read more*: model, view, or controller? The controller! A controller is meant for handling user interactions. Clicking a link is a user interaction. So, you should work on your blog controller to deal with users clicking *Read more*. Here's a small code change in `controllers/blog.php` that will output the primary key of the clicked blog entry:

```
<?php
//complete code for controllers/blog.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );

//new code starts here
$isEntryClicked = isset( $_GET['id'] );
if ($isEntryClicked ) {
    //show one entry ... soon
    $entryId = $_GET['id'];
    $blogOutput = "will soon show entry with entry_id = $entryId";
```

```
} else {
    //list all entries
    $entries = $entryTable->getAllEntries();
    $blogOutput = include_once "views/list-entries-html.php";
}
//end of changes

return $blogOutput;
```

Load `http://localhost/blog/index.php` in your browser to see what changed. By default, you'll still see a list of all entries, but if you click *Read more* on the first entry, you'll see a different output, as follows:

```
will soon show entry with entry_id = 1
```

The `entry_id` is the primary key of blog entries. Now that your code knows the primary key of the clicked blog entry, it will be an almost trivial task to display it. Before tackling that, I think you should take a little time to reflect: why can you find the `entry_id` in `$_GET['id']` when *Read more* is clicked?

The *Read more* link is created with a somewhat special `href`. If you click such a link in your browser, you can see the requested URL in the browser's address bar. It'll be something like the following:

```
http://localhost/blog/index.php?page=blog&id=1
```

Notice that there are two URL variables encoded: one called `page` and another called `id`. The two URL variables are separated with an & (ampersand) character. One URL can hold multiple URL variables, as long as each URL variable is separated with the & character.

You can probably work out that the `id` URL variable holds the `entry_id` attribute of the clicked entry. That's how you can find the primary key of a clicked entry; it's encoded in the URL. But how did it get encoded there? The answer lies in `views/list-entries-html.php`. Take an extra look at the `href` for the `<a>` element:

```
//partial code for views/list-entries-html.php
//no code changes - please just read the code again
while ( $entry = $entries->fetchObject() ) {
    $href  = "index.php?page=blog&amp;id=$entry->entry_id";
    $entriesHTML .= "<li>
        <h2>$entry->title</h2>
        <div>$entry->intro
            <p><a href='$href'>read more</a></p>
        </div>
    </li>";
}
```

And there it is, plain to see. The URL variable `id` gets its value from the corresponding entry's `entry_id` attribute. It might be a little strange to look at the `&`.

It's no big deal. An & character is called an ampersand, and `&` is an HTML character entity representing an ampersand character. An HTML character entity is a short code representing a special character.

---

■ **Note** You can read more about HTML entities at www.w3schools.com/html/html_entities.asp.

---

## Getting Entry Data

It is time for you to tackle the trivial problem of displaying the entry. To do that, you get data from the `blog_entry` table. You already have a class that provides access to the `blog_entry` table. You can continue to use that, so you have a single point of access to the table. You can declare a new method in `models/Blog_Entry_Table.class.php`.

You already have the `entry_id` available in the blog controller. So, you can declare a method that takes an `entry_id` as argument and returns an `StdClass` object with all the content for the corresponding blog entry, as in the following:

```
//partial code for models/Blog_Entry_Table.class.php
//declare a new method inside the class code block
//do not change any existing methods
public function getEntry( $id ) {
    $sql = "SELECT entry_id, title, entry_text, date_created
            FROM blog_entry
            WHERE entry_id = ?";
    $statement = $this->db->prepare( $sql );
    $data = array( $id );
    try{
        $statement->execute( $data );
    } catch (Exception $e) {
        $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                <p>Exception: $e</p>";
        trigger_error($exceptionMessage );
    }
    $model = $statement->fetchObject();
    return $model;
}
```

You can see that this new method is quite similar to the other methods inside the `Blog_Entry_Table` class. First, you declare an SQL string. Next, you use the `prepare()` method to convert the SQL string to a `PDOStatement` object and `try()` to `execute()` the statement. Finally, you fetch the first row of data from the returned MySQL table and return it as an `StdClass` object.

Note the use of a prepared statement. Remember to always use a prepared statement when you are creating SQL statements, using input received from the browser. The `$id` came from a URL variable and as such, it should be treated as unsafe. A malicious hacker might try to exploit it and attempt an SQL injection attack. A prepared statement stops such attempts.

You're using a prepared statement with unnamed placeholders (represented by ? characters). To replace the placeholder with an actual value, you create an array of values to use and pass the array to the `execute()` method.

You've already seen this way of using a prepared statement—you used it for building the entry editor. Feel free to consult Chapter 7 to refresh your memory and deepen your understanding of the concept.

## Creating a Blog View

To show an entry, you need a view for it, a view that supplies an HTML structure and merges it with data for the entry. Create a new file, `views/entry-html.php`, as follows:

```
<?php
//complete source code for views/entry-html.php
//check if required data is available
$entryDataFound = isset( $entryData );
```

```
if ( $entryDataFound === false ) {
    trigger_error('views/entry-html.php needs an $entryData object');
}
//properties available in $entry: entry_id, title, entry_text, date_created
return "<article>
    <h1>$entryData->title</h1>
    <div class='date'>$entryData->date_created</div>
    $entryData->entry_text
</article>";
```

The preceding code is a repetition of the approach you have seen a few times since developing the poll. The essence is quite basic: merge some data stored in an `StdClass` object with a predefined HTML structure. The view requires an `StdClass` object saved in a variable called `$entryData`. So, the first few lines of code check the availability of `$entryData`. If it is not found, the code will trigger a custom error, so it will be easy to find and correct the mistake.

## Displaying an Entry

You've got the model; you've got the view. The final step is to update your blog controller. It is responsible for fetching entry data from the model, sharing it with the entry view, and returning the resulting HTML string to `index.php`, where it will be displayed. Your controller is only two lines of code away from completion:

```
<?php
//complete code for controllers/blog.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );
$isEntryClicked = isset( $_GET['id'] );
if ($isEntryClicked ) {
    $entryId = $_GET['id'];
    //new code begins here
    $entryData = $entryTable->getEntry( $entryId );
    $blogOutput = include_once "views/entry-html.php";
    //end of code changes
} else {
    $entries = $entryTable->getAllEntries();
    $blogOutput = include_once "views/list-entries-html.php";
}
return $blogOutput;
```

Test your progress by loading `http://localhost/blog/index.php?page=blog` in your browser. Click *Read more*, and you should see the full content of the clicked blog entry displayed in your browser.

You have a functional blog, complete with an `index.php` for normal users and `admin.php` through which you can create new blog entries. Now would be a good time to celebrate your progress. You have come a long way since Chapter 1. You know something about object-oriented programming with PHP. You even have some experience with a few design patterns. You know how to work with databases. You are no longer an absolute beginner.

## Code Smell: Duplicate Code

Can you smell it? There is a bad smell coming from your code. It's one of those classic code smells every coder knows about. And it's one of those things you should learn to avoid, as your proficiency with code grows.

■ **Note** Find a long list of typical code smells at http://en.wikipedia.org/wiki/Code_smell.

Duplicate code is *when identical or similar code exists in multiple places* in your code. Do you already know where to find the smell? It's in `models/Blog_Entry_Table.class.php`. Here's an example:

```
//partial code for models/Blog_Entry_Table.class.php
public function getEntry( $id ){
    $sql = "SELECT entry_id, title, entry_text, date_created
            FROM blog_entry
            WHERE entry_id = ?";
    $statement = $this->db->prepare( $sql );
    $data = array( $id );
    try {
        $statement->execute( $data );
    } catch (Exception $e){
        $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                <p>Exception: $e</p>";
        trigger_error($exceptionMessage);
    }
    $model = $statement->fetchObject();
    return $model;
}
```

There are a couple of other methods in the `Blog_Entry_Table` class. They all `prepare()` a `PDOStatement` and `try()` to `execute()` it. So, in all three methods, you can find five to seven lines of code that are nearly identical. That's bad!

## Staying DRY with Curly

Duplicate code is bad for a number of reasons. One is that you are simply using more lines than necessary. Your code is unnecessarily long. Longer code is worse, because more code invariably means more errors. Less code means fewer errors. Another reason duplicate code is bad is that chances are you'll be changing the code you write. Someday, you'll want to make some changes. If you have identical or very similar code spread out in ten different methods, you will have to make identical or very similar code changes in ten different methods. If you keep the code in a separate method, you will only have to change the code in one method, and it will affect the other ten methods automatically.

It's really just another case of coding by Curly's law, or at least a variant of Curly's law. Curly's original law was: *Do one thing*. This particular variant should perhaps be: *Do one thing once*. There's another geek expression for it: staying DRY. *DRY* is an acronym that means "don't repeat yourself."

## Refactoring with Curly

*Refactoring* is the process of changing your code without changing what it does. It's a big deal for coders. You should refactor your code when you realize the project requirements have outgrown the code architecture, or, in other words, when the code architecture doesn't support the features your project needs in a beautiful way.

It is time to refactor the `Blog_Entry_Table` class, so it becomes more DRY. Let's begin by encapsulating the code that prepares an SQL statement into a separate method. Declare a new method in `models/Blog_Entry_Table.class.php`, as follows:

```
//Partial code for models/Blog_Entry_Table.class.php

//declare a new method in the Blog_Entry_Table class
//$sql argument must be an SQL string
//$data must be an array of dynamic data to use in the SQL
public function makeStatement ( $sql, $data) {
    //create a PDOStatement object
    $statement = $this->db->prepare( $sql );
    try{
        //use the dynamic data and run the query
        $statement->execute( $data );
    } catch (Exception $e) {
        $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                <p>Exception: $e</p>";
        trigger_error($exceptionMessage);
    }
    //return the PDOStatement object
    return $statement;
}
```

With the new method declared, you can try to refactor one of your existing methods to use the new method. You could begin with the most recently declared method, the `getEntry()` method.

```
//Partial code for models/Blog_Entry_Table.class.php

//edit existing method getEntry
public function getEntry( $id ){
    $sql = "SELECT entry_id, title, entry_text, date_created
            FROM blog_entry
            WHERE entry_id = ?";
    $data = array($id);
    //call the new DRY method
    $statement = $this->makeStatement($sql, $data);
    $model = $statement->fetchObject();
    return $model;
}
```

Let's just test that the new method works as intended. Navigate your browser to `http://localhost/blog/index.php` and click *Read more* to see one blog entry. If you can see one entry in your browser, you know the refactored `getEntry()` method works.

See how using the new `makeStatement()` method makes your `getEntry()` method a little shorter? You can work through the other methods and replace the hideous duplicate code with a beautiful DRY solution.

There is a little syntactical detail to notice. See how you need the `$this` keyword when one method calls another method declared in the same class? It's not really different from using `$this` to get to a property. In both cases, `$this` is an object's reference to itself. It's the object-oriented way of saying "my."

Let's continue refactoring and move on to the `saveEntry()` method:

```
//Partial code for models/Blog_Entry_Table.class.php

//edit existing method saveEntry
public function saveEntry ( $title, $entry ) {
    $entrySQL = "INSERT INTO blog_entry ( title, entry_text )
                 VALUES ( ?, ?)";
    $formData = array($title, $entry);
    //changes start here
    //$this is the object's way of saying 'my'
    //so $this->makeStatement calls makeStatement of Blog_Entry_Table
    $entryStatement = $this->makeStatement( $entrySQL, $formData );
    //end of changes
}
```

You can probably trust this to work perfectly, but just to be absolutely sure, you should test. Load `http://localhost/blog/admin.php?page=editor` in your browser and test that you can still create new blog entries with your entry editor.

Refactoring is truly joyful work, isn't it? You spot an ugly corner in your code, and you make it beautiful. You really ought to refactor `getAllEntries()` also. Rewrite the existing function `getAllEntries()` in `models/Blog_Entry_Table.class.php`:

```
//Partial code for models/Blog_Entry_Table.class.php

//edit existing method getAllEntries
public function getAllEntries () {
    $sql = "SELECT entry_id, title, SUBSTRING(entry_text, 1, 150) AS intro  FROM blog_entry";
    $statement = $this->makeStatement($sql);
    return $statement;
}
```

Hang on, there's a problem here. Do you see it? You're calling `makeStatement()` with a single argument, but it needs two arguments. Up until now, you have called it with two arguments. The second argument has been an array of data to use in an SQL string. But in this case, you have no data you want to use in the SQL string. You have no second argument to pass on.

Sometimes, you want to call `makeStatement()` with one argument, and sometimes you want to call it with two arguments. You want the second argument to be optional. Luckily, PHP has a very easy way to make an argument optional. You can simply declare the argument with a default value, which will be used if nothing is passed. Here's how to do it:

```
//Partial code for models/Blog_Entry_Table.class.php

//edit existing method makeStatement
//change code: declare a default value of NULL for the $data argument
public function makeStatement ( $sql, $data = NULL ){
    //end of code changes
    $statement = $this->db->prepare( $sql );
    try{
        $statement->execute( $data );
```

```
    } catch (Exception $e){
        $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                <p>Exception: $e</p>";
        trigger_error($exceptionMessage );
    }
    return $statement;
}
```

In the preceding code, the argument `$data` gets a default value of `NULL`. So, if `makeStatement()` is called without a second argument, the created `PDOStatement` object will execute with `NULL`. That means no dynamic values will replace placeholders in the prepared statement. And that is exactly what you want, because there are no placeholders in the SQL for this statement.

---

■ **Note** You can consult www.w3schools.com/php/php_functions.asp to learn a little more about function arguments with default values.

---

In the other cases where `makeStatement()` is called with two arguments, the second argument will be used to replace SQL placeholders with actual values. Using optional arguments is a very powerful concept in code. It can often lead to a clean solution when you are encapsulating nearly duplicate code into a single separate method, just as you've just done.

You can trust your code to work when you have tested it. Navigate your browser to `http://localhost/blog/index.php` to confirm that all entries are listed as before. Remember: to refactor is to change code without changing what it does. So, a successful test is confirmed when the code behaves exactly as it did before refactoring. Refactoring is done with the sole purpose of making the code easier to work with for coders.

## Using the Private Access Modifier

The `makeStatement()` method is a fragile member of the `Blog_Entry_Table` class. It is only meant to be called internally, and only by other `Blog_Entry_Table` methods. It is certainly not meant to be called from outside the class. The `makeStatement()` can be understood as a sub-method used by the other `Blog_Entry_Table` methods.

Right now, it is possible to call it from the outside. In fact, it can be called from anywhere in your code. That means the `makeStatement()` method can easily be used in a wrong way, if you or a fellow developer forgets that it should only be called internally. Because of that, `makeStatement()` is almost an error just waiting to happen.

It is so easy to remedy. You can simply use the `private` access modifier, and `makeStatement()` can only be called internally. No other piece of code can ever get access to calling it. Here's how to use the `private` access modifier:

```
//Partial code for models/Blog_Entry_Table.class.php

//edit existing method makeStatement
//code change: make it private
private function makeStatement ( $sql, $data = NULL ){
    //end of code changes
    $statement = $this->db->prepare( $sql );
    try {
        $statement->execute( $data );
    } catch (Exception $e) {
```

```
            $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                    <p>Exception: $e</p>";
            trigger_error($exceptionMessage);
        }
        return $statement;
    }
```

With this tiny change, your `Blog_Entry_Table` class has improved considerably. The improvement is that it is now impossible to call `makeStatement()` from the outside. Only a `Blog_Entry_Table` object can call the method. That means it's just become a lot harder for you or a fellow coder to use `Blog_Entry_Table` in a wrong way.

You may have noticed the `private` keyword before? I have used it for the `$db` property, without explaining it. A `private` object property can't be accessed from the outside. As a rule of thumb, you should declare object properties `private` by default. Use a different access modifier only if you specifically need to. That way, only the object itself can manipulate its properties.

Remember: The Single Responsibility Principle, also known as Curly's law, applied to classes. A class should have a single purpose, and all its properties and behaviors should be strictly aligned with that purpose. A class with one responsibility is simpler than a class with many, and a simple class is easier to use than a complex class. By hiding some properties and methods using a `private` access modifier, you present a public interface that's even simpler and even easier to use. So, as a rule of thumb, use `private` by default, `public` when you need it.

---

■ **Note** There is a third access modifier: `protected`. In PHP, it is like `private`, except it can be shared with subclasses through *inheritance*. You can find a nice tutorial covering inheritance, access modifiers, and other central OOP topics at www.killerphp.com/tutorials/object-oriented-php/.

---

You have come across a few examples where `public` object properties have been used—every time you created a view that depends on object properties. You have been using `StdClass` objects with `public` properties. You have used such objects to represent a row of data from a database table.

# Summary

In this chapter, you have created the public "face" of your blog. In the process, you have seen a little more SQL, and you have had a few extra opportunities to understand MVC.

You have seen how to reuse your own code. The `Blog_Entry_Table` is now used by `admin.php` and by `index.php`. The `Blog_Entry_Table` is a table data gateway that provides a single point of access from your PHP code to your `blog_entry` database table.

You have also seen how refactoring can eliminate code smells and how the private access modifier and optional arguments can be used to keep your code DRY.

CHAPTER 9



# Deleting and Updating Entries

This is how it will be: one chapter will improve the public face of your blog in some way, and the next chapter will focus on improving the secret blog administration module, and they will keep alternating like that. This chapter focuses on improving the secret blog administration module.

Let's continue down the model-view-controller (MVC) path with objects and classes. This second iteration of the administration module will show you how to update and delete existing entries, through the entry editor. In the process of improving the entry manager, you will learn about writing small, informal code tests. The tests are meant to emphasize the experimental process of developing a blog, rather than just showing you the finished code for a blog. If you integrate testing into your development process, you will improve your overall code quality and decrease debugging time.

In this chapter, you will also learn how you can create HTML forms that can communicate changes to users. In addition, you will get to revisit the idea of progressive enhancement with JavaScript. Working through this chapter may change your perspective a little: you will no longer just focus on how your code works. You will begin to focus on how to use code to design a system that works for your users.

## Creating a Model for Administrative Links

Take a look at your existing editor at `http://localhost/blog/admin.php?page=editor`. You can clearly see that there are already buttons that can be used for saving or deleting existing entries. You can also see that something is missing. Where would you click to load an existing entry into the entry editor, so that you could edit or delete it?

I am sure you can think of many clever ways to go about this task. I propose an approach that will be quite similar to what you just did with the blog: I propose that you display a list of all entries to administrators. Clicking one entry should load it into the entry editor. My main motivation for taking this approach is its similarity to what you've already done for blog entries. Doing it again will present a nice little learning loop for you. In my experience as a teacher, repetition is essential—especially for beginners.

Another advantage is that you already have a `getAllEntries()` method in the `Blog_Entry_Table` class. You can reuse the existing method. So, you have already dealt with the model. Just to be certain that you can get access to entry data from the controller, you can write a little code to test the assumption. Open `controllers/admin/entries.php` in your code editor and rewrite it completely, as follows:

```
<?php
//complete code for controller/admin/entries.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );
//get a PDOStatement object to get access to all entries
$allEntries = $entryTable->getAllEntries();
//test that you can get the first row as a StdClass object
$oneEntry = $allEntries->fetchObject();
```

```
//prepare test output
$testOutput = print_r( $oneEntry, true );
//return test output to front controller, to admin.php
return "<pre>$testOutput</pre>";
```

If everything works as intended, you should see a `StdClass` representation of one entry printed when you load `http://localhost/blog/admin.php?page=entries` in your browser. The output should be something like the following:

```
stdClass Object (
     [entry_id] => 1
     [title] => Testing title
     [intro] => bla bla
)
```

This little test confirms that the entries controller does get access to entry data. You can see the content for the first entry printed in the browser.

You can learn a few things from the preceding output. You can see that your `$testOutput` variable holds an object of the type `StdClass`. You can see that it has three properties, called `entry_id`, `title`, and `intro`. You can even see the values of those three properties.

You must learn to understand PHP behavior. It is particularly important that you understand the PHP code you use when you build your own web application. Here's a good opportunity. You can take a look at the PHP code that produced this output. See if you can work your way through the code and understand every little process involved in creating the preceding output. Here are three questions that may guide you:

- How does the output get from `controllers/admin/entries.php` to `admin.php`, where your browser sees it?
- Why are there three properties? Why not one or four or some other number?
- What does the `fetchObject()` method do?

Take your time and work through the questions to arrive at your own answers. Once you truly understand the answers to those questions, you will be much better equipped to start developing your own PHP/MySQL projects.

## Displaying Administrative Links

With the model tried, tested, and understood, you can start working on a view. What you're after is a list of clickable blog entry titles. That means you will have to iterate through all entries found in the database.

You can take an approach very similar to that you've taken with the blog. You can use a `while` loop to iterate through database records by way of a `PDOStatement` object. Every row of data from the database table will be represented by a separate `<li>` element.

You can wrap the individual blog titles in `<a>` elements to create a clickable list of entries. Create a new file in `views/admin/entries-html.php`, as follows:

```
<?php
//complete code for views/admin/entries-html.php
if( isset( $allEntries ) === false ) {
    trigger_error('views/admin/entries-html.php needs $allEntries');
}
```

```
$entriesAsHTML = "<ul>";
while ( $entry = $allEntries->fetchObject() ) {
    //notice two URL variables are encoded in the href
    $href = "admin.php?page=editor&amp;id=$entry->entry_id";
    $entriesAsHTML .= "<li><a href='$href'>$entry->title</a></li>";
}

$entriesAsHTML .= "</ul>";
return $entriesAsHTML;
```

Take a closer look at the `href` values generated. Clicking an entry will request a URL like `admin.php?page=editor&id=2`. This way, the editor controller will have access to the `entry_id` of the clicked entry.

You can see that this code is very similar to that in `views/list-entries-html.php`. Actually, it is perhaps too similar. If you were to refactor the blog code, this would be a candidate for refactoring. One might argue that you're not breaking Curly's law, but you are certainly entering a gray zone. You could refactor the code and use the same view file for listing all entries in slightly different ways. On the other hand, such a solution would lead to greater complexity in the code. I prefer to keep my views as simple as possible. So, I propose you keep this code as it is, because it works, and it's not overly complicated.

I would like you to see that there are no right or wrong solutions to organizing your code. You can write a solution to a task in many different ways. How you decide to organize your code depends on how you think about your code. In the preceding case, I had to choose between short, abstract code or duplicate, simple code. I prefer shorter code over longer code, but I also prefer simple code over abstract code. In the example, I decided to favor simple code over short code.

To have the new view displayed, you must update the entries controller, so that it loads the view. Right now, your code in `controllers/admin/entries.php` outputs a `StdClass` object. You did this to test whether the relevant model code works as intended. You can delete the test code in your entries controller entirely, so that it returns your newly created view. Update the code in `controllers/admin/entries.php`, as follows:

```
<?php
//complete code for controller/admin/entries.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );
$allEntries = $entryTable->getAllEntries();

$entriesAsHTML = include_once "views/admin/entries-html.php";
return $entriesAsHTML;
```

You can test your code now. Load `http://localhost/blog/admin.php?page=entries` in your browser, and you should see a well-formed list of clickable blog entry titles. If you click a title, the empty entry editor will be displayed. You can change this, so that the entry editor will be loaded with the contents of the clicked blog entry displayed inside the editor.

## Populating Your Form with the Entry to Be Edited

Sometimes, the entry editor form should be displayed with blank fields, so that you can create new entries. At other times, the editor should display an existing entry, so that it can be edited. A user should click a blog title to load it into the editor.

Clicking such a blog title will encode the entry's `entry_id` into the HTTP request as a URL variable. It follows that if an entry's `entry_id` is available as a URL variable, you should load the corresponding entry into the editor. If no such URL variable is found, you should display a blank editor.

You can achieve that by adding a few placeholders in the editor view. If the view finds data for an entry, it should display it; otherwise, it should display empty editor fields. Here's the updated `views/admin/editor-html.php`:

```
<?php
//complete code for views/admin/editor-html.php

//new code added here
//check if required data is available
$entryDataFound = isset( $entryData );
if( $entryDataFound === false ){
    //default values for an empty editor
    $entryData = new StdClass();
    $entryData->entry_id = 0;
    $entryData->title = "";
    $entryData->entry_text = "";
}

//changes in existing code below
//notice object properties used in <input> and <textarea>
return "
<form method='post' action='admin.php?page=editor' id='editor'>
    <input type='hidden' name='id' value='$entryData->entry_id' />
    <fieldset>
        <legend>New Entry Submission</legend>
        <label>Title</label>
        <input type='text' name='title' maxlength='150'
               value='$entryData->title' />

        <label>Entry</label>
        <textarea name='entry'>$entryData->entry_text</textarea>
        <fieldset id='editor-buttons'>
            <input type='submit' name='action' value='save' />
            <input type='submit' name='action' value='delete' />
        </fieldset>
    </fieldset>
</form>";
```

The main principle to notice here is the use of object properties as content placeholders. For example: Whatever data PHP finds inside `$entryData->entry_text` will be displayed inside the `<teaxtarea>` element. If `$entryData->entry_text` is an empty string, the `<textarea>` will be empty. On the other hand, if `$entryData->entry_text` holds data from the database, the `<textarea>` will display content from the database.

You can see a new `<input>` type used in the editor view. There is a *hidden* input. A hidden input will not be visible to users. You can use it to store values you require to process submitted form input correctly. This hidden input will store the currently displayed entry's `entry_id`, or `0`, if the editor fields are blank.

The last step to populating the otherwise empty editor with an existing entry is to get the content for the clicked entry. You already have a `getEntry()` method declared in your `Blog_Entry_Table` class. You can use that from your editor controller. Update the code in `controllers/admin/editor.php`, as follows:

```
//partial code for controllers/admin/editor.php
//add this code near the end of the script
//in my example this is line 21
//introduce a new variable: get entry id from URL
```

```
$entryRequested = isset( $_GET['id'] );
//create a new if-statement
if ( $entryRequested ) {
    $id = $_GET['id'];
    //load model of existing entry
    $entryData = $entryTable->getEntry( $id );
    $entryData->entry_id = $id;
}
//no new code below

$editorOutput = include_once "views/admin/editor-html.php";
return $editorOutput;
```

You can get the entry data from your database, because you have its `entry_id` available as a URL variable. Because you have an `entry_id`, you can get all content for that particular blog entry. See it for yourself. Save your work and point your browser to `http://localhost/blog/admin.php?page=entries`. Click a title to see your editor populated with the `title` and `entry_text` of the clicked entry.

The editor isn't quite perfect yet. You can see any existing entry in the editor, but you cannot save any changes. If you click the Save button, you can see that a new entry will be inserted, even if you tried to edit an existing entry. Your editor cannot edit existing entries yet, nor can it delete existing entries. Deleting is very easy, so let's implement that first.

## Handling Entry Deletion

Obviously, it should be possible to delete entries. It is a wonderfully uncomplicated operation and a good place to start. You will need some changes to the editor's model and controller.

The model should have some code to actually delete entry data from the database. The view already has a Delete button, so no changes are necessary here. The controller must be updated so it reacts when a user clicks Delete.

## Deleting Entries from the Database

To delete a row of data from the `blog_entry` table in the database, you will need SQL. You will also need `PDO` to prepare a `PDOStatement` with your SQL. Executing the `PDOStatement` should delete the identified entry. Take a minute to reflect. Where would you write the code to delete an entry?

To delete an entry, you want PHP to manipulate the `blog_entry` database table. You already have a table data gateway object, your `Blog_Entry_Table` object. The purpose of a table data gateway is to provide a single point of access to a given table. Whenever you want access to that table, you should use the relevant table data gateway object. So, the PHP code to delete an entry in the `blog_entry` table belongs in your `Blog_Entry_Table`. You can implement it by declaring a new method in `models/Blog_Entry_Table.class.php`, as follows:

```
//partial code for models/Blog_Entry_Table.class.php

//declare a new method inside the Blog_Entry_Table class
public function deleteEntry ( $id ) {
    $sql = "DELETE FROM blog_entry WHERE entry_id = ?";
    $data = array( $id );
    $statement = $this->makeStatement( $sql, $data );
}
```

Deleting data from a database is a final action; there is no undo! It is important that you never accidentally delete something you shouldn't have. Luckily, your database table is properly designed: every record has a primary key. That means every single record can be uniquely identified, if you have the primary key of an entry. It follows that you can safely delete a `blog_entry` by its `entry_id`, because you can trust the right entry to be deleted. There will be no accidental loss of data here!

## Responding to Delete Requests

With the editor model ready to delete entries, it is time to update the controller with code, to determine if the Delete button was clicked. If the Delete button was clicked, the controller should call the model, to have the relevant entry deleted. How can you know if the Delete button was clicked? Take a look at the HTML for the editor form:

```
//partial code for views/admin/editor-html.php
//two buttons, one name, different values
<fieldset id='editor-buttons'>
    <input type='submit' name='action' value='save' />
    <input type='submit' name='action' value='delete' />
</fieldset>
```

Clicking any of the submit buttons in the editor form will encode a URL variable named `action`. The `action` will have a value of `save`, if you clicked the Save button, and a value of `delete`, if you clicked the Delete button. The value of the URL variable `action` can tell you which button a user clicked. Update the code in `controllers/admin/editor.php`, as follows:

```
<?php
//complete code for controllers/admin/editor.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );

$editorSubmitted = isset( $_POST['action'] );
if ( $editorSubmitted ) {
    $buttonClicked = $_POST['action'];
    $insertNewEntry = ( $buttonClicked === 'save' );
    // new code: was "delete" button clicked
    $deleteEntry = ( $buttonClicked === 'delete' );
    //new code: get the entry id from the hidden input in editor form
    $id = $_POST['id'];

    if ( $insertNewEntry ) {
        $title = $_POST['title'];
        $entry = $_POST['entry'];
        $entryTable->saveEntry( $title, $entry );
        //new code here
    } else if ( $deleteEntry ) {
        $entryTable->deleteEntry( $id );
    }
    //end of new code. No changes below
}
```

```
$entryRequested = isset( $_GET['id'] );
if ( $entryRequested ) {
    $id = $_GET['id'];
    $entryData = $entryTable->getEntry( $id );
    $entryData->entry_id = $id;
}

$editorOutput = include_once "views/admin/editor-html.php";
return $editorOutput;
```

Test your work! Load `http://localhost/blog/admin.php?page=entries` in your browser; click an entry to load it into the editor; and delete the entry. Clicking the Delete button should delete the entry and reload the empty editor. Confirm that the selected entry was, in fact, deleted.

There is a little detail I would like to bring to your attention. The `entry_id` can be found in two different places. In one part of the code, you look for the `entry_id` in `$_POST['id']`; in another part of your code, you look in `$_GET['id']`. It is a little peculiar that they hold identical values but serve different purposes. Perhaps it is helpful to consider where these URL variables were encoded.

The `$_GET['id']` gets encoded every time a user clicks a blog title listed on `http://localhost/blog/admin.php?page=entries`. So, `$_GET['id']` represents the `entry_id` of a blog entry a user would like to see in the entry editor.

The `$_POST['id']` gets encoded every time an entry has been loaded into the entry editor. It represents the `entry_id` of the entry the user has just seen in the editor. So, `$_GET['id']` represents the entry to be loaded, whereas `$_POST['id']` represents the already loaded entry.

## Preparing a Model to Update Entries in the Database

You have an editor that can create new entries and delete existing ones. The next step is to update your editor code, so that you can update existing entries. Updating an existing entry in the database is definitely a job for a model. You can add an `updateEntry()` method to your `Blog_Entry_Table` class, as follows:

```
//Partial code for models/Blog_Entry_Table.class.php

//declare new method
public function updateEntry ( $id, $title, $entry) {
    $sql = "UPDATE blog_entry
            SET title = ?,
            entry_text = ?
            WHERE entry_id = ?";
    $data = array( $title, $entry, $id );
    $statement = $this->makeStatement( $sql, $data) ;
    return $statement;
}
```

Remember SQL update statements from the poll? Here they are again! There should be no surprises in the preceding code. The next task is to call the new method at the right time, i.e., call `updateEntry()` when a user clicks Save while an existing entry is loaded into the entry editor.

## Controller: Should I Insert or Update?

When a user clicks Save, the displayed entry should either be inserted or updated in the database. Which action to take depends on which entry was displayed in the entry editor form. Remember the hidden input from the editor view? It stores the currently displayed entry's `entry_id`, or 0, if the editor fields are blank. You can use that to check whether the admin user is trying to insert a new row in the `blog_entry` table or update an existing row.

When the Save button is clicked, you can get the value from the hidden input. If the editor was empty, the hidden input holds a value of 0. That means the admin user has just created a new entry. Your code should insert a new row into `blog_entry`. If it holds any other integer, you should update the `blog_entry` with the corresponding `entry_id`. The code to deal with user interaction belongs in the controller. It is time to change some code in an `if` statement in `controllers/admin/editor.php`, as follows:

```
//partial code for controllers/admin/editor.php
//this is line 6 in my script
$editorSubmitted = isset( $_POST['action'] );
if ( $editorSubmitted ) {
    $buttonClicked = $_POST['action'];

    //new code begins here
    $save = ( $buttonClicked === 'save' );
    $id = $_POST['id'];
    //id id = 0 the editor was empty
    //so user tries to save a new entry
    $insertNewEntry  = ( $save and $id === '0' );
    //comment out or delete the line below
    //$insertNewEntry = ( $buttonClicked === 'save' );
    $deleteEntry = ($buttonClicked === 'delete');
    //if $insertNewEntry is false you know that entry_id was NOT 0
    //That happens when an existing entry was displayed in editor
    //in other words: user tries to save an existing entry
    $updateEntry = ( $save and $insertNewEntry === false );
    //get title and entry data from editor form
    $title = $_POST['title'];
    $entry = $_POST['entry'];

    if ( $insertNewEntry ) {
        $entryTable->saveEntry( $title, $entry );
        //new code below
    } else if ( $updateEntry ){
        $entryTable->updateEntry( $id, $title, $entry );
        //end of code changes
    } else if ( $deleteEntry ) {
        $entryTable->deleteEntry( $id );
    }
}
```

Keep in mind that the user might click the Save button in two of the following different scenarios:

- The user wants to save a new entry.
- The user wants to save some changes in an existing entry.

Your code must be able to differentiate between these two user actions. This is where it is really good to know that when the empty editor is displayed, the hidden input named `entry_id` will have a value of 0. So, when the user clicks the Save button, and `entry_id` is 0, the user is really trying to insert a new entry. Take a look in the preceding code and note how that is expressed in code. Read the following code slowly, and let the meaning sink in. Don't rush this step of your learning process:

```
//code fragments from controllers/admin/editor.php – make no changes

//$save becomes TRUE if the button with the value of 'save' was clicked
$save = ( $buttonClicked === 'save' );

//and later in the same script...
//$insertNewEntry becomes TRUE only if $save is TRUE and $id is 0. Both conditions must be TRUE
$insertNewEntry  = ( $save and $id === '0' );
```

It is also good to keep in mind that if an existing entry was displayed in the editor, the hidden `entry_id` will have a value different from 0. So, if the user clicks Save and `entry_id` is not 0, the user is really trying to update an existing entry. It is a small task to call either `saveEntry()` or `updateEntry()`, once your code has determined which action a user has requested. You can see it expressed in code in the `if-else if` statement. Test the progress of your work. You should be able to load an existing entry into the editor and change it. You should also be able to create a new entry.

## Communicating Changes

The entry editor make any changes "silently"—it does not inform the user if an entry was saved or not. You can improve the editor, so that it provides feedback to users. There are, as you can imagine, many possible approaches. I propose you show a message *Entry was saved* whenever a new or existing entry has been saved. You can change the code, so that changes are communicated clearly to users, and continue to show the saved entry in the editor. These improvements will require code changes in model, view, and controller.

### Step 1: Update Model

PHP cannot really *continue* to show a saved entry, because the entire entry editor is generated from scratch with every new HTTP request. It is impossible to change only a small part of a PHP-generated HTML page. PHP will change all or nothing at all.

---

■ **Note** Well, perhaps not exactly impossible. You can do it, if you combine PHP and JavaScript with AJAX, but that is a topic beyond the scope of this book.

---

But there is always another way. You can give users the impression that an entry continues to be loaded in the editor. You can simply reload it immediately. To be able to reload the saved entry in the editor, you need its `entry_id`. You already know the `entry_id` of any updated entry, but you don't know the `entry_id` of a new entry only just inserted into the database. Update the `saveEntry()` method in `models/Blog_Entry_Table.class.php`, so that it returns the saved entry's `entry_id`, as follows:

```
//partial code for models/Blog_Entry_Table.class.php
//edit existing method
public function saveEntry ( $title, $entry ) {
    $entrySQL = "INSERT INTO blog_entry ( title, entry_text )
                                VALUES ( ?, ?)";
    $formData = array($title, $entry);
    $entryStatement = $this->makeStatement( $entrySQL, $formData );
    //new code below
    //return the entry_id of the saved entry
    return $this->db->lastInsertId();
}
```

Note the `lastInsertId()` method. It is a standard PDO method that can often be very handy. It does what you would expect: it returns the id of the most recently inserted row. All it requires is that the table in question be created with an auto-incrementing primary key.

## Step 2: Update Controller

Now that the `saveEntry()` method returns an `entry_id`, you also have to change some code in the controller, to remember the returned `entry_id`. It can be done with a tiny change in `controllers/admin/editor.php`. You simply have to declare a variable that stores the returned value, as follows:

```
//partial code for controllers/admin/editor.php

//this is line 16 in my script
//update existing if-statement
if ( $insertNewEntry ) {

    //introduce a variable to hold the id of a saved entry
    $savedEntryId = $entryTable->saveEntry( $title, $entry );
} else if ( $updateEntry ){
    $entryTable->updateEntry( $id, $title, $entry );

    //in case an entry was updated
    //overwrite the variable with the id of the updated entry
    $savedEntryId = $id;
} else if ( $deleteEntry ) {
    $entryTable->deleteEntry( $id );
}
```

■ **Note** This project is quickly becoming more complex, and scripts are growing longer. It is probably getting harder for you to know exactly where to implement code changes. If you get stuck, you can always download the complete source code for a particular chapter from the book's companion site at www.apress.com.

With the preceding code changes implemented, the editor controller now knows the `entry_id` of the entry that was just submitted through the entry editor form. If PHP can find a variable called `$savedEntryID`, you know that an entry was just saved or updated.

If PHP finds `$savedEntryID`, you should show a message telling users that *Entry was saved.* You want the editor to display the created or updated entry. So, you have to get a `StdClass` object, with entry data to have it rendered. But not only that: you want to display a confirmation message, indicating whether an entry was saved or updated. You can do that in the controller, in `controllers/admin/editor.php` just before the view is loaded.

```
//partial code for controllers/admin/editor.php

//update existing if-statement
$entryRequested = isset( $_GET['id'] );
if ( $entryRequested ) {
    $id = $_GET['id'];
    $entryData = $entryTable->getEntry( $id );
    $entryData->entry_id = $id;
    //new code: show no message when entry is loaded initially
    $entryData->message = "";
}

//new code below: an entry was saved or updated
$entrySaved = isset( $savedEntryId );
if ( $entrySaved ) {
    $entryData = $entryTable->getEntry( $savedEntryId );
    //display a confirmation message
    $entryData->message = "Entry was saved";
}
//end of new code

$editorOutput = include_once "views/admin/editor-html.php";
return $editorOutput;
```

At this point, your entry editor should reload a saved or updated blog entry. You can test it quite easily. Load `http://localhost/blog/admin.php?page=entries` in your browser. Click an entry title, to load the entry into the entry editor form. Change the entry a little and click Save. You should see that the form is reloaded and continues to show the entry.

It is an improvement, compared to before, when clicking Save would have resulted in an empty entry editor form. Try to imagine yourself as a normal user. From that perspective, you will probably agree that it is nicer to have clear feedback from the system, indicating that the entry has really been saved. Wait a second. What about the feedback message? I don't see it! You nearly have it. A feedback message lives in PHP memory, as `$entryData->message`. It will be a small task to update the view and show the feedback message.

## Step 3: Update View

At this point, you get a new or updated entry's `entry_id` from the model. Your controller determines whether an entry has just been saved or updated and adds an appropriate confirmation message. The final step is to update the view, so that the confirmation message is displayed along with the entry. Update the code in `views/admin/editor-html.php`, as follows:

```
<?php
//complete code for views/admin/editor-html.php

$entryDataFound = isset( $entryData );
if( $entryDataFound === false ){
    //default values for an empty editor
    $entryData = new StdClass();
    $entryData->entry_id = 0;
    $entryData->title = "";
    $entryData->entry_text = "";
    //notice $entryData->message is blank when the editor is empty
    $entryData->message = "";
}

//notice new code below: $entryData->message is embedded
return "
<form method='post' action='admin.php?page=editor' id='editor'>
 <input type='hidden' name='id' value='$entryData->entry_id'/>
    <fieldset>
        <legend>New Entry Submission</legend>
        <label>Title</label>
        <input type='text' name='title' maxlength='150'  value='$entryData->title' />

        <label>Entry</label>
        <textarea name='entry'>$entryData->entry_text</textarea>

        <fieldset id='editor-buttons'>
            <input type='submit' name='action' value='save' />
            <input type='submit' name='action' value='delete' />
            <p id='editor-message'>$entryData->message</p>
        </fieldset>

    </fieldset>
</form>";
```

Test your work by loading the editor in your browser. If you create a new entry and save it, you should see the editor message *Entry was saved*. This is great! Your editor provides clear feedback, and you know that your new entry has been saved. You can also try to load an existing entry to make some editorial changes and save. Again, you should see that *Entry was saved*.

From a user's perspective, your entry editor is now greatly improved. Communicating changes is almost as vital as actually making changes.

## Insisting on a Title

We're slowly starting to focus on usability as much as functionality. Now that you are working on the entry editor anyway, I'd like to point out a usability flaw. It is possible to create a new blog entry without specifying a title for it! I have done it here, for demonstration purposes (see Figure 9-1). The problem is that you use blog entry titles to list all entries in the administration module.

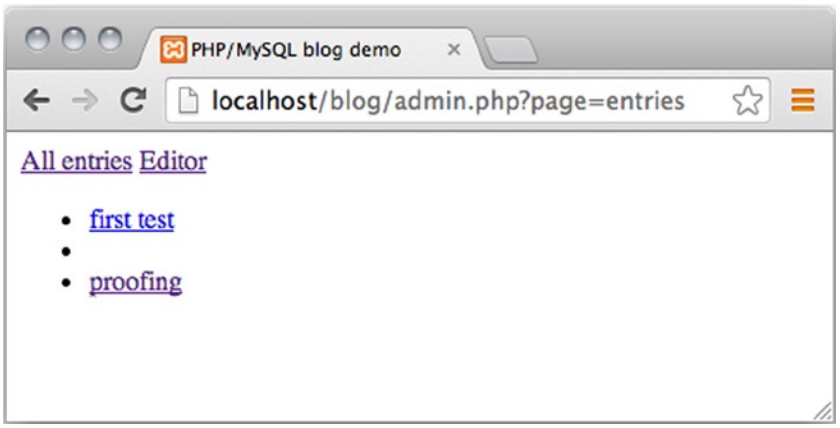


***Figure 9-1.*** *A title-less blog entry listed in the admin module*

The problem is that a blog entry without a title cannot be clicked, and consequently, such an entry is not loaded into the entry editor form. That's a little unfortunate, but perhaps not a huge problem, because the created blog entry will be displayed to ordinary users visiting `index.php`.

If you really wanted to edit the blog entry, you could go through phpMyAdmin. You might say that this is a usability problem, rather than a functional problem. Functionally speaking, it *is* possible to edit the entry, but it would be much more convenient, and much easier for users, if all blog entries could be edited through the entry editor.

One solution is to change your entry editor, so it insists that a title must be declared for blog entries. You can simply add a `required` attribute on the `<input>` element for the title. Do that, and it will be impossible to submit the entry editor form if no title is declared. Update one line of code in `views/admin/editor-html.php`, as follows:

```
//partial code for views/admin/editor-html.php

//notice the added required attribute
<input type='text' name='title' maxlength='150' value='$entryData->title' required />
```

Implement that small change in your code and try to save a new entry without a title. As you try to save, you will see that the form is not submitted. All you get is a little warning, as shown in Figure 9-2.

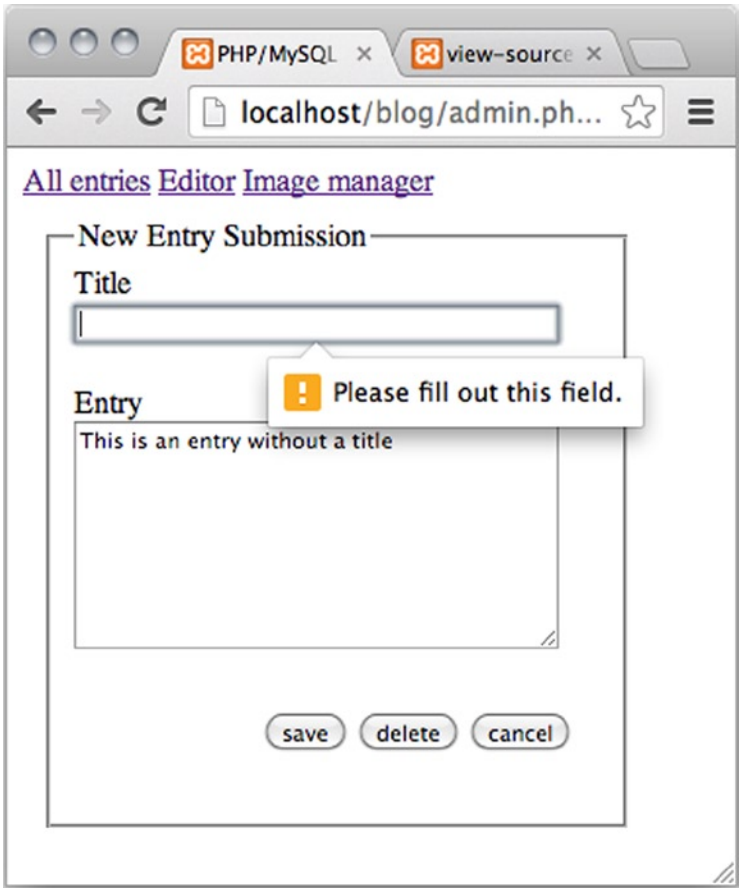


***Figure 9-2.*** *Trying to create a blog entry without a title*

This solution will work beautifully in most modern browsers, but at the time of this writing, there are some notable exceptions. It will not work in Safari, nor will it work in a number of mobile browsers.

---

■ **Note** Using the `required` attribute for client-side form validation is new to HTML5. You can see which browsers currently support such form validation at `http://caniuse.com/form-validation`.

---

## Improving Editor Usability with Progressive Enhancement

Fixing a usability flaw with a simple HTML attribute is great, because it is so easy to implement. Unfortunately, not all browsers support the `required` attribute. But that is great, too, as this is an opportunity for you to exercise your JavaScript.

The `required` attribute can be used to prevent form submission for users employing updated versions of Chrome, Firefox, Opera, and Internet Explorer. Apple's Safari browser doesn't block form submissions, even if a required field is not filled out. A possible solution would be to use JavaScript to detect whether the `required` attribute is supported. If it is not, you could use JavaScript to prevent incomplete form submissions. The problem is that when JavaScript asks Safari if it supports `required`, Safari will claim support. But, in fact, Safari doesn't really support the `required` attribute.

You could consider using JavaScript to detect the name of the visiting browser. That is possible through JavaScript's `navigator.userAgent`. Unfortunately, Chrome's `navigator.userAgent` string has the word *Safari* in it. So, your JavaScript could easily mistake Chrome for Safari. And as if that weren't enough, user agent strings can easily be spoofed. So, user agent strings aren't very reliable.

I propose a different solution: provide a JavaScript solution for all modern browsers. Continue to use the `required` attribute for fully HTML5-compliant browsers, with JavaScript disabled. Accept that Safari users with JavaScript disabled will be able to create a blog entry with no title.

First of all, you want to make sure that your JavaScript only runs in modern browsers. You don't want ancient browsers choking on our modern JavaScript. You can use the approach you've already tried in Chapter 5. Create a new folder for your JavaScript files. You could call it `js`. Create a new JavaScript file in `js/editor.js`.

```
//complete code for js/editor.js
function init(){
    console.log('your browser understands DOMContentLoaded');
}

document.addEventListener("DOMContentLoaded", init, false);
```

## Embedding Your External JavaScript

The preceding script uses the `DOMContentLoaded` event to hide your JavaScript from older browsers. But you can't test your JavaScript code just yet. Your browser must load the JavaScript file first. To do that, you can open `admin.php` in your code editor and embed the JavaScript file using the `Page_Data->addScript()` method you created back in Chapter 5, as follows:

```
//partial code for admin.php
include_once "models/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "PHP/MySQL blog demo";
$pageData->addCSS("css/blog.css");
//new code: add the Javascript file here
$pageData->addScript("js/editor.js");
//no changes below
```

Now, you can save and test progress. Open your browser and its JavaScript console. If you're using Chrome, you can open the console with Cmd+Alt+J, if you are working on a Mac, or Ctrl+Alt+J, on a Windows machine.

Once the browser's JavaScript console is open, you can navigate your browser to `http://localhost/blog/admin.php?page=entries`. Note the little output in the console, shown in Figure 9-3. It confirms that your browser runs the JavaScript code.

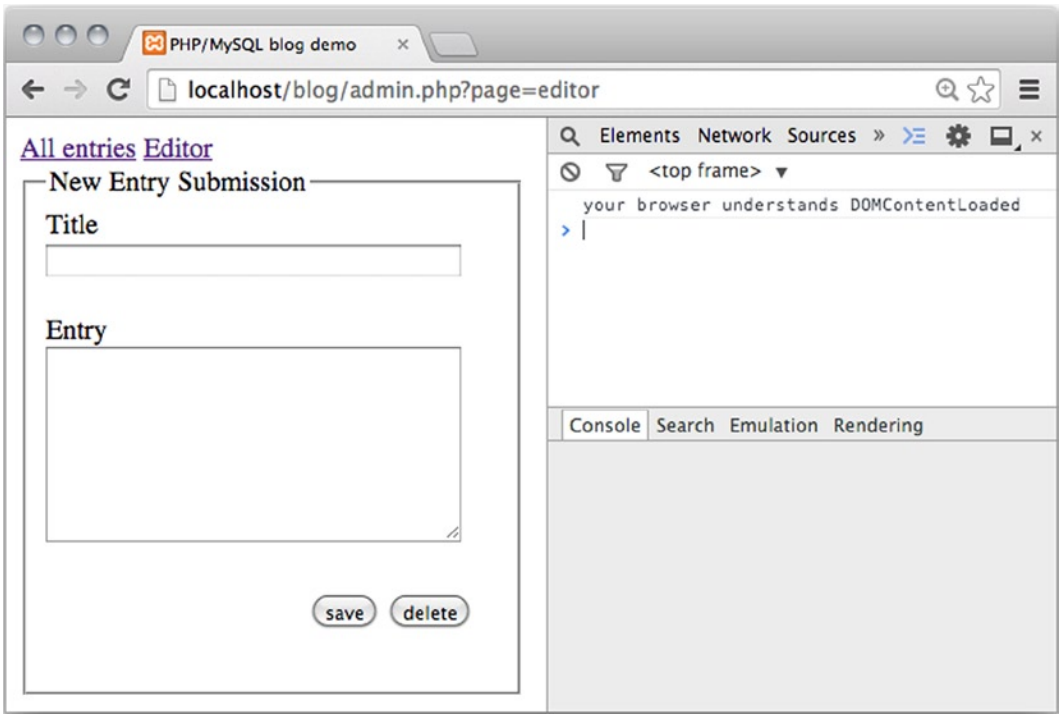


***Figure 9-3.*** *A console message in Chrome*

## Showing a Warning If Title Is Empty

You want your system to tell the user to enter a title before allowing form submission. So, you need an HTML element to output a message to users. Many polyfills you can find online will create and style such an HTML element dynamically. I want to keep the JavaScript really simple, so I don't want to create HTML elements with JavaScript. I propose the simpler but less elegant approach of hard-coding an empty HTML element into the view, as follows:

```
//partial code for views/admin/editor-html.php
//notice the <p id='title-warning'> element added
<form method='post' action='admin.php?page=editor' id='editor'>
 <input type='hidden' name='id' value='$entryData->entry_id' />
    <fieldset>
        <legend>New Entry Submission</legend>
        <label>Title</label>
        <input type='text' name='title' maxlength='150' value='$entryData->title' required/>
        <p id='title-warning'></p>
        <label>Entry</label>
```

You can use JavaScript to detect when the entry editor form is submitted. If the title is empty, you can prevent form submission and show an error message. The form should only be submitted if the title is not empty. Here's how to express that in JavaScript:

```
//Complete code for js/editor.js
//declare new function
function checkTitle (event) {
    var title = document.querySelector("input[name='title']");
    var warning = document.querySelector("form #title-warning");
    //if title is empty...
    if (title.value === "") {
        //preventDefault, ie don't submit the form
        event.preventDefault();
        //display a warning
        warning.innerHTML = "*You must write a title for the entry";
    }
}

//edit existing function
function init(){
    var editorForm = document.querySelector("form#editor");
    editorForm.addEventListener("submit", checkTitle, false);
}

document.addEventListener("DOMContentLoaded", init, false);
```

The preceding code would work, but fully HTML5-compliant browsers will never execute your `checkTitle()` function, because the `required` attribute will cause the browser to show a standard warning. So, a browser such as Chrome would show a standard warning, while Safari would show your JavaScripted warning. It's not really much of a problem, but the standard warnings of different browsers look different. You can see for yourself, if you test your blog entry editor in Chrome and Firefox. If you want your editor to look similar across multiple browsers, you should write JavaScript to suppress the standard browser warning and show your JavaScript warning instead. It is easy to do, with a tiny change in the `init()` function declared in `js/editor.js`, as follows:

```
//partial code for js/editor.js

//edit existing function
function init(){
    var editorForm = document.querySelector("form#editor");
    var title = document.querySelector("input[name='title']");
    //this will prevent standard browser treatment of the required attribute
    title.required = false;
    editorForm.addEventListener("submit", checkTitle, false);
}
```

That's it! If a user comes along with a modern browser, your code will rely on client-side validation using JavaScript. You can see it in action in Safari, in Figure 9-4. Users with fully HTML5-compliant browsers and JavaScript disabled will be served client-side validation using the HTML5 `required` attribute. Safari users with JavaScript disabled are out of luck: they will be able to create a blog entry with no title. Users running legacy browsers that don't support the `required` attribute and don't support modern JavaScript will simply have to accept that they can create blog entries without titles. Users with such outdated technology will get a less optimal experience, but at least they can create blog entries.

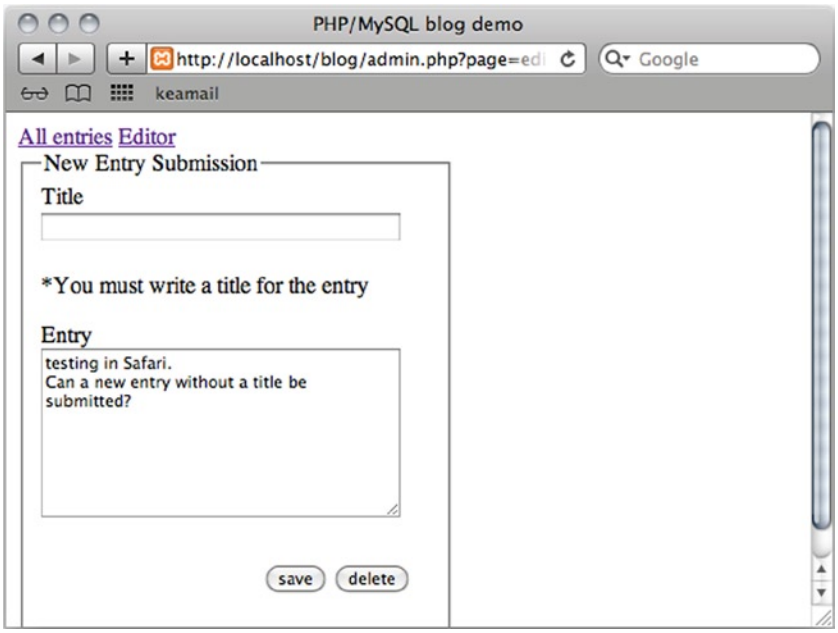


***Figure 9-4.*** *Testing editor in Safari with JavaScript enabled. A user cannot submit an entry without a title*

## Other Usability Flaws

There are other usability flaws in the editor, at this point. One is that administrators have to know a bit of HTML to format the entries in any way. Also, it is a big disadvantage that administrators will probably find it rather cumbersome to use images as part of a blog entry.

As if that weren't enough, the confirmation message indicating whether an entry has been saved or not can be a bit misleading at times. If you create a new entry and save it, you will see confirmation that the entry was saved. If you continue to edit the entry further, the confirmation message will still show that the entry was saved, even though that is no longer true.

That is not so great. Generally, you don't want misleading messages! You will continue to improve the usability of the blog entry editor in Chapter 11.

## A Coding Challenge: Fix a Usability Flaw

I'd like you to notice a usability flaw that you can try to remedy with PHP. When you load an existing blog entry into the editor, the `<legend>` element will have a value of New Entry Submission. That's a misleading `<legend>`. It fits fine when you create a new blog entry, but when you edit an existing entry, it is bad!

You can change it with PHP. There are, as always, many ways to approach the problem. One approach would be to argue that the problem relates only to the view. That would mean you only have to change code in `views/admin/editor-html.php`.

If you look in your code, you will see that you already have an `$entryData->message`. You use it to show different messages at different times. You could take a similar approach and declare a `$entryData->legend`. You would replace the hard-coded `<legend>` value with the new `$entryData->legend`. The bigger problem is assigning an appropriate value to `$entryData->legend`. When you load an existing entry into the editor, the value of `$entryData->legend` could be *Edit Entry*. When you load an empty editor, the value could be *New Entry Submission*. The challenge is for you to make it happen.

# Summary

In this chapter, you have greatly improved the entry editor. It can now delete or edit existing entries, and it provides some client-side validation to enhance the user experience. Back in Chapter 5 you were introduced to the notion of progressive enhancement, which is a common approach to using JavaScript. In this chapter, you have seen another common JavaScript task: using a JavaScript polyfill to remedy inconsistencies across browsers.

In terms of code, this chapter was mostly a repetition of principles already covered. The code is getting more complex, especially the editor controller. Despite the increased complexity, you're still seeing previously covered principles.

But something has changed radically: you have started to use code to design user experiences. You are no longer solely concerned with the basic problem of getting the code to work. You're starting to focus more on how it works from a user's perspective. You are writing code to design systems that communicate with users. This is a topic you will pursue further in subsequent chapters.

CHAPTER 10

# Improving Your Blog with User Comments and Search

This chapter returns to the public face of your blog. At this point, blog visitors can see a list of all the blog entries you wrote and can click an entry to read all of it.

One of the most important features of modern web applications is the ability to allow users to interact via a commenting system. Nearly every blog in existence allows its readers to comment on entries. This adds to the experience of the users and the blog author, by enabling everyone to continue the conversations the author initiates with his posts. This chapter shows you how to add a commenting system to the blog.

This will be an opportunity for you to strengthen your grasp of some of the things you have learned already and pick up another few skills. In this chapter you will

- Design a form for user comments
- Deal with complex views
- Develop a controller script to deal with users interacting with the form
- Create a new database table for user comments
- Use a foreign key to associate comments with blog entries
- Use inheritance to avoid redundant code
- Write a class to provide access to the comment table

## Building and Displaying the Comment Entry Form

An essential part of any commenting system is the user interface. You need a form that allows users to compose comments for blog entries. The form is a *view*. You will also need a corresponding model and a controller. You have to start somewhere. Create a new file in `views/comment-form-html.php`, as follows:

```
<?php
//complete code for views/comment-form-html.php

$idIsFound = isset($entryId);

if( $idIsFound === false ) {
    trigger_error('views/comments-html.php needs an $entryId');
}

return "
<form action='index.php?page=blog&amp;id=$entryId' method='post' id='comment-form'>
    <input type='hidden' name='entry-id' value='$entryId' />
    <label>Your name</label>
    <input type='text' name='user-name' />
    <label>Your comment</label>
    <textarea name='new-comment'></textarea>
    <input type='submit' value='post!' />
</form>";
```

To display the comment form, you need a comment controller. Its job at this early stage will simply be to load the view and return the HTML to have the comment form displayed.

```
<?php
//complete code for controllers/comments.php
$comments = include_once "views/comment-form-html.php";
return $comments;
```

So far, the code is short, to the point, and very much like previous code examples. You may notice that the comment form will not show up anywhere in your browser yet. The comment controller loads the comment view. But who should load the comment controller and actually show the comment form?

## A Combined View

A comment form should only be displayed when a complete entry is displayed. So, the page that shows an entry should also show a comment form: it is a complex view composed of other views. Figure 10-1 shows a simple solution to combine views. Figure out a hierarchy of the parts and load the “secondary controller” from the “primary controller.”

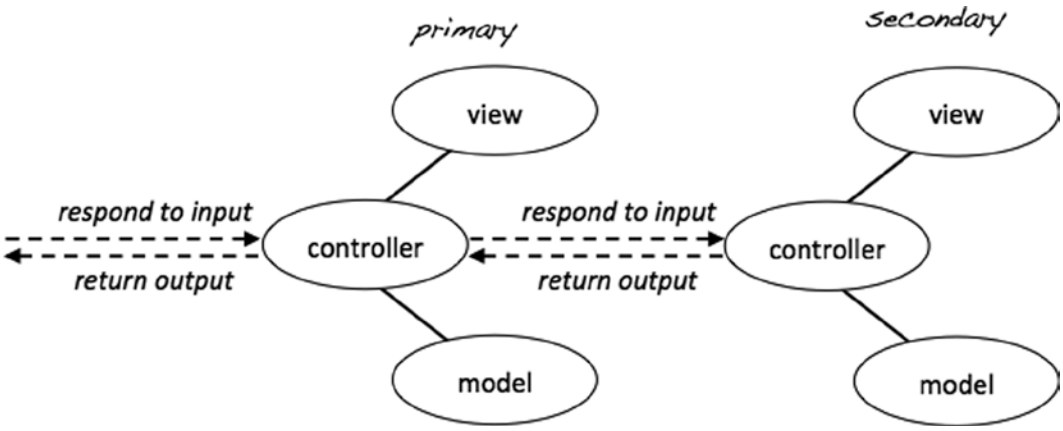


***Figure 10-1.*** *Constructing complex views*

You have already done this from the front controllers you have made. Take the code in `admin.php`, for example. It loads a model and a view for making HTML5 pages. Generating an HTML5 page is a major concern for `admin.php`. But depending on conditions, `admin.php` further loads either the editor controller or the list-entries controller, each of which will return some content to be embedded on the generated page. So, `admin.php` is the primary controller, and the subsequently loaded controller is a secondary controller.

The task at hand is to display a blog entry and an HTML form, so users can comment on the entry. In this case, it should be obvious that the primary controller is the blog controller. The comment form is only meaningful in the context of a blog entry. The blog controller loads blog entries. The blog controller should also load the comment controller. Here's how you can express that in `controllers/blog.php`:

```
<?php
//complete code for controllers/blog.php
include_once "models/Blog_Entry_Table.class.php";
$entryTable = new Blog_Entry_Table( $db );

if ( $isEntryClicked ) {
    $entryId = $_GET['id'];
    $entryData = $entryTable->getEntry( $entryId );
    $blogOutput = include_once "views/entry-html.php";

    //new code here: load the comments
    $blogOutput .=include_once "controllers/comments.php";
    //no other changes
} else {
    $entries = $entryTable->getAllEntries();
    $blogOutput = include_once "views/list-entries-html.php";
}
return $blogOutput;
```

Now you can start testing your progress! Navigate to `http://localhost/blog/index.php` and click *Read more* to see one entry displayed. At the very end of the blog entry, you should see the comment form displayed. It shouldn't come as a big surprise that you cannot submit any new comments just yet. You can see that the form is completely unstyled. It is not a pretty sight. You can style your comment form however you prefer. Here's a little CSS to get you started. I have added these new CSS rules to my style sheet in `css/blog.css`.

```
/*partial code for css/blog.css*/

form#comment-form{
    margin-top:2em;
    padding-top: 0.7em;
    border-top:1px solid grey;
}

form#comment-form label, form#comment-form input[type='submit']{
    padding-top:0.7em;
    display:block;
}
```

## Creating a Comment Table in the Database

Before you can begin working with comments, you need to have a place to store them. Create a table named *comment* in the `simple_blog` database. You'll use this to store all information about comments. You have to store several different kinds of information in this table, as follows:

- `comment_id`: A unique identifier for the comment. This is the table's primary key. You can use the `AUTO_INCREMENT` property, so that new comments are automatically assigned a unique id number.
- `entry_id`: The identifier of the blog entry to which the comment corresponds. This column is an `INT` value. The `entry_id` refers to a primary key in another table. The `entry_id` is a so-called foreign key.
- `author`: The name of the comment author. This column accepts a maximum of 75 characters and is of the `VARCHAR` type.
- `txt`: The actual comment text. I would have called the column *text*, but *text* is a reserved SQL keyword, so I can't use that. The column's data type should be `TEXT`.
- `date`: The date the comment was posted stored as a `TIME_STAMP`. You can set a default value for this column: the `CURRENT_TIMESTAMP`, which will provide a `TIME_STAMP` for the exact date and time when a user adds a new comment to the table.

To create this table, navigate to `http://localhost/phpmyadmin` in a browser, select the `simple_blog` database, and open the SQL tab. Execute the following command to create the comment table:

```
CREATE TABLE comment (
    comment_id INT NOT NULL AUTO_INCREMENT,
    entry_id INT NOT NULL,
    author VARCHAR( 75 ),
    txt TEXT,
    date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (comment_id),
    FOREIGN KEY (entry_id) REFERENCES blog_entry (entry_id)
)
```

A comment is a user's response to one particular blog entry. So, every new comment must be uniquely associated with one blog entry. Your blog will soon show comments, but all comments should not be displayed all the time. When a particular blog entry is displayed, only comments related to that blog entry should be displayed.

It follows that your database must be designed in such a way as to represent a relationship between blog entries and comments. The database design must support that any one comment can be related to only one blog entry. A logical solution is to create the comment table with a column for `entry_id`, as shown in Table 10-1. That way, one comment will know the `entry_id` of the entry it is related to.

***Table 10-1.*** *Comment Rows Related to Specific Entries*

| comment_id | entry_id | author | txt | date |
|---|---|---|---|---|
| 1 | 1 | Thomas | [...] | 2014-03-02 12:54:15 |
| 2 | 8 | Thomas | [...] | 2014-03-02 13:25:41 |
| 3 | 1 | Brennan | [...] | 2014-03-07 01:43:19 |

Take a look at the populated comment table shown in Table 10-1. See how any one comment will be explicitly related to a particular entry's `entry_id`? This way, every comment knows which blog entry it is related to. Notice also that one blog entry may have many associated comments. The entry with `entry_id = 1` has two comments in the preceding example. This kind of relationship is known as a *one-to-many relationship* in relational database terminology.

## Using a Foreign Key

It is very common to use a *foreign key constraint* when establishing relationships between two tables. You can see how a foreign key constraint is declared in the previous SQL statement. But what is it really for?

A foreign key constraint is just that—a *constraint*—that is, it restricts something. When a table field is declared with a constraint, you can't just insert anything into it. A constraint puts a limit on what data will be accepted by the field.

A foreign key constraint is also a *reference* to a primary *key* column in a *foreign* table. In the preceding example, the comment table's `entry_id` is a reference to the `blog_entry` table's `entry_id` column.

The used foreign key constraint helps maintain data integrity, because the comment table will only accept comments with an `entry_id` that can be found in the `blog_entry` table. In other words, the comment table will only accept comments related to blog entries that exist.

Now you have a database table ready for comments, and every blog entry is displayed with a form for accepting new comments from users. In MVC terminology, you have to write a *model* for inserting comments and update your *controller,* so it can respond to form submissions. Generally speaking, controllers should deal with user interactions, whereas models should deal with core logic and manipulate your database. You can start coding either the model or the controller. It is not so important where you start.

## Building a Comment_Table Class

Begin with the model, to start somewhere. You have just created a new table in your database. As you may remember, you have already used the *table data gateway* design pattern for accessing your `blog_entry` table.

A table data gateway provides a single point of access from your PHP code to one database table. Let's continue to use the table data gateway pattern by creating a new class to provide a single point of access to the comment table. You can call the new class `Comment_Table`, so its name clearly indicates what it represents.

---

■ **Note** Here's a simple rule of thumb for the table data gateway pattern: each one table should have a corresponding table data gateway class. One table, one class!

---

If you think back to the `Blog_Entry_Table` class, you can get a pretty good idea about what you will need in the new class. To get access to a database table, you'll need a PDO object. Whenever you instantiate a `Blog_Entry_Table`, you pass a PDO object as an argument to the constructor. The received PDO object is stored in a `$db` property, and it seems to do the job nicely. You can reuse that code as it is.

It could also be handy to have a `makeStatement()` method, just like in `Blog_Entry_Table`. With a `makeStatement()` method, your code in `Comment_Table` could stay nice and DRY (don't repeat yourself), or at least DRYish, as you will soon see.

To get off to a quick and painless start, you could simply copy the relevant code from the `Blog_Entry_Table` class and paste it into a new `Comment_Table` class. Once you have copied the identical code, you could start to declare new methods unique to the `Comment_Table`. Make a new file called `Comment_Table.class.php` in the `models` folder:

```
<?php
//complete code for models/Comment_Table.class.php
class Comment_Table {
    //code below can be copied from models/Blog_Entry_Table.class.php
    private $db;

    public function __construct ( $db ) {
        $this->db = $db;
    }

    private function makeStatement ( $sql, $data = null ){
        $statement = $this->db->prepare( $sql );
        try{
            $statement->execute( $data );
        } catch (Exception $e) {
            $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                    <p>Exception: $e</p>";
            trigger_error($exceptionMessage);
        }
        return $statement;
    }
} //end of class
```

Do you remember DRY? Isn't it a bit of a shame to have two different classes that have two identical methods and one identical property? Duplicate code is a recognized code smell, and the code above surely stinks! The problem is that you have two classes that in some ways are identical.

It is a very common problem—there is even a name for it. There are many solutions for such common problems. I'd like to show you a solution that is unique to object-oriented programming. That solution is called *inheritance*.

## Staying DRY with Inheritance

Using *inheritance*, you can create a single class definition whereby you keep the code to be shared among several classes. Next, you can create separate subclasses in which you keep the code that's unique for individual classes, such as your `Comment_Table` and your `Blog_Entry_Table`. You can see the idea expressed in Figure 10-2.

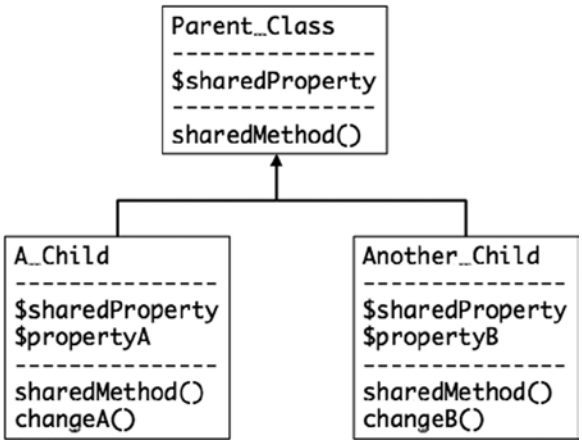


***Figure 10-2.*** *Subclasses inherit properties and methods from a parent class*

Figure 10-2 illustrates how some code you want to share among a number of classes can be declared in a *parent* class. All *children* of that parent will be *born* with those properties. In Figure 10-2, you can see that both `A_Child` and `Another_Child` have a `$sharedProperty` and a `sharedMethod()`. These were inherited from the parent class. In Figure 10-2, you can also see that `A_Child` and `Another_Child` each have special properties and methods. These are declared in the child class definition. For example, only `A_Child` has a `changeA()` method.

You could use that idea to share `$db` and `makeStatement()` between the `Blog_Entry_Table` and the `Comment_Table` classes. You can see such an architecture in Figure 10-3.

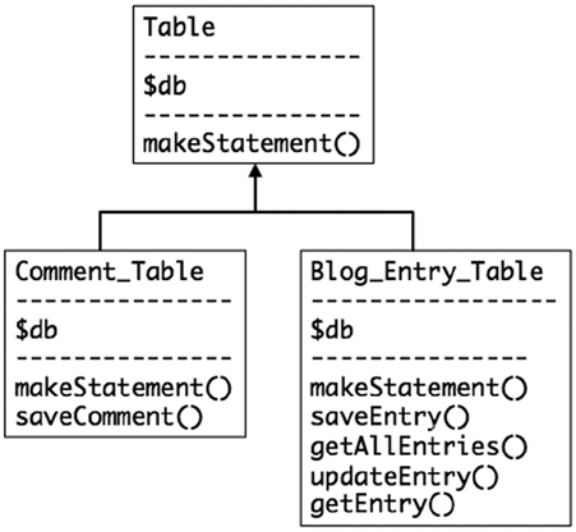


***Figure 10-3.*** *Using inheritance to make DRY table data gateway classes*

The `Comment_Table` and `Blog_Entry_Table` classes will both be born with a `$db` property and a `makeStatement()` method *inherited* from the parent, from the `Table` class. The code for `$db` and `makeStatement()` would be written only once in the `Table` class. The `$db` and `makeStatement()` would still be accessible in `Comment_Table` and `Blog_Entry_Table`, because they are both children of the same parent.

## Is-a Relationships

In object-oriented terminology, the relationship between a parent class and child class is referred to as an *is-a* relationship. A `Comment_Table` *is a* `Table`. The `Table` is a general abstraction that represents the general concept of a database table. The `Comment_Table` is a representation of a specific database table.

The concept of is-a relationships between objects is something you use in your everyday thinking. Coffee is a beverage. Orange juice also is a beverage. Orange juice and coffee share some characteristics, though they are clearly different. A beverage is the abstract idea of a consumable liquid. Coffee and orange juice are specific kinds of consumable liquids. You could probably come up with many other examples of abstract concepts and their concrete implementations. Object-oriented programming has borrowed a widely used human mode of reasoning and used it to bring hierarchical order to computer programs.

## Using Inheritance in Your Code

You can create the solution indicated in Figure 10-3. The code you want to share among subclasses must have *public* or *protected* access modifiers. Any property or method with a private access modifier will not be shared through inheritance. Here's what a general `Table` class could look like. Create a new file called `Table.class.php` in the `models` folder:

```
<?php
//complete code for models/Table.class.php

class Table {
    //notice protected, not private
    protected $db;

    public function __construct ( $db ) {
        $this->db = $db;
    }

    //notice protected, not private
    protected function makeStatement ( $sql, $data = null ){
        $statement = $this->db->prepare( $sql );
        try {
            $statement->execute( $data );
        } catch (Exception $e) {
            $exceptionMessage = "<p>You tried to run this sql: $sql <p>
                    <p>Exception: $e</p>";
            trigger_error($exceptionMessage);
        }
        return $statement;
    }
}
```

The protected access modifier is quite similar to the private access modifier you have already used. Protected methods and properties cannot be accessed from the outside. They can only be accessed from inside the class itself. But if you used private here, the `makeStatement()` method and the `$db` property would not be available to subclasses.

To make this code available to a subclass such as `Comment_Table`, you will have to include the `Table` class definition script in your code and use the keyword `extends`. Here's what the almost completely rewritten DRY `Comment_Table` should look like:

```
<?php
//complete code for models/Comment_Table.class.php

//include parent class definition
include_once "models/Table.class.php";

//extend current class from parent class
class Comment_Table extends Table{
    //delete all previous code inside class
    //it should be completely empty
}
```

See how the `Comment_Table` class is simply an empty code block at this point? It is not at all obvious from the code, but the `Comment_Table` was "born" with a `makeStatement()` method and a `$db` property. You cannot see them in the `Comment_Table` class, but they are available here. They are inherited from the `Table` class, because the `extends` keyword is used; the `Comment_Table` extends the `Table`. Because of that, all public and protected methods and properties declared in `Table` are directly available in the `Comment_Table`. A `Comment_Table` *is a* `Table`. Figure 10-4 shows the code architecture.

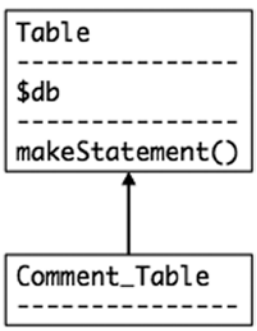


***Figure 10-4.*** *A `Comment_Table` is a special kind of table*

## Inserting New Comments into the Database

You can add a method to the `Comment_Table`, so that you can insert new comments into the database. Call the new method `saveComment()`:

```
<?php
//complete code for models/Comment_Table.class.php

include_once "models/Table.class.php";
```

```
class Comment_Table extends Table{

    //declare a new method inside the Comment_Table class
    public function saveComment ( $entryId, $author, $txt ) {
        $sql = "INSERT INTO comment ( entry_id, author, txt)
                VALUES (?, ?, ?)";
        $data = array( $entryId, $author, $txt );
        $statement = $this->makeStatement($sql, $data);
        return $statement;
    }

}
```

That code should drive home what inheritance is. See how `$this->makeStatement()` can be used in `Comment_Table`? It is possible because the `makeStatement()` method was inherited from the `Table` class. The `Comment_Table` is "born" with all public and protected properties and methods declared in the `Table` class.

## Testing the saveComment() Method

It is time to test whether the `Comment_Table` class and its `saveComment()` method work. You can hard-code a preliminary comment and insert it just for testing purposes. The comment controller should be responsible for user interaction related to comments, so edit `controllers/comments.php`:

```
<?php
//complete code for controllers/comments.php

//include class definition
include_once "models/Comment_Table.class.php";
//create a new object, pass it a PDO database connection object
$commentTable = new Comment_Table($db);
//insert a test comment for entry_id = 1
//assuming an entry_id of 1.
$commentTable->saveComment( 1, "me", "testing, testing" );

$comments = include_once "views/comment-form-html.php";
return $comments;
```

The preceding test code assumes that you have an `entry_id = 1`. If you don't, you can use another `entry_id` present in your `blog_entry` database table. When you have written the test code, you can navigate your browser to `http://localhost/blog/index.php?page=blog` and click any *Read more* link to run your test code. The code should insert a test comment into your comment table. To see if your code worked, you will have to load `http://localhost/phpmyadmin` and browse the comment table. You should expect to see one row inserted into your comment table, as shown in Figure 10-5.


***Figure 10-5.*** *A row inserted into the comment table, as seen in phpMyAdmin*

Notice that the comment_id and date field values were created automatically. The entry_id, author, and txt values are received from the comments controller and inserted by way of the saveComment() method in the Comment_Table object.

## Retrieving All Comments for a Given Entry

It is great to have a comment for entry_id = 1 in the database. But it isn't a particularly useful comment, as long as blog visitors cannot see it in their browser. To display all comments for a given entry, you have to get the data for all comments associated with the given entry's entry_id. Your Comment_Table should be the single point of access from PHP to the comment database table. Declare a new method in models/Comment_Table.class.php to get all entries for a particular entry_id, as follows:

```
//partial code for models/Comment_Table.class.php

//declare new method inside the Comment_Table class
public function getAllById ( $id ) {
    $sql = "SELECT author, txt, date FROM comment
            WHERE entry_id = ?
            ORDER BY comment_id DESC";
    $data = array($id);
    $statement = $this->makeStatement($sql, $data);
    return $statement;
}
```

Take a moment to read the SQL statement used in the preceding code. It will SELECT author, txt, and date columns for all comments associated with a particular entry_id. Remember that the entry_id was declared as a *foreign key*. It is a reference to the primary key of the blog_entry table. Through the entry_id, you can unambiguously identify one particular blog_entry: you know which blog entry a comment is related to.

The comments will be ordered chronologically by comment_id values in DESCending order. In other words, comments with a higher comment_id value will be listed before comments with a lower comment_id value. The comment_id column is declared as auto_incrementing, which means the very first comment inserted will automatically get a comment_id value of 1. The next comment will get a comment_id value of 2, and so forth. So, the newest comments will have the highest comment_id value. It follows that the SQL statement above lists new comments first, older comments later.

Once again, you can appreciate that the makeStatement() method is inherited from the Table class. You can use the makeStatement() inside the Comment_Table class. Actually, you may recall that makeStatement() was declared with a protected access modifier. That means the makeStatement() method can only be called internally, from inside the class itself. It will be impossible to call makeStatement() from any other PHP script in your system.

## Testing getAllById()

You know it already. You have probably experienced it many times. Whenever you type some code, you may introduce a *bug*—a programming error—into your code. The only sane thing to do is to write a few lines of code and then test whether the code behaves as intended. Catch bugs in their infancy!

Open the comments controller and write a little code to test if the `getAllById()` method works as it should. If you call `getAllById()` with an `entry_id` of an entry you know has at least one comment associated, you should get back a `PDOStatement`. Call `fetchObejct()` of the `PDOStatement` and you should get a `StdClass` object representing one row of data from the comment table. You can rewrite `controllers/comments.php` to test the new `getAllById()` method, as follows:

```
<?php
//complete code for controllers/comments.php

include_once "models/Comment_Table.class.php";
$commentTable = new Comment_Table($db);

//query database
$allComments = $commentTable->getAllById( $entryId );
//get first row as a StdClass object
$firstComment = $allComments->fetchObject();

$testOutput = print_r( $firstComment, true );
die( "<pre>$testOutput</pre>" );

//PHP dies before coming to these lines
$comments = include_once "views/comment-form-html.php";
return $comments;
```

Navigate your browser to `http://localhost/blog/index.php?page=blog` and click *Read more* to run the test. If everything went well, you should see an output such as the following:

```
stdClass Object (
     [author] => me
     [txt] => testing, testing
     [date] => 2014-03-03 10:29:33
)
```

If you see a similar output, you have confirmed that `getAllById()` works as intended. The `die()` function will kill the PHP process, so it will effectively stop PHP in its tracks. It can sometimes be useful to kill a PHP script prematurely for debugging or testing.

## Creating a View for Listing Comments

At this point, you should have a confirmed test. You will require a view for displaying all comments. Create a new file in `views/comments-html.php`:

```
<?php
//complete code for views/comments-html.php

$commentsFound = isset( $allComments );
if($commentsFound === false){
    trigger_error('views/comments-html.php needs $allComments' );
}
```

```
$allCommentsHTML = "<ul id='comments'>";
//iterate through all rows returned from database
while ($commentData = $allComments->fetchObject() ) {
    //notice incremental concatenation operator .=
    //it adds <li> elements to the <ul>
    $allCommentsHTML .= "<li>
        $commentData->author wrote:
        <p>$commentData->txt</p>
    </li>";
}
//notice incremental concatenation operator .=
//it helps close the <ul> element
$allCommentsHTML .= "</ul>";
return $allCommentsHTML;
```

## Hooking Up View and Model to Display Comments

You can comment out or delete the test code in your comments controller. The final step to displaying comments is to load the view that will display all comments retrieved from the database:

```
<?php
//complete code for controllers/comments.php
include_once "models/Comment_Table.class.php";
$commentTable = new Comment_Table($db);
$comments = include_once "views/comment-form-html.php";

//new code starts here
$allComments = $commentTable->getAllById( $entryId );
//notice the incremental concatenation operator .=
$comments .=include_once "views/comments-html.php";
//no changes below

return $comments;
```

Check out any blog entry in your browser. You should expect to see the blog entry, then the comment form, and finally, a list of all comments associated with that blog entry.

# Inserting a Comment Through the Comment Form

You have a comment form, and comments are displayed. You have also established that your `saveComment()` method works. It should be a small task to insert new comments received from users through the form. To get input from a form, you need to know what HTTP method the form uses and what name attributes were used in the form.

Using PHP to retrieve form input is a very common task for web developers. It's something you really should understand thoroughly by the end of this book. I could just assume you know exactly what I mean and that you already understand the topic perfectly, but I'd rather give you an opportunity to test your own understanding. Take a look at the following code and see if you can figure out what *method* the comment form uses and which *name* attributes are used for the input and text area elements.

```
//partial code for views/comment-form-html.php
//make NO code changes
return "
<form action='index.php?page=blog&amp;id=$entryId' method='post' id='comment-form'>
    <input type='hidden' name='entry-id' value='$entryId' />
    <label>Your name</label>
    <input type='text' name='user-name' />
    <label>Your comment</label>
    <textarea name='new-comment'></textarea>
    <input type='submit' value='post!' />
</form>";
```

It wasn't a particularly hard challenge, was it? The form method is `post`. There is an `<input>` field named `user-name`, a hidden `<input>` named `entry-id`, and a `<textarea>` named `new-comment`. Knowing this, it is easy to write a little PHP in the comment controller to insert new comments from users, as follows:

```
<?php
//complete code for controllers/comments.php
include_once "models/Comment_Table.class.php";
$commentTable = new Comment_Table($db);

//new code here
$newCommentSubmitted = isset( $_POST['new-comment'] );
if ( $newCommentSubmitted ) {
    $whichEntry = $_POST['entry-id'];
    $user = $_POST['user-name'];
    $comment = $_POST['new-comment'];
    $commentTable->saveComment( $whichEntry, $user, $comment );
}
//end of new code

$comments = include_once "views/comment-form-html.php";
$allComments = $commentTable->getAllById( $entryId );
$comments .=include_once "views/comments-html.php";

return $comments;
```

Point your browser to any blog entry and submit a new comment through the form. You should expect to see the submitted comment listed alongside any other comments you may have for that blog entry. Your commenting system works!

## Practice Makes Perfect

You have come a long way since you started reading this book. You can see it when you look at your blog: it is a far cry from the "Hello from PHP" you wrote in Chapter 1. The most important development is the change that has taken place in your mind. You now know some PHP and MySQL. You still need many hours of experience before you get really familiar with what you have learned.

How about trying to apply some of the learned lessons without my instructions? You could change the `Blog_Entry_Table` class, so that it inherits from `Table`, to practice inheritance. It would be very similar to what you did with the `Comment_Table` class.

Or you could provide feedback to users, to let them know that the system has registered a new comment submitted. It would be very similar to the confirmation message you provided in the entry manager.

Here's another task you could work on: when a user reads a blog entry that has no comments, you could display a message similar to the following:

```
Be the first to comment this article
```

You could do that in the view. You should have PHP count how many rows of comments were returned from the database. If 0 rows were returned, you know there are no comments for the blog entry, and you should output a message like the one preceding.

Now that you have a commenting system, users can stumble on a slightly perplexing system behavior. Imagine that a user reads a blog entry and posts a comment. After that, the user would like to return to the list of all blog entries. So, she clicks the browser's Back button. But wait! That won't take the user back to the index. Wouldn't it be nice if every blog entry had an `<a>` linking back to `index.php`? Can you implement such a link?

You will probably find that it slows you down to write your own code. That is only to be expected. But you only learn to write your own code if you start to write your own code—you might as well get started sooner rather than later.

# Searching for Entries

You have come a long way with your blogging system. You have probably added a few entries through the entry editor already. Somebody visiting your blog might be looking for something specific you wrote at one time, and they might not remember which entry you wrote it in. You need to give them an option to search through entries.

You should show a search form, so that visitors can enter a search text. You should use any entered search text to perform a search in your database and return any entries that match the entered search term.

Can you see the three responsibilities? You'll need a *view* to show a search form and another *view* to show search results. You'll need a *model* to perform the database search and return a result. You'll need a *controller* to respond to user interactions. If the form was submitted, the controller should show search results; if not, show the search form.

## The Search View

It's always a good idea to begin with a small step. You can create an HTML form for the search view. It will be nothing fancy. Create a new file in `views/search-form-html.php`:

```
<?php
//complete code for views/search-form-html.php
return "<aside id='search-bar'>
    <form method='post' action='index.php?page=search'>
        <input type='search' name='search-term' />
        <input type='submit' value='search'>
    </form>
</aside>";
```

The view will display an HTML search form. You may be unfamiliar with the *search* type attribute used on the input element. A *search* field is simply a special kind of single-line text field. Search fields will remember a user's previous search terms and present the user with a drop-down list suggesting previous search terms.

Not all browsers support the search type. But any browser that doesn't support it will default to a basic `<input type='text'>`, so the search form will still work, even if a browser doesn't support the search type.

■ **Note** To check which browser supports which HTML5 elements, consult http://caniuse.com.

To display the search form, you should consider when you want it to be displayed. It would be nice to display the search form on every page view. To show the search form regardless of what else is displayed, you could load it from the front controller, from `index.php`. Add one line of code near the end of `index.php`:

```
//partial code for index.php
//new code: include the search view before the blog controller
$pageData->content .=include_once "views/search-form-html.php";
//end of new code
$pageData->content .=include_once "controllers/blog.php";

$page = include_once "views/page.php";
echo $page;
```

Test your progress by loading `http://localhost/blog/index.php` in your browser. You should see the search form displayed before the list of all blog entries. If you click *Read more*, you should see one particular blog entry. Notice that the search form is still displayed.

## Responding to a User Search

With a search form displayed, it is tempting to write a search term and submit it. Try it, and you will see no result in your browser—search form submissions are not detected. Eventually, you'd want to show a list of search results instead. You can begin with a preliminary search controller. Create a new file in `controllers/search.php`:

```
<?php
//complete code for controllers/search.php
return "You just searched for something";
```

You want the search controller to be loaded from the index, when a search has been performed. If no search has been performed, `index.php` should load the blog controller. Notice the `action` attribute of the search form:

```
//partial code from views/search-form-html.php, don't change anything
<form method='post' action='index.php?page=search'>
```

Whenever a user submits the search form, a URL variable named `page` with a value of `search` will be encoded as part of the request. So when `page` has a value of `search`, your web application should show search results. This will be quite easy to achieve from `index.php`:

```
//partial code for index.php

//new code starts here, in line 17 in my index.php
$pageRequested =  isset( $_GET['page'] );
//default controller is blog
$controller = "blog";
if ( $pageRequested ) {
    //if user submitted the search form
    if ( $_GET['page'] === "search" ) {
        //load the search by overwriting default controller
        $controller = "search";
    }
}
```

```
$pageData->content .=include_once "views/search-form-html.php";
//comment out or delete this line
//$pageData->content .=include_once "controllers/blog.php";
$pageData->content .=include_once "controllers/$controller.php";

//end of changes
$page = include_once "views/page.php";
echo $page;
```

That's it. Your front controller will only show either the blog or the search page. The search page will only be displayed when a search has been performed. Test progress by navigating your browser to `http://localhost/blog/index.php`. You should see the list of blog entries. Now make some search. You should see a short message from the search controller. It says, "You just searched for something," which basically confirms that the search controller was loaded.

## The Search Model

You have a search form view, and you have a preliminary search controller. It is time to work on a search model, so that you can perform an actual search. To perform a search, you will have to query your `blog_entry` database table. You already have a `Blog_Entry_Table` class to provide a single point of access to that table. The sensible thing to do would be to add another method to the `Blog_Entry_Table`. Do the sensible thing in `models/Blog_Entry_Table.class.php`, as follows:

```
//Declare new method in Blog_Entry_Table class
public function searchEntry ( $searchTerm ) {
    $sql = "SELECT entry_id, title FROM blog_entry
            WHERE title LIKE ?
            OR entry_text LIKE ?";
    $data = array( "%$searchTerm%", "%$searchTerm%" );
    $statement = $this->makeStatement($sql, $data);
    return $statement;
}
```

Perhaps the `$data` array warrants a few words of explanation. Isn't it strange to create an array with two separate items *when the items are identical*? Well, the number of unnamed placeholders in your SQL must exactly match the number of items in the array you execute. Because there are two placeholders in the SQL, you need an array with two values to use in the search. In the preceding example, your code will search for the same search term in two different table columns.

---

■ **Note** A ? represents an unnamed placeholder in prepared SQL statements.

---

Because you search two columns, you need two placeholders in the SQL statement. Because you search for the same search term in both columns, the two placeholders should be replaced with the same value. That is why the `$data` array should have a length of two, even if the two items are identical.

## Searching with a LIKE Condition

The SQL statement above demonstrates an SQL keyword I haven't used in the previous examples in this book: `LIKE`. Let's go through a slightly simpler example and gradually work up to the syntax used previously:

```
SELECT entry_id, title FROM blog_entry WHERE title LIKE 'test'
```

That query would return a result set with `entry_id` and `title` attributes of any `blog_entry` rows with a `title` attribute of exactly `test`. So, a row with a title of "This is a test" would *not* be part of the result.

Such a query quite clearly illustrates how `LIKE` works. But as a search, it is not very useful. To make a more useful search, you could add a *wildcard* character to the `LIKE` condition.

```
SELECT entry_id, title FROM blog_entry WHERE title LIKE 'test%'
```

The `%` character represents a wildcard character. A wildcard character represents anything. So the query would return a result set of all rows where the title begins with "test" followed by anything. A row with a *title* of "test if it works" would be returned. A row with a title of "This is a test" would *not* be returned. Surely you can work out how two wildcards can greatly improve the query.

```
SELECT entry_id, title FROM blog_entry WHERE title LIKE '%test%'
```

Such a query would return a row with a title of "test if it works" and a row with a title of "This is a test." So, the preceding query would be perfect, if you only wanted to search for entries with the word *test* in the title. You could easily widen the search and also look for matches in the `entry_text` column:

```
SELECT entry_id, title FROM blog_entry WHERE title LIKE '%test%'
OR entry_text LIKE '%test%'
```

That's great when searching for the word *test*. But you can safely assume that your blog visitors will search for other words or phrases. So, you need to create an SQL statement prepared with empty placeholders. PHP can retrieve a visitor's search term and insert that where the placeholders are. Here is the final SQL statement for searching is:

```
SELECT entry_id, title FROM blog_entry WHERE title LIKE '%?%'
OR entry_text LIKE '%?%'
```

This final SQL statement will return `entry_id` and `title` for any rows where `title` or `entry_text` match the search term submitted from a user.

## Test Model

Code in small steps, interspersed with informal tests! You can use `print_r()` in your controller to test if your `searchEntry()` method works. You could hard-code a search for a term you know should return a result. I know I have used the word *test* in a blog entry, so I hard-code a search for *test* and expect a result. To do that, I update `controllers/search.php`, as follows:

```
<?php
//complete code for controllers/search.php
//load model
include_once "models/Blog_Entry_Table.class.php";
$blogTable = new Blog_Entry_Table( $db );

//get PDOStatement object from model
$searchData = $blogTable->searchEntry( "test" );
//get first row from result set
$firstResult = $searchData->fetchObject();
//inspect first row
$searchOutput = print_r($firstResult, true);
$searchForm = include_once "views/search-form-html.php";
$searchOutput .= $searchForm;
//display all output on index.php
return $searchOutput;
```

Save your code and load `http://localhost/blog/index.php` in your browser. Now perform some search—it doesn't matter what you search for. When you submit the search form, you should expect to see the following: a printed object. It will look something like the following:

```
stdClass Object ( [entry_id] => 3 [title] => This is a test )
```

## A Search Result View

The output you just saw is a representation of a search result. It confirms that the `searchEntry()` method works. To show a search result in a way a user might appreciate, you'll need a search view. You have to wrap some HTML around the returned data. Create a new file in `views/search-result-html.php`:

```
<?php
//complete code for views/search-results-html.php
$searchDataFound = isset( $searchData );
if( $searchDataFound === false ){
    trigger_error('views/search-results-html.php needs $searchData');
}

$searchHTML = "<section id='search'> <p>
    You searched for <em>$searchTerm</em></p><ul>";

while ( $searchRow = $searchData->fetchObject() ){
    $href = "index.php?page=blog&amp;id=$searchRow->entry_id";
    $searchHTML .= "<li><a href='$href'>$searchRow->title</li>";
}

$searchHTML .= "</ul></section>";
return $searchHTML;
```

The preceding code assumes the existence of a `$searchData` variable. If it is not found, an error will be triggered. If the `$searchData` variable is found, the code will iterate through the result set with a `while` statement. The `while` loop will create an `<li>` element for each `blog_entry` that matches the search.

## Loading a Search Result View from the Controller

To display your search results in the browser, you must load the search result view. Hooking up a view with a model is a task for a controller. Update your search controller, update your code in `controllers/search.php`:

```
<?php
//complete code for controllers/search.php
include_once "models/Blog_Entry_Table.class.php";
$blogTable = new Blog_Entry_Table( $db );

$searchOutput = "";
if ( isset($_POST['search-term']) ){
    $searchTerm = $_POST['search-term'];
    $searchData = $blogTable->searchEntry( $searchTerm ) ;
    $searchOutput = include_once "views/search-results-html.php";
}
//delete all the code your wrote for testing the searchEntry method
return $searchOutput;
```

That's it. Your front controller will only show either the blog or the search page. The search page will show search results—even if there were no matches.

## Exercise: Improving Search

Did you notice a small problem with the search? Try to search for a term you absolutely know has no match in the database. I tried to search for *"#€%* and, obviously, there was no match. I took a look in the generated HTML source code, and the following is what I found:

```
<section id='search'>
    <p>You searched for <em>"#€%</em></p>
    <ul></ul>
</section>
```

That will never be valid HTML, nor is it particularly user-friendly. A small change of code could append a single `<li>` element to the `<ul>`. Perhaps something such as *No entries match your search*. Could you make that change? You will probably need to know that a `PDOStatement` will hold the value `FALSE`, if it comes back without a result set.

# Summary

You covered a lot of ground in this chapter, both in terms of learning and in improving your blog. Blog visitors will probably mostly notice the commenting system. The commenting system is a game changer, as far as interactive communication is concerned. All of a sudden, you're not just publishing your thoughts for the world to see. With a commenting system, you are inviting two-way communication between you and your readers.

I really enjoyed the opportunity to show you something about one of the classic characteristics of object-oriented programming: inheritance. It is a very smart way of staying DRY, as long as you don't overuse it.

It is possible to code long inheritance chains. You could make a Dog class, which *is a* child of Wolf, which *is a* child of Canine, which *is a* child of Quadruped, which *is a* child of Mammal. But experience shows that shallow inheritance relationships are preferable. Long inheritance chains lead to dependency issues, because the Dog would depend on the presence of the Wolf, which, in turn, depends on Canine, which depends on the Quadruped. Keep your inheritance chains short, and you'll be fine.

In this chapter, you came face to face with foreign keys in database design and with searching database tables with wildcard characters. Both are important topics you are bound to come across again in your future as a web developer.

CHAPTER 11



# Adding Images to Blog Entries

The face of your blog is complete! Sure there's room for improvement, but it is fully functional. The administration module can be used to create blog entries, but there are some serious flaws.

- It cannot delete blog entries if there are comments in the database related to the entry.
- It cannot add images to blog entries, and I am sure you will want to spice up your blog entries with images.
- Administrators have to know HTML in order to write blog entries, and not all great writers are equally great with HTML.
- Administrators can create an entry without a title, effectively making the blog entry inaccessible to users.

I am sure you can come up with other features that you would like to have added to the administration module. This chapter will implement the features just listed. In the process, you will learn about the following:

- Deleting entries and related comments
- Using a WYSIWYG editor for your entry editor
- Uploading images to the server
- Deleting image files from the server

## Problem: Cannot Delete an Entry with Comments

I am sure you will agree that the commenting system is a great improvement. Unfortunately, the comments have also introduced unwanted system behavior in the administration module. It has become impossible to delete an entry with comments.

I want you to see the problem before fixing it. Go to `http://localhost/blog/admin.php?page=editor` and create a new blog entry. Now point your browser to `http://localhost/blog/index.php` and click to read the entry you just created. Add one or two comments to the new entry, through the comment form. Now you have a blog entry with comments.

Load `http://localhost/blog/admin.php?page=entries` in your browser and click the title of the blog entry you just wrote. The blog entry will be loaded into your blog entry editor. Click Delete, in an attempt to delete the post. Clicking Delete should give you an error message similar to what you see following.

```
Exception: exception 'PDOException' with message 'SQLSTATE[23000]:
Integrity constraint violation: 1451 Cannot delete or update a parent row: a foreign key
constraint fails
```

## Understanding Foreign Key Constraints

It is always a bit annoying to come across errors, but this one is a friendly error. It is really preventing you from undermining the integrity of your database. Table 11-1 takes a look inside my tables to see a blog entry I created through the editor form.

***Table 11-1.*** *One Row from the* `blog_entry` *Table*

| entry_id | title | entry_text | created |
|---|---|---|---|
| 17 | delete me | testing testing | 2014-03-23 10:26:18 |

I also created a comment related to that particular blog entry. Table 11-2 shows the corresponding row from the comment table.

***Table 11-2.*** *One Row from the Comment Table*

| comment_id | entry_id | author | txt | date |
|---|---|---|---|---|
| 4 | 17 | Thomas | test comment | 2014-03-23 10:26:40 |

Imagine you delete the `blog_entry` with `entry_id = 17`. You would have a comment related to a blog entry that no longer exists. Comments are only meaningful in the right context. The comment with `comment_id = 4` would have lost its context; it would have lost its integrity. You can imagine what would happen if you inserted a new blog entry with `entry_id = 17`. That blog entry would be born with a completely irrelevant comment.

The purpose of a foreign key constraint is to maintain data integrity. So, when you try to delete a `blog_entry` that has a comment, MySQL will stop you, because the delete action would leave a renegade comment floating around in your system, without a meaningful context. Only blog entries without comments can be deleted without losing data integrity.

## Deleting Comments Before Blog Entry

Once you see that *only blog entries without comments can be deleted without losing data integrity*, it is easy to see the solution: when deleting a blog entry, you should first delete any comments related to the entry. You already have a class that provides a single point of access to the comment table. You can add a new method to delete all comments related to a particular `entry_id`, as follows:

```
//partial code for models/Comment_Table.class.php

//declare new method
public function deleteByEntryId( $id ) {
    $sql = "DELETE FROM comment WHERE entry_id = ?";
    $data = array( $id );
    $statement = $this->makeStatement( $sql, $data );
}
```

This method should be called before a blog entry is deleted. Blog entries are deleted from the `Blog_Entry_Table` class. One solution would be to have the `Blog_Entry_Table` class use the `Comment_Table` class, as follows:

```
//partial code for models/Blog_Entry_Table.class.php

//edit existing method
public function deleteEntry ( $id ) {
    //new code: delete any comments before deleting entry
    $this->deleteCommentsByID( $id );
    $sql = "DELETE FROM blog_entry WHERE entry_id = ?";
    $data = array( $id );
    $statement = $this->makeStatement( $sql, $data );
}

//new code: declare a new private method inside Blog_Entry_Table.class.php
private function deleteCommentsByID( $id ) {
    include_once "models/Comment_Table.class.php";
    //create a Comment_Table object
    $comments = new Comment_Table( $this->db );
    //delete any comments before deleting entry
    $comments->deleteByEntryId( $id );
}
```

With the preceding code completed, you're ready to test. You should be able to delete any blog entry through the editor. If there are any comments related to the `blog_entry`, those comments will be deleted first, to avoid violating foreign key constraints.

# Improving Usability with WYSIWYG

WYSIWYG is an acronym for *What You See Is What You Get*. With a WYSIWYG HTML editor, an administrator with little or no understanding of HTML can write blog entries that are marked up programmatically as HTML. A WYSIWYG editor looks a lot like any other word processing software, but it can save texts as HTML, rather than as `.doc` or `.odf` files.

## Integrating TinyMCE

TinyMCE is a popular open source WYSIWYG editor. It is integrated in many popular CMS systems, such as Drupal, Joomla, and WordPress. Soon, it will be integrated into your entry editor, and it will be a wonderful improvement.

It is quite easy to integrate TinyMCE. The first step is to download it from www.tinymce.com/download/download.php. Unzip the downloaded file and save a copy of the `tinymce` folder in your existing `js` folder. Take a look inside `js/tinymce` and check that you can find the main JavaScript file `tinymce.min.js`. That is the JavaScript file your `admin.php` should use.

The TinyMCE JavaScript and CSS files can turn your plain `<textarea>` element into a WYSIWYG editor. All you have to do is initialize TinyMCE and point it to your text area. You can do that from `views/admin/editor-html.php`, as follows:

```
//partial code for views/admin/editor-html.php
//notice the new <script> elements added at the end
return "
```

```
<form method='post' action='admin.php?page=editor' id='editor'>
    <input type='hidden' name='entry_id'
           value='$entryData->entry_id' />
    <fieldset>
        <legend>New Entry Submission</legend>
        <label>Title</label>
        <input type='text' name='title' maxlength='150'
               value='$entryData->title' />

        <label>Entry</label>
        <textarea name='entry'>$entryData->entry_text</textarea>

        <fieldset id='editor-buttons'>
            <input type='submit' name='action' value='save' />
            <input type='submit' name='action' value='delete' />
            <p id='editor-message'>$entryData->message</p>
        </fieldset>

    </fieldset>
</form>
<script type='text/javascript' src='js/tinymce/tinymce.min.js'> </script>
<script type='text/javascript'>
tinymce.init({
    selector: 'textarea',
    plugins: 'image'
 });
</script>";
```

Take a look at those two `<script>` elements. The first embeds the main TinyMCE JavaScript file. The second initializes TinyMCE and configures it with a couple of parameters.

The `selector` parameter indicates that TinyMCE should convert every `<textarea>` element on the page to fancy WYSIWYG editors. On this particular page, there is only one `<textarea>` element.

The `plugins` parameter indicates which TinyMCE plug-ins should be activated for the editor. The image plug-in will allow users to insert images in blog entries.

Most of the time, you shouldn't scatter small `<script>` elements throughout your generated HTML. The more `<script>` elements you keep in your code, the harder it may be for you to track down JavaScript errors. But in this case, I'll accept the discomfort, because I want the TinyMCE editor only on this particular page. I specifically do not want TinyMCE to take control over any other `<textarea>` elements in my blogging system. To see what TinyMCE looks like, see Figure 11-1. You can also navigate your browser to `http://localhost/blog/admin.php?page=editor`.

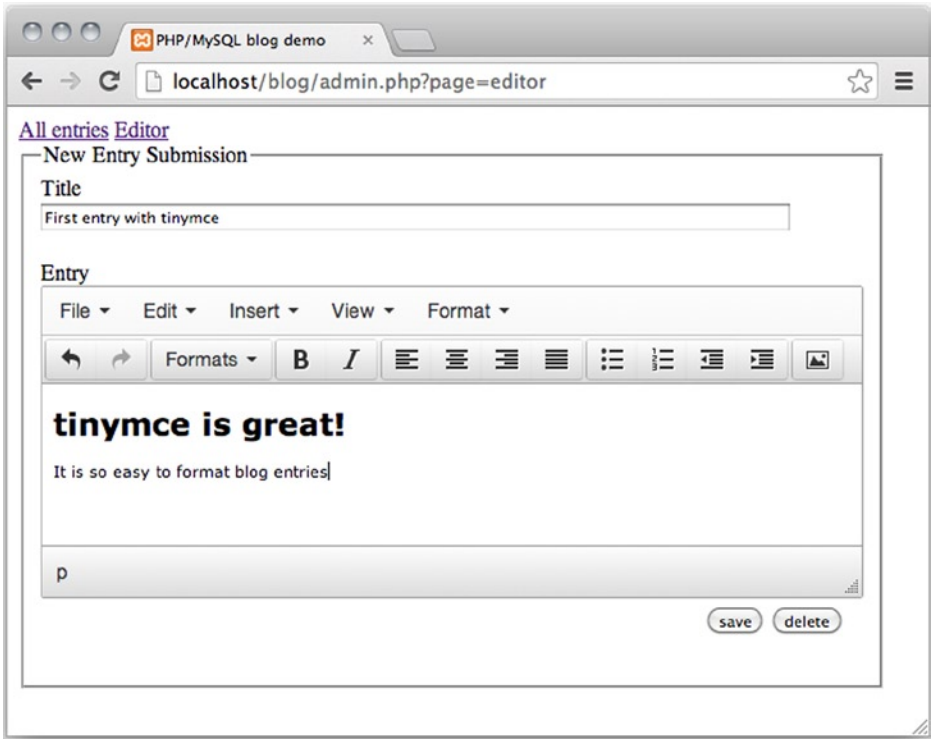


***Figure 11-1.*** *The TinyMCE WYSIWYG editor*

In Figure 11-1, I have changed my CSS a little to make the editor `<form>` 625 pixels wide. I made the change in `css/blog.css`.

# Creating an Image Manager

The `admin` module has two different page views at this point: the list of entries and the entry editor. You can create a third page for uploading and deleting the images that you'll be using in your blog entries. You can start by creating the menu item for the image manager. Update the code in `views/admin/admin-navigation.php`, as follows:

```
<?php
//complete code for views/admin/admin-navigation.php

//notice item added for image manager
return "
<nav id='admin-navigation'>
    <a href='admin.php?page=entries'>All entries</a>
    <a href='admin.php?page=editor'>Editor</a>
    <a href='admin.php?page=images'>Image manager</a>
</nav>";
```

See the `href` for the Image manager menu item. Clicking the item will encode a URL variable named `page` and set its value to `images`. Because of the way you have coded your front controller in `admin.php`, you can probably guess the next step: you need a new controller script called `images.php`. Also, it is important that it be saved in `controllers/admin`. That way, the controller will be loaded automatically from `admin.php` whenever the menu item Image manager is clicked. As always, start with a tiny step to catch errors while they're still easy to correct. Create a new file in `controllers/admin/images.php`.

```
<?php
//complete code for controllers/admin/images.php
$imageManagerHTML = "Image manager coming soon!";
return $imageManagerHTML;
```

Load `http://localhost/blog/admin.php?page=images` in your browser, and you should see the output that confirms your script is working as intended.

```
Image manager coming soon!
```

## Showing a Form for Uploading Images

Now that you have the image manager controller script set up, you can move on to creating and outputting an image manager view. Let's start with a basic HTML form, which you can eventually use to upload images. Create a new php file and save it as `views/admin/images-html.php`:

```
<?php
//complete code for views/admin/images-html.php
if ( isset( $uploadMessage ) === false ){
    $uploadMessage = "Upload a new image";
}

return "
<form method='post' action='admin.php?page=images'
      enctype='multipart/form-data'>
    <p>$uploadMessage</p>
    <input type='file' name='image-data' accept='image/jpeg' />
    <input type='submit' name='new-image' value='upload' />
</form>
";
```

You can see that the `views` code is prepared with a placeholder for displaying upload messages to users. The default upload message is *Upload a new image*. Soon, your system will let users know if an upload was successful or not. But before you get to that, I'd like to repeat the basics of uploading.

To allow users to upload files such as images, you need an HTML form. You must use the HTTP method `POST`, and the form's encoding type must explicitly be declared, to allow for file upload. By default, HTML forms are set to `application/x-www-form-urlencoded`. However, this won't work when you're uploading files. Instead, you have to set the enctype of the form to `multipart/form-data`, which will accept files *and* ordinary URL encoded form data.

An `<input>` element with `type=file` will create a *file chooser,* allowing users to browse their local computers for a file to upload. This particular file chooser has an `accept` attribute limiting the kinds of file types that can be chosen. This file chooser will only accept JPEG images. Keep in mind that client-side validation can improve usability but not security. A malicious user can be trusted to be able to work around any kind of client-side validation. To protect your

system from attacks, you must implement server-side validation. I'll give you some hints later. First, update the image manager controller, so the upload form is displayed. Change the code in `controllers/admin/images.php`:

```
<?php
//complete code for controllers/admin/images.php

$imageManagerHTML = include_once "views/admin/images-html.php";
return $imageManagerHTML;
```

Save and point your browser to `http://localhost/blog/admin.php?page=images`, to confirm that the form is, in fact, displayed in your browser.

## A Quick Refresher on the $_FILES Superglobal Array

You learned about the `$_FILES` superglobal in Chapter 4, but it might be helpful to review what it does, before moving on. Whenever a file is uploaded via an HTML form, that file is stored in temporary memory, and information about the file is passed in the `$_FILES` superglobal. You can see this in action by taking a look at what's being passed to your `images` controller. Add this code at the top of `controllers/admin/images.php`, as follows:

```
<?php
//complete code for controllers/admin/images.php
$imageSubmitted = isset( $_POST['new-image'] );

if ( $imageSubmitted ) {
    $testOutput = "<pre>";
    $testOutput .=print_r($_FILES, true);
    $testOutput .= "</pre>";
    return $testOutput;
}

$imageManagerHTML = include_once "views/admin/images-html.php";
return $imageManagerHTML;
```

You can test your code by loading `http://localhost/blog/admin.php?page=images` in your browser. Use the form to select an image to upload. Your code will not save the uploaded file on your server, but you can see from the following output that PHP gets access to the file:

```
Array (
   [image-data] => Array (
       [name] => alberte-lea.jpg
       [type] => image/jpeg
       [tmp_name] => /Applications/XAMPP/xamppfiles/temp/phprDui5l
       [error] => 0
       [size] => 119090
   )
)
```

You can see that `$_FILES` is an `Array`. The first `Array` holds another `Array` under the named index `image-data`. `$_FILES` will provide information about file name, type, `tmp_name`, `error`, and `size`, by default. But an important question you have to ask yourself is, where did the name `image-data` come from?

The answer is that you provided it! The `image-data` is there because of the name attribute you wrote for the file chooser element. If you look in your code in `views/admin/images-html.php`, you can find the place where you set the name attribute of the file chooser:

```
//one line of code from views/admin/images-html.php
<input type='file' name='image-data' accept='image/jpeg' />
```

Because you set `image-data` as the name attribute of the file chooser, PHP can find the related file data in an array nested inside `$_FILES`, under the named index `image-data`.

The nested array holds information about the image you tried to upload with this particular file chooser. You can see the original image name and its mimetype. You can see that the image data is saved temporarily on your server, under a temporary name `tmp_name`. You can see the error code `0`, which indicates that no errors occurred during upload. You can also see the `size` of the image file, measured in bytes.

If you see something similar in your browser, you are just one or two lines of code away from uploading the image. All you have to do is to save the file data on your server. The file data is already uploaded and saved temporarily under `tmp_name`, inside `image-data` array, inside `$_FILES`. To grab the file data, you would simply have to write something along the following lines:

```
//don't write this anywhere...yet
$fileData = $_FILES['image-data']['tmp_name'];
```

Note how your code accesses `$_FILES` to find the array `image-data`. Inside `image-data`, PHP finds `tmp_name`. To save the file data, you would simply move it from its temporary location and save it in a destination folder under a name.

But I propose you don't do that in `controllers/admin/images.php`, mostly because you will want to write code to deal with some of the errors that might occur during upload. Someday in the future, you will need to upload files through a form again. So, instead of reinventing a solution every time you need one, you can write a reusable class for uploading. That way, you can reuse your upload code in many projects without changing it.

Come to think of it, you already wrote an `Uploader` class back in Chapter 4. You could use that as it is. Start by simply reusing the `Uploader`. In the process, I will point out ways to improve the `Uploader` further. Make a copy of the `Uploader` class in `XAMPP/htdocs/ch4/classes/Uploader.class.php` and save the copy in `XAMPP/htdocs/blog/models/Uploader.class.php`. Alternatively, you can get the `Uploader` class, if you download the source code for Chapter 4, from the book's companion web site at www.apress.com.

## Uploading an Image

You can try to upload an image using the `Uploader` class. Begin by creating a new folder for images. I have created a folder called `img` in the root of my project. I'll be uploading images to the `img` folder.

When the image manager upload form is submitted, your code should try to upload the indicated file. Submitting a form is a user interaction, so the code belongs in a controller, in `controllers/admin/images.php`:

```
<?php
//complete code for controllers/admin/images.php
//new code: include Uploader class definition
include_once "models/Uploader.class.php";
$imageSubmitted = isset( $_POST['new-image'] );

//if the upload form was submitted
if ( $imageSubmitted ) {
    //new code below
    //create an Uploader object
    $uploader = new Uploader( 'image-data' );
```

```
    //indicate destination folder on server
    //please check that you have an img folder in your project folder
    $uploader->saveIn( "img" );
    $uploader->save();
    $uploadMessage = "file probably uploaded!";
    //end of new code
}
$imageManagerHTML = include_once "views/admin/images-html.php";
return $imageManagerHTML;
```

Everything is ready for uploading. You can test your code by pointing your browser to `http://localhost/blog/admin.php?page=images`. Now, try to upload an image through the form. Your code should be able to upload files now, so you should be able to find the image you upload in the `img` folder.

The single most common error to encounter at this point relates to folder permissions. This kind of problem is particularly common when you develop on a local web server like XAMPP. If your destination folder is write-protected, PHP cannot save the upload file. So, if you cannot upload a file, try to change the folder permissions of the `img` folder. Change permission settings to `read & write`.

## What Could Possibly Go Wrong?

A few things may go wrong when you try to upload files through a form. There are eight possible error codes related to `$_FILES`, one of which is an error code representing *No errors*. Actually, there are more than seven potential problems lurking around the upload process. It would be a shame if your entire blog system breaks down temporarily when you encounter one of these errors. Systems can be built to deal with errors without breaking down. It is time for you to learn about writing code that fails gracefully.

## Updating the Uploader Class

You can change the `Uploader` class, so that it will fail gracefully and provide meaningful error messages when it fails. To that end, you will need a property for storing error messages and another property for storing any standard PHP error codes encountered. Add a few properties in `models/Uploader.class.php`.

```
//partial code for models/Uploader.class.php
//edit existing Uploader class
class Uploader {
    private $filename;
    private $fileData;
    private $destination;
    //new code: add a property for an error message
    private $errorMessage;
    //new code: add a property for standard PHP error codes
    private $errorCode;
```

Look at the code and note that the new properties don't have any values assigned. You have to assign actual values to those properties whenever your code prepares to attempt an upload. The `$_FILES` array will provide an error code immediately. It would be an obvious choice to grab the current error code in the `Uploader`'s constructor method, because you know the constructor will run only once, just as a new `Uploader` object is created:

```
//partial code for models/Uploader.class.php

//edit existing constructor method
public function __construct( $key ) {
    $this->filename = $_FILES[$key]['name'];
    $this->fileData = $_FILES[$key]['tmp_name'];
    //new code: remember the current upload error code
    $this->errorCode = ( $_FILES[$key]['error'] );
}
```

I suspect you read most of this code back in Chapter 4, as you typed it in. I also strongly suspect that you didn't quite understand every little detail. If you are like most learners, you will have to allow your understanding of code to grow in small steps. With every small step, you will learn something, and there will always be something you haven't learned. By repeating lessons already learned, you can gradually develop a more comprehensive understanding. You can take a closer look at the `$key` argument used in the `Uploader` class. Remember that to get to the file data, you could write something like the following:

```
$fileData = $_FILES['image-data']['tmp_name'];
```

The `tmp_name` is a default name provided by `$_FILES`, and the `image-data` is a name you provide when you set the `name` attribute of the HTML file chooser input element. You want your `Uploader` class to be able to upload files regardless of the name attribute of the file chooser. It would be a terrible design decision to always depend on `image-data`. So, the design decision I have made for you is that the `Uploader` class should be given the used `name` attribute as an argument to the constructor method. It is the `$key` argument. Here's an example to give you the idea:

```
//Example: don't write any of this code anywhere
//this would upload from <input type='file' name='image-data' />
$imgUploader = new Uploader("image-data");
//this would upload from <input type='file' name='video-file' />
$vidUploader = new Uploader("video-file");
```

You can probably see that because the relevant name attribute is provided as an argument, the `Uploader` class is easier to reuse in different situations.

## Error: Restrictive Folder Permissions

When you are developing on a local web server like XAMPP, it is very common to encounter folder permissions that are too restrictive. If a folder is read-only, PHP cannot write file data for an uploaded image to the folder. It is not among the standard errors reported through the `$_FILES` error codes. But it is quite simple to test whether a

destination folder is writable or not. You already did it back in Chapter 4, but I'd like to change the code a little. You can begin by declaring a new method in `Uploader` to check for the problem, as follows:

```
//partial code for models/Uploader.class.php
//declare a new private method in the Uploader class
private function readyToUpload(){
    $folderIsWriteAble = is_writable( $this->destination );
    if( $folderIsWriteAble === false ){
        //provide a meaningful error message
        $this->errorMessage = "Error: destination folder is ";
        $this->errorMessage .= "not writable, change permissions";
        //indicate that code is NOT ready to upload file
        $canUpload = false;
    } else {
        //assume no other errors - indicate we're ready to upload
        $canUpload = true;
    }
    return $canUpload ;
}
```

As always, methods will not run until they are explicitly called. So, you should call this new method, just as the code tries to save the upload file. You should completely rewrite the existing method `save()` in `models/Uploader.class.php`, as follows:

```
//partial code for models/Uploader.class.php

// rewrite existing method save() completely
public function save () {
    //call the new method to look for upload errors
    //if it returns TRUE, save the uploaded file
    if ( $this->readyToUpload() ) {
        move_uploaded_file(
             $this->fileData,
            "$this->destination/$this->filename" );
    } else {
        //if not create an exception - pass error message as argument
        $exc = new Exception( $this->errorMessage );
        //throw the exception
        throw $exc;
    }
}
```

The native PHP function `move_uploaded_file()` saves the uploaded file data in a new destination on the server. It is a sensitive process, which your code should only attempt if there are no potential errors waiting to happen.

Remember how you already used a `try-catch` statement when you established a connection to a database with PDO? I suspect you found it a little abstract to try *something that could go wrong* and then *catch* any exceptions. What is an exception anyway? Well, an exception is a PHP object made from PHP's native `Exception` class. PHP can throw an exception when something that can go wrong does go wrong.

Metaphorically, you can think of an exception as a letter with bad news. Throwing the exception is like the act of sending the letter on its way. To read the bad news, you have to receive the letter. Exceptions are like that, but they are *caught*, not delivered to a mailbox.

Throwing an exception is very similar to triggering an error, but there is a notable difference: exceptions can easily be caught, so your code can continue to run, and your code can work around the problem gracefully. When you call a method that may throw an exception, you can wrap the method call in a `try-catch` statement and, thus, code a system that fails gracefully.

---

■ **Note** It is also possible to intercept ordinary PHP errors. Catching exceptions is the object-oriented way.

---

You're nearly there. Your `Uploader` class checks whether an upload file is too big, and if it is, an exception is thrown. To create an image manager that fails gracefully, you have to write code to catch any exceptions that might be thrown. You can do this from `controllers/admin/images.php`, as follows:

```
//partial code for controllers/admin/images.php

//edit existing if-statement
if ( $imageSubmitted ) {
    $uploader = new Uploader( 'image-data' );
    $uploader->saveIn( "img" );
    //try to save the upload file
    try {
        $uploader->save();
        //create an upload message that confirms succesful upload
        $uploadMessage = "file uploaded!";
    //catch any exception thrown
    } catch ( Exception $exception ) {
        //use the exception to create an upload message
        $uploadMessage = $exception->getMessage();
    }
}
```

It's time to test, but first, you should make sure there is a problem. You can change the folder permission for your `img` folder. Make the folder `Read-only`. Now, load `http://localhost/blog/admin.php?page=images` in your browser and try to upload an image file through your form.

The image should not be uploaded, and you should not see any standard PHP error message. You should see a message dispatched from your PHP script.

---

```
Error: destination folder is not writable, change permissions
```

---

Once you have confirmed that your code spots the folder permission problem and provides an error message, you can change folder permissions back to `Read & Write`, so that you can upload images again.

## Error: Upload File Too Big

You have managed to build an image manager that fails gracefully, but only if the destination folder is not writable. There are many other potential problems lurking around the file upload process. One is that PHP is configured with a maximum file upload size. Because there is a maximum file size for uploads, some users may try to upload a file that is too big. Users are bound to break anything that can be broken. You system should provide a meaningful error message for such users.

Testing for one upload error involved a fair bit of code changes in a few different files. Testing for one more upload error is very easy. Simply add an extra `else-if` block in the private `readyToUpload` method:

```
//partial code for models/Uploader.class.php

//edit existing method in the Uploader class
private function readyToUpload(){
    $folderIsWriteAble = is_writable( $this->destination );
    if( $folderIsWriteAble === false ){
        $this->errorMessage = "Error: destination folder is ";
        $this->errorMessage .= "not writable, change permissions";
        $canUpload = false;
    //new code: add an else-if code block to test for error code 1
    } else if ( $this->errorCode === 1 ) {
        $maxSize = ini_get( 'upload_max_filesize' );
        $this->errorMessage = "Error: File is too big. ";
        $this->errorMessage .= "Max file size is $maxSize";
        $canUpload  = false;
        //end of new code
    } else {
        //assume there are no other errors
        //indicate that we're ready to upload
        $canUpload = true;
    }
    return $canUpload ;
}
```

Now, your code is ready for testing. You should have a system that fails gracefully, if there is a problem uploading a file that is too big. To test your work, you should try to upload a file bigger than the maximum upload file size. *Uhm, what is my maximum upload file size?* I can almost hear your question. I am so glad you asked, because that will give me an excuse to write a little more about PHP configurations.

To see how your PHP is configured, you can create a new PHP file, for example, in your blog folder. Call it `test.php`:

```
<?
//complete source code for test.php
phpinfo();
```

Save the file and load `http://localhost/blog/test.php` in your browser. You can see a lot of information about your PHP installation—I mean, *a lot!*

I found that my installation has an `upload_max_filesize` of 128MB. This setting declares the upper limit for files to upload.

To test whether my `Uploader` class will provide an error message when a user tries to upload too big a file, I could try to upload a JPEG image bigger than 128MB! I don't have JPEG images anywhere near that size. I will change the `upload_max_filesize` instead.

## Configuring via ini.php or .htaccess

You have a couple of options when trying to configure PHP. If you host your own server, you can change the `upload_max_filesize` directly in the PHP configuration file. It is called `ini.php`, and the output from `phpinfo()` can tell you where that `ini.php` is saved on your computer, if you look under `Loaded Configuration File`. You can open `ini.php` with Komodo Edit, or any other text editor, and manually change the `upload_max_filesize`. Once you have saved your changes, you should restart your Apache server.

When you publish your PHP project on the Internet, you will often use a shared hosting solution. You will buy a domain through some web hosting service provider. It is cheap and convenient, but you probably won't get access to the `ini.php` file. Most web hosts allow you to make some configurations through a so-called `.htaccess` file, and you might as well get acquainted with `.htaccess` right away. Luckily, changing the `upload_max_filesize` is a very simple task.

Create a new file with Komodo Edit, or whatever editor you use. Save the new file as `.htaccess` and save it in the `blog` folder. Note that the complete file name should not be `.htaccess.txt` or `.htaccess.php`. The file name must only be `.htaccess`. Here's the complete contents of the `.htaccess` file:

```
php_value upload_max_filesize 1M
```

Once you have saved that, you can reload `http://localhost/blog/test.php` in your browser. You should see that the `local values for upload_max_filesize` is now 1M (meaning 1MB). The `Master Values` remain 128M, but for your blog folder and sub-folders, it is now changed to 1M.

---

■ **Note** You can learn more about what you can do with `.htaccess` at http://htaccess-guide.com. It is quite common to use `.htaccess` to restrict access to folders. Another common task is to rewrite URLs dynamically, to provide search-engine friendly URLs.

---

Now, it'll be easier to test your image manager's upload form. It shouldn't be difficult to find a JPEG bigger than 1MB. Try to upload such a file through your form, and you will find that the file will not be uploaded. You should see an error message telling you that the upload file was too big. Now that you have tested your system and confirmed it fails gracefully, you can change the `upload_max_filesize` back to whatever you think is appropriate for your blog.

## Detecting Other Errors

You know the `$_FILES` superglobal natively supports seven different errors. You're only testing for one of those: the `Upload file is too big`.

---

■ **Note** You can learn more about the error codes and what they mean at
www.php.net/manual/en/features.file-upload.errors.php.

---

I'm not going to go into detail about the other kinds of errors you may experience when uploading files. But I will show you how you can provide a default error message, in case any of the remaining six errors should occur. It is as easy as you might think. You simply add an extra `else-if` code block in your `readyToUpload()` method, as follows:

```
//partial code for models/Uploader.class.php

//edit existing function
private function readyToUpload(){
    $folderIsWriteAble = is_writable( $this->destination );
    if( $folderIsWriteAble === false ){
        $this->errorMessage = "Error: destination folder is ";
        $this->errorMessage .= "not writable, change permissions";
        $canUpload = false;
```

```
    } else if ( $this->errorCode === 1 ) {
        $maxSize = ini_get( 'upload_max_filesize' );
        $this->errorMessage = "Error: File is too big. ";
        $this->errorMessage .= "Max file size is $maxSize";
        $canUpload = false;

    //new code: add code block for other errors
    //the error codes have values 1 to 8
    //we already check for code 1
    //if error code is greater than one, some other error occurred
    } else if ( $this->errorCode > 1 ) {
        $this->errorMessage = "Something went wrong! ";
        $this->errorMessage .= "Error code: $this->errorCode";
        $canUpload = false;
    //end of new code
    //if error code is none of the above
    //we can safely assume no error occurred
    } else {
        //indicate that we're ready to upload
        $canUpload = true;
    }
    return $canUpload;
}
```

Excellent! Your code now checks for two common problems and provides custom error messages for these issues. At this point, your system should even provide a default error message, in case any other error should occur. You can test if such errors are caught by forcing an error. Navigate your browser to `http://localhost/blog/admin.php?page=images` and click the Upload button *without choosing a file to upload*. Your code will try to upload *nothing*, and that will cause an error 4: no file uploaded.

If you see the error message, you have confirmed that your system provides a default error message when one of the typical file upload errors occurs.

## Further Refinements

You should be able to improve the error messages further. You could simply check for the remaining error codes and provide custom error messages for them. It would be a wonderful learning exercise for you in several ways.

- You would have to learn the meaning of the remaining error codes.
- You would have to learn how to trigger these errors.
- You would consider how to communicate meaningful error messages to users.

I hope you will take the time to do it. There are also other improvements you could consider to implement on your own. You could look into an unfortunate feature of the image upload. Imagine you have uploaded an image `test.jpg`. Now try to upload a different image also named `test.jpg`. The result will be that the first image is overwritten by the next image.

It would be a significant improvement if you could check for name conflicts before upload. Perhaps your system should throw an exception and prompt the user to rename the image before upload? Or perhaps you could even change the upload form to allow users to rename the image through the form?

Another significant improvement could be to throw an exception if a user tries to upload a file that is not of type `image/jpeg`. You already have client-side validation, but a malicious user could easily work around this. How about implementing server-side validation for file type? You would need your code to compare the mimetype of the upload file with one or more accepted mimetypes. PHP can find the mimetype of the uploaded file in the `$_FILES` superglobal.

# Displaying Images

You have an image manager that allows image upload, and you have integrated the TinyMCE rich text editor in your entry editor. It should be possible for a blog administrator to see all available images and use any one of those in a blog entry.

To embed an image in a blog entry with TinyMCE, you have to know the path to the image file. It is time to update the image manager, so it shows thumbnails for all uploaded images and also the path to each image. That way, a blog administrator could copy the path relatively easily to a given image and paste it into TinyMCE to use an image in a blog entry. Listing all images would be very, very similar to the code you wrote for the image gallery in Chapter 4.

You could create a `<dl>` element to list all images. You could use a `DirectoryIterator` object to iterate through the `img` folder and find all JPEG images. Each image should be displayed as an `<img>` element. Immediately below the image you should display the path to the image. It would be nice if images could be deleted again. So, you might as well provide a link for deleting each image. Update the code in `views/admin/images-html.php`, as follows:

```
<?php
//complete source code for views/admin/images-html.php
if ( isset( $uploadMessage ) === false ){
    $uploadMessage = "Upload a new image";
}

//new code starts here
//declare a variable to hold HTML for all your images
$imagesAsHTML = "<h1>Images</h1>";
$imagesAsHTML .= "<dl id='images'>";
$folder = "img";
$filesInFolder = new DirectoryIterator( $folder);
//loop through all files in img folder
while ( $filesInFolder->valid() ) {
    $file = $filesInFolder->current();
    $filename = $file->getFilename();
    $src = "$folder/$filename";
    $fileInfo = new Finfo( FILEINFO_MIME_TYPE );
    $mimeType = $fileInfo->file( $src );
    //if file is a jpg...
    if ( $mimeType === 'image/jpeg' ) {
        //display image and its src
        $href = "admin.php?page=images&amp;delete-image=$src";
        $imagesAsHTML .= "<dt><img src='$src' /></dt>
                          </dd>Source: $src  <a href='$href'>delete</a></dd>";
    }
    $filesInFolder->next();
}
```

```
$imagesAsHTML .= "</dl>";

//notice that $imagesAsHTML is added at the end of the returned HTML
return "
<form method='post' action='admin.php?page=images' enctype='multipart/form-data'>
    <p>$uploadMessage</p>
    <input type='file' name='image-data' accept='image/jpeg' />
    <input type='submit' name='new-image' value='upload' />
</form>
<div>
    $imagesAsHTML
</div>";
//end of changes
```

Take a look at `http://localhost/blog/admin.php?page=images` to see how all images in the `img` folder are displayed with a path to each image and a delete link for each image.

## Deleting Images

The image manager lists all images in their original size at this point. It is not very pretty, but it is good enough for you to use the image manager to browse images. You could easily write some CSS to improve the aesthetics. You could even reuse some of the JavaScript ideas from the image gallery in Chapter 5. Please consider doing that, when you're done with this chapter. I have another few functional enhancements I'd like you to work on before you get started with the aesthetics. First of all: deleting images.

Clicking a link to delete a file is a user interaction. So, the code to respond to the click belongs in a controller. One might argue that the code for actually deleting a file belongs in a model script. But it is just one line of code, so I'll keep it all in the controller, in `controllers/admin/images.php`:

```
<?php
//complete code for controllers/admin/images.php
include_once "models/Uploader.class.php";

$imageSubmitted = isset( $_POST['new-image'] );
if ( $imageSubmitted ) {
    $uploader = new Uploader( 'image-data' );
    $uploader->saveIn( "img" );
    try{
        $uploader->save();
        $uploadMessage = "file uploaded!";
    } catch ( Exception $exception )  {
        $uploadMessage = $exception->getMessage();
    }
}
```

```
//new code starts here: if a delete link was clicked...
$deleteImage = isset( $_GET['delete-image'] );
if ( $deleteImage ) {
    //grab the src of the image to delete
    $whichImage = $_GET['delete-image'];
    unlink($whichImage);
}
//end of new code

$imageManagerHTML = include_once "views/admin/images-html.php";
return $imageManagerHTML;
```

You should be able to delete images with your image manager now. Try it first, and then you can read on, to learn how it happens.

The native PHP function to delete a file is `unlink`. To delete a file, you pass the path and name of the file to `unlink()` as an argument. It is important for you to understand how PHP found the path and name of the image. If you look in the code, you can see that PHP grabs the source of the image to delete from a URL variable `delete-image`, which is retrieved with the `$_GET` method. The next question to ponder is, how did the source of the image get encoded as a URL variable?

You could view the HTML source code of the image manager in your browser. You would see that every *delete* link encodes a URL variable holding an image source. You can see it, if you take a closer look at the `href` attributes. They'll be something like the following:

```
<a href='admin.php?page=images&amp;delete-image=img/coffee.jpg'>
    delete
</a>
```

See how a URL variable `page` is set to `images`, and another URL variable, `delete-image`, is set to `img/coffee.jpg`. So, clicking this delete link will load the `images` controller and delete the image found in `img/coffee.jpg`.

A last question to consider is, where in your PHP code did you create the delete links? You have a choice: either you think about the question and look through the code to find your own answer, or you continue reading to see my answer.

Delete links are created in `views/admin/images-html.php`. Inside the `while` loop, you can find the PHP that creates the HTML for showing one image, its source, and a delete link:

```
//partial code for views/admin/images-html.php
//make no changes to any code
$file = $filesInFolder->current();
$filename = $file->getFilename();
//$src holds the relative path to the file
$src = "$folder/$filename";
$fileInfo = new Finfo( FILEINFO_MIME_TYPE );
$mimeType = $fileInfo->file( $src );

if ( $mimeType === 'image/jpeg' ) {
    //href for delete link created below
    $href = "admin.php?page=images&amp;delete-image=$src";
    $imagesAsHTML .= "<dt><img src='$src' /></dt>
                <dd>Source: $src
                <a href='$href'>delete</a>
                </dd>";
```

There you have it. The variable `$src` holds the relative path to a file PHP, found by looping through a folder with a `DirectoryIterator` object. If the found file is a JPEG image, it will be displayed as HTML. Notice how the `$src` is used to create an `href` attribute for the delete link.

# Using an Image in a Blog Entry

With all your images listed, it should be an uncomplicated task to embed an image in one of your blog entries. Find an image you would like use by looking though all your images, at `http://localhost/blog/admin.php?page=images`. Note how the *Source* of each image is displayed in your browser. Select and copy the Source of some image you would like to use in a blog post. Now, load an existing blog entry into your blog editor. Note how TinyMCE provides a button for inserting images. Click the Image button, to bring up TinyMCE's image dialog pop-up, as shown in Figure 11-2. Embed the image by pasting the image `src` into the Source field.

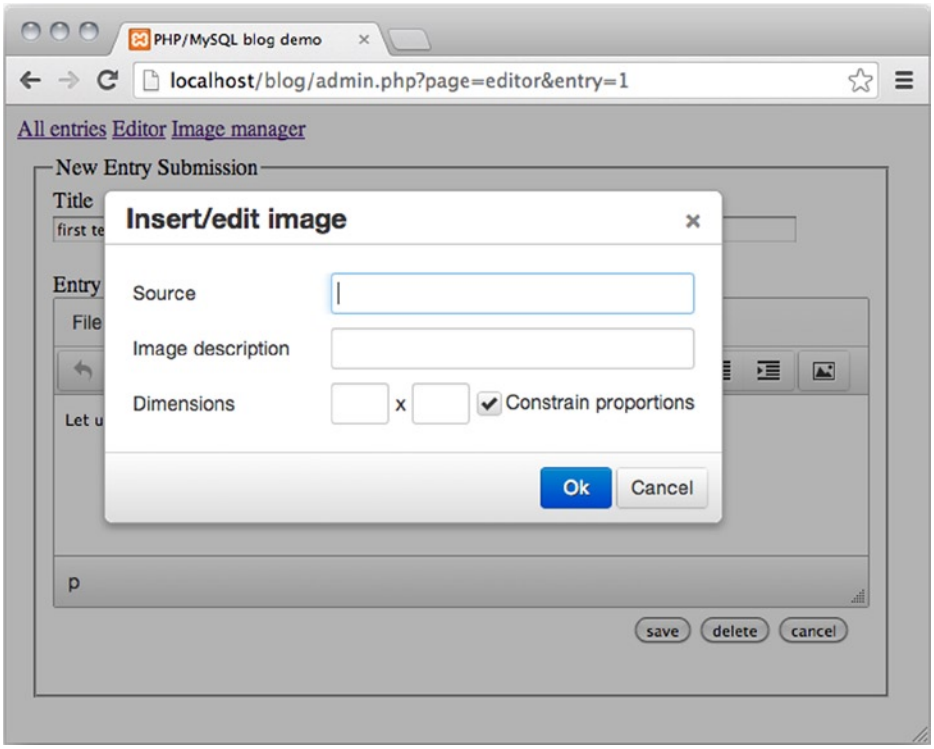


***Figure 11-2.*** *The image dialog pop-up in TinyMCE*

Once you have embedded an image in a blog entry, you should save it and then load `http://localhost/blog/index.php` in your browser, to check that the image is, in fact, displayed in the blog entry.

## Improving Editor Usability

You have a marvelous administration module for your blog now. Through the administration module, you can create, update, and delete blog entries. No more features will be added under my guidance.

The entry editor has a usability flaw I would like to point out and improve. Load an existing entry into the entry editor, change the entry a little, and save the changes. This should reload the saved changes in your entry editor, and you should see a little message indicating that *Entry was saved*, telling you clearly that what you see in the editor is saved in the database.

But if you change the entry a bit more, the message will still insist that *Entry was saved*, even though you and I know it is misleading. Changes are not saved until you click the Save button. That is bad usability! Misleading feedback from a system can be even worse than no feedback at all. How can you remedy the problem? Is this a task that calls for PHP or JavaScript? Take a minute to reflect before you read on.

It is a task that absolutely requires JavaScript! PHP is a server-side scripting language. It runs once every time a PHP script on the server is contacted. PHP runs once for every HTTP request, if the request looks for a PHP resource. Your entry editor is simply an HTML form with a `<textarea>` element enhanced with a bit of JavaScript and CSS. PHP will only run whenever you submit the form. Typing a few characters into the `<textarea>` will not submit the form.

JavaScript is a client-side scripting language. It runs perfectly well in the browser. You have already seen how JavaScript functions can be called when certain browser events occur. So far, you have learned how JavaScript functions can be called when the HTML DOM content is loaded into the browser, when a user clicks, and when a form is submitted. JavaScript can also respond whenever a user presses a key on the keyboard. You can use that event here. If a user presses a key while the `<input name='title'>` is in focus, you know that the title was changed. If the title was changed, and the form has not been submitted, you know that changes were not saved in the database yet. Open your `editor.js` JavaScript file and declare a new function for updating the update message. While you're at it, add an event listener, to call the new function every time the user changes the title, as follows:

```
//partial code for js/editor.js

//new code: declare a new function
function updateEditorMessage () {
   console.log( "editor changes not saved yet!" );
}

//edit existing function
function init(){
    var editorForm = document.querySelector("form#editor");
    var title = document.querySelector("input[name='title']");
    title.required = false;

    //code changes start here
    title.addEventListener("keyup", updateEditorMessage, false);
    //end of changes

    editorForm.addEventListener("submit", checkTitle, false);
}
```

The `updateEditorMessage()` function doesn't do anything meaningful yet. It simply outputs a message to the console. But you can use it, to test your progress. Open your browser and its JavaScript console. If you're using Chrome, you can open the console with Cmd+Alt+J, if you are working on a Mac, or Ctrl+Alt+J, if you are using a Windows machine.

Once the browser's JavaScript console is open, you can navigate your browser to `http://localhost/blog/admin.php?page=entries`. Click an entry to load it in your blog entry editor. Edit the entry a little and save changes. Note the usual *Entry was saved* message created by PHP. Now, make a little change in the entry title, and note the output in your console: JavaScript noticed that something happened. Note also that the *Entry was saved* message has now become misleading (see Figure 11-3): The most recent changes in the entry were *not* saved!

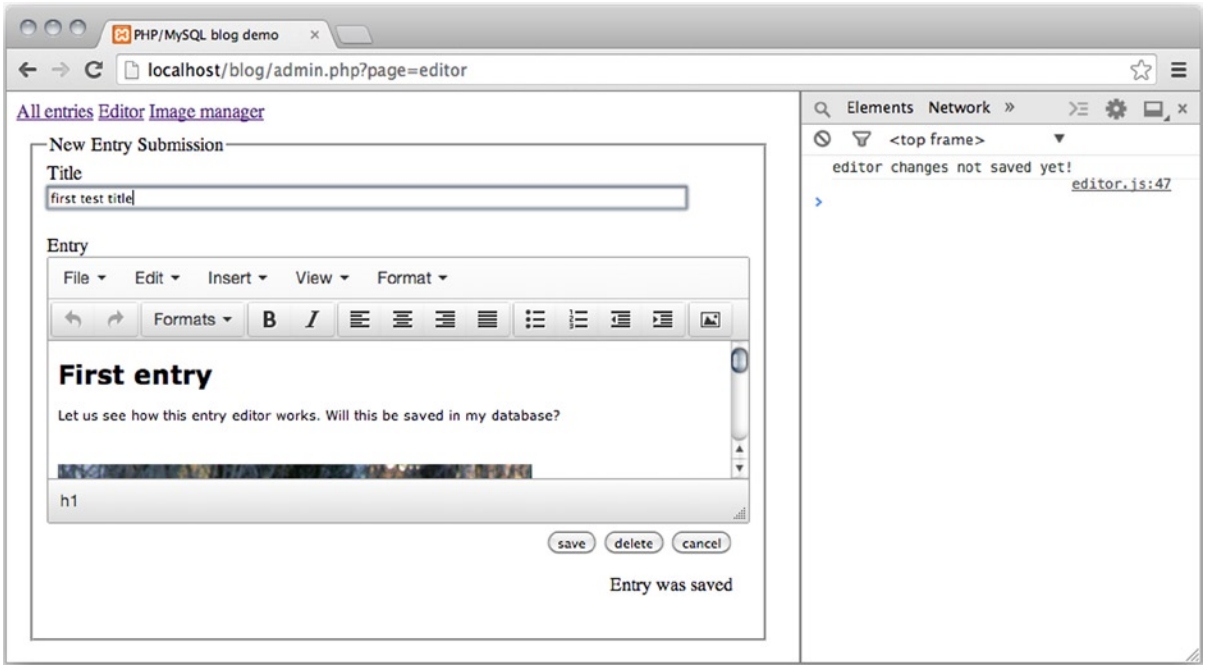


***Figure 11-3.*** *A message in Chrome's console, and a misleading editor message*

---

■ **Note** If you don't see the expected output in the console, it may be because Google Chrome keeps a cached version of your previous JavaScript file. In other words, Google Chrome hasn't noticed that you have updated your JavaScript. You simply have to open the "Clear browsing data" dialog box (Mac: Cmd+Shift+Delete, Windows: Ctrl+Shift+Delete), to clear "Cached images and files."

---

It is great to see that JavaScript noticed something happened in the editor. JavaScript knows that the changes have not been saved in the database yet. JavaScript knows that the editor message *Entry was saved* is no longer valid, but actually misleading. Now that JavaScript has detected a change in the entry editor's title field, it is time to change feedback for users. It takes another couple of lines of JavaScript in `js/editor.js`:

```
//partial code for js/editor.js

//edit existing function
function updateEditorMessage() {
    var p = document.querySelector("#editor-message");
    p.innerHTML = "Changes not saved!";
}
```

That's it! Whenever there is an unsaved change in the <title>, users will be notified. It will be quite simple to add an event listener to call updateEditorMessage() whenever the <textarea> is changed also. You might think you could add an extra event listener in the init function—something like this:

```
//partial code for js/editor.js

//edit existing function
function init(){
    var editorForm = document.querySelector("form#editor");
    var title = document.querySelector("input[name='title']");
    title.required = false;

    //changes start here
    //sadly, this code will not solve the problem...
    var textarea = document.querySelector("form textarea");
    textarea.addEventListener("keyup", updateEditorMessage, false);
    //end of changes

    title.addEventListener("keyup", updateEditorMessage, false);
    editorForm.addEventListener("submit", checkTitle, false);
}
```

It may surprise you to see that the preceding code doesn't update the editor message whenever there is a change in the <textarea>. The problem is that your JavaScript code conflicts with the TinyMCE JavaScript. It could seem you're out of luck. Luckily, TinyMCE can be configured. The TinyMCE developers have even taken the effort to document configuration options. A bit of searching in the TinyMCE documentation told me that the TinyMCE JavaScript can be brought to call your updateEditorMessage() whenever the TinyMCE editor changes. Open views/admin/editor-html.php in Komodo Edit and change the code a little, as follows:

```
//partial code for views/admin/editor-html.php
<script type='text/javascript'>
//change the existing call to tinymce.init
//add the code to call your updateEditorMessage function...
//...whenever the tinymce editor changes

tinymce.init ({
    selector: 'textarea',
    plugins: 'image',
    setup: function (editor) {
        editor.on ('change', function (e) {
            updateEditorMessage();
        });
    }
});
</script>
```

That code will call the function `updateEditorMessage()` whenever the TinyMCE editor is changed in any way. That was exactly what you were after.

---

■ **Note** Learn about configuring TinyMCE at `www.tinymce.com/wiki.php/Configuration`.

---

It's time for the final test. Load an existing entry into the entry editor, change it a little, and save it. Now, you should see an editor message stating that *Entry was saved*. Make another little change to the entry and see how the editor message immediately changes into *Changes not saved*! This is the work of your JavaScript function `updateEditorMessage()`, which was called by way of a TinyMCE `change` event, triggered by any change in the TinyMCE editor.

One thing to bear in mind is that TinyMCE relies on JavaScript. The entry editor isn't very impressive without JavaScript enabled. Any blog administrators will have to accept that they need a modern browser with JavaScript enabled.

The next chapter shows you how to restrict access to the administration module. You will create a login form, so that only users with a correct username and password will be allowed to use the administration module.

# Summary

This chapter has led to some great improvements to the administration module. In the process, you have had another opportunity to come to terms with PHP file uploads and how to delete files with PHP.

You have had a closer look at `try-catch` statements used with custom PHP exceptions. You have seen how you can use `try-catch`, so that your PHP code can fail gracefully. You have used exceptions to provide meaningful error messages to users. You should know that some developers would object to this use of exceptions. Some developers would say that any error you can predict isn't really an exception. Exceptions should *not* be used for these cases. The final decision is yours: At least now, you have the choice.

I hope your work with TinyMCE has offered you an appetizer. There are many solid, well-documented JavaScripts to be found online, and integrating such JavaScripts into your projects can greatly improve the quality of your work. On the other hand, you should approach with care libraries and frameworks that you don't fully understand. If you deliver a project to a client and subsequently identify a bug in the system, you may have a very hard time fixing the bug if you don't understand the code used. As was the case with the exceptions, it all comes down to making an informed decision.

CHAPTER 12

# Password Protection

One of the last things you need to do before you can call your blog "web-ready" is to hide the administration module from users who aren't authorized to see it. In this chapter, you'll learn how to build a system that lets you create new administrators and require administrators to log in with a password before they can create, edit, and delete entries on the blog. Creating this system requires that you perform the following tasks:

- Create an admin table in the `simple_blog` database
- Use one-way encryption of passwords
- Create an HTML form for creating new administrators
- Insert one or more administrators into the admin table
- Create a login form for administrators
- Hide administration module from unauthorized users
- Use sessions to persist a login state across multiple HTTP requests

## Creating an admin_table in the Database

Enabling authorized administrators for your site requires that you create a table to store their data. I have created an admin table to store the following information about administrators:

- `admin_id`: a unique number identifying one administrator
- `email`: the administrator's e-mail address
- `password`: the administrator's password

You will encrypt the password with the so-called MD5 algorithm into a string of 32 characters. Consequently, the password column can use the `VARCHAR` data type and limit input to 32 characters.

To create the admin table, navigate to `http://localhost/phpmyadmin` in a browser, select your `simple_blog` database and open the SQL tab. Enter the following command to create your table:

```
create table admin(
    admin_id INT NOT NULL AUTO_INCREMENT,
    email TEXT,
    password VARCHAR(32),
    primary key (admin_id)
)
```

## Encrypting Passwords with MD5

Once you get around to inserting new users into your admin table, you will see how to encrypt a string with the MD5 algorithm. The MD5 algorithm provides so-called one-way encryption.

Encrypting passwords is a good practice, because an encrypted password is harder to steal. Let's imagine your password is *test*. Run it through MD5, and *test* becomes *098f6bcd4621d373cade4e832627b4f6*. No matter how long or how short your password is, it becomes a 32-character string, once it has been through MD5.

Even if your database suffers a serious attack, and all usernames and passwords are exposed, attackers are still unable to misuse usernames and passwords.

## One-Way Encryption

Passwords are often protected through so-called one-way encryption. If you encrypt a password with a one-way encryption algorithm, it should be practically impossible to decrypt. So, if you have an encrypted password such as *098f6bcd4621d373cade4e832627b4f6*, it should be practically impossible to figure out that the unencrypted password was *test.* This means that even if your database is compromised, passwords are still protected.

Any system that can recover a lost password is inherently unsafe. If your original password can be recovered, the system must somehow remember your password. If your password is remembered by the system, it is likely that system administrators can see your password. If your password can be seen, it is vulnerable. A secure system would only store encrypted passwords. If you should forget your password, a secure system would provide you with an opportunity to reset your password. With one-way encryption, sensitive data such as passwords can be kept hidden from system administrators.

In the context of your blogging system, it means that administrator passwords are protected from you or other blog administrators. If a user loses her password, you cannot send it to her again. You can offer her a chance to reset her password, but the password is completely safe—as it should be.

## Sufficient Security

Assume that even the most secure systems can be hacked. The question is how much time and effort is required from the attacker. Your task is to provide enough security to discourage attackers from your content. If your content is very valuable, you need very serious security. Most personal blogs are probably not particularly attractive targets for IT criminals.

MD5 is a one-way encryption algorithm. It is often used in PHP, because it is very easy to use in PHP applications. By itself, MD5 is not sufficient to create optimal password security. Despite all its merits, MD5 has been compromised. But for our purposes, it will suffice. You have already taken measures against SQL injection attacks. Soon, you will also have encrypted passwords, so you have probably deterred any attackers from your blogging system.

---

■ **Note** There is another common type of attack known as a cross-site scripting (XSS) attack. You could consider doing a bit of Internet research about preventing XSS attacks on PHP sites.

---

You should know that there is more to learn about building secure PHP applications. You cannot expect MD5 to keep attackers at bay, if you develop a system that holds valuable data. You might consider consulting Chris Snyder's *Pro PHP Security* (Apress, 2010), to learn much more about this topic.

## Adding Administrators in the Database

You have a database table to save administrator credentials; you're ready to start creating blog administrators. Your first step is to create a form that allows you to enter an e-mail address and a corresponding password. Once you accomplish this, you must store the information in the database for later use.

# Building an HTML Form

As you have already learned, it is best to create forms that provide feedback to users. You might as well prepare the form for user feedback right away. Create a new file in `views/admin/new-admin-form-html.php`. Here's what my form looks like:

```
<?php
//complete code for views/admin/new-admin-form-html.php
if( isset($adminFormMessage) === false ) {
    $adminFormMessage = "";
}

return "<form method='post' action='admin.php?page=users'>
    <fieldset>
        <legend>Create new admin user</legend>
        <label>e-mail</label>
        <input type='text' name='email' required/>
        <label>password</label>
        <input type='password' name='password' required/>
        <input type='submit' value='create user' name='new-admin'/>
    </fieldset>
    <p id='admin-form-message'>$adminFormMessage</p>
</form>
";
```

To display the form, you need a controller to load it and return it to `admin.php`. You can create a new navigation item and a corresponding controller. Begin by updating your navigation with an additional link, as follows:

```
<?php
//complete code for views/admin/admin-navigation.php

//new code: notice item added for a 'Create admin user' page
return "
<nav id='admin-navigation'>
    <a href='admin.php?page=entries'>All entries</a>
    <a href='admin.php?page=editor'>Editor</a>
    <a href='admin.php?page=images'>Image manager</a>
    <a href='admin.php?page=users'>Create admin user</a>
</nav>
";
```

With the form's view and a new navigation item created, the next step could be to create a controller to load the view and have it displayed. In the previous example, I created an `href` for the new link that requests a new page called *users*. So, my system expects a controller called `controllers/admin/users.php`. Create such a file and write the following code:

```
<?php
//complete code for controllers/admin/users.php
$newAdminForm = include_once "views/admin/new-admin-form-html.php";
return $newAdminForm;
```

Save your work and load `http://localhost/blog/admin.php?page=users` in your browser. You should see an unstyled, yet valid, HTML form. You can't really use it yet: it cannot insert new administrators into the database table.

# Saving New Administrators in the Database

Inserting new rows into a database table is a job for a *model* script. You have already made good use of the table data gateway design pattern a couple of times. You have even prepared a base `Table` class, which you can extend to create a table data gateway for the admin table.

At this point, you don't need to be able to do much: you want to avoid name conflicts, so before a new admin user is created, the code should check whether the e-mail is already used in the table. If the e-mail is not in use, the model script should insert a new entry into the admin table. Create a new file, `models/Admin_Table.class.php`, and write a new class definition, as follows:

```
<?php
//complete code for models/Admin_Table.class.php
//include parent class' definition
include_once "models/Table.class.php";

class Admin_Table extends Table {

    public function create ( $email, $password ) {
        //check if e-mail is available
        $this->checkEmail( $email );
        //encrypt password with MD5
        $sql = "INSERT INTO admin ( email, password )
                VALUES( ?, MD5(?) )";
        $data= array( $email, $password );
        $this->makeStatement( $sql, $data );
    }

    private function checkEmail ($email) {
        $sql = "SELECT email FROM admin WHERE email = ?";
        $data = array( $email );
        $this->makeStatement( $sql, $data );
        $statement = $this->makeStatement( $sql, $data );
        //if a user with that e-mail is found in database
        if ( $statement->rowCount() === 1 ) {
            //throw an exception > do NOT create new admin user
            $e = new Exception("Error: '$email' already used!");
            throw $e;
        }
    }
}
```

The preceding code is just waiting to be called from a controller. The `create()` method takes two arguments for a new e-mail and the associated password. The private method `checkEmail()` is used to check whether the provided e-mail already exists in the database. If it does, an `Exception` object is created with a relevant message, and the created exception is thrown. This means you can use a `try-catch` statement in your controller. You will soon write code that will try to create a new admin user. If the operation fails, your code will fail gracefully, by catching the exception and providing relevant feedback to the user.

Take a look at the `$sql` variable in the `create()` method. See how the received password is encrypted with MD5 as it is inserted into the database? That's possible because the SQL language has a built-in MD5 function. Let's go back to the controller script in `controllers/admin/users.php` and write some code, so that input from the form will be used to insert new admin users in the database:

```
<?php
//complete code for controllers/admin/users.php
//new code starts here
include_once "models/Admin_Table.class.php";

//is form submitted?
$createNewAdmin = isset( $_POST['new-admin'] );
//if it is...
if( $createNewAdmin ) {
    //grab form input
    $newEmail = $_POST['email'];
    $newPassword = $_POST['password'];
    $adminTable = new Admin_Table($db);
    try {
        //try to create a new admin user
        $adminTable->create( $newEmail, $newPassword );
        //tell user how it went
        $adminFormMessage = "New user created for $newEmail!";
    } catch ( Exception $e ) {
        //if operation failed, tell user what went wrong
        $adminFormMessage = $e->getMessage();
    }
}
//end of new code

$newAdminForm = include_once "views/admin/new-admin-form-html.php";
return $newAdminForm;
```

Save your code and test it by running `http://localhost/blog/admin.php?page=users` in your browser. Try to create an admin user. You should get a confirmation message displayed in the form. Check in `http://localhost/phpmyadmin` to see whether the admin user was in fact inserted into the admin database table.

Once you have created a new admin user, you could try to create one more admin user using the same e-mail address. It shouldn't be allowed, and you should receive an error message in the form. To be more specific, you should see the message of the `Exception` thrown from the `checkEmail()` method of the `Admin_Table` class.

# Planning Login

The rest of this chapter focuses on restricting access to the administration module. Only authenticated users should be allowed to create, edit, and delete blog entries.

You can continue to use the MVC idea for organizing the code. You'll need two *views*: A login form and a logout form. You'll need a *controller* to handle user interactions received from these two views and a *model* to actually perform login and logout. The model should also remember a state: it should remember if the user is logged in or not.

## Creating a Login Form

Begin with a login form. You should end up with a system in which a user must provide a valid e-mail and a matching password to be allowed access to the blog administration module. So, you need a form with e-mail and password fields. Create a new file in `views/admin/login-form-html.php`:

```
<?php
//complete code for views/admin/login-form-html.php
return " <form method='post' action='admin.php'>
    <p>Login to access restricted area</p>
    <label>e-mail</label><input type='email' name='email' required />
    <label>password</label>
    <input type='password' name='password' required />
    <input type='submit' value='login' name='log-in' />
</form>";
```

You should also create a simple controller. At this point, it should only load the view and output it. Create a new file in `controllers/admin/login.php`:

```
<?php
//complete code for controllers/admin/login.php
$view = include_once "views/admin/login-form-html.php";
return $view;
```

The admin navigation does not provide a menu item for the login, and it shouldn't. You want the login form to be displayed as the default view of `admin.php`. Only when a user is authenticated and logged in should the user be allowed to see the blog administration module.

You can create a class to represent an admin user. This class should remember whether the current visitor is logged in. Initially, you can safely assume that a user is not logged in. It should be possible for users to log in and log out. You can represent the state (that is, whether the user is logged in or not) with a property. You'll need methods to log in and log out. Each method should manipulate the login state represented by a property. Create a new file in `models/Admin_User.class.php`, as follows:

```
<?php
//complete code for models/Admin_User.class.php

class Admin_User {
    private $loggedIn = false;

    public function isLoggedIn(){
        return $this->loggedIn;
    }

    public function login () {
        $this->loggedIn = true;
    }
```

```
    public function logout () {
        $this->loggedIn = false;
    }

}
```

## Hiding Controls from Unauthorized Users

It is very important that a login can actually hide parts of a system from unauthorized users. You might argue that this is the whole point of a login. So far, the administration module has been readily available to anybody visiting `admin.php`. You need to make a few changes here.

1. Create an `Admin_User` object to remember login state.
2. If a visitor is not logged in, show only the login form.
3. If a user is logged in, show the admin navigation and the administration module.

To do that, you have to make a few changes in `admin.php`. Some of the changes will involve commenting out, or deleting, existing code. But you also have to add some new lines of code, as follows:

```
<?php
//complete code for blog/admin.php
error_reporting( E_ALL );
ini_set( "display_errors", 1 );

include_once "models/Page_Data.class.php";
$pageData = new Page_Data();
$pageData->title = "PHP/MySQL blog demo";
$pageData->addCSS("css/blog.css");
$pageData->addScript("js/editor.js");
//code changes start here: comment out navigation
//$pageData->content = include_once "views/admin/admin-navigation.php";

$dbInfo = "mysql:host=localhost;dbname=techreview_blog";
$dbUser = "root";
$dbPassword = "";
$db = new PDO( $dbInfo, $dbUser, $dbPassword );
$db->setAttribute( PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION );
//code changes here: comment out some of the existing code in admin.php
//$navigationIsClicked = isset( $_GET['page'] );
//if ($navigationIsClicked ) {
//    $contrl = $_GET['page'];
//} else {
//    $contrl = "entries";
//}

//new code added below
include_once "models/Admin_User.class.php";
$admin = new Admin_User();
//load the login controller, which will show the login form
$pageData->content = include_once "controllers/admin/login.php";
```

```
//add a new if statement
//show admin module only if user is logged in
if( $admin->isLoggedIn() ) {
    $pageData->content .=include "views/admin/admin-navigation.php";
    $navigationIsClicked = isset( $_GET['page'] );
    if ($navigationIsClicked ) {
        $controller = $_GET['page'];
    } else {
        $controller = "entries";
    }
    $pathToController = "controllers/admin/$controller.php";
    $pageData->content .=include_once $pathToController;
} //end if-statement
//end of new code

$page = include_once "views/page.php";
echo $page;
```

With this code, you're ready to test your login. If you navigate your browser to `http://localhost/blog/admin.php`, you should expect to see nothing but the login form. That is as it should be, as the `loggedIn` property of the `$admin` object is `false` by default, and when a user is not logged in, the user should not be allowed into the administration area.

## Logging In a User

It is time to allow users to log in. At this point, I will not really authenticate users. I will not write code to check whether the supplied e-mail and password are valid. I will show you an approach to that later in this chapter. For now, you'll simply log in anybody who submits the login form. Dealing with such user interactions is a job for the login controller. Update your code in `controllers/admin/login.php`:

```
<?php
//complete code for controllers/admin/login.php
//new code here
$loginFormSubmitted = isset( $_POST['log-in'] );
if( $loginFormSubmitted ) {
    $admin->login();
}
//end of new code
$view = include_once "views/admin/login-form-html.php";
return $view;
```

Changes are small and should be easily accessible for you by now. Try to point your browser to `http://localhost/blog/admin.php` and log in. You can use any credentials!

It should work like a charm, and you should see the admin navigation and a clickable list of all blog entries. But wait! Click one of the entries to load it into the entry editor. What happens? Oh no! You don't see the entry editor. Instead, you find yourself looking at the login form once again. Apparently, you were instantly logged out? Why?!?

## HTTP Is Stateless

The HyperText Transfer Protocol is the foundation of much data communication on the Internet. It is *stateless*, which means it treats each new request as a separate transaction. In practical terms, it means that all PHP variables and objects are created from scratch with every new request.

That has some consequences for you. When you submit the login form, you really make an HTTP request. PHP will run, and the `$admin` object will remember that you are logged in. Next, you click an entry. This will make a new HTTP request, and thus, a new `$admin` object will be created. PHP will not remember that you just logged in, because the new HTTP request is treated as an independent, separate transaction. Whatever happened with the previous HTTP request is completely forgotten.

## Superglobals Revisited: $_SESSION

This stateless HTTP is bad news for your login. No matter how many times you log in, PHP will forget about it with every new request. It is very impractical, and of course, there is a solution: persist state across requests. You need a way to force PHP to remember that a given user is logged in. You need a PHP session.

When a PHP session is started, the visiting user's browser will be assigned a unique identification number: a session id. The server will create a small, temporary file on the server-side. Any information you require your application to remember across requests should be stored in this file. PHP will handle this temporary file for you. A PHP session has a default lifetime of 20 minutes, and duration can be configured in the `php.ini` file.

---

■ **Note** Read more at www.php.net/manual/en/intro.session.php.

---

PHP provides a superglobal to make session handling easier. You've already met a few superglobals: `$_GET`, `$_POST`, and `$_FILES`. Now, it's time for you to meet `$_SESSION`.

## Persisting State with a Session

To use a session, your code must start a session. With that in place, you can create a session variable, which is a variable whose value is *stateful*, meaning a variable whose value can persist across HTTP requests.

You already have an `Admin_User` class, which uses a normal property to remember login state. You can use a session variable instead. Here's how that can be done:

```
<?php
//complete code for models/Admin_User.class.php

class Admin_User {

    //declare a new method, a constructor
    public function __construct(){
        //start a session
        session_start();
    }

    //edit existing method
    public function isLoggedIn(){
        $sessionIsSet = isset( $_SESSION['logged_in'] );
        if ( $sessionIsSet ) {
            $out = $_SESSION['logged_in'];
        } else {
            $out = false;
        }
        return $out;
    }
```

```
    //edit existing method
    public function login () {
        //set session variable ['logged_in'] to true
        $_SESSION['logged_in'] = true;
    }

    //edit existing method
    public function logout () {
        //set session variable ['logged_in'] to false
        $_SESSION['logged_in'] = false;
    }

}
```

Save the code changes and load `http://localhost/blog/admin.php` in your browser. You can still log in with any credentials, but this time, PHP will remember that you are logged in. So, clicking a blog entry will actually load the clicked entry into the entry editor. This is great news: you can log in as an administrator, and once logged in, you can use the administration module. But all is not perfect yet. Anybody can log in, because the e-mail and password aren't really compared to database records of valid administrators. Also, logged-in users cannot log out. Let's tackle the easiest task first.

## Logging Users Out

It is customary to provide a logout option for logged-in users. It is also a good idea: you don't want administrators to stay logged in with no option for logging out. What if an administrator has to leave the computer to get a fresh cup of coffee? You wouldn't want to leave the administration module exposed. Let's create a new view for logging out. Create a new file in `views/admin/logout-form-html.php`, as follows:

```
<?php
//complete code for views/admin/logout-form-html.php
return "
<form method='post' action='admin.php'>
    <label>logged in as administrator</label>
    <input type='submit' value='log out' name='logout' />
</form>";
```

This view should be displayed whenever a user is logged in. But simply showing a logout form won't actually log out users. If a user clicks the Logout button, this should run a script to log out the user. These are tasks for the controller. You have to write some code for the login controller in `controllers/admin/login.php`:

```
<?php
//complete code for controllers/admin/login.php

$loginFormSubmitted = isset( $_POST['log-in'] );
if( $loginFormSubmitted ) {
    $admin->login();
}
```

```
//new code below
$loggingOut = isset ( $_POST['logout'] );
if ( $loggingOut ){
    $admin->logout();
}

if ( $admin->isLoggedIn() ) {
    $view = include_once "views/admin/logout-form-html.php";
} else {
    $view = include_once "views/admin/login-form-html.php";
}
//comment out the former include statement
//$view = include_once "views/admin/login-form-html.php";
//end of code changes
return $view;
```

You're making excellent progress! You should be able to log in and log out now, and PHP will remember your current state through a session variable. It is a bit of a problem that anybody can log in using any credentials. The code does not check if the supplied username and password are correct.

## Allowing Authorized Users Only

The login is nearly complete. All you have to do is check whether the supplied e-mail and password match exactly one record in the database. The information is available in the admin table, and you already have a table data gateway called `Admin_Table`. You can create a new method to check whether submitted credentials are valid.

```
//partial  code for models/admin/Admin_Table.class.php
//declare new method in Admin_Table class
public function checkCredentials ( $email, $password ){
    $sql = "SELECT email FROM admin
            WHERE email = ? AND password = MD5(?)";
    $data = array($email, $password);
    $statement = $this->makeStatement( $sql, $data );
    if ( $statement->rowCount() === 1 ) {
        $out = true;
    } else {
        $loginProblem = new Exception( "login failed!" );
        throw $loginProblem;
    }
    return $out;
}
```

See how the received password is encrypted with MD5? In the database, you have MD5-encrypted passwords. To compare a password in the database with a password received from the login form, your code will have to encrypt the received password. If you were to compare an encrypted password with an un-encrypted password, users would not be able to login, because *test* is not identical to *098f6bcd4621d373cade4e832627b4f6*.

Note how the method will create a new Exception object and throw it, if the submitted e-mail and password do not match exactly one row of data in the admin table. With this method declared, you're ready to call it from your login controller whenever a user tries to log in.

```
<?php
//complete code for controllers/admin/login.php
//new code: include the new class definition
include_once "models/Admin_Table.class.php";

$loginFormSubmitted = isset( $_POST['log-in'] );
if( $loginFormSubmitted ) {
    //code changes start here: comment out the existing login call
    //$admin->login();
    //grab submitted credentials
    $email = $_POST['email'];
    $password = $_POST['password'];
    //create an object for communicating with the database table
    $adminTable = new Admin_Table( $db );
    try {
        //try to login user
        $adminTable->checkCredentials( $email, $password );
        $admin->login();
    } catch ( Exception $e ) {
        //login failed
    }
    //end of code changes
}

$loggingOut = isset ( $_POST['logout'] );
if ( $loggingOut ){
    $admin->logout();
}
if ( $admin->isLoggedIn() ) {
    $view = include_once "views/admin/logout-form-html.php";
} else {
    $view = include_once "views/admin/login-form-html.php";
}
return $view;
```

This is a huge improvement in terms of security. With the preceding code saved, you should only be able to log in with valid user credentials. But don't take my word for it. Try to log in with bad credentials; it should be impossible.

## Exercises

You should consider a few additions related to the login. You've tried it all in previous chapters, so these additions should be great learning exercises for you.

Your login form uses a required attribute. As you have learned in Chapter 10, different browsers treat the required attribute differently. You could use JavaScript to provide a more consistent behavior across modern browsers. This would improve the usability, and it would also give you an opportunity to practice a little JavaScript.

The login form fails silently at this point. An `Exception` object is thrown when a user login fails. The exception is even caught, but the exception message is not displayed as feedback to the user. Perhaps you could grab the exception message and display it in the login form. It would be a lot like creating an `$adminFormMessage` in the beginning of this chapter.

Once you have a basic user feedback implemented, you could try something more advanced for the login form. When a user login fails, check whether the e-mail exists in the database. If it does, let the user know. *Wrong password, please try again.* If the e-mail does not exist, tell the user *The supplied e-mail does not match any record in the system, please try again*.

# Summary

This was a short chapter. With a one-way encryption for privacy and a session for stateful memory, you have managed to effectively restrict access to the administration module of your blog. I hope you will agree it was very rewarding to implement a login system. I especially enjoyed providing you with extra opportunities for creating forms that communicate with users.

Your blog is web-ready. The next chapter walks you through the process of uploading your project to an online web host, so that your blog can be published on the Internet.

CHAPTER 13

# Going Public with Your Blog

You have developed a complete, database-driven blog and even an administration module. But it is all running on your local computer only. Blogs are really about writing for an audience. Your blog will certainly have a limited audience, if it only runs on your local computer. This chapter will show you all you need to know to migrate your blog from your local development environment to an online web host.

The process of bringing your blog online on the Internet involves a few steps.

1. You'll need access to a web hosting service. It should offer an A-pache web server, MySQL, and PHP.
2. You'll need an FTP program, so that you can upload your files from your local computer to your web host.
3. You'll need to export your local database, so that you can import it into your web host's MySQL database.

## Web Host Requirements

You will need a web host for your online blog. It should support the technologies you have used: PHP 5.4 and MySQL 5.x. Your web host should also support FTP, which will allow you to upload your PHP files to the web host.

You have used phpMyAdmin to access your MySQL database. There are other programs that can provide access to MySQL. If this is your first PHP/MySQL project, you should probably look for a web host that provides phpMyAdmin. But understand that phpMyAdmin is not a fixed requirement. I recommend it, because it is the program you have already used.

Do a search for "web host" and look for a solution that suits you. There are free solutions, cheap solutions, and expensive solutions. I assume this is your first PHP/MySQL project, so I recommend you look for a web host that offers live chat support. You may run into problems along the way, and a live chat with a competent support person can solve many small problems.

## Exporting and Importing Your Database

Once you have settled on a hosting solution, you can start moving your database from your local development environment to your new web host. Moving a database involves creating an SQL file that includes all SQL statements for creating and populating your tables. Luckily, it is quite easy to do.

Start by opening your XAMPP control panel, and make sure that both Apache and MySQL are running. Point your browser to `http://localhost/phpmyadmin` and select your `simple_blog` database. Now, click the Export tab in the top navigation (see Figure 13-1).

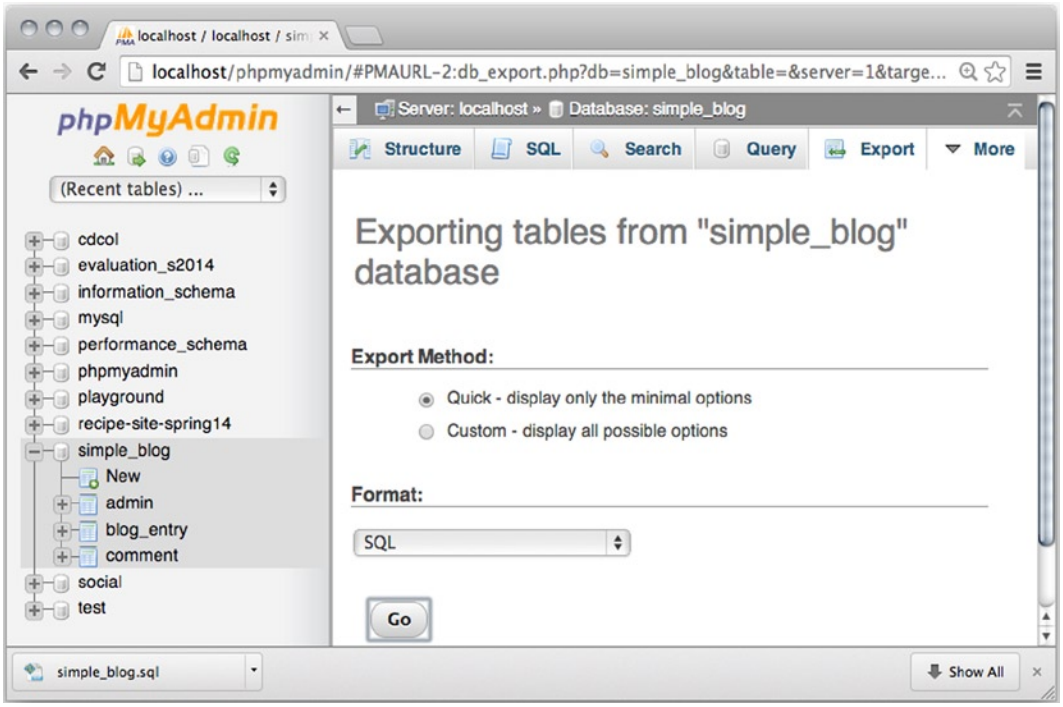


***Figure 13-1.*** *Exporting the* `simple_blog` *database with phpMyAdmin*

To export your `simple_blog` database with all its tables and content, you simply accept SQL as the export format and then click Go. phpMyAdmin will generate an SQL file called `simple_blog.sql`. You can find the generated file in your `downloads` folder. The `simple_blog.sql` file holds a backup of everything in your `simple_blog` database.

The next step is for you to access the phpMyAdmin of your web host. You will need to consult information from your web host to find the right URL for accessing phpMyAdmin on your web host. You will require a valid database username and password to log in to phpMyAdmin.

Once you are logged in to phpMyAdmin on your web host, you can prepare to import your `simple_blog` database. On some web hosts, users are allowed to create a new database. If you can create a new database, I suggest you do so for your project. You could call the new database `simple_blog`.

If your web host does not allow you to make new databases, you can simply use any premade database for your project. It shouldn't pose a problem for you.

Select the database you will use for the project and click the Import option in phpMyAdmin's navigation bar. All you need to do to import your `simple_blog` is to upload the `simple_blog.sql` file from your local computer to your web host. Once you click Go, all tables and all data from your `simple_blog` database should be imported into your new database, running on your web host. You can see for yourself, by browsing the database and the tables, using phpMyAdmin.

Your database is ready, and you can safely move on to the next step.

# Preparing Your PHP Files for Upload

Once your database is imported to your web host, you can prepare your PHP files for upload. It doesn't take much effort. You only have to change the database credentials used in `index.php`, as follows:

```
//Partial code for index.php

//comment out or delete localhost credentials
//$dbInfo = "mysql:host=localhost;dbname=simple_blog";
//$dbUser = "root";
//$dbPassword = "";

//declare valid web host credentials
$dbInfo = "mysql:host=[????];dbname= [????]";
$dbUser = "[????]";
$dbPassword = "[????]";
```

If you use the preceding code example, you will have to replace all instances of `[????]` with valid values required by your web host. You will probably have to consult your web host documentation, to see which `mysql:host` to use. The `dbname` should be the database you used when you imported your `simple_blog` database in the previous step. `Username` and `password` should be whatever username and password you used to log in to phpMyAdmin on your web host.

Once those three lines in `index.php` are updated with valid credentials, you're ready to upload your PHP files to your web host.

## Uploading Files with FileZilla FTP

FTP (File Transfer Protocol) is the standard protocol used to upload files from a local computer to a web host. Many different programs can be used to work with FTP. You could use a browser or even Komodo Edit.

Perhaps you have already used FTP for uploading an HTML project to a web host. If you are already familiar with an FTP program, I suggest you continue using that.

If this is your first time using FTP, I suggest you download and install the FileZilla FTP client from `https://filezilla-project.org/download.php?type=client`. I like FileZilla FTP because it is simple, versatile, and easy to use.

To upload your blog files to your web host, you must establish an FTP connection. You will have to consult your web host documentation to see which URL or host to connect to. In FileZilla FTP, you can type the URL of your FTP host into the Host field, as shown in Figure 13-2. You will also have to enter a username and password. Once again, you will have to consult your web host documentation to find your username and password. Once you have entered your credentials, you can click Quickconnect to establish an FTP connection.

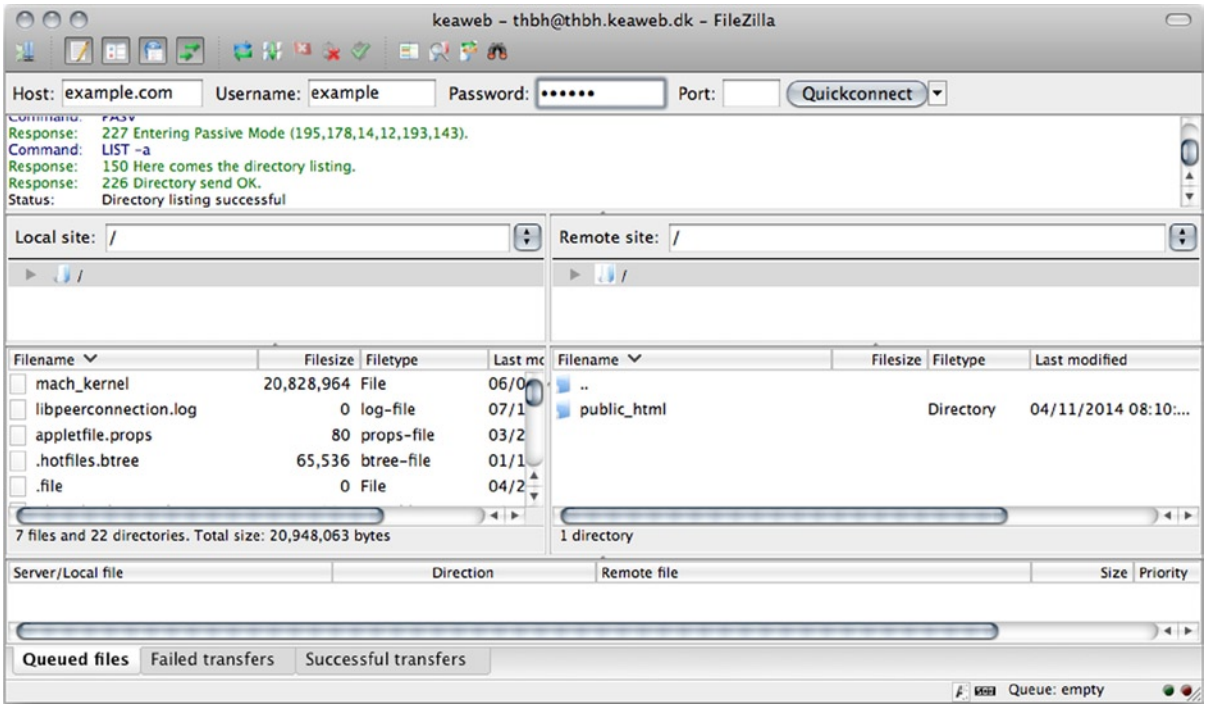


***Figure 13-2.*** *FileZilla FTP connected to a web host*

Figure 13-2 shows FileZilla connected to a web host. The window on the right-hand side shows files and folders on the web host. The window to the left shows files and folders on my local computer.

Remember how you had to put your PHP files in XAMPP's `htdocs` folder? Some web hosts also have such a folder. Whatever you upload to that folder will be available for anybody on the Internet. On your web host, that folder may be called something like `htdocs`, `public_html`, or `www`. Or perhaps your web host will make anything you upload with FTP available on the Internet. Once again, you will have to consult your web host documentation to see how your web host is set up.

You can simply drag and drop files and folders to upload your blog. You would want to upload all the files from `XAMPP/htdocs/blog` on your local computer and transfer those files and folders to your web host.

Once your PHP files are uploaded, your blog is live on the Internet! To see your work, you should navigate your browser to your domain's URL. You should see your blog, and you should also be able to log in with your blog administrator credentials, if you navigate to `admin.php`. Your blog is live, and you can start to write captivating blog posts for your audience: everybody on the Internet.

# Summary

This was a short but important chapter. With a few steps, you have taken the blog project and published it on the Internet for the world to see.

You have also come to the end of the book, but that doesn't mean there is nothing more to learn. Actually, you will probably have to create a few more PHP/MySQL-driven web sites before all the lessons you have learned in this book are fully integrated. The process of developing your competencies further will probably be fun, fascinating, and frustrating. You might consider picking up *Beginning PHP and MySQL, From Novice to Professional* by W. Jason Gilmore (Apress, 2010). It covers PHP and MySQL concepts in greater detail, and it will take you well beyond the basics covered in this book.

# Index

## C



## D

## L



## M, N, O



## P



## Q

## R



## S



## T



## U, V



## W



## X, Y, Z

# PHP for Absolute Beginners



Thomas Blom Hansen
Jason Lengstorf

Apress®

**PHP for Absolute Beginners**



ISBN-13 (pbk): 978-1-4302-6815-4

ISBN-13 (electronic): 978-1-4302-6814-7



Publisher: Heinz Weinheimer
Lead Editor: Ben Renow-Clarke
Technical Reviewer: Adam Shackelford
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Jim DeWolf, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Steve Weiss
Coordinating Editor: Christine Ricketts
Copy Editor: Michael G. Laraque
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science+Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

# Contents

# About the Authors

**Thomas Blom Hansen** has extensive experience teaching web programming in the Digital section of the Copenhagen School of Design and Technology. When he is not teaching, you can find Thomas fly-fishing for sea-run brown trout in the coastal waters around Denmark or possibly hiking some wilderness area in southern Scandinavia. Thomas lives in a small village with his wife, three kids, too few fly rods, and a lightweight camping hammock.

**Jason Lengstorf** is a turbogeek from Portland, Oregon. He started building web sites in his late teens, when his band couldn't afford to pay someone to do it, and he continued building web sites after he realized his band wasn't actually very good. He's been a full-time freelance web developer since 2007 and expanded his business under the name Copter Labs, which is now a distributed freelance collective, keeping about ten freelancers worldwide busy. He is also the author of *Pro PHP and jQuery* (Apress, 2010).

# About the Technical Reviewer

**Adam Shackelford** has been architecting and developing web and mobile applications for the past ten years. He is currently the chief technology officer and lead developer at Caravan Interactive, a technology company he cofounded in Brooklyn in 2009. Prior to Caravan, Adam worked for several agencies in New York City, developing web sites and web applications. He currently resides in the Hudson Valley area of New York.

# Acknowledgments

The authors would like to thank Ben Renow-Clarke and Christine Ricketts for orchestrating the project that led to this book. Thanks to Corbin Collins and Michael G. Laraque for your contributions to copy and organization of the text. Also, thanks to Adam Shackelford. Your technical expertise and keen eye has made the code more accessible for readers.