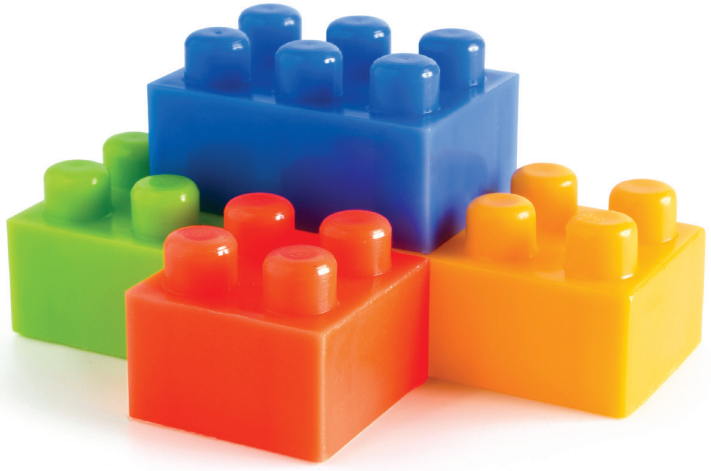
sitepoint®

BUILD YOUR OWN DATABASE DRIVEN WEB SITE USING PHP & MYSQL

BY **KEVIN YANK**

4TH EDITION



LEARNING PHP & MYSQL HAS NEVER BEEN SO EASY!

Thank-you for Downloading This Book

You Too Can Easily Create Impressive Database Driven Web Sites Using PHP and MySQL!

Thank-you for downloading the sample chapters of *Build Your Own Database Driven Web Site Using PHP and MySQL (4th Edition)*, by Kevin Yank, published by SitePoint.

Build Your Own Database Driven Web Site Using PHP & MySQL (4th Edition) is a practical, hands-on guide to learning all the tools, principles, and techniques needed to build a fully functional database driven web site using PHP and MySQL. This book covers everything from installing PHP and MySQL on Windows, Linux, and Mac computers, through to building a live, web-based content management system.

This excerpt includes:

- a summary of contents
- information about the author, editors, and SitePoint
- the Table of Contents
- the Preface
- the first 4 chapters of the book
- the Index

If you enjoy these first 4 chapters, and you're ready to start building your own database driven web sites, you can order yourself a copy now!

For more information, visit http://www.sitepoint.com/launch/3eb28e.

What's in This Excerpt?

Preface

Chapter 1: Installation
Making sure that you have the right tools for the job

Chapter 2: Introducing MySQL
An introduction to databases in general, and the MySQL relational database management system in particular

Chapter 3: Introducing PHP
Here's where the fun really starts, an introduction to the PHP scripting language

Chapter 4: Publishing MySQL Data on the Web
Create some of your first database driven web pages

Index

What's in the Rest of the Book?

Chapter 5: Relational Database Design
Learn the essential principles of good database design

Chapter 6: Structured PHP Programming
Learn simple techniques to keep your code manageable and maintainable

Chapter 7: A Content Management System
The climax of the book: construct a basic content management system

Chapter 8: Content Formatting with Regular Expressions
Some neat tweaks you can make to the page that displays the contents of your database

Chapter 9: Cookies, Sessions, and Access Control
Explore how cookies and sessions work in PHP and use them to build a web site access control system

Chapter 10: MySQL Administration

Learn how to make backups of, manage access to, and repair your MySQL database

Chapter 11: Advanced SQL Queries

Covers some of the more advanced features of this language to help you juggle complex data like a pro

Chapter 12: Binary Data

Learn the ins and outs of file upload and storage, and working with binary data in MySQL

BUILD YOUR OWN DATABASE DRIVEN WEB SITE USING PHP & MYSQL

BY KEVIN YANK

4TH EDITION

Build Your Own Database Driven Web Site Using PHP & MySQL

by Kevin Yank

Managing Editor: Chris Wyness
Editor: Kelly Steele
Technical Editor: Andrew Tetlaw
Cover Design: Alex Walker
Indexer: Russell Brooks
Printing History:
1st Ed. Aug. 2001, 2nd Ed. Feb. 2003,
3rd Ed. Oct. 2004
Fourth Edition: July 2009
Latest Update: July 2009

Notice of Rights

Notice of Liability

Trademark Notice

Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066.
Web: www.sitepoint.com
Email: business@sitepoint.com

ISBN 978-0-9805768-1-8
Printed and bound in the United States of America

About the Author

As Technical Director for SitePoint, Kevin Yank keeps abreast of all that is new and exciting in web technology. Best known for the book you are reading right now, he also co-authored *Simply JavaScript* (http://www.sitepoint.com/books/javascript1/) with Cameron Adams and *Everything You Know About CSS Is Wrong!* (http://www.sitepoint.com/books/csswrong1/) with Rachel Andrew. In addition, Kevin hosts the *SitePoint Podcast* (http://www.sitepoint.com/podcast/) and writes the *SitePoint Tech Times*, a free email newsletter that goes out to over 240,000 subscribers worldwide.

Kevin lives in Melbourne, Australia and enjoys speaking at conferences, as well as visiting friends and family in Canada. He's also passionate about performing improvised comedy theater with Impro Melbourne (http://www.impromelbourne.com.au/) and flying light aircraft. Kevin's personal blog is *Yes, I'm Canadian* (http://yesimcanadian.com/).

About the Technical Editor

Andrew Tetlaw has been tinkering with web sites as a web developer since 1997. At SitePoint he is dedicated to making the world a better place through the technical editing of SitePoint books, kits, articles, and newsletters. He is also a busy father of five, enjoys coffee, and often neglects his blog at http://tetlaw.id.au/.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for Web professionals. Visit http://www.sitepoint.com/ to access our blogs, books, newsletters, articles, and community forums.

To my parents, Cheryl and Richard, for making all this possible.

Table of Contents

Preface

PHP and MySQL have changed.

Back in 2001, when I wrote the first edition of this book, readers were astonished to discover that you could create a site full of web pages without having to write a separate HTML file for each page. **PHP** stood out from the crowd of programming languages, mainly because it was easy enough for almost anyone to learn and free to download and install. The **MySQL** database, likewise, provided a simple and free solution to a problem that, up until that point, had been solvable only by expert programmers with corporate budgets.

Back then, PHP and MySQL were special—heck, they were downright miraculous! But over the years, they have gained plenty of fast-moving competition. In an age when anyone with a free WordPress[1] account can set up a full-featured blog in 30 seconds flat, it's no longer enough for a programming language like PHP to be easy to learn; nor is it enough for a database like MySQL to be free.

Indeed, as you sit down to read this book, you probably have ambitions that extend beyond what you can throw together using the free point-and-click tools of the Web. You might even be thinking of building an exciting, new point-and-click tool of your own. WordPress, after all, is built using PHP and MySQL, so why limit your vision to anything less?

To keep up with the competition, and with the needs of more demanding projects, PHP and MySQL have had to evolve. PHP is now a far more intricate and powerful language than it was back in 2001, and MySQL is a vastly more complex and capable database. Learning PHP and MySQL today opens up a lot of doors that would have remained closed to the PHP and MySQL experts of 2001.

That's the good news. The bad news is that, in the same way that a butter knife is easier to figure out than a Swiss Army knife (and less likely to cause self-injury!), all these dazzling new features and improvements have indisputably made PHP and MySQL more difficult for beginners to learn.

[1] http://wordpress.com/

Worse yet, PHP has completely abandoned several of the beginner-friendly features that gave it a competitive advantage in 2001, because they turned out to be oversimplifications, or could lead inexperienced programmers into building web sites with gaping security holes. This is a problem if you're the author of a beginner's book about PHP and MySQL.

PHP and MySQL have changed, and those changes have made writing this book a lot more difficult. But they have also made this book a lot more important. The more twisty the path, the more valuable the map, right?

In this book, I'll provide you with a hands-on look at what's involved in building a database driven web site using PHP and MySQL. If your web host provides PHP and MySQL support, you're in great shape. If not, I'll show you how to install them on Windows, Mac OS X, and Linux computers, so don't sweat it.

This book is your map to the twisty path that every beginner must navigate to learn PHP and MySQL today. Grab your favorite walking stick; let's go hiking!

Who Should Read This Book

This book is aimed at intermediate and advanced web designers looking to make the leap into server-side programming. You'll be expected to be comfortable with simple HTML, as I'll make use of it without much in the way of explanation. No knowledge of Cascading Style Sheets (CSS) or JavaScript is assumed or required, but if you *do* know JavaScript, you'll find it will make learning PHP a breeze, since these languages are quite similar.

By the end of this book, you can expect to have a grasp of what's involved in building a database driven web site. If you follow the examples, you'll also learn the basics of PHP (a server-side scripting language that gives you easy access to a database, and a lot more) and **Structured Query Language** (**SQL**—the standard language for interacting with relational databases) as supported by MySQL, the most popular free database engine available today. Most importantly, you'll come away with everything you need to start on your very own database driven site!

What's in This Book

This book comprises the following 12 chapters. Read them in order from beginning to end to gain a complete understanding of the subject, or skip around if you only need a refresher on a particular topic.

Chapter 1: ***Installation***

Before you can start building your database driven web site, you must first ensure that you have the right tools for the job. In this first chapter, I'll tell you where to obtain the two essential components you'll need: the PHP scripting language and the MySQL database management system. I'll step you through the setup procedures on Windows, Linux, and Mac OS X, and show you how to test that PHP is operational on your web server.

Chapter 2: ***Introducing MySQL***

Although I'm sure you'll be anxious to start building dynamic web pages, I'll begin with an introduction to databases in general, and the MySQL relational database management system in particular. If you have never worked with a relational database before, this should definitely be an enlightening chapter that will whet your appetite for what's to come! In the process, you'll build up a simple database to be used in later chapters.

Chapter 3: ***Introducing PHP***

Here's where the fun really starts. In this chapter, I'll introduce you to the PHP scripting language, which you can use to build dynamic web pages that present up-to-the-moment information to your visitors. Readers with previous programming experience will probably only need a quick skim of this chapter, as I explain the essentials of the language from the ground up. This is a must-read chapter for beginners, however, as the rest of this book relies heavily on the basic concepts presented here.

Chapter 4: ***Publishing MySQL Data on the Web***

In this chapter you'll bring together PHP and MySQL, which you'll have seen separately in the previous chapters, to create some of your first database driven web pages. You'll explore the basic techniques of using PHP to retrieve information from a database and display it on the Web in real time. I'll also show you how to use PHP to create web-based forms for adding new entries to, and modifying existing information in, a MySQL database on the fly.

Chapter 5: *Relational Database Design*

Although you'll have worked with a very simple sample database in the previous chapters, most database driven web sites require the storage of more complex forms of data than you'll have dealt with to this point. Far too many database driven web site designs are abandoned midstream or are forced to start again from the beginning, because of mistakes made early on during the design of the database structure. In this critical chapter you'll learn the essential principles of good database design, emphasizing the importance of data normalization. If you're unsure what that means, then this is definitely an important chapter for you to read!

Chapter 6: *Structured PHP Programming*

Techniques to better structure your code are useful in all but the simplest of PHP projects. The PHP language offers many facilities to help you do this, and in this chapter, I'll cover some of the simple techniques that exist to keep your code manageable and maintainable. You'll learn to use include files to avoid having to write the same code more than once when it's needed by many pages of your site, and I'll show you how to write your own functions to extend the built-in capabilities of PHP and to streamline the code that appears within your scripts.

Chapter 7: *A Content Management System*

In many ways the climax of the book, this chapter is the big payoff for all you frustrated site builders who are tired of updating hundreds of pages whenever you need to make a change to a site's design. I'll walk you through the code for a basic content management system that allows you to manage a database of jokes, their categories, and their authors. A system like this can be used to manage simple content on your web site; just a few modifications, and you'll have a site administration system that will have your content providers submitting content for publication on your site in no time—all without having to know a shred of HTML!

Chapter 8: *Content Formatting with Regular Expressions*

Just because you're implementing a nice, easy tool to allow site administrators to add content to your site without their knowing HTML, that content can still be jazzed up, instead of settling for just plain, unformatted text. In this chapter, I'll show you some neat tweaks you can make to the page that displays the

contents of your database—tweaks that allow it to incorporate simple formatting such as bold or italicized text, among other options.

Chapter 9: *Cookies, Sessions, and Access Control*

What are sessions, and how are they related to cookies, a long-suffering technology for preserving stored data on the Web? What makes persistent data so important in current ecommerce systems and other web applications? This chapter answers all those questions by explaining how PHP supports both cookies and sessions, and explores the link between the two. You'll then put these pieces together to build a simple shopping cart system, as well as an access control system for your web site.

Chapter 10: *MySQL Administration*

While MySQL is a good, simple database solution for those without the need for many frills, it does have some complexities of its own that you'll need to understand if you're going to rely on a MySQL database to store your content. In this section, I'll teach you how to perform backups of, and manage access to, your MySQL database. In addition to a couple of inside tricks (like what to do if you forget your MySQL password), I'll explain how to repair a MySQL database that has become damaged in a server crash.

Chapter 11: *Advanced SQL Queries*

In Chapter 5 we saw what was involved in modeling complex relationships between pieces of information in a relational database like MySQL. Although the theory was quite sound, putting these concepts into practice requires that you learn a few more tricks of Structured Query Language. In this chapter, I'll cover some of the more advanced features of this language to help you juggle complex data like a pro.

Chapter 12: *Binary Data*

Some of the most interesting applications of database driven web design include some juggling of binary files. Online file storage services are prime examples, but even a system as simple as a personal photo gallery can benefit from storing binary files (that is, pictures) in a database for retrieval and management on the fly. In this chapter, I'll demonstrate how to speed up your web site by creating static copies of dynamic pages at regular intervals—using PHP, of course! With these basic file-juggling skills in hand, you'll go on to develop a simple online

file storage and viewing system, and learn the ins and outs of working with binary data in MySQL.

Where to Find Help

PHP and MySQL are moving targets, so chances are good that, by the time you read this, some minor detail or other of these technologies has changed from what's described in this book. Thankfully, SitePoint has a thriving community of PHP developers ready and waiting to help you out if you run into trouble, and we also maintain a list of known errata for this book you can consult for the latest updates.

The SitePoint Forums

The SitePoint Forums[2] are discussion forums where you can ask questions about anything related to web development. You may, of course, answer questions, too. That's how a discussion forum site works—some people ask, some people answer and most people do a bit of both. Sharing your knowledge benefits others and strengthens the community. A lot of fun and experienced web designers and developers hang out there. It's a good way to learn new stuff, have questions answered in a hurry, and just have fun.

The SitePoint Forums include separate forums for PHP and MySQL, as well as a separate forum covering advanced PHP Application Design:

- PHP: http://www.sitepoint.com/forums/forumdisplay.php?f=34
- PHP Application Design: http://www.sitepoint.com/forums/forumdisplay.php?f=147
- MySQL: http://www.sitepoint.com/forums/forumdisplay.php?f=182

The Book's Web Site

Located at http://www.sitepoint.com/books/phpmysql1/, the web site that supports this book will give you access to the following facilities:

The Code Archive

As you progress through this book, you'll note a number of references to the code archive. This is a downloadable ZIP archive that contains each and every line of

[2] http://www.sitepoint.com/forums/

example source code that's printed in this book. If you want to cheat (or save yourself from carpal tunnel syndrome), go ahead and download the archive.[3]

Updates and Errata

No book is perfect, and we expect that watchful readers will be able to spot at least one or two mistakes before the end of this one. The Errata page on the book's web site will always have the latest information about known typographical and code errors.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters, such as *SitePoint Tech Times*, *SitePoint Tribune*, and *SitePoint Design View*, to name a few. In them, you'll read about the latest news, product releases, trends, tips, and techniques for all aspects of web development. Sign up to one or more SitePoint newsletters at http://www.sitepoint.com/newsletter/.

Your Feedback

If you're unable to find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is `books@sitepoint.com`. We have a well-staffed email support system set up to track your inquiries, and if our support team members are unable to answer your question, they'll send it straight to us. Suggestions for improvements, as well as notices of any mistakes you may find, are especially welcome.

[3] http://www.sitepoint.com/books/phpmysql1/code.php

Conventions Used in This Book

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park. The birds
were singing and the kids were all back at school.</p>
```

If the code is to be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer {
  background-color: #CCC;
  border-top: 1px solid #333;
}
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css *(excerpt)*

```
  border-top: 1px solid #333;
```

If additional code is to be inserted into an existing example, the new code will be displayed in bold:

```
function animate() {
  new_variable = "Hello";
}
```

Also, where existing code is required for context, rather than repeat all the code, a vertical ellipsis will be displayed:

```
function animate() {
  ⋮
  return new_variable;
}
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➥ indicates a line break that exists for formatting purposes only, and should be ignored.

```
URL.open("http://www.sitepoint.com/blogs/2007/05/28/user-style-she
➥ets-come-of-age/");
```

Tips, Notes, and Warnings

Hey, You!

Tips will give you helpful little pointers.

Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

Make Sure You Always ...

... pay attention to these important points.

Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Chapter 1

Installation

In this book, I'll guide you as you take your first steps beyond the static world of building web pages with pure HTML. Together, we'll explore the world of database driven web sites and discover the dizzying array of dynamic tools, concepts, and possibilities that they open up. Whatever you do, don't look down!

Okay, maybe you *should* look down. After all, that's where the rest of this book is. But remember, you were warned!

Before you build your first dynamic web site, you must gather together the tools you'll need for the job. In this chapter, I'll show you how to download and set up the two software packages you'll need. Can you guess what they are? I'll give you a hint: their names feature prominently on the cover of this book! They are, of course, PHP and MySQL.

If you're used to building web sites with HTML, CSS, and perhaps even a smattering of JavaScript, you're probably used to uploading to another location the files that make up your site. Maybe this is a web hosting service that you've paid for; maybe it's a free service provided by your Internet Service Provider (ISP); or maybe it's a web server set up by the IT department of the company that you work for. In any

case, once you copy your files to their destination, a software program called a **web server** is able to find and serve up copies of those files whenever they are requested by a web browser like Internet Explorer or Firefox. Common web server software programs you may have heard of include Apache and Internet Information Services (IIS).

PHP is a **server-side scripting language**. You can think of it as a plugin for your web server that enables it to do more than just send exact copies of the files that web browsers ask for. With PHP installed, your web server will be able to run little programs (called **PHP scripts**) that can do tasks like retrieve up-to-the-minute information from a database and use it to generate a web page on the fly before sending it to the browser that requested it. Much of this book will focus on writing PHP scripts to do exactly that. PHP is completely free to download and use.

For your PHP scripts to retrieve information from a database, you must first *have* a database. That's where **MySQL** comes in. MySQL is a **relational database management system**, or **RDBMS**. We'll discuss the exact role it plays and how it works later, but briefly it's a software program that's able to organize and manage many pieces of information efficiently while keeping track of how all of those pieces of information are related to each other. MySQL also makes that information really easy to access with server-side scripting languages like PHP. MySQL, like PHP, is completely free for most uses.

The goal of this first chapter is to set you up with a web server equipped with PHP and MySQL. I'll provide step-by-step instructions that work on recent Windows, Mac OS X, and Linux computers, so no matter what flavor of computer you're using, the instructions you need should be right here.

Your Own Web Server

If you're lucky, your current web host's web server already has PHP and MySQL installed. Most do—that's one of the reasons why PHP and MySQL are so popular. If your web host is so equipped, the good news is that you'll be able to publish your first database driven web site without having to shop for a web host that supports the right technologies.

The bad news is that you're still going to need to install PHP and MySQL yourself. That's because you need your own PHP-and-MySQL-equipped web server to test your database driven web site on before you publish it for all the world to see.

When developing static web sites, you can often load your HTML files directly from your hard disk into your browser to see how they look. There's no web server software involved when you do this, which is fine, because web browsers can understand HTML code all by themselves.

When it comes to dynamic web sites built using PHP and MySQL, however, your web browser needs some help! Web browsers are unable to understand PHP scripts; rather, PHP scripts contain instructions for a PHP-savvy web server to execute in order to *generate* the HTML code that browsers can understand. So in addition to the web server that will host your site publicly, you also need your own private web server to use in the development of your site.

If you work for a company that has an especially helpful IT department, you may find that there's already a development web server provided for you. The typical setup is that you must work on your site's files on a network drive that's hosted by an internal web server that can be safely used for development. When you're ready to deploy the site to the public, your files are copied from that network drive to the public web server.

If you're lucky enough to work in this kind of environment, you can skip most of this chapter. However, you'll want to ask the IT boffins responsible for the development server the same questions I've covered in the section called "What to Ask Your Web Host". That's because you'll need to have that critical information handy when you start using the PHP and MySQL support they've so helpfully provided.

Windows Installation

In this section, I'll show you how to start running a PHP-and-MySQL-equipped web server on a Windows XP, Windows Vista, or Windows 7 computer. If you're using an operating system other than Windows, you can safely skip this section.

All-in-one Installation

I normally recommend that you install and set up your web server, PHP, and MySQL individually, using the official installation packages for each. This is especially

useful for beginners, because it gives you a strong sense of how these pieces all fit together. If you're in a rush, however, or if you need to set up a temporary development environment to use just for a day or two, the following quick-and-dirty solution may be preferable.

You can skip ahead to the section called "Installing Individual Packages" if you want to take the time to install each piece of the puzzle separately.

WampServer (where *Wamp* stands for Windows, Apache, MySQL, and PHP) is a free, all-in-one program that includes built-in copies of recent versions of the Apache web server, PHP, and MySQL. Let me take you through the process of installing it:

1. Download the latest version from the WampServer web site.[1] After downloading the file (as of this writing, WampServer 2.0g is about 16MB in size), double-click it to launch the installer, as shown in Figure 1.1.

Figure 1.1. The WampServer installer

2. The installer will prompt you for a location to install WampServer. The default of **c:\wamp** shown in Figure 1.2 is an ideal choice for most purposes, but if you have strong feelings about where it's installed, feel free to specify your preferred location.

[1] http://www.wampserver.com/en/

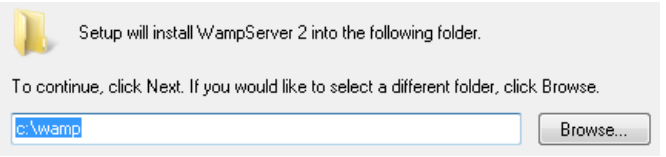


Figure 1.2. The default installation directory is a good choice

3. At the end of the installation, WampServer will ask you to choose your default browser. This is the web browser it will launch when you use the included system tray icon tool to launch your browser. If you have Firefox installed it will ask if you'd like to use it as your default browser. If you answer **No**, or have a different browser installed, it will ask you to select the executable file for the browser you want to use. As shown in Figure 1.3, it selects Internet Explorer (**explorer.exe**) for you, which is fine. If you're using an alternative browser such as Safari or Opera, you can browse to find the **.exe** file for your browser if you want to.

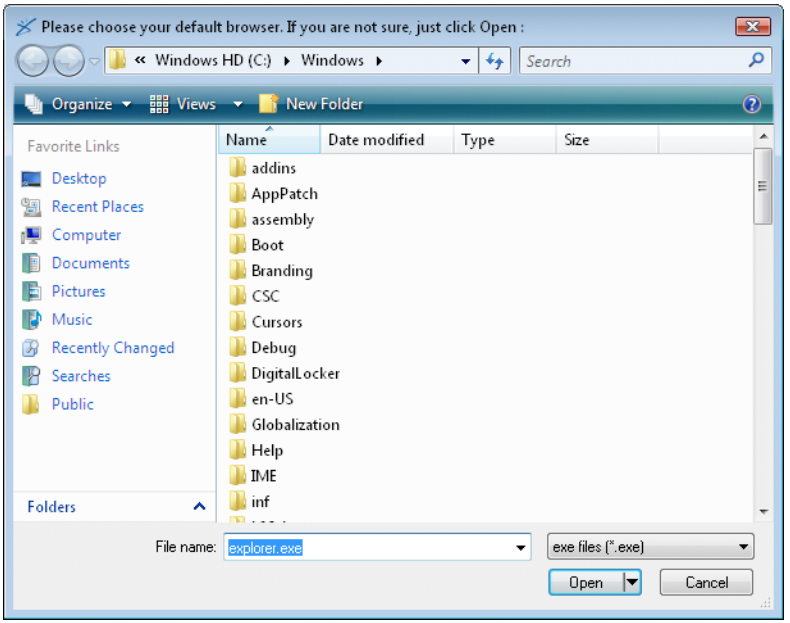


Figure 1.3. The default choice of Internet Explorer is fine

4. As WampServer is installed, it fires up its built-in copy of the Apache HTTP Server, a popular web server for PHP development. Windows will likely display

a security alert at this point, like the one in Figure 1.4, since the web server attempts to start listening for browser requests from the outside world.

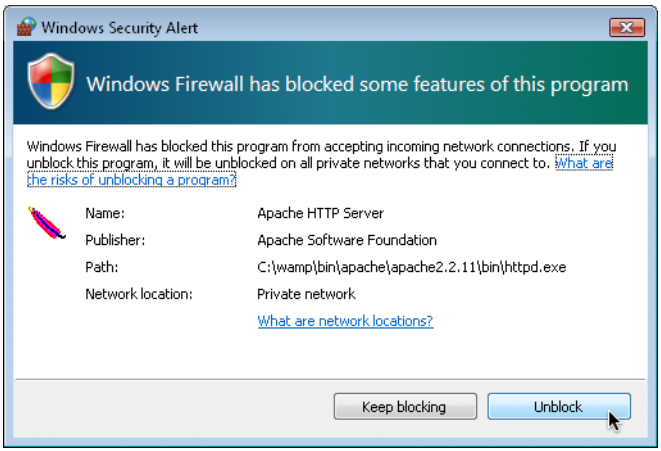


Figure 1.4. This security alert tells you Apache is doing its job

If you want to make absolutely sure that Apache rejects connections from the outside world, and that only a web browser running on your own computer can view web pages hosted on your development server, feel free to click **Keep blocking**. WampServer has its own built-in option to block connections from the outside world when you want to, however, so I recommend clicking **Unblock** in order to have the flexibility to grant access to your development server if and when you need to.

5. Next, as shown in Figure 1.5, the WampServer installer will prompt you for your SMTP server and email address. A PHP script can send an email message, and these settings tell it the outgoing email server, and the default "from" address to use. Type in your email address, and if you can remember your Internet Service Provider's SMTP server address, type it in too. You can always leave the default value for the time being, though, and set it manually if and when you need to send email using a PHP script.

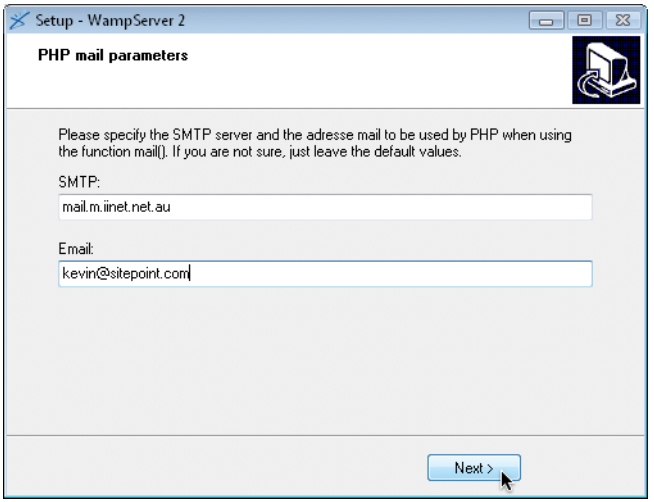


Figure 1.5. Fill in your Internet Service Provider's SMTP server address if you know it

Once the installation is complete, you can fire up WampServer. An icon will appear in your Windows System Tray. Click on it to see the WampServer menu shown in Figure 1.6.

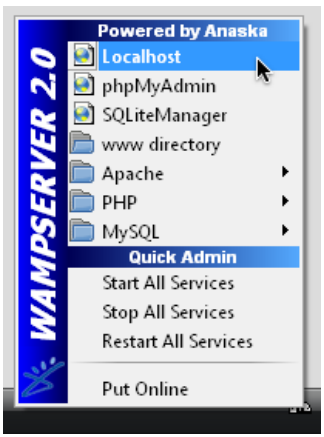


Figure 1.6. The WampServer menu

By default, your server can only be accessed by web browsers running on your own computer. If you click the **Put Online** menu item, your server will become accessible to the outside world.

To test that WampServer is working properly, click the **Localhost** menu item at the top of the WampServer menu. Your web browser will open to display your server's home page, shown in Figure 1.7.

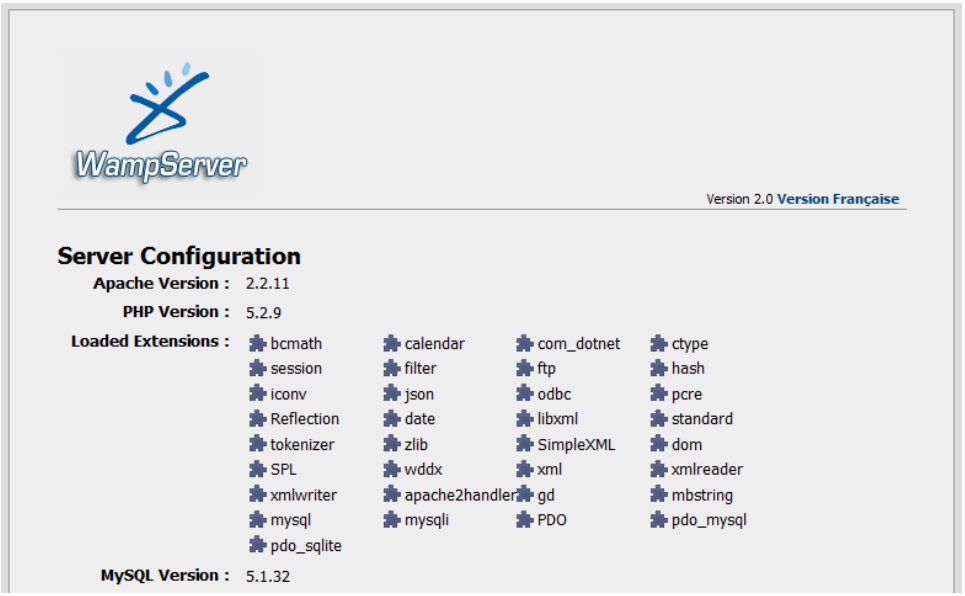


Figure 1.7. The home page provided by WampServer confirms Apache, PHP, and MySQL are installed

When you're done working with WampServer, you can shut it down (along with its built-in servers) by *right-clicking* the System Tray icon and choosing **Exit**. When you're next ready to do some work on a database driven web site, just fire it up again!

Later in this book, you'll need to use some of the programs that come with the MySQL server built into WampServer. To work properly, these programs must be added to your Windows system path.

To add the MySQL command prompt programs that come with WampServer to your Windows system path, follow these instructions:

1. Open the Windows Control Panel. Locate and double-click the **System** icon.

2. Take the appropriate step for your version of Windows:
   - In Windows XP, switch to the **Advanced** tab of the **System Properties** window.
   - In Windows Vista or Windows 7, click the **Advanced system settings** link in the sidebar.
3. Click the **Environment Variables…** button.
4. In the list labeled **User variables for *user***, look for a variable named **PATH**.
   - If it exists, select it and click the **Edit…** button.
   - If there's no variable, click the **New…** button and fill in the **Variable name** by typing **PATH**.
5. Add the path to WampServer's MySQL **bin** directory[2] as the **Variable value**:
   - If the **Variable value** is empty, just type in the path.
   - If there is already text in the **Variable value** field, add a semicolon (**;**) to the end of the value, then type the path after that.
6. Click the **OK** button in each of the open windows to apply your changes.

Installing Individual Packages

Installing each individual package separately is really the way to go if you can afford to take the time. That way you learn how all the pieces fit together, but have the freedom to update each of the packages independently of the others. Ultimately, it's always worthwhile becoming familiar with the inner workings of any software with which you'll be spending a lot of time.

Installing MySQL

As I mentioned above, you can download MySQL free of charge. Simply proceed to the MySQL Downloads page[3] and click the **Download** link for the free MySQL

[2] The exact path will depend on where you've installed WampServer and which version of MySQL it contains. On my system, the path is **C:\wamp\bin\mysql\mysql5.1.34\bin**. Use Explorer to take a look inside your WampServer installation's files to figure out the exact path on your system.
[3] http://dev.mysql.com/downloads/

Community Server. This will take you to a page with a long list of download links for the current recommended version of MySQL (as of this writing, it's MySQL 5.1).

At the top of the list you'll see links for Windows and Windows x64. If you're positive you're running a 64-bit version of Windows, go ahead and follow the **Windows x64** link to download the **Windows Essentials (AMD64 / Intel EM64T)** package (about 28MB in size). If you know you're running a 32-bit version of Windows, or if you're at all unsure, follow the **Windows** link and download the **Windows Essentials (x86)** package (about 35MB)—it'll work even if it turns out you're running a 64-bit version of Windows. Although a little obscure, the **Pick a mirror** link shown in Figure 1.8 is the one you need to click to download the file.

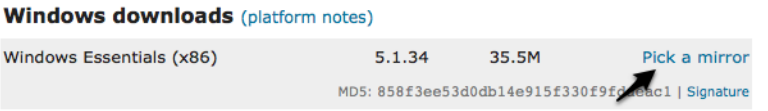


Figure 1.8. Finding the right link can be tricky—here it is!

Once you've downloaded the file, double-click it and go through the installation as you would for any other program. Choose the **Typical** option when prompted for the setup type, unless you have a particular preference for the directory in which MySQL is installed. When you reach the end, you'll be prompted to choose whether you want to **Configure the MySQL Server now**. Select this to launch the configuration wizard,[4] and choose **Detailed Configuration**, which we'll use to specify a number of options that are vital to ensuring compatibility with PHP. For each step in the wizard, select the options indicated here:

1. **Server Type**

   Assuming you're setting up MySQL for development purposes on your desktop computer, choose **Developer Machine**.

[4] In my testing, I found that the configuration wizard failed to actually launch automatically, even with this option checked. If you run into the same problem, just launch the MySQL Server Instance Config Wizard from the Start Menu after the installation has completed.

2. **Database Usage**

   Unless you know for a fact that you will need support for transactions (as such support is usually superfluous for most PHP applications), choose **Non-Transactional Database Only**.

3. **Connection Limit**

   Select **Decision Support (DSS)/OLAP** to optimize MySQL for a relatively modest number of connections.

4. **Networking Options**

   Uncheck the **Enable Strict Mode** option to ensure MySQL's compatibility with older PHP code that you might need to use in your own work.

5. **Default Character Set**

   Select **Best Support For Multilingualism** to tell MySQL to assume you want to use UTF-8 encoded text, which supports the full range of characters that are in use on the Web today.

6. **Windows Options**

   Allow MySQL to be installed as a Windows Service that's launched automatically; also select **Include Bin Directory in Windows PATH** to make it easier to run MySQL's administration tools from the command prompt.

7. **Security Options**

   Uncheck the **Modify Security Settings** option. It's best to learn how to set the root password mentioned at this juncture without the assistance of the wizard, so I'll show you how to do this yourself in the section called "Post-Installation Set-up Tasks".

Once the wizard has completed, your system should now be fully equipped with a running MySQL server!

To verify that the MySQL server is running properly, type **Ctrl+Alt+Del** and choose the option to open the Task Manager. Click the **Show processes from all users** button unless it's already selected. If all is well, the server program (**mysqld.exe**) should be

listed on the **Processes** tab. It will also start up automatically whenever you restart your system.

Installing PHP

The next step is to install PHP. Head over to the PHP Downloads page[5] and choose the **PHP 5.2.x zip package** under **Windows Binaries**; avoid the **installer** version, which is easier to install, but lacks the same flexibility attained by installing PHP manually.

What about PHP 4?

At the time of writing, PHP 5 is firmly entrenched as the preferred version of PHP. For several years after PHP 5's initial release, many developers chose to stick with PHP 4 due to its track record of stability and performance, and indeed today many bargain-basement web hosts have yet to upgrade to PHP 5. There's no longer any excuse for this, however; PHP 5 is by far the better choice, and development of PHP 4 has been completely discontinued. If your web host is still living in the PHP 4 past, you're better off finding a new web host!

PHP was designed to run as a plugin for existing web server software such as Apache or Internet Information Services, so before you can install PHP, you must first set up a web server.

Many versions of Windows come with Microsoft's powerful Internet Information Services (IIS) web server, but not all do. Windows XP Home, Windows Vista Home, and Windows 7 Home Basic (among others) are without IIS, so you need to install your own web server on these versions of Windows if you want to develop database driven web sites. On top of that, assorted versions of Windows come with different versions of IIS, some of which vary dramatically in how you configure them to work with PHP.

With that in mind, if you're still considering IIS, you should know it's also relatively uncommon to host web sites built using PHP with IIS in the real world. It's generally less expensive and more reliable to host PHP-powered sites on servers running some flavor of the Linux operating system, with the free Apache web server installed. About the only reason for hosting a PHP site on IIS is if your company has already invested in Windows servers to run applications built using ASP.NET (a Microsoft

[5] http://www.php.net/downloads.php

technology built into IIS), and you want to reuse that existing infrastructure to host a PHP application as well.

Although it's by no means a requirement, it's generally easiest to set up your development server to match the environment in which your web site will be deployed publicly as closely as possible. For this reason, I recommend using the Apache web server—even for development on a Windows computer. If you insist (or your boss insists) on hosting your PHP-based site using IIS, you will find the necessary installation instructions in the **install.txt** file contained in the PHP zip package you downloaded from the PHP web site.

If you need to install Apache on your computer, surf on over to The Apache HTTP Server Project[6] and look for the version of Apache described as the best available (as of writing it's version 2.2.11, as shown in Figure 1.9).

Figure 1.9. The best available version—accept no substitutes!

Once you get to the Download page, scroll down to find the links to the various versions available. The one you'll want is **Win32 Binary without crypto**, shown in Figure 1.10.

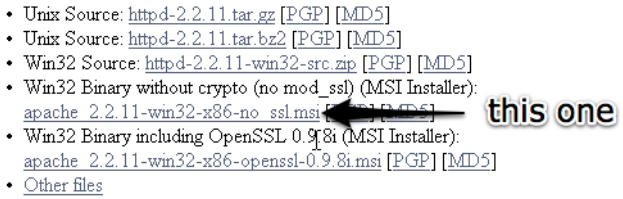


Figure 1.10. This is the one you need

[6] http://httpd.apache.org/

Once the file has downloaded, double-click on it as usual to start the installation wizard. After a few steps, you'll arrive at the **Server Information** screen.

If you were setting up a web server to be accessed publicly on the Web, the options on this screen would be important. For the purposes of setting up a development server, you can type whatever you like. If you know your computer's network name, type that in for the Server Name. Feel free to put in your correct email address if, like me, you're a stickler for the details. If you already have a web server running on your computer (for example, if you have also set up IIS to do some ASP.NET development on the same computer), you may need to select the **only for the Current User, on Port 8080, when started Manually** option on this screen, so as to avoid a conflict with the existing web server running on port 80.

On the next screen, choose the **Typical** option for the **Setup Type**, and follow the wizard from there to complete the installation. When it's done, you should see a new icon for the Apache Service Monitor running in your System Tray. If you chose the default option to have Apache start up automatically, the status indicator should be green, as shown in Figure 1.11; otherwise, you'll need to start Apache manually as shown in Figure 1.12 before you can use it.

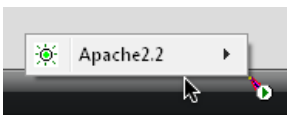


Figure 1.11. The green light means Apache is up and running

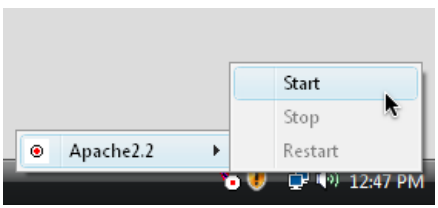


Figure 1.12. Choose **Start** to fire up Apache manually

You can also use the Apache Service Monitor icon to stop Apache running, once you've finished your web development work for the day.

When you have Apache up and running, open your web browser of choice and type http://localhost into the location bar. If you chose the option to run Apache on port

8080, you will need to type http://localhost:8080 instead. Hit **Enter**, and you should see a page like that shown in Figure 1.13 that confirms Apache is working correctly.

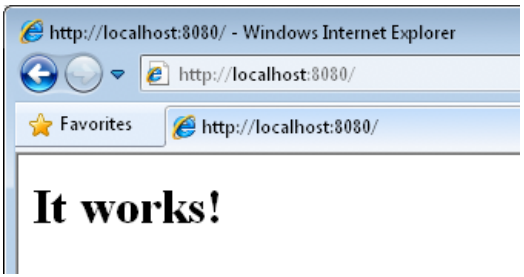


Figure 1.13. You can take my word for it!

With Apache standing on its own two feet, you can now install PHP. Follow these steps:

1. Unzip the file you downloaded from the PHP web site into a directory of your choice. I recommend **C:\PHP** and will refer to this directory from this point forward, but feel free to choose another directory if you like.

2. Find the file called **php.ini-dist** in the PHP folder and make a duplicate copy of it. The easiest way to do it is to right-click and drag the file's icon a short distance, drop it in the same Explorer window, and choose **Copy Here** from the pop-up menu. This will leave you with a new file named along the lines of **php - Copy.ini-dist** (depending on the version of Windows you're using). Find this new file and rename it to **php.ini**. Windows will ask if you're sure about changing the filename extension (from **.ini-dist** to **.ini**); click **Yes**.

Windows Hides Known Filename Extensions by Default

When you rename the file to **php.ini**, you might notice that the new filename that appears next to the icon is actually just **php**. If this happens, it's because your copy of Windows is set up to hide the filename extension if it recognizes it. Since Windows knows that **.ini** files are Configuration Settings files, it hides this filename extension.

As you can imagine, this feature can cause a certain amount of confusion. When you return to edit the **php.ini** file in the future, it would help to be able to see its full filename so you could tell it apart from the **php.gif** and **php.exe** files in the same folder.

To switch off filename extension hiding, open the Windows Control Panel and search for Folder Options. Open the Folder Options window and switch to the **View** tab. Under **Files and Folders**, uncheck the **Hide extensions for known file types** checkbox, as shown in Figure 1.14.

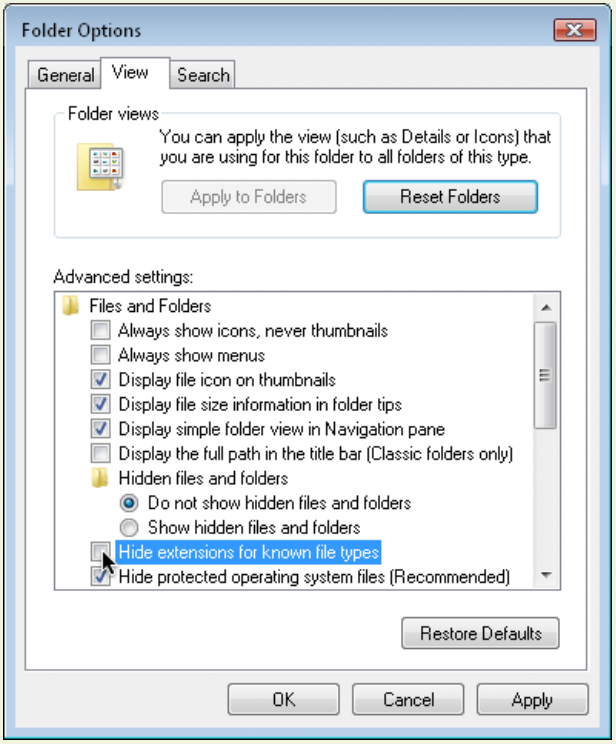


Figure 1.14. Make filename extensions visible for all files

3. Open the **php.ini** file in your favorite text editor. If you have no particular preference, just double-click the file to open it in Notepad. It's a large file with a lot of confusing options, but look for the line that begins with `doc_root` (Notepad's **Edit** > **Find...** feature will help). Out of the box, this line looks like this:

```
doc_root =
```

To the end of this line, add the path to your web server's document root directory. For the Apache server, this is the **htdocs** folder in the main Apache web server directory. If you installed Apache in the default location, the path should be "**C:\Program Files\Apache Software Foundation\Apache2.2\htdocs**". If you installed it elsewhere, find the **htdocs** folder and type its path:

```
doc_root = "C:\Program Files\Apache Software Foundation\Apache2.
➥2\htdocs"
```

Just a little further down in the file, look for the line that begins with `extension_dir`, and set it so that it points to the **ext** subfolder of your PHP folder:

```
extension_dir = "C:\PHP\ext"
```

Scroll further down in the file, and you'll see a bunch of lines beginning with `;extension=`. These are optional extensions to PHP, disabled by default. We want to enable the MySQL extension so that PHP can communicate with MySQL. To do this, remove the semicolon from the start of the **php_mysqli.dll** line:

```
extension=php_mysqli.dll
```

`php_mysqli`, not `php_mysql`

Just above the line for `php_mysqli.dll` there is a line for `php_mysql.dll`. The *i* in `php_mysqli` stands for *improved*. You want to enable the new improved MySQL extension. The one without the *i* is obsolete, and some of its features are incompatible with current versions of MySQL.

Keep scrolling even further down in the file, and look for a line that starts with `;session.save_path`. Once again, remove the semicolon to enable this line, and set it to your Windows **Temp** folder:

```
session.save_path = "C:\Windows\Temp"
```

Save the changes you made and close your text editor.

That takes care of setting up PHP. Now you can set up your Apache server to use it as a plugin:

1. Run Notepad as Administrator. This is necessary because the Apache configuration file, by default, can only be edited by an administrator. To do this, find the **Notepad** icon in your Start Menu (under **All Programs** > **Accessories**) and right-click on it. Click the **Run as administrator** menu item.

2. Choose **File** > **Open...** in Notepad. Browse to the **conf** subfolder in your Apache installation folder (by default, **C:\Program Files\Apache Software Foundation\Apache2.2\conf**), and select the **httpd.conf** file located there. In order to make this file visible for selection, you'll need to select **All Files (*.*)** from the file type drop-down menu at the bottom of the **Open** window.

3. Look for the existing line in this file that begins with `DirectoryIndex`, shown here:

```
<IfModule dir_module>
    DirectoryIndex index.html
</IfModule>
```

This line tells Apache which filenames to use when it looks for the default page for a given directory. Add **index.php** to the end of this line:

```
<IfModule dir_module>
    DirectoryIndex index.html index.php
</IfModule>
```

4. All of the remaining options in this long and intimidating configuration file should have been set up correctly by the Apache install program. All you need to do is add the following lines to the very end of the file:

```
LoadModule php5_module "C:/PHP/php5apache2_2.dll"
AddType application/x-httpd-php .php
PHPIniDir "C:/PHP"
```

Make sure the `LoadModule` and `PHPIniDir` lines point to your PHP installation directory, and note the use of forward slashes (`/`) instead of backslashes (`\`) in the paths.

PHP and Future Apache Versions

Historically, major new versions of the Apache server have required new versions of the **.dll** file you see referenced in the `LoadModule` line above. If you take another look in your PHP installation directory, for example, you'll see there are also **php5apache.dll** and **php5apache2.dll** files there. These files were provided for use with Apache 1.3 and Apache 2.0, respectively.

By the time you read this, it's possible that Apache has undergone another major release (for instance, Apache 2.3), which might need yet another new **.dll** file. For example, Apache 2.3 might require you to use a new file named **php5apache2_3.dll**.

If you *are* using a subsequent version of Apache, and if you *do* see a **.dll** file that looks like it might correspond to your Apache version, try adjusting the `LoadModule` line accordingly. You can always return and edit this file again later if Apache fails to load PHP correctly.

5. Save your changes and close Notepad.
6. Restart Apache using the Apache Service Monitor system tray icon. If all is well, Apache will start up again without complaint.
7. Double-click the Apache Service Monitor icon to open the Apache Service Monitor window. If PHP is installed correctly, the status bar of this window should indicate the version of PHP you have installed, as shown in Figure 1.15.
8. Click OK to close the Apache Service Monitor window.

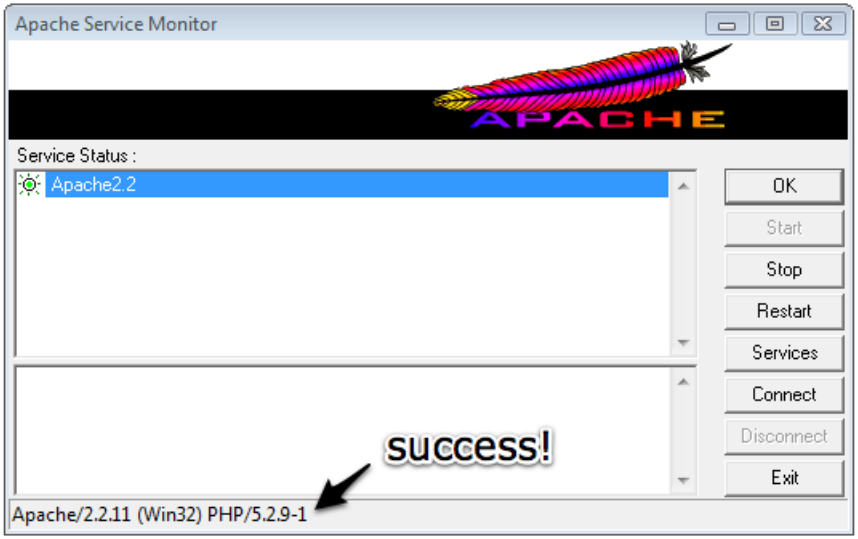


Figure 1.15. The PHP version number indicates Apache is configured to support PHP

With MySQL, Apache, and PHP installed, you're ready to proceed to the section called "Post-Installation Set-up Tasks".

Mac OS X Installation

Mac OS X distinguishes itself by being the only consumer OS to install both Apache and PHP as components of every standard installation. That said, these take a few tweaks to switch on, and you'll need to install the MySQL database as well.

In this section, I'll show you how to start running a PHP-and-MySQL-equipped web server on a Mac computer running Mac OS X version 10.5 (Leopard). If you're using an alternative to a Mac, you can safely skip this section.

All-in-one Installation

I normally recommend that you install and set up your web server, PHP, and MySQL individually, using the official installation packages for each. This process is especially useful for beginners, because it gives you a strong sense of how these pieces all fit together. If you're in a rush, however, or if you need to set up a temporary development environment to use just for a day or two, a quick-and-dirty solution may be preferable.

You can skip ahead to the section called "Installing Individual Packages" if you want to take the time to install each piece of the puzzle separately.

MAMP (which stands for Mac, Apache, MySQL, and PHP) is a free, all-in-one program that includes built-in copies of recent versions of the Apache web server, PHP, and MySQL. Let me take you through the process of installing it:

1. Download the latest version from the MAMP web site.[7] After downloading the file (as of this writing, MAMP 1.7.2 is about 130MB in size), double-click it to unzip the disk image (**MAMP_1.7.2.dmg**), then double-click the disk image to mount it, as shown in Figure 1.16.

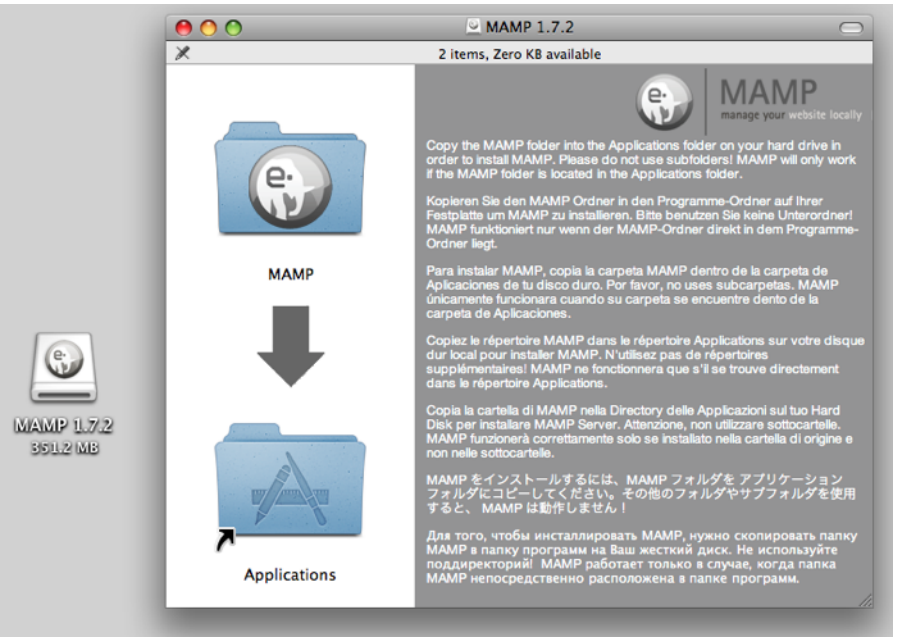


Figure 1.16. The MAMP package

2. As instructed in the disk image window, drag the **MAMP** folder icon over to the **Applications** folder icon to install MAMP on your system. After the copy operation has completed, you can drag the **MAMP** icon on your desktop to the **Trash** icon on your dock to eject it (it will turn into an **Eject** icon), then delete the disk image, as well as the original **.zip** file you downloaded.

[7] http://www.mamp.info

Browse to your **Applications** folder and find the new **MAMP** folder there. Open it, and double-click the **MAMP** icon inside to launch MAMP. As MAMP starts up, the following will happen. First, the MAMP window shown in Figure 1.17 will appear. The two status indicators will switch from red to green as the built-in Apache and MySQL servers start up. Next, MAMP will open your default web browser and load the MAMP welcome page, shown in Figure 1.18.

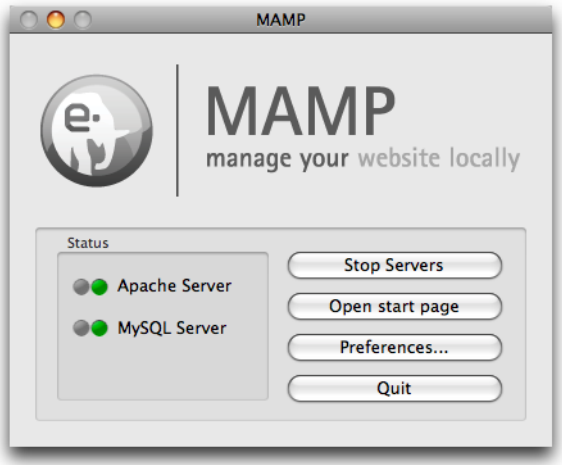


Figure 1.17. The MAMP window

Figure 1.18. The MAMP welcome page confirms Apache, PHP, and MySQL are up and running

When you're done working with MAMP, you can shut it down (along with its built-in servers) by clicking the **Quit** button in the MAMP window. When you're next ready to do some work on a database driven web site, just fire it up again!

Later in this book, you'll need to use some of the programs that come with the MySQL server built into MAMP. To work properly, these programs must be added to your Mac OS X system path.

To add the MySQL command prompt programs that come with MAMP to your Mac OS X system path, follow these instructions:

1. Open a Terminal window.[8]

   - If you're running Mac OS X 10.5 (Leopard) or later, type these commands:

```
Machine:~ user$ sudo su
Password: (type your password)
sh-3.2# echo '/Applications/MAMP/Library/bin' >> /etc/paths.d
➥/MAMP
sh-3.2# exit
```

What to Type

The *Machine*:~ *user*$ portion (where *Machine* is your computer's name) represents the prompt that's already displayed. You only need to type the command, which is shown in bold.

   - If you're running Mac OS X 10.4 (Tiger) or earlier, type these commands:

```
Machine:~ user$ touch .profile
Machine:~ user$ open .profile
```

   This should open the hidden **.profile** file in TextEdit. This file contains a list of Terminal commands that are executed automatically whenever you open a new Terminal window. If you've never installed command prompt programs on your system before, this file will be completely empty. In any case, add this line to the end of the file:

```
export PATH=$PATH:/Applications/MAMP/Library/bin
```

   Save your changes, and quit TextEdit.

2. Close the Terminal window to allow this change to take effect.

[8] To open a Terminal window, launch the **Terminal** application, which you can find in the **Utilities** folder in the **Applications** folder.

Installing Individual Packages

Installing each individual package separately is really the way to go if you can afford to take the time. You gain the opportunity to learn how all the pieces fit together, and you have the freedom to update each of the packages independently of the others. Besides, it's always worthwhile being familiar with the inner workings of any software with which you'll be spending a lot of time.

The following instructions assume you're running Mac OS X 10.5 (Leopard) or later. If you're running an earlier version of Mac OS OX, you should stick with the all-in-one option.

Installing MySQL

Apple maintains a fairly comprehensive guide to installing MySQL on Mac OS X on its Mac OS X Internet Developer site[9] if you want to compile MySQL yourself. It's much easier, however, to obtain the precompiled binary version directly from the MySQL web site.

Start by visiting the The MySQL Downloads page.[10] Click the **Download** link for the free MySQL Community Server. This will take you to a page with a long list of download links for the current recommended version of MySQL (as of this writing, it's MySQL 5.1).

Click the **Mac OS X (package format)** link. You will be presented with the list of downloads shown in Figure 1.19. Which one you need to choose depends on your operating system version and platform architecture. If your system is running Mac OS X version 10.5 (Leopard), you can ignore the Mac OS X 10.4 links. If you know your Mac has a 64-bit processor, you can safely pick the **Mac OS X 10.5 (x86_64)** version. If you're at all unsure, your best bet is the **Mac OS X 10.5 (x86)** version—all it requires is that you have an Intel-based Mac (to be sure, check the processor information in the **About This Mac** window, which you can access from the Apple menu). If you have an older, PowerPC-based Mac, you'll need one of the PowerPC versions. The 32-bit version is the safe bet, since it will run on 64-bit systems too.

[9] http://developer.apple.com/internet/macosx/osdb.html
[10] http://dev.mysql.com/downloads/

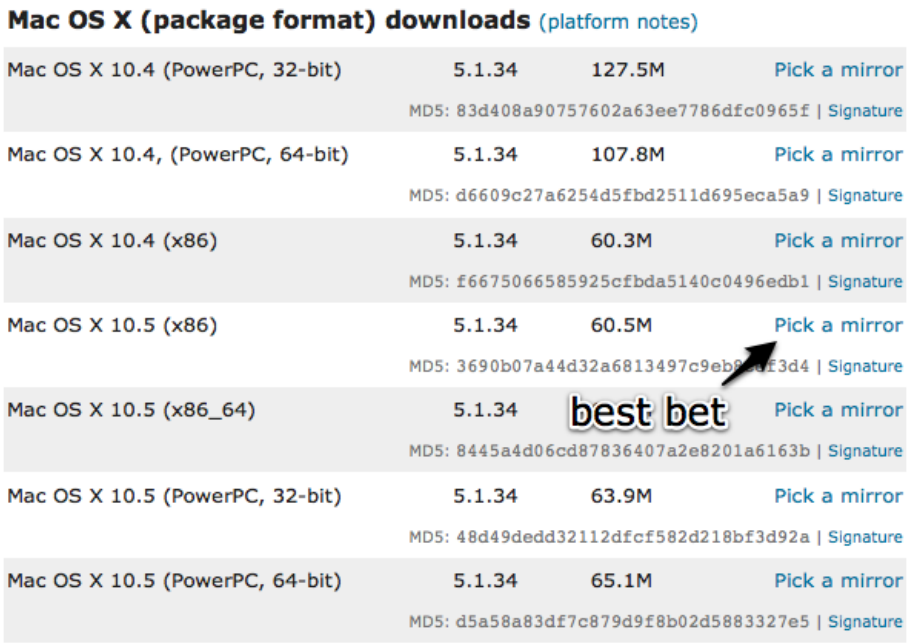


Figure 1.19. The 32-bit version of MySQL for Intel processors will work on most current Macs

Once you've downloaded the **mysql-*version*-osx*version*-*platform*.dmg** file, double-click it to mount the disk image. As shown in Figure 1.20, it contains the installer in **.pkg** format, as well as a **MySQLStartupItem.pkg** file. Double-click the installer, which will guide you through the installation of MySQL.

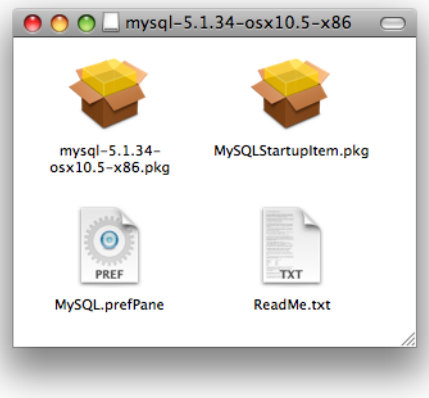


Figure 1.20. The MySQL Mac OS X package contains lots of goodies

Once MySQL is installed, you can launch the MySQL server. Open a Terminal window[11] and type this command:

```
Machine:~ user$ sudo /usr/local/mysql/bin/mysqld_safe
```

What to Type

The `Machine:~ user$` portion (where *Machine* is your computer's name) represents the prompt that's already displayed. You only need to type the command, which is shown in bold.

Once you have typed the command, hit **Enter**.

This command runs the **mysqld_safe** script with administrator privileges. You'll be prompted to input your password to do this, then a status message will confirm that MySQL is running.

Once MySQL is running, you can switch it to background execution by typing **Ctrl+Z** to stop the process, and then typing this command to let it continue running in the background:

```
Machine:~ user$ bg
```

You can then quit the Terminal application and MySQL will continue to run as a server on your system. When you want to shut down the MySQL server, open a new Terminal window and type this command:

```
Machine:~ user$ sudo /usr/local/mysql/bin/mysqladmin shutdown
```

Though you'll gain plenty of geek cred for memorizing these commands, there's a much less tedious way to control your MySQL server. Back in the installation disk image shown in Figure 1.20, you'll notice a file named **MySQL.prefPane**. Double-click this to install a new pane in Mac OS X's System Preferences, and the window shown in Figure 1.21 will open.

[11] To open a Terminal window, launch the **Terminal** application, which you can find in the **Utilities** folder in the **Applications** folder.

Figure 1.21. The MySQL System Preferences pane

This window will tell you if your MySQL server is running or not, and lets you start it up and shut it down with the click of a button!

Presumably, you'll want your system to launch the MySQL server at startup automatically so that you can avoid having to repeat the above process whenever you restart your system. The system preferences pane has a checkbox that does this, but for this checkbox to do anything you must first install the **MySQLStartupItem.pkg** from the installation disk image.

When you have everything set up the way you want it, you can safely drag the MySQL installation disk icon on your desktop to the trash, then delete the **.dmg** file you downloaded.

One last task you'll want to do is add the **/usr/local/mysql/bin** directory to your system path. Doing this enables you to run programs like **mysqladmin** and **mysql** (for which we'll have plenty of use later in this book) in the Terminal without typing out their full paths. Pop open a new Terminal window and type these commands:

```
Machine:~ user$ sudo su
Password: (type your password)
sh-3.2# echo '/usr/local/mysql/bin' >> /etc/paths.d/mysql
sh-3.2# exit
```

Close the Terminal window and open a new one to allow this change to take effect. Then, with your MySQL server running, try running the **mysqladmin** program from your home directory:

```
Machine:~ user$ mysqladmin status
```

If everything worked the way it's supposed to, you should see a brief list of statistics about your MySQL server.

Installing PHP

Mac OS X 10.5 (Leopard) comes with Apache 2.2 and PHP 5 built right in! All you need to do to use them for development is switch them on:

1. Open System Preferences (**System Preferences...** on the Apple menu).
2. In the main System Preferences menu, click **Sharing** under **Internet & Network**.
3. Make sure that **Web Sharing** is checked, as shown in Figure 1.22.

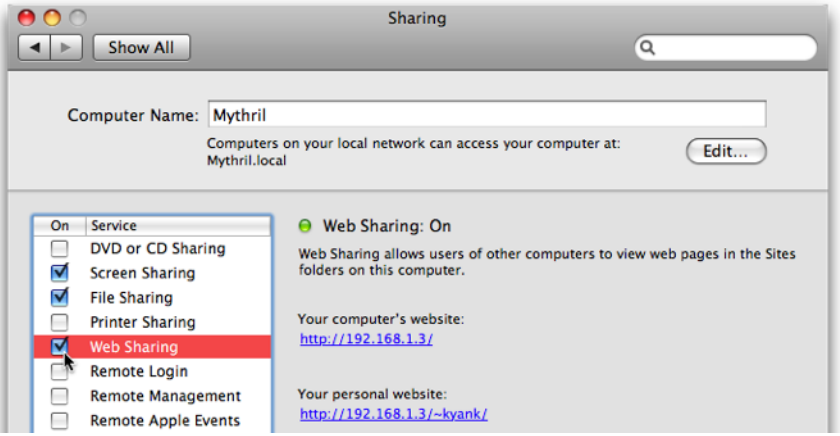


Figure 1.22. Enable Web Sharing in Mac OS X

4. Quit System Preferences.
5. Open your browser, type http://localhost into the address bar, and hit **Enter**. Your browser should display the standard Apache welcome message shown in Figure 1.23.

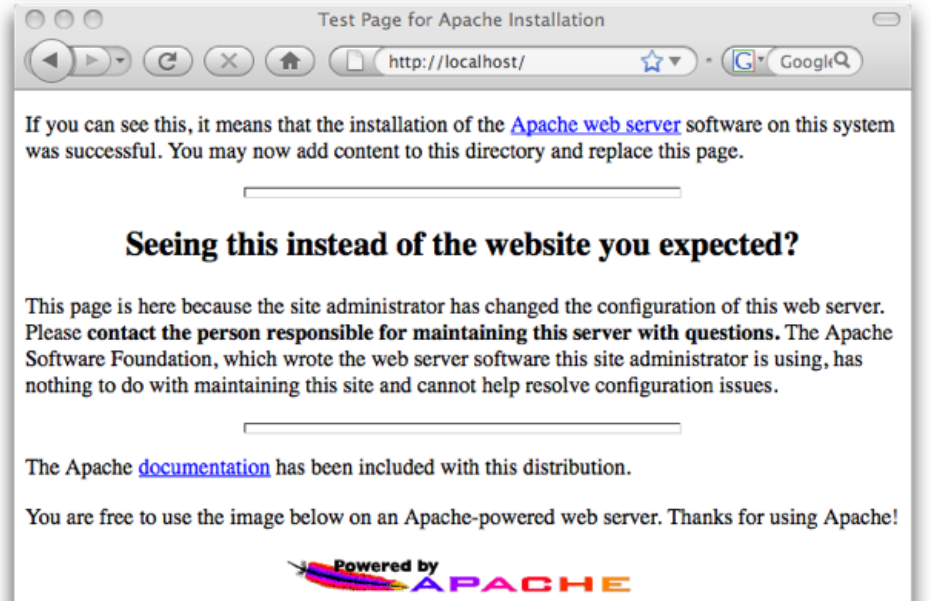


Figure 1.23. The standard Apache welcome page

With this procedure complete, Apache will be run at startup automatically on your system. You're now ready to enhance this server by enabling PHP support:

1. In the Finder menu bar, choose **Go** > **Go to folder** (⇧+⌘+**G**), and type **/private/etc/apache2/** before clicking **Go**.

2. In the Finder window that opens, there should be a file named **httpd.conf**. This is the Apache configuration file. By default, it's read-only. Right-click the file and choose **Get Info** (⌘+**I**) to open the file's properties. Scroll down to the bottom of the **httpd.conf Info** window to find the **Sharing & Permissions** setting.

   By default, the settings in this section are disabled. Click the little lock icon shown in Figure 1.24 to enable them. Enter your password when prompted.

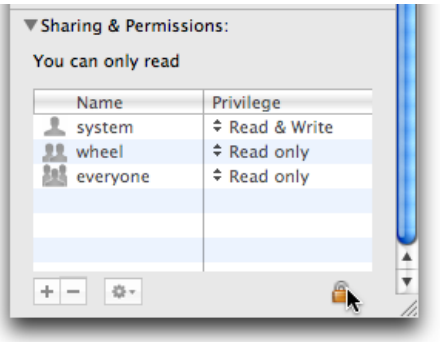


Figure 1.24. Click the lock to make changes to these settings

To make this file editable, change the value in the **Privilege** column for **everyone** to **Read & Write**, as shown in Figure 1.25.

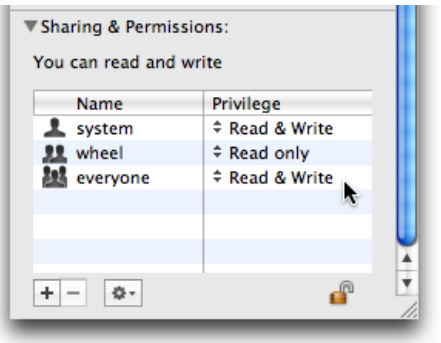


Figure 1.25. Set the permissions for **everyone** to **Read & Write**

3. Back in the Finder window for the **apache2** folder, right-click in the background of the folder window and choose **Get Info** to open the folder's properties. As in the previous step, set the **Sharing & Permissions** settings from **everyone** to **Read & Write**.

4. Finally, double-click the `httpd.conf` file to open it in TextEdit.

5. In the **httpd.conf** file, search for this line:

```
#LoadModule php5_module        libexec/apache2/libphp5.so
```

Enable this command by deleting the hash (`#`) character at the start of the line.

6. Save your changes, and quit TextEdit.

7. If you like to tidy up after yourself, you can go back and reset the privileges on the **httpd.conf** file and the **apache2** folder. This will keep other users of your computer from making changes to the Apache configuration.

8. Open a Terminal window and type this command to restart Apache:

```
Machine:~ user$ sudo /usr/sbin/apachectl restart
```

   Type your password when prompted.

9. Load http://localhost in your browser again to make sure that Apache is still running.

Your computer is now equipped with an Apache web server with PHP support. If you need to make changes to Apache's configuration, you know how to edit its **httpd.conf** file using the instructions above. The PHP plugin, however, has its own configuration file, named **php.ini**, and you need to edit that file to tell PHP how to connect to your MySQL server.

With the version of PHP built into Mac OS X, there is no **php.ini** file by default—PHP just runs with the default settings. In order to modify those settings, you'll need to open Terminal and copy the **/private/etc/php.ini.default** file to **/private/etc/php.ini**:

```
Machine:~ user$ cd /private/etc
Machine:etc user$ sudo cp php.ini.default php.ini
Password: (type your password)
```

To make this new **php.ini** file editable by users like yourself, use the same procedure described above for editing **httpd.conf**: in Finder use **Go** > **Go to folder** to open **/private/etc**, modify the permissions of both the **php.ini** file and the folder that contains it, then open the file with TextEdit.

Scroll down through the file or use **Edit** > **Find** > **Find...** (⌘+**F**) to locate the `mysql.default_socket` option. Edit this line of the **php.ini** file so that it looks like this:

```
mysql.default_socket = /tmp/mysql.sock
```

You should only have to add the portion in bold.

Scroll down further to locate the `mysqli.default_socket` option (`mysqli`, not `mysql`), and make the same change:

```
mysqli.default_socket = /tmp/mysql.sock
```

Save your changes, quit TextEdit, and restore the file and directory permissions if you want to. Finally, open a Terminal window and type this command to restart Apache once more:

```
Machine:~ user$ sudo /usr/sbin/apachectl restart
```

Type your password when prompted. Once Apache is up and running again, load http://localhost in your browser once more to make sure that all is well.

That's it! With MySQL, Apache, and PHP installed, you're ready to proceed to the section called "Post-Installation Set-up Tasks".

Linux Installation

This section will show you the procedure for manually installing Apache, PHP, and MySQL under most current distributions of Linux. These instructions were tested under Ubuntu 8.10;[12] however, they should work on other distributions such as Fedora,[13] Debian,[14] openSUSE,[15] and Gentoo[16] without much trouble. The steps involved will be very similar, almost identical.

Most Linux distributions come with a **package manager** of one kind or another. Ubuntu's Synaptic Package Manager[17] is a graphical front end to APT,[18] the Debian package manager. Other distributions use the older RPM package manager. Regardless of which distribution you use, prepackaged versions of Apache, PHP, and MySQL should be readily available. These prepackaged versions of software are really easy

[12] http://www.ubuntu.com
[13] http://fedoraproject.org
[14] http://www.debian.org
[15] http://www.opensuse.org
[16] http://www.gentoo.org
[17] https://help.ubuntu.com/community/SynapticHowto
[18] http://www.debian.org/doc/user-manuals#apt-howto

to install; unfortunately, they also limit the software configuration options available to you. For this reason—and because any attempt to document the procedures for installing the packaged versions across all popular Linux distributions would be doomed to failure—I will instead show you how to install them manually.

If you already have Apache, PHP, and MySQL installed in packaged form, feel free to use those versions, and skip forward to the section called "Post-Installation Set-up Tasks". If you encounter any problems, you can always uninstall the packaged versions and return here to install them by by hand.

Installing MySQL

Start by downloading MySQL. Simply proceed to the MySQL Downloads page[19] and click the **Download** link for the free MySQL Community Server. This will take you to a page with a long list of download links for the current recommended version of MySQL (as of this writing, it's MySQL 5.1).

Click the link near the top of the list to go to the **Linux (non RPM packages)**. Now you need to choose the package that corresponds to your system architecture. If you're positive you're running a 64-bit version of Linux, go ahead and download the **Linux (AMD64/Intel EM64T)** package (about 120MB in size). If you're running a 32-bit version of Linux, download the **Linux (x86)** package (about 115MB)—it'll work even if it turns out you're running a 64-bit version of Linux. It may be a little unclear, but the **Pick a mirror** link shown in Figure 1.26 is the one you need to click to download the file.

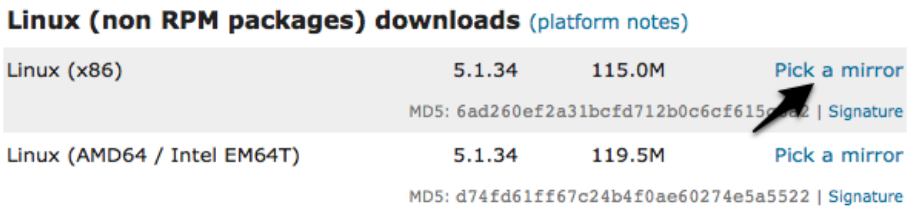


Figure 1.26. Finding the right link can be tricky—here it is!

Once you've downloaded the file, open a Terminal and log in as the root user:

```
user@machine:~$ sudo su
```

[19] http://dev.mysql.com/downloads/

You will, of course, be prompted for your password.

Change directories to **/usr/local** and unpack the downloaded file:

```
root@machine:/home/user# cd /usr/local
root@machine:/usr/local# tar xfz ~user/Desktop/mysql-version-linux-
➥platform.tar.gz
```

The second command assumes you left the downloaded file on your desktop, which is the **Desktop** directory in your home directory. You'll need to replace *user* with your username, *version* with the MySQL version you downloaded, and *platform* with the architecture and compiler version of the release you downloaded; this is so that the command exactly matches the path and filename of the file you downloaded. On my computer, for example, the exact command looks like this:

```
root@mythril:/usr/local# tar xfz ~kyank/Desktop/mysql-5.1.34-linux-x
➥86_64-glibc23.tar.gz
```

After a minute or two, you'll be returned to the command prompt. A quick `ls` will confirm that you now have a directory named **mysql-*version*-linux-*platform***. This is what it looks like on my computer:

```
root@mythril:/usr/local# ls
bin  games    lib  mysql-5.1.34-linux-x86_64-glibc23  share
etc  include  man  sbin                               src
```

Next, create a symbolic link to the new directory with the name **mysql** to make accessing the directory easier. Then enter the directory:

```
root@machine:/usr/local# ln -s mysql-version-linux-platform mysql
root@machine:/usr/local# cd mysql
```

While you can run the server as the root user, or even as yourself (if, for example, you were to install the server in your home directory), you should normally set up on the system a special user whose sole purpose is to run the MySQL server. This will remove any possibility of an attacker using the MySQL server as a way to break into the rest of your system. To create a special MySQL user, type the following commands (still logged in as root):

```
root@machine:/usr/local/mysql# groupadd mysql
root@machine:/usr/local/mysql# useradd -g mysql mysql
```

Now give ownership of your MySQL directory to this new user:

```
root@machine:/usr/local/mysql# chown -R mysql .
root@machine:/usr/local/mysql# chgrp -R mysql .
```

MySQL is now installed, but before it can do anything useful, its database files need to be installed, too. Still in the new **mysql** directory, type the following command:

```
root@machine:/usr/local/mysql# scripts/mysql_install_db --user=mysql
```

Now everything's prepared for you to launch the MySQL server for the first time. From the same directory, type the following command:

```
root@machine:/usr/local/mysql# bin/mysqld_safe --user=mysql &
```

If you see the message `mysql daemon ended`, then the MySQL server was prevented from starting. The error message should have been written to a file called ***hostname*.err** (where *hostname* is your machine's host name) in MySQL's **data** directory. You'll usually find that this happens because another MySQL server is already running on your computer.

If the MySQL server was launched without complaint, the server will run (just like your web or FTP server) until your computer is shut down. To test that the server is running properly, type the following command:

```
root@machine:/usr/local/mysql# bin/mysqladmin -u root status
```

A little blurb with some statistics about the MySQL server should be displayed. If you receive an error message, check the ***hostname*.err** file to see if the fault lies with the MySQL server upon starting up. If you retrace your steps to make sure you followed the process described above, and this fails to solve the problem, a post to the SitePoint Forums[20] will help you pin it down in little time.

[20] http://www.sitepoint.com/forums/

If you want your MySQL server to run automatically whenever the system is running, you'll have to set it up to do so. In the **support-files** subdirectory of the **mysql** directory, you'll find a script called **mysql.server** that can be added to your system startup routines to do this. For most versions of Linux, you can do this by creating a link to the **mysql.server** script in the **/etc/init.d** directory, then create two links to that: **/etc/rc2.d/S99mysql** and **/etc/rc0.d/K01mysql**. Here are the commands to type:

```
root@machine:/usr/local/mysql# cd /etc
root@machine:/etc# ln -s /usr/local/mysql/support-files/mysql.server
➥ init.d/
root@machine:/etc# ln -s /etc/init.d/mysql.server rc2.d/S99mysql
root@machine:/etc# ln -s /etc/init.d/mysql.server rc0.d/K01mysql
```

That's it! To test that this works, reboot your system, and request the status of the server with `mysqladmin` as you did above.

One final thing you might like to do for the sake of convenience is to place the MySQL client programs—which you'll use to administer your MySQL server later on—in the system path. To this end, you can place symbolic links to **mysql**, **mysqladmin**, and **mysqldump** in your **/usr/local/bin** directory:

```
root@machine:/etc# cd /usr/local/bin
root@machine:/usr/local/bin# ln -s /usr/local/mysql/bin/mysql .
root@machine:/usr/local/bin# ln -s /usr/local/mysql/bin/mysqladmin .
root@machine:/usr/local/bin# ln -s /usr/local/mysql/bin/mysqldump .
```

Once you've done this, you can log out of the root account. From this point on, you can administer MySQL from any directory on your system:

```
root@machine:/usr/local/bin# exit
user@machine:~$ mysqladmin -u root status
```

Installing PHP

As mentioned above, PHP is more a web server plugin module than a program. There are actually three ways to install the PHP plugin for Apache:

- as a CGI program that Apache runs every time it needs to process a PHP-enhanced web page
- as an Apache module compiled right into the Apache program
- as an Apache module loaded by Apache each time it starts up

The first option is the easiest to install and set up, but it requires Apache to launch PHP as a program on your computer every time a PHP page is requested. This activity can really slow down the response time of your web server, especially if more than one request needs to be processed at a time.

The second and third options are almost identical in terms of performance, but the third option is the most flexible, since you can add and remove Apache modules without having to recompile it each time. For this reason, we'll use the third option.

Assuming you don't already have Apache running on your computer, surf on over to the Apache HTTP Server Project[21] and look for the version of Apache described as "the best available version" (as of this writing it's version 2.2.11, as shown in Figure 1.27).

Figure 1.27. The best available version—accept no substitutes!

[21] http://httpd.apache.org/

Once you get to the Download page, scroll down to find the links to the various versions available. The one you want is **Unix Source**, shown in Figure 1.28. Both the **.tar.gz** or the **.tar.bz2** are the same; just grab whichever archive format you're used to extracting.

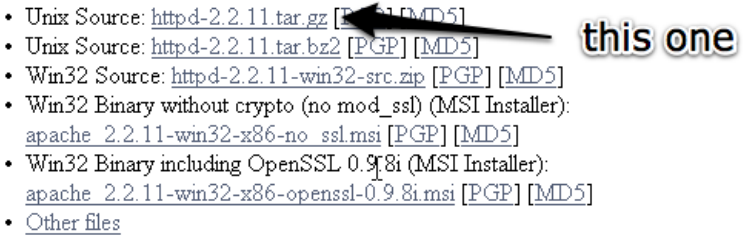


Figure 1.28. This is the one you need

What you've just downloaded is actually the source code for the Apache server. The first step, then, is to compile it into an executable binary installation. Pop open a Terminal, navigate to the directory where the downloaded file is located, then extract it, and navigate into the resulting directory:

```
user@machine:~$ cd Desktop
user@machine:~/Desktop$ tar xfz httpd-version.tar.gz
user@machine:~/Desktop$ cd httpd-version
```

The first step in compiling Apache is to configure it to your requirements. Most of the defaults will be fine for your purposes, but you'll need to enable dynamic loading of Apache modules (like PHP), which is off by default. Additionally, you should probably enable the URL rewriting feature, upon which many PHP applications rely (although it's unnecessary for the examples in this book). To make these configuration changes, type this command:

```
user@machine:~/Desktop/httpd-version$ ./configure --enable-so --enab
➥le-rewrite
```

A long stream of status messages will parade up your screen. If the process stops with an error message, your system may be missing some critical piece of software that's required to compile Apache. Some Linux distributions lack the essential development libraries or even a C compiler installed by default. Installing these should enable you to return and run this command successfully. Current versions of Ubuntu, however, should come with everything that's needed.

After several minutes, the stream of messages should come to an end:

```
⋮
config.status: creating build/rules.mk
config.status: creating build/pkg/pkginfo
config.status: creating build/config_vars.sh
config.status: creating include/ap_config_auto.h
config.status: executing default commands
user@machine:~/Desktop/httpd-version$
```

You're now ready to compile Apache. The one-word command `make` is all it takes:

```
user@machine:~/Desktop/httpd-version$ make
```

Again, this process will take several minutes to complete, and should end with the following message:

```
⋮
make[1]: Leaving directory `/home/user/Desktop/httpd-version'
user@machine:~/Desktop/httpd-version$
```

To install your newly-compiled copy of Apache, type `sudo make install` (the `sudo` is required, since you need root access to write to the installation directory).

```
user@machine:~/Desktop/httpd-version$ sudo make install
```

Enter your password when prompted.

As soon as this command has finished copying files, your installation of Apache is complete. Navigate to the installation directory and launch Apache using the `apachectl` script:

```
user@machine:~/Desktop/httpd-version$ cd /usr/local/apache2
user@machine:/usr/local/apache2$ sudo bin/apachectl -k start
```

You'll likely see a warning message from Apache complaining that it was unable to determine the server's fully qualified domain name. That's because most personal computers are without one. Don't sweat it.

Fire up your browser and type http://localhost into the address bar. If Apache is up and running, you should see a welcome message like the one in Figure 1.29.

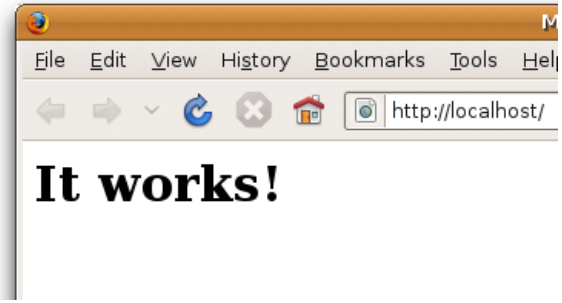


Figure 1.29. You can take my word for it!

As with your MySQL server, you'll probably want to configure Apache to start automatically when your system boots. The procedure to do this is similar; just copy and link the **apachectl** script from your Apache installation:

```
user@machine:/usr/local/apache2$ sudo su
root@machine:/usr/local/apache2# cd /etc
root@machine:/etc# ln -s /usr/local/apache2/bin/apachectl init.d/
root@machine:/etc# ln -s /etc/init.d/apachectl rc2.d/S99httpd
root@machine:/etc# ln -s /etc/init.d/apachectl rc0.d/K01httpd
```

To test that this works, restart your computer and then hit the http://localhost page in your browser again.

With a shiny new Apache installation up and running, you're now ready to add PHP support to it. To start, download the PHP Complete Source Code package from the PHP Downloads page.[22] Again, the **.tar.gz** and **.tar.bz2** versions are identical; just download whichever you're used to extracting.

The file you downloaded should be called **php-*version*.tar.gz** (or **.bz2**). Pop open a new Terminal window, navigate to the directory containing the downloaded file, extract it, and move into the resulting directory:

[22] http://www.php.net/downloads.php

```
user@machine:~$ cd Desktop
user@machine:~/Desktop$ tar xfz php-version.tar.gz
user@machine:~/Desktop$ cd php-version
```

To install PHP as an Apache module, you'll need to use the Apache **apxs** program. This will have been installed along with the Apache server if you followed the instructions above to compile it yourself; but if you're using the copy that was installed with your distribution of Linux, you may need to install the Apache development package to access Apache **apxs**. You should be able to install this package by using the package manager included with your Linux distribution. For example, on Debian Linux, you can use `apt-get` to install it as follows:

```
user@machine:~$ sudo apt-get install apache-dev
```

Now, to install PHP, you must be logged in as root:

```
user@machine:~/Desktop/php-version$ sudo su
[sudo] password for user: (type your password)
root@machine:/home/user/Desktop/php-version#
```

The first step is to configure the PHP installation program by telling it which options you want to enable, and where it should find the programs it needs to know about (such as Apache **apxs** and MySQL). The command should look like this (all on one line):

```
root@machine:/home/user/Desktop/php-version# ./configure
➥ --prefix=/usr/local/php --with-apxs2=/usr/local/apache2/bin/apxs
➥ --with-mysqli=/usr/local/mysql/bin/mysql_config
```

The `--prefix` option tells the installer where you want PHP to be installed (**/usr/local/php** is a good choice).

The `--with-apxs2` option tells the installer where to find the Apache **apxs** program mentioned above. When installed using your Linux distribution's package manager, the program is usually found at **/usr/sbin/apxs**. If you compiled and installed Apache yourself as described above, however, it will be in the Apache binary directory, at **/usr/local/apache2/bin/apxs**.

The `--with-mysqli` option tells the installer where to find your MySQL installation. More specifically, it must point to the **mysql_config** program in your MySQL installation's **bin** directory (**/usr/local/mysql/bin/mysql_config**).

Again, a parade of status messages will appear on your screen. When it stops, check for any error messages and install any files it identifies as missing. On a default Ubuntu 8.10 installation, for example, you're likely to see an error complaining about an incomplete libxml2 installation. To correct this particular error, open Synaptic Package Manager, then locate and install the **libxml2-dev** package (**libxml2** should already be installed). Once it's installed, try the `configure` command again.

After you watch several screens of tests scroll by, you'll be returned to the command prompt with the comforting message "`Thank you for using PHP.`" The following two commands will compile and then install PHP:

```
root@machine:/home/user/Desktop/php-version# make
root@machine:/home/user/Desktop/php-version# make install
```

Take a coffee break: this will take some time.

Upon completion of the `make install` command, PHP will be installed in **/usr/local/php** (unless you specified a different directory with the `--prefix` option of the **configure** script above). Now you just need to configure it!

The PHP configuration file is called **php.ini**. PHP comes with two sample **php.ini** files called **php.ini-dist** and **php.ini-recommended**. Copy these files from your installation work directory to the **/usr/local/php/lib** directory, then make a copy of the **php.ini-dist** file and call it **php.ini**:

```
root@machine:/home/user/Desktop/php-version# cp php.ini* /usr/local/
➥php/lib/
root@machine:/home/user/Desktop/php-version# cd /usr/local/php/lib
root@machine:/usr/local/php/lib# cp php.ini-dist php.ini
```

You may now delete the directory from which you compiled PHP—it's no longer needed.

We'll worry about fine-tuning **php.ini** shortly. For now, we need to tweak Apache's configuration to make it more PHP-friendly. Locate your Apache **httpd.conf** config-

uration file. This file can usually be found in the **conf** subdirectory of your Apache installation (**/usr/local/apache2/conf/httpd.conf**).

To edit this file you must be logged in as root, so launch your text editor from the Terminal window where you're still logged in as root:

```
root@machine:/usr/local/php/lib# cd /usr/local/apache2/conf
root@machine:/usr/local/apache2/conf# gedit httpd.conf
```

In this file, look for the line that begins with `DirectoryIndex`. This line tells Apache which filenames to use when it looks for the default page for a given directory. You'll see the usual **index.html**, but you need to add **index.php** to the list:

```
<IfModule dir_module>
  DirectoryIndex index.html index.php
</IfModule>
```

Finally, go right to the bottom of the file and add these lines to tell Apache that files with names ending in **.php** should be treated as PHP scripts:

```
<FilesMatch \.php$>
  SetHandler application/x-httpd-php
</FilesMatch>
```

That should do it! Save your changes and restart your Apache server with this command:

```
root@machine:/usr/local/apache2/conf# /usr/local/apache2/bin/
➥apachectl -k restart
```

If it all goes according to plan, Apache should start up without any error messages. If you run into any trouble, the helpful individuals in the SitePoint Forums[23] (myself included) will be happy to help.

[23] http://www.sitepoint.com/forums/

Post-Installation Set-up Tasks

Regardless of which operating system you're running, or how you set up your web server—once PHP is installed and the MySQL server is functioning, the very first action you need to perform is assign a **root password** for MySQL.

MySQL only allows authorized users to view and manipulate the information stored in its databases, so you'll need to tell MySQL who's authorized and who's unauthorized. When MySQL is first installed, it's configured with a user named `root` that has access to do most tasks without even entering a password. Your first task should be to assign a password to the `root` user so that unauthorized users are prohibited from tampering with your databases.

Why Bother?

It's important to realize that MySQL, just like a web server, can be accessed from any computer on the same network. If you're working on a computer connected to the Internet, then, depending on the security measures you've taken, anyone in the world could connect to your MySQL server. The need to pick a difficult-to-guess password should be immediately obvious!

To set a root password for MySQL, you can use the **mysqladmin** program that comes with MySQL. If you followed the instructions to install MySQL separately (as explained earlier in this chapter), the **mysqladmin** program should be on your system path. This means you can pop open a Terminal window (or in Windows, a Command Prompt) and type the name of the program without having to remember where it's installed on your computer.

Go ahead and try this now, if you've yet to already. Open a Terminal or Command Prompt and type this command:[24]

```
mysqladmin -u root status
```

When you hit **Enter** you should see a line or two of basic statistics about your MySQL server, like this:

[24] If you're using Windows and are unfamiliar with the Command Prompt, check out my article *Kev's Command Prompt Cheat Sheet* [http://www.sitepoint.com/article/command-prompt-cheat-sheet/] for a quick crash course.

```
Uptime: 102261  Threads: 1  Questions: 1  Slow queries: 0  Opens: 15
 Flush tables: 1  Open tables: 0  Queries per second avg: 0.0
```

If you're seeing a different message entirely, it's probably one of two options. First, you might see an error message telling you that the **mysqladmin** program was unable to connect to your MySQL server:

```
mysqladmin: connect to server at 'localhost' failed
error: 'Can't connect to MySQL server on 'localhost' (10061)'
Check that mysqld is running on localhost and that the port is 3306.
You can check this by doing 'telnet localhost 3306'
```

This message normally means that your MySQL server simply isn't running. If you have it set up to run automatically when your system boots, double-check that the setup is working. If you normally launch your MySQL server manually, go ahead and do that before trying the command again.

Second, if you're using MAMP on the Mac, you'll probably see this error message instead:

```
mysqladmin: connect to server at 'localhost' failed
error: 'Access denied for user 'root'@'localhost' (using password: N
➥O)'
```

This error message means that the `root` user on your MySQL server already has a password set. It turns out that, with your security in mind, MAMP comes with a root password already set on its built-in MySQL server. That password, however, is *root*—so you're probably still going to want to change it using the instructions below.

One way or the other, you should now be able to run the **mysqladmin** program. Now you can use it to set the root password for your MySQL server:

```
mysqladmin -u root -p password "newpassword"
```

Replace *newpassword* with whatever password you'd like to use for your MySQL server. Make sure it's one you can remember, because if you forget your MySQL root password, you might need to erase your entire MySQL installation and start

over from scratch! As we'll see in Chapter 10, it's usually possible to recover from such a mishap, but it's definitely a pain in the neck.

Here's a spot for you to record your MySQL root password in case you need to:

My MySQL Root Password

`root` user password: _____

When you hit **Enter**, you'll be prompted to enter the current password for the `root` MySQL user. Just hit **Enter** again, since the `root` user has no password at this point, unless you've used MAMP to set up MySQL on your Mac; in this case you should type **`root`**, the default `root` MySQL password on MAMP.

Let me break this command down for you, so you can understand what each part means:

`mysqladmin`
: This, of course, is the name of the program you wish to run.

`-u root`
: This specifies the MySQL user account you wish to use to connect to your MySQL server. On a brand new server, there is only one user account: `root`.

`-p`
: This tells the program to prompt you for the current password of the user account. On a brand new MySQL server, the `root` account has no password, so you can just hit **Enter** when prompted. It's a good idea, however, to make a habit of including this option, since most of the time you *will* need to provide a password to connect to your MySQL server.

`password "`*`newpassword`*`"`
: This instructs the `mysqladmin` program to change the password of the user account to *newpassword*. In this example, whatever password you specify will become the new password for the `root` MySQL user.

Now, to try out your new password, request once again that the MySQL server tell you its current status at the system command prompt, but this time include the `-p` option:

```
mysqladmin -u root -p status
```

Enter your new password when prompted. As before, you should see a line or two of statistics about your MySQL server.

Since the `root` account is now password-protected, attempting to run this command without the `-p` switch will give you an "Access Denied" error.

You're done! With everything set up and running, you're ready to write your first PHP script. Before we do that, however, you might want to write a short email to your web host.

What to Ask Your Web Host

While you tinker with PHP and MySQL on your own computer, it might be good to start collecting the information you'll need when it comes time to deploy your first database driven web site to the public. Here's a rundown of the details you should be asking your web host for.

First, you'll need to know how to transfer files to your web host. You'll upload PHP scripts to your host the same way you normally send the HTML files, CSS files, and images that make up a static web site, so if you already know how to do that, it's unnecessary to bother your host. If you're just starting with a new host, however, you'll need to be aware of what file transfer protocol it supports (FTP or SFTP), as well as knowing what username and password to use when connecting with your (S)FTP program. You also have to know what directory to put files into so they're accessible to web browsers.

In addition to these, you'll also need to find out a few details about the MySQL server your host has set up for you. It's important to know the host name to use to connect to it (possibly `localhost`), and your MySQL username and password, which may or may not be the same as your (S)FTP credentials. Your web host will probably also have provided an empty database for you to use, which prevents you from interfering with other users' databases who may share the same MySQL server with you. If they have provided this, you should establish the name of that database.

Have you taken in all that? Here's a spot to record the information you'll need about your web host:

My Hosting Details

File transfer protocol: ☐ FTP ☐ SFTP

(S)FTP host name: _____

(S)FTP username: _____

(S)FTP password: _____

MySQL host name: _____

MySQL username: _____

MySQL password: _____

MySQL database name: _____

Your First PHP Script

It would be unfair of me to help you install everything—but stop short of giving you a taste of what a PHP script looks like until Chapter 3. So here's a little morsel to whet your appetite.

Open your favorite text or HTML editor and create a new file called **today.php**. Type this into the file:

chapter1/today.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Today&rsquo;s Date</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>Today&rsquo;s date (according to this web server) is
      <?php

      echo date('l, F dS Y.');

      ?>
    </p>
  </body>
</html>
```

Editing PHP Scripts in Windows with Notepad

Windows users should note that, to save a file with a `.php` extension in Notepad, you'll need to either select *All Files* as the file type, or surround the filename with quotes in the **Save As** dialog box; otherwise, Notepad will unhelpfully save the file as **today.php.txt**, which will fail to work.

Editing PHP Scripts in Mac OS X with TextEdit

Mac OS X users are advised to be careful when using TextEdit to edit **.php** files, as it saves them in Rich Text Format, with an invisible **.rtf** filename extension by default. To save a new **.php** file, you must first remember to convert the file to plain text by selecting **Format** > **Make Plain Text** (⇧+⌘+**T**) from the TextEdit menu.

TextEdit also has a nasty habit of mistaking existing **.php** files for HTML documents when opening them, and attempting to display them as formatted text. To avoid this, you must select the **Ignore rich text commands** checkbox in the **Open** dialog box.

Try a Free IDE!

As you can tell from the preceding warnings, the text editors provided with current operating systems are a little unsuitable for editing PHP scripts. There are a number of solid text editors and Integrated Development Environments (IDEs) with rich support for editing PHP scripts that you can download for free. Here are a few that work on Windows, Mac OS X, and Linux:

NetBeans	http://www.netbeans.org/features/php/
Aptana	http://www.aptana.com/php
Komodo Edit	http://www.activestate.com/komodo_edit/

If you'd prefer to avoid typing out all the code, you can download this file—along with the rest of the code in this book—from the code archive. See the Preface for details on how to download the code archive.

Save the file, and move it to the **web root** directory of your local web server.

Where's My Server's Web Root Directory?

If you're using an Apache server you installed manually, the web root directory is the **htdocs** directory within your Apache installation (that is, **C:\Program Files\Apache Software Foundation\Apache2.2\htdocs** on Windows, **/usr/local/apache2/htdocs** on Linux).

For Apache servers built into WampServer, the web root directory is the **www** directory within your WampServer directory. You can reach it quickly by selecting the **www directory** menu item from the WampServer menu in your Windows System Tray.

If the Apache server you're using is built into Mac OS X, the web root directory is **/Library/WebServer/Documents**.

The Apache server built into MAMP has a web root directory in the **htdocs** folder inside the MAMP folder (**/Applications/MAMP/htdocs**). If you prefer using a different folder as your web root, you can change it on the **Apache** tab of the MAMP application's Preferences.

Open your web browser of choice, and type http://localhost/today.php (or http://localhost:*port*/today.php if Apache is configured to run on a port other than the default of 80) into the address bar to view the file you just created.[25]

You Must Type the URL

You might be used to previewing your web pages by double-clicking on them, or by using the **File** > **Open**... feature of your browser. These methods tell your browser to load the file directly from your computer's hard drive, and so they'll fail to work with PHP files.

As previously mentioned, PHP scripts require your web server to read and execute the PHP code they contain before sending the HTML code that's generated to the browser. Only if you type the URL (http://localhost/today.php) will your browser request the file from your web server so that this can happen.

Figure 1.30 shows what the web page generated by your first PHP script should look like.

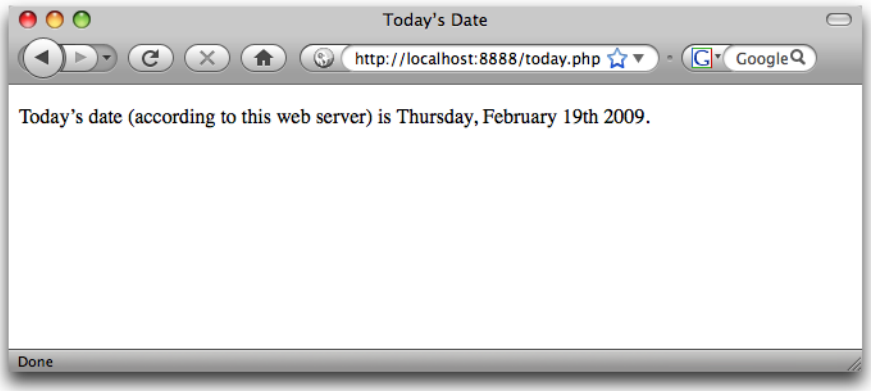


Figure 1.30. See your first PHP script in action!

Neat, huh? If you use the **View Source** feature in your browser, all you'll see is a regular HTML file with the date in it. The PHP code (everything between `<?php` and `?>` in the code above) was interpreted by the web server and converted to normal text before it was sent to your browser. The beauty of PHP, and other server-side

[25] If you installed Apache on Windows, you may have selected the option to run it on port 8080. If you're using MAMP, it's configured by default to run Apache on port 8888.

scripting languages, is that the web browser can remain ignorant—the web server does all the work!

Be reassured also that before too long you'll know code (like this example) as well as the back of your hand.

If the date is missing, or if your browser prompts you to download the PHP file instead of displaying it, then something is wrong with your web server's PHP support. If you can, use **View Source** in your browser to look at the code of the page. You'll probably see the PHP code right there in the page. Since the browser fails to understand PHP, it just sees `<?php ... ?>` as one long, invalid HTML tag, which it ignores. Double-check that you have requested the file from your web server rather than your hard disk (that is, make sure the location bar in your browser shows a URL beginning with http://localhost), and make sure that PHP support has been properly installed on your web server using the instructions provided earlier in this chapter.

Full Toolbox, Dirty Hands

You should now be fully equipped with a web server that supports PHP scripts, a MySQL database server, and a basic understanding of how to use each of these. You should even have dirtied your hands by writing and successfully testing your first PHP script!

If the **today.php** script was unsuccessful for you, drop by the SitePoint Forums[26] and we'll be glad to help you figure out the problem.

In Chapter 2, you'll learn the basics of relational databases and start working with MySQL. I'll also introduce you to the language of database: Structured Query Language. If you've never worked with a database before, it'll be a real eye-opener!

[26] http://www.sitepoint.com/forums/

Chapter 2

Introducing MySQL

In Chapter 1, we installed and set up two software programs: the Apache web server with PHP, and the MySQL database server.

As I explained in that chapter, PHP is a server-side scripting language that lets you insert into your web pages instructions that your web server software (in most cases, Apache) will execute before it sends those pages to browsers that request them. In a brief example, I showed how it was possible to insert the current date into a web page every time it was requested.

Now, that's all well and good, but things *really* become interesting when a database is added to the mix. In this chapter, we'll learn what a database is, and how to work with your own MySQL databases using Structured Query Language.

An Introduction to Databases

A database server (in our case, MySQL) is a program that can store large amounts of information in an organized format that's easily accessible through programming languages like PHP. For example, you could tell PHP to look in the database for a list of jokes that you'd like to appear on your web site.

In this example, the jokes would be stored entirely in the database. The advantages of this approach would be twofold: First, instead of having to write an HTML page for each of your jokes, you could write a single PHP script that was designed to fetch any joke from the database and display it by generating an HTML page for it on the fly. Second, adding a joke to your web site would be a simple matter of inserting the joke into the database. The PHP code would take care of the rest, automatically displaying the new joke along with the others when it fetched the list from the database.

Let's run with this example as we look at how data is stored in a database. A database is composed of one or more **tables**, each of which contains a list of **items**, or *things*. For our joke database, we'd probably start with a table called `joke` that would contain a list of jokes. Each table in a database has one or more **columns**, or **fields**. Each column holds a certain piece of information about each item in the table. In our example, our `joke` table might have one column for the text of the jokes, and another for the dates on which the jokes were added to the database. Each joke stored in this way would then be said to be a **row** or **entry** in the table. These rows and columns form a table that looks like Figure 2.1.

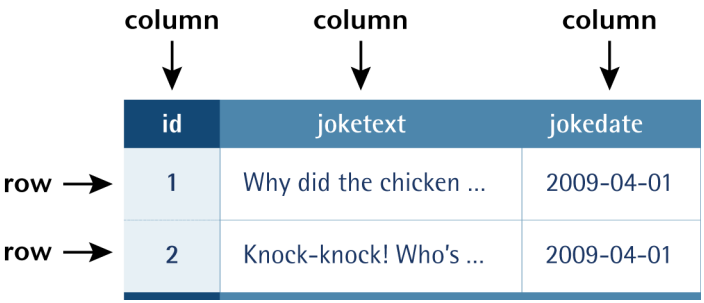


Figure 2.1. A typical database table containing a list of jokes

Notice that, in addition to columns for the joke text (`joketext`) and the date of the joke (`jokedate`), I've included a column named `id`. As a matter of good design, a database table should always provide a means by which we can identify each of its rows uniquely. Since it's possible that a single joke could be entered more than once on the same date, the `joketext` and `jokedate` columns can't be relied upon to tell all the jokes apart. The function of the `id` column, therefore, is to assign a unique number to each joke so that we have an easy way to refer to them and to keep track of which joke is which. We'll take a closer look at database design issues like this in Chapter 5.

So, to review, the table in Figure 2.1 is a three-column table with two rows, or entries. Each row in the table contains three fields, one for each column in the table: the joke's ID, its text, and the date of the joke. With this basic terminology under your belt, you're ready to dive into using MySQL.

Logging On to MySQL

Just as a web server is designed to respond to requests from a client (a web browser), the MySQL database server responds to requests from **client programs**. Later in this book, we'll write our own MySQL client programs in the form of PHP scripts, but for now we can use some of the client programs that come included with the MySQL server.

mysqladmin is an example of a MySQL client program. If you followed the instructions in Chapter 1, after setting up a MySQL server of your own, you used the **mysqladmin** client program to connect to the server, establish a password for the root user, and view basic statistics about the running server.

Another client program that comes with the MySQL server is called **mysql**. This program provides the most basic interface for working with a MySQL server, by establishing a connection to the server and then typing commands one at a time.

The **mysql** program can be found in the same place as **mysqladmin**, so if you followed the instructions in Chapter 1 to add this location to your system path, you should be able to open a Terminal window (or Command Prompt if you're using a Windows system) and type this command to run the **mysql** client program:

```
mysql --version
```

If everything is set up right, this command should output a one-line description of the version of the **mysql** client program that you've installed. Here's what this looks like on my Mac:

```
mysql  Ver 14.14 Distrib 5.1.31, for apple-darwin9.5.0 (i386) using
➥readline 5.1
```

If instead you receive an error message complaining that your computer is unable to recognize the `mysql` command, you should probably revisit the installation instructions provided in Chapter 1. Once you're able to run the `mysqladmin` commands

in that chapter, the `mysql` command should work too. If you're still stuck, drop by the SitePoint Forums[1] and ask for some help.

Assuming the **mysql** program is running for you, you can now use it to connect to your MySQL server. First, make sure that server is running, then type this command and hit **Enter**:

```
mysql -u root -p
```

The *`-u root`* and *`-p`* parameters perform the same function for this program as they did for **mysqladmin** in Chapter 1. *`-u root`* tells the program you wish to connect to the server using the `root` user account, and *`-p`* tells it you're going to provide a password.

What you should see next is an `Enter password:` prompt. Enter the root password you chose for yourself in Chapter 1, and hit **Enter**.

If you typed everything correctly, the MySQL client program will introduce itself and dump you on the MySQL command prompt:

```
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.1.31 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Let's use a few simple commands to take a look around your MySQL server.

The MySQL server can actually keep track of more than one database. This allows a web host to set up a single MySQL server for use by several of its subscribers, for example. So, your first step after connecting to the server should be to choose a database with which to work. First, let's retrieve a list of databases on the current server.

[1] http://www.sitepoint.com/forums/

Connecting to a Remote MySQL Server

The instructions in this chapter assume you're working with a MySQL server running on your own computer. Of course, when it comes time to publish your first PHP-and-MySQL-powered web site, you will need to know how to work with the MySQL server provided by your web host, or by your company's IT department.

Technically, the **mysql** program we're using in this chapter can connect to remote MySQL servers too. You just have to add an additional parameter when running it:

```
mysql -h hostname -u username -p
```

The `-h hostname` parameter (where *hostname* is the host name of the MySQL server to which you want to connect) tells the program to connect to a remote MySQL server instead of one running on the same computer. If you do this, you'll probably also need to specify a username other than `root`, since the administrator responsible for the MySQL server will probably want to keep the `root` password secret for security reasons.

In practice, most remote MySQL servers will block connections from client programs running on untrusted computers like yours. Disallowing this type of connection is a common security measure for MySQL servers used in production.

To work with a remote MySQL server, you might be able to connect to a trusted computer and run the **mysql** program from there, but a far more common approach is to use a program called phpMyAdmin to manage your remote databases. phpMyAdmin is a sophisticated PHP script that lets you work with your MySQL databases using a web-based interface in your browser. phpMyAdmin connects to the remote MySQL server in the same way as the PHP scripts we'll be writing later in this book.

I'll show you how to install and use phpMyAdmin in Chapter 10. For now, let's focus on learning to work with the MySQL server you've installed on your computer.

Type this command (including the semicolon!) and press **Enter**:[2]

```
mysql> SHOW DATABASES;
```

MySQL will show you a list of the databases on the server. If you're working on a brand new server, the list should look like this:

```
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| test               |
+--------------------+
3 rows in set (0.00 sec)
```

The MySQL server uses the first database, named `information_schema`, to keep track of all the other databases on the server. Unless you're doing some very advanced stuff, you'll probably leave this database alone.

The second database, `mysql`, is special too. MySQL uses it to keep track of users, their passwords, and what they're allowed to do. We'll steer clear of this for now, though we'll revisit it in Chapter 10, when we discuss MySQL administration.

The third database, named `test`, is a sample database. You can actually delete this database because I'll show you how to create your own database in a moment.

No `test` on WampServer

As of this writing, WampServer's initial MySQL database has no `test` database in it. No need to be alarmed though; the developers of WampServer just thought it was as useless as I do, I guess!

Deleting stuff in MySQL is called "dropping" it, and the command for doing so is appropriately named:

```
mysql> DROP DATABASE test;
```

[2] As in Chapter 1, the `mysql>` prompt should already be visible on your screen; just type the command that comes after it.

If you type this command and press **Enter**, MySQL will obediently delete the database, displaying "`Query OK`" in confirmation. Notice that there's no confirmation prompt like "Are you sure?". You have to be very careful to type your commands correctly in the **mysql** client program because, as this example shows, you can obliterate your entire database—along with all the information it contains—with a single command!

Before we go any further, let's learn a couple of fundamentals about the MySQL command prompt. As you may have noticed, all commands in MySQL are terminated by a semicolon (`;`). If you forget the semicolon, MySQL will think you're still typing your command, and will let you continue on another line:

```
mysql> SHOW
    -> DATABASES;
```

MySQL shows that it's waiting for you to type more of your command by changing the prompt from `mysql>` to `->`. This handy feature allows you to spread long commands over several lines.

Case Sensitivity in SQL Queries

Most MySQL commands are not case-sensitive, which means you can type `SHOW DATABASES`, `show databases`, or `ShOw DaTaBaSeS`, and it will know what you mean. Database names and table names, however, are case-sensitive when the MySQL server is running on an operating system with a case-sensitive file system (like Linux or Mac OS X, depending on your system configuration).

Also, table, column, and other names must be spelled exactly the same when they're used more than once in the same command.

For consistency, this book will respect the accepted convention of typing database commands in all capitals, and database entities (databases, tables, columns, and so on) in all lowercase.

If you're halfway through a command and realize that you made a mistake early on, you may want to cancel the current command entirely and start over from scratch. To do this, type `\c` and press **Enter**:

```
mysql> DROP DATABASE\c
mysql>
```

MySQL will ignore the command you had begun to type and will return to the `mysql>` prompt to await another command.

Finally, if at any time you want to exit the MySQL client program, just type `quit` or `exit` (either will work). This is the only command where the semicolon is unnecessary, but you can use one if you want to.

```
mysql> quit
Bye
```

Structured Query Language

The set of commands we'll use to direct MySQL throughout the rest of this book is part of a standard called **Structured Query Language**, or **SQL** (pronounced as either "sequel" or "ess-cue-ell"—take your pick). Commands in SQL are also referred to as **queries**; I'll use these two terms interchangeably.

SQL is the standard language for interacting with most databases, so, even if you move from MySQL to a database like Microsoft SQL Server in the future, you'll find that most of the commands are identical. It's important that you understand the distinction between SQL and MySQL. MySQL is the database server software that you're using. SQL is the language that you use to interact with that database.

Learn SQL in Depth

In this book, I'll teach you the essentials of SQL that every PHP developer needs to know.

If you decide to make a career out of building database driven web sites, you'll find that it pays to know some of the more advanced details of SQL, especially when it comes to making your sites run as quickly and smoothly as possible.

If you'd like to dive deeper into SQL, I highly recommend the book *Simply SQL*[3] by Rudy Limeback (Melbourne: SitePoint, 2008).

[3] http://www.sitepoint.com/books/sql1/

Creating a Database

When the time comes to deploy your first database driven web site on the Web, you'll likely find that your web host or IT department has already created a MySQL database for you to use. Since you're in charge of your own MySQL server, however, you'll need to create your own database to use in developing your site.

It's just as easy to create a database as it is to delete one:

```
mysql> CREATE DATABASE ijdb;
```

I chose to name the database `ijdb`, for Internet Joke Database,[4] because that fits with the example I gave at the beginning of this chapter—a web site that displays a database of jokes. Feel free to give the database any name you like, though.

Now that you have a database, you need to tell MySQL that you want to use it. Again, the command is easy to remember:

```
mysql> USE ijdb;
```

You're now ready to use your database. Since a database is empty until you add some tables to it, our first order of business will be to create a table that will hold your jokes (now might be a good time to think of some!).

Creating a Table

The SQL commands we've encountered so far have been reasonably simple, but as tables are so flexible, it takes a more complicated command to create them. The basic form of the command is as follows:

```
mysql> CREATE TABLE table_name (
    ->   column1Name column1Type column1Details,
    ->   column2Name column2Type column2Details,
    ->   ⋮
    -> ) DEFAULT CHARACTER SET charset;
```

[4] With a tip of the hat to the Internet Movie Database. [http://www.imdb.com]

Let's continue with the `joke` table I showed you in Figure 2.1. You'll recall that it had three columns: `id` (a number), `joketext` (the text of the joke), and `jokedate` (the date on which the joke was entered). This is the command to create that table:

```
mysql> CREATE TABLE joke (
    ->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->   joketext TEXT,
    ->   jokedate DATE NOT NULL
    -> ) DEFAULT CHARACTER SET utf8;
```

Looks scary, huh? Let's break it down:

`CREATE TABLE joke (`
: This first line is fairly simple; it says that we want to create a new table named `joke`. The opening parenthesis (`(`) marks the beginning of the list of columns in the table.

`id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,`
: This second line says that we want a column called `id` that will contain an integer (`INT`), that is, a whole number. The rest of this line deals with special details for the column:

1. First, when creating a row in this table, this column is not allowed to be left blank (`NOT NULL`).
2. Next, if we omit specifying a particular value for this column when we add a new entry to the table, we want MySQL to automatically pick a value that is one more than the highest value in the table so far (`AUTO_INCREMENT`).
3. Finally, this column is to act as a unique identifier for the entries in the table, so all values in this column must be unique (`PRIMARY KEY`).

`joketext TEXT,`
: This third line is super simple; it says that we want a column called `joketext`, which will contain text (`TEXT`).

`jokedate DATA NOT NULL`
: This fourth line defines our last column, called `jokedate`; this will contain a date (`DATE`), which cannot be left blank (`NOT NULL`).

`) DEFAULT CHARACTER SET utf8;`

The closing parenthesis ()) marks the end of the list of columns in the table.

`DEFAULT CHARACTER SET utf8` tells MySQL that you will be storing UTF-8 encoded text in this table. UTF-8 is the most common encoding used for web content, so you should use it in all your database tables that you intend to use on the Web.

Finally, the semicolon tells the **mysql** client program that you've finished typing your query.

Note that we assigned a specific data type to each column we created. `id` will contain integers, `joketext` will contain text, and `jokedate` will contain dates. MySQL requires you to specify in advance a data type for each column. This helps to keep your data organized, and allows you to compare the values within a column in powerful ways, as we'll see later. For a complete list of supported MySQL data types, see Appendix C.

Now, if you typed the above command correctly, MySQL will respond with "`Query OK`", and your first table will be created. If you made a typing mistake, MySQL will tell you there was a problem with the query you typed, and will try to indicate where it had trouble understanding what you meant.

For such a complicated command, "`Query OK`" is a fairly underwhelming response. Let's have a look at your new table to make sure it was created properly. Type the following command:

```
mysql> SHOW TABLES;
```

The response should look like this:

```
+----------------+
| Tables_in_ijdb |
+----------------+
| joke           |
+----------------+
1 row in set (0.02 sec)
```

This is a list of all the tables in your database (which we named `ijdb` above). The list contains only one table: the `joke` table you just created. So far, everything seems fine. Let's take a closer look at the `joke` table itself using a `DESCRIBE` query:

```
mysql> DESCRIBE joke;
+----------+---------+------+-----+---------+----------------+
| Field    | Type    | Null | Key | Default | Extra          |
+----------+---------+------+-----+---------+----------------+
| id       | int(11) | NO   | PRI | NULL    | auto_increment |
| joketext | text    | YES  |     | NULL    |                |
| jokedate | date    | NO   |     | NULL    |                |
+----------+---------+------+-----+---------+----------------+
3 rows in set (0.10 sec)
```

As you can see, there are three columns (or fields) in this table, which appear as the three rows in this table of results. The details are a little cryptic, but if you look at them closely, you should be able to figure out what they mean. It's nothing to be too worried about, though. You have better things to do, like adding some jokes to your table!

We need to look at just one more task before you get to that, though: deleting a table. This task is as frighteningly easy as deleting a database. In fact, the command is almost identical. *Don't* run this command with your `joke` table, unless you actually do want to be rid of it!

```
mysql> DROP TABLE tableName;
```

Inserting Data into a Table

Your database is created and your table is built; all that's left is to put some actual jokes into the database. The command that inserts data into a database is called, appropriately enough, `INSERT`. This command can take two basic forms:

```
mysql> INSERT INTO tableName SET
    ->   column1Name = column1Value,
    ->   column2Name = column2Value,
    ->   ⋮
    -> ;
```

```
mysql> INSERT INTO tableName
    ->   (column1Name, column2Name, …)
    ->   VALUES (column1Value, column2Value, …);
```

So, to add a joke to our table, we can use either of these commands:

```
mysql> INSERT INTO joke SET
    -> joketext = "Why did the chicken cross the road? To get to
    ">  the other side!",
    -> jokedate = "2009-04-01";
```

```
mysql> INSERT INTO joke
    -> (joketext, jokedate) VALUES (
    -> "Why did the chicken cross the road? To get to the other
    ">  side!",
    -> "2009-04-01"
    -> );
```

Note that in both forms of the `INSERT` command, the order in which you list the columns must match the order in which you list the values. Otherwise, the order of the columns is unimportant.

As you typed this query, you'll have noticed that we used double quotes (`"`) to mark where the text of the joke started and ended. A piece of text enclosed in quotes this way is called a **text string**, and this is how you represent most data values in SQL. You'll notice, for instance, that the dates are typed as text strings as well, in the form `"YYYY-MM-DD"`.

If you prefer, you can type text strings surrounded with single quotes (`'`) instead of double quotes:

```
mysql> INSERT INTO joke SET
    -> joketext = 'Why did the chicken cross the road? To get to
    '>  the other side!',
    -> jokedate = '2009-04-01';
```

You might be wondering what happens when the text of a joke itself contains quotes. Well, if the text contains single quotes, the easiest thing to do is surround it with double quotes. Conversely, if the text contains double quotes, surround it with single quotes.

If the text you want to include in your query contains both single *and* double quotes, you'll have to **escape** the conflicting characters within your text string. You escape a character in SQL by adding a backslash (\) immediately before it. This tells MySQL to ignore any "special meaning" this character might have. In the case of single or double quotes, it tells MySQL not to interpret the character as the end of the text string.

To make this as clear as possible, here's an `INSERT` command for a joke containing both single and double quotes:

```
mysql> INSERT INTO joke
    -> (joketext, jokedate) VALUES (
    -> 'Knock-knock! Who\'s there? Boo! "Boo" who?
    '>  Don\'t cry; it\'s only a joke!',
    -> "2009-04-01");
```

As you can see, I've marked the start and end of the text string for the joke text using single quotes. I've therefore had to escape the three single quotes within the string by putting backslashes before them. MySQL sees these backslashes and knows to treat the single quotes as characters within the string, rather than end-of-string markers.

If you're especially clever, you might now be wondering how to include actual backslashes in SQL text strings. The answer is to type a double-backslash (`\\`), which MySQL will see and treat as a single backslash in the string of text.

Now that you know how to add entries to a table, let's see how we can view those entries.

Viewing Stored Data

The command we use to view data stored in database tables, `SELECT`, is the most complicated command in the SQL language. The reason for this complexity is that the chief strength of a database is its flexibility in data retrieval. At this early point in our experience with databases we need only fairly simple lists of results, so we'll just consider the simpler forms of the `SELECT` command here.

This command will list everything that's stored in the `joke` table:

```
mysql> SELECT * FROM joke;
```

Read aloud, this command says "select everything from `joke`." If you try this command, your results will resemble the following:

```
+----+-------------------------------------------------------------
-+------------+
| id | joketext
 | jokedate   |
+----+-------------------------------------------------------------
-+------------+
|  1 | Why did the chicken cross the road? To get to the other side!
 | 2009-04-01 |
+----+-------------------------------------------------------------
-+------------+
1 row in set (0.00 sec)
```

The results look a little disorganized because the text in the `joketext` column is so long that the table is too wide to fit on the screen properly. For this reason, you might want to tell MySQL to leave out the `joketext` column. The command for doing this is as follows:

```
mysql> SELECT id, jokedate FROM joke;
```

This time, instead of telling it to "select everything," we told it precisely which columns we wanted to see. The results look like this:

```
+----+------------+
| id | jokedate   |
+----+------------+
|  1 | 2009-04-01 |
+----+------------+
1 row in set (0.00 sec)
```

That's okay, but we'd like to see at least *some* of the joke text? As well as being able to name specific columns that we want the `SELECT` command to show us, we can use functions to modify each column's display. One function, called `LEFT`, lets us tell MySQL to display a column's contents up to a specified maximum number of characters. For example, let's say we wanted to see only the first 20 characters of the `joketext` column. Here's the command we'd use:

```
mysql> SELECT id, LEFT(joketext, 20), jokedate FROM joke;
+----+----------------------+------------+
| id | LEFT(joketext, 20)   | jokedate   |
+----+----------------------+------------+
|  1 | Why did the chicken  | 2009-04-01 |
+----+----------------------+------------+
1 row in set (0.00 sec)
```

See how that worked? Another useful function is `COUNT`, which lets us count the number of results returned. If, for example, you wanted to find out how many jokes were stored in your table, you could use the following command:

```
mysql> SELECT COUNT(*) FROM joke;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.02 sec)
```

As you can see, you have just one joke in your table.

So far, all the examples have fetched all the entries in the table; however, you can limit your results to include only those database entries that have the specific attributes you want. You set these restrictions by adding what's called a **`WHERE` clause** to the `SELECT` command. Consider this example:

```
mysql> SELECT COUNT(*) FROM joke WHERE jokedate >= "2009-01-01";
```

This query will count the number of jokes that have dates greater than or equal to January 1, 2009. In the case of dates, "greater than or equal to" means "on or after." Another variation on this theme lets you search for entries that contain a certain piece of text. Check out this query:

```
mysql> SELECT joketext FROM joke WHERE joketext LIKE "%chicken%";
```

This query displays the full text of all jokes that contain the text "chicken" in their `joketext` column. The `LIKE` keyword tells MySQL that the named column must match the given pattern. In this case, the pattern we've used is `"%chicken%"`. The

% signs indicate that the text "chicken" may be preceded and/or followed by any string of text.

Additional conditions may also be combined in the WHERE clause to further restrict results. For example, to display knock-knock jokes from April 2009 only, you could use the following query:

```
mysql> SELECT joketext FROM joke WHERE
    -> joketext LIKE "%knock%" AND
    -> jokedate >= "2009-04-01" AND
    -> jokedate < "2009-05-01";
```

Enter a few more jokes into the table and experiment with SELECT queries. A good familiarity with the SELECT command will come in handy later in this book.

You can do a lot with the SELECT command. We'll look at some of its more advanced features later, when we need them.

Modifying Stored Data

Having entered your data into a database table, you might like to change it. Whether you want to correct a spelling mistake, or change the date attached to a joke, such alterations are made using the UPDATE command. This command contains elements of the SELECT and INSERT commands, since the command both picks out entries for modification and sets column values. The general form of the UPDATE command is as follows:

```
mysql> UPDATE tableName SET
    ->   colName = newValue, …
    -> WHERE conditions;
```

So, for example, if we wanted to change the date on the joke we entered above, we'd use the following command:

```
mysql> UPDATE joke SET jokedate = "2010-04-01" WHERE id = "1";
```

Here's where that id column comes in handy: it enables you to single out a joke for changes easily. The WHERE clause used here works just as it did in the SELECT com-

mand. This next command, for example, changes the date of all entries that contain the word "chicken":

```
mysql> UPDATE joke SET jokedate = "2010-04-01"
    -> WHERE joketext LIKE "%chicken%";
```

Deleting Stored Data

Deleting entries in SQL is dangerously easy, which, if you've yet to notice, is a recurring theme. Here's the command syntax:

```
mysql> DELETE FROM tableName WHERE conditions;
```

To delete all chicken jokes from your table, you'd use the following query:

```
mysql> DELETE FROM joke WHERE joketext LIKE "%chicken%";
```

Careful With That Enter Key!

Believe it or not, the `WHERE` clause in the `DELETE` command is actually optional.

Consequently, you should be very careful when typing this command! If you leave the `WHERE` clause out, the `DELETE` command will then apply to *all entries in the table*.

This command will empty the `joke` table in one fell swoop:

```
mysql> DELETE FROM joke;
```

Scary, huh?

Let PHP Do the Typing

There's a lot more to the MySQL database server software and SQL than the handful of basic commands I've presented here, but these commands are by far the most commonly used.

At this stage, you might be thinking that databases seem a little cumbersome. SQL can be fairly tricky to type—its commands tend to be rather long and verbose com-

pared to other computer languages. You're probably already dreading the thought of typing in a complete library of jokes in the form of `INSERT` commands.

Don't sweat it! As we proceed through this book, you'll be surprised at how few SQL queries you actually type by hand. Generally, you'll be writing PHP scripts that type your SQL for you. If you want to be able to insert a bunch of jokes into your database, for example, you'll typically create a PHP script for adding jokes that includes the necessary `INSERT` query, with a placeholder for the joke text. You can then run that PHP script whenever you have jokes to add. The PHP script prompts you to enter your joke, then issues the appropriate `INSERT` query to your MySQL server.

For now, however, it's important for you to gain a good feel for typing SQL by hand. It will give you a strong sense of the inner workings of MySQL databases, and will make you appreciate the work that PHP will save you all the more!

To date, we've only worked with a single table, but to realize the true power of a relational database, you'll also need to learn how to use multiple tables together to represent potentially complex relationships between the items stored in your database. I'll cover all this and more in Chapter 5, in which I'll discuss database design principles and show off some more advanced examples.

For now, though, we've accomplished our objective, and you can comfortably interact with MySQL using the **mysql** client program. In Chapter 3, the fun continues as we delve into the PHP language, and use it to create several dynamically-generated web pages.

If you like, you can practice with MySQL a little before you move on by creating a decent-sized `joke` table. This knowledge will come in handy in Chapter 4.

Chapter 3

Introducing PHP

PHP is a **server-side language**. This concept may be a little difficult to grasp, especially if you're used to designing pages using only client-side languages like HTML, CSS, and JavaScript.

A server-side language is similar to JavaScript in that it allows you to embed little programs (scripts) into the HTML code of a web page. When executed, these programs give you greater control over what appears in the browser window than HTML alone can provide. The key difference between JavaScript and PHP is the stage of loading the web page at which these embedded programs are executed.

Client-side languages like JavaScript are read and executed by the web browser, after downloading the web page (embedded programs and all) from the web server. In contrast, server-side languages like PHP are run by the web *server*, before sending the web page to the browser. Whereas client-side languages give you control over how a page behaves once it's displayed by the browser, server-side languages let you generate customized pages on the fly before they're even sent to the browser.

Once the web server has executed the PHP code embedded in a web page, the results of that code's execution take the place of the PHP code in the page. When the browser

receives the page, all it sees is standard HTML code, hence the name: server-side language. Let's look back at the **today.php** example presented in Chapter 1:

chapter3/today.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Today&rsquo;s Date</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>Today&rsquo;s date (according to this web server) is
      <?php

      echo date('l, F dS Y.');

      ?>
    </p>
  </body>
</html>
```

Most of this is plain HTML; however, the line between `<?php` and `?>` is PHP code. `<?php` marks the start of an embedded PHP script and `?>` marks the end of such a script. The web server is asked to interpret everything between these two delimiters, and to convert it to regular HTML code before it sends the web page to the requesting browser. The browser is presented with the following:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Today&rsquo;s Date</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>Today&rsquo;s Date (according to this web server) is
      Wednesday, April 1st 2009.    </p>
  </body>
</html>
```

Notice that all signs of the PHP code have disappeared. In its place, the output of the script has appeared, and it looks just like standard HTML. This example demonstrates several advantages of server-side scripting:

No browser compatibility issues
: PHP scripts are interpreted by the web server alone, so there's no need to worry about whether the language you're using is supported by the visitor's browser.

Access to server-side resources
: In the above example, we placed the date, according to the web server, into the web page. If we had inserted the date using JavaScript, we'd only be able to display the date according to the computer on which the web browser was running. Granted, there are more impressive examples of the exploitation of server-side resources; a better example might be inserting content pulled out of a MySQL database (*hint, hint …*).

Reduced load on the client
: JavaScript can delay the display of a web page on slower computers significantly, as the browser must run the script before it can display the web page. With server-side code, this burden is passed to the web server machine, which you can make as beefy as your application requires.

Basic Syntax and Statements

PHP syntax will be very familiar to anyone with an understanding of C, C++, C#, Java, JavaScript, Perl, or any other C-derived language. If you're unfamiliar with any of these languages, or if you're new to programming in general, there's no need to worry about it!

A PHP script consists of a series of commands, or **statements**. Each statement is an instruction that must be followed by the web server before it can proceed to the next. PHP statements, like those in the above-mentioned languages, are always terminated by a semicolon (`;`).

This is a typical PHP statement:

```
echo 'This is a <strong>test</strong>!';
```

This is an `echo` statement, which is used to generate content (usually HTML code) to be sent to the browser. An `echo` statement simply takes the text it's given, and inserts it into the page's HTML code at the position of the PHP script that contains it.

In this case, we have supplied a string of text to be output: `'This is a <strong>test</strong>!'`. Notice that the string of text contains HTML tags (`<strong>` and `</strong>`), which is perfectly acceptable. So, if we take this statement and put it into a complete web page, here's the resulting code:

chapter3/echo.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Simple PHP Example</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p><?php echo 'This is a <strong>test</strong>!'; ?></p>
  </body>
</html>
```

If you place this file on your web server, a browser that requests the page will receive this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Simple PHP Example</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>This is a <strong>test</strong>!</p>
  </body>
</html>
```

The **today.php** example we looked at earlier contained a slightly more complex `echo` statement:

chapter3/today.php *(excerpt)*

```
echo date('l, F dS Y.');
```

Instead of giving `echo` a simple string of text to output, this statement invokes a **built-in function** called `date` and passes *it* a string of text: `'l, F dS Y.'`. You can think of built-in functions as tasks that PHP knows how to do without your needing to spell out the details. PHP has many built-in functions that let you do everything from sending email to working with information stored in various types of databases.

When you invoke a function in PHP, you're said to be **calling** that function. Most functions **return** a value when they're called; PHP then replaces the function call with that value when it executes the statement. In this case, our `echo` statement contains a call to the `date` function, which returns the current date as a string of text (the format of which is specified by the text string in the function call). The `echo` statement therefore outputs the value returned by the function call.

You may wonder why we need to surround the string of text with both parentheses (`()`) and single quotes (`''`). As in SQL, quotes are used in PHP to mark the beginning and end of strings of text, so it makes sense for them to be there. The parentheses serve two purposes. First, they indicate that `date` is a function that you want to call. Second, they mark the beginning and end of a list of **parameters** (or **arguments**) that you wish to provide, in order to tell the function what to do. In the case of the `date` function, you need to provide a string of text that describes the format in which you want the date to appear.[1] Later on, we'll look at functions that take more than one parameter, and we'll separate those parameters with commas. We'll also consider functions that take no parameters at all. These functions will still need the parentheses, though it's unnecessary to type anything between them.

[1] A full reference is available in the online documentation for the `date` function [http://www.php.net/date/].

Variables, Operators, and Comments

Variables in PHP are identical to variables in most other programming languages. For the uninitiated, a **variable** can be thought of as a name that's given to an imaginary box into which any **literal value** may be placed. The following statement creates a variable called `$testvariable` (all variable names in PHP begin with a dollar sign) and assigns it a literal value of `3`:

```
$testvariable = 3;
```

PHP is a **loosely typed** language. This means that a single variable may contain any type of data, be it a number, a string of text, or some other kind of value, and may change types over its lifetime. So the following statement, if you were to type it after the statement above, assigns a new value to the existing `$testvariable`. In the process, the variable changes type: where it used to contain a number, it now contains a string of text:

```
$testvariable = 'Three';
```

The equals sign we used in the last two statements is called the **assignment operator**, as it's used to assign values to variables. Other operators may be used to perform various mathematical operations on values:

```
$testvariable = 1 + 1;  // Assigns a value of 2
$testvariable = 1 - 1;  // Assigns a value of 0
$testvariable = 2 * 2;  // Assigns a value of 4
$testvariable = 2 / 2;  // Assigns a value of 1
```

From the above examples, you can probably tell that `+` is the **addition operator**, `-` is the **subtraction operator**, `*` is the **multiplication operator**, and `/` is the **division operator**. These are all called **arithmetic operators**, because they perform arithmetic on numbers.

Each of the lines above ends with a **comment**. Comments are a way to describe what your code is doing. They insert explanatory text into your code—text that the PHP interpreter will ignore. Comments begin with `//` and they finish at the end of the same line. If you need a comment to span several lines, you can instead start your comment with `/*`, and end it with `*/`. The PHP interpreter will ignore everything

between these two delimiters. I'll use comments throughout the rest of this book to help explain some of the code I present.

Returning to the operators, there's another one that sticks strings of text together, called the **string concatenation operator**:

```
$testvariable = 'Hi ' . 'there!';  // Assigns a value of 'Hi there!'
```

Variables may be used almost anywhere that you use a literal value. Consider this series of statements:

```
$var1 = 'PHP';          // Assigns a value of 'PHP' to $var1
$var2 = 5;              // Assigns a value of 5 to $var2
$var3 = $var2 + 1;      // Assigns a value of 6 to $var3
$var2 = $var1;          // Assigns a value of 'PHP' to $var2
echo $var1;             // Outputs 'PHP'
echo $var2;             // Outputs 'PHP'
echo $var3;             // Outputs '6'
echo $var1 . ' rules!'; // Outputs 'PHP rules!'
echo "$var1 rules!";    // Outputs 'PHP rules!'
echo '$var1 rules!';    // Outputs '$var1 rules!'
```

Notice the last two lines in particular. You can include the name of a variable right inside a text string, and have the value inserted in its place if you surround the string with double quotes instead of single quotes. This process of converting variable names to their values is known as **variable interpolation**. However, as the last line demonstrates, a string surrounded with single quotes will not interpolate the variable names it contains.

Arrays

An **array** is a special kind of variable that contains multiple values. If you think of a variable as a box that contains a value, then an array can be thought of as a box with compartments, where each compartment is able to store an individual value.

The simplest way to create an array in PHP is to use the built-in `array` function:

```
$myarray = array('one', 2, '3');
```

This code creates an array called `$myarray` that contains three values: `'one'`, `2`, and `'3'`. Just like an ordinary variable, each space in an array can contain any type of value. In this case, the first and third spaces contain strings, while the second contains a number.

To access a value stored in an array, you need to know its **index**. Typically, arrays use numbers, starting with zero, as indices to point to the values they contain. That is, the first value (or element) of an array has index 0, the second has index 1, the third has index 2, and so on. In general, therefore, the index of the *n*th element of an array is *n*–1. Once you know the index of the value you're interested in, you can retrieve that value by placing that index in square brackets after the array variable name:

```
echo $myarray[0];       // Outputs 'one'
echo $myarray[1];       // Outputs '2'
echo $myarray[2];       // Outputs '3'
```

Each value stored in an array is called an **element** of that array. You can use an index in square brackets to add new elements, or assign new values to existing array elements:

```
$myarray[1] = 'two';    // Assign a new value
$myarray[3] = 'four';   // Create a new element
```

You can add elements to the end of an array using the assignment operator as usual, but leaving empty the square brackets that follow the variable name:

```
$myarray[] = 'the fifth element';
echo $myarray[4];       // Outputs 'the fifth element'
```

However, numbers are only the most common choice for array indices; there's another possibility. You can also use strings as indices to create what's called an **associative array**. This type of array is called associative because it associates values with meaningful indices. In this example, we associate a date (in the form of a string) with each of three names:

```
$birthdays['Kevin'] = '1978-04-12';
$birthdays['Stephanie'] = '1980-05-16';
$birthdays['David'] = '1983-09-09';
```

The `array` function also lets you create associative arrays, if you prefer that method. Here's how we'd use it to create the `$birthdays` array:

```
$birthdays = array('Kevin' => '1978-04-12',
    'Stephanie' => '1980-05-16', 'David' => '1983-09-09');
```

Now, if we want to know Kevin's birthday, we look it up using the name as the index:

```
echo 'My birthday is: ' . $birthdays['Kevin'];
```

This type of array is especially important when it comes to user interaction in PHP, as we'll see in the next section. I'll demonstrate other uses of arrays throughout this book.

User Interaction and Forms

For most database driven web sites these days, you need to do more that just dynamically generate pages based on database data; you must also provide some degree of interactivity, even if it's just a search box.

Veterans of JavaScript tend to think of interactivity in terms of event handlers, which let you react directly to the actions of the user—for example, the movement of the cursor over a link on the page. Server-side scripting languages such as PHP have a more limited scope when it comes to support for user interaction. As PHP code is only activated when a request is made to the server, user interaction can occur only in a back-and-forth fashion: the user sends requests to the server, and the server replies with dynamically generated pages.[2]

The key to creating interactivity with PHP is to understand the techniques we can use to send information about a user's interaction along with a request for a new web page. As it turns out, PHP makes this fairly easy.

[2] To some extent, the rise of Ajax techniques in the JavaScript world over the past few years has changed this. It's now possible for JavaScript code, responding to a user action such as mouse movement, to send a request to the web server, invoking a PHP script. For the purposes of this book, however, we'll stick to non-Ajax applications. If you'd like to learn how to use PHP with Ajax, check out *Build Your Own AJAX Web Applications* [http://www.sitepoint.com/books/ajax1/] by Matthew Eernisse (Melbourne: SitePoint, 2006).

The simplest method we can use to send information along with a page request is to use the **URL query string**. If you've ever seen a URL in which a question mark followed the file name, you've witnessed this technique in use. For example, if you search for "SitePoint" on Google, it will take you to the following URL to see the search results:

```
http://www.google.com/search?hl=en&q=SitePoint&btnG=Google+Search&
➥meta=
```

See the question mark in the URL? See how the text that follows the question mark contains things like your search query (`SitePoint`) and the name of the button you clicked (`Google+Search`)? That information is being sent along with the request for http://www.google.com/search.

Let's code up an easy example of our own. Create a regular HTML file called **welcome1.html** (no **.php** file name extension is required, since there will be no PHP code in this file) and insert this link:

chapter3/welcome1.html *(excerpt)*

```
<a href="welcome1.php?name=Kevin">Hi, I&rsquo;m Kevin!</a>
```

This is a link to a file called **welcome1.php**, but as well as linking to the file, you're also passing a variable along with the page request. The variable is passed as part of the query string, which is the portion of the URL that follows the question mark. The variable is called `name` and its value is `Kevin`. To restate, you have created a link that loads **welcome1.php**, and informs the PHP code contained in that file that `name` equals `Kevin`.

To really understand the effect of this link, we need to look at **welcome1.php**. Create it as a new HTML file, but, this time, note the **.php** file name extension—this tells the web server that it can expect to interpret some PHP code in the file. In the `<body>` of this new web page, type the following:

chapter3/welcome1.php *(excerpt)*

```
<?php
$name = $_GET['name'];
echo 'Welcome to our web site, ' . $name . '!';
?>
```

Now, put these two files (**welcome1.html** and **welcome1.php**) onto your web server, and load the first file in your browser (the URL should be similar to http://localhost/welcome1.html, or http://localhost:8080/welcome1.html if your web server is running on a port other than 80). Click the link in that first page to request the PHP script. You should see that the resulting page says "Welcome to our web site, Kevin!", as shown in Figure 3.1.

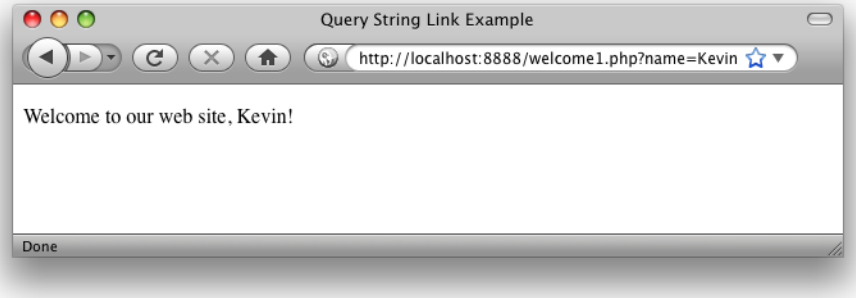


Figure 3.1. Greet users with a personalized welcome message

Let's take a closer look at the code that made this possible. The most important line is this one:

chapter3/welcome1.php *(excerpt)*

```
$name = $_GET['name'];
```

If you were paying close attention in the section called "Arrays", you'll recognize what this line does. It assigns to a new variable called `$name` the value stored in the `'name'` element of the array called `$_GET`. But where does the `$_GET` array come from?

It turns out that `$_GET` is one of a number of variables that PHP automatically creates when it receives a request from a browser. PHP creates `$_GET` as an array variable that contains any values passed in the query string. `$_GET` is an associative array, so the value of the `name` variable passed in the query string can be accessed as `$_GET['name']`. Your **welcome1.php** script assigns this value to an ordinary PHP variable (`$name`), then displays it as part of a text string using an `echo` statement:

chapter3/welcome1.php *(excerpt)*

```
echo 'Welcome to our web site, ' . $name . '!';
```

The value of the `$name` variable is inserted into the output string using the string concatenation operator (`.`) that we looked at in the section called "Variables, Operators, and Comments".

But look out! There is a **security hole** lurking in this code! Although PHP is an easy programming language to learn, it turns out it's also especially easy to introduce security issues into web sites using PHP if you're unaware of what precautions to take. Before we go any further with the language, I want to make sure you're able to spot and fix this particular security issue, since it's probably the most common kind of security issue on the Web today.

The security issue here stems from the fact that the **welcome1.php** script is generating a page containing content that is under the control of the user—in this case, the `$name` variable. Although the `$name` variable will normally receive its value from the URL query string in the link on the **welcome1.html** page, a malicious user could edit the URL to send a different value for the `name` variable.

To see how this would work, click the link in **welcome1.html** again. When you see the resulting page (with the welcome message containing the name "Kevin"), take a look at the URL in the address bar of your browser. It should look similar to this:

```
http://localhost/welcome1.php?name=Kevin
```

Edit the URL to insert a `<b>` tag before the name, and a `</b>` tag following the name, like this:

```
http://localhost/welcome1.php?name=<b>Kevin</b>
```

Hit **Enter** to load this new URL, and notice that the name in the page is now bold, as shown in Figure 3.2.

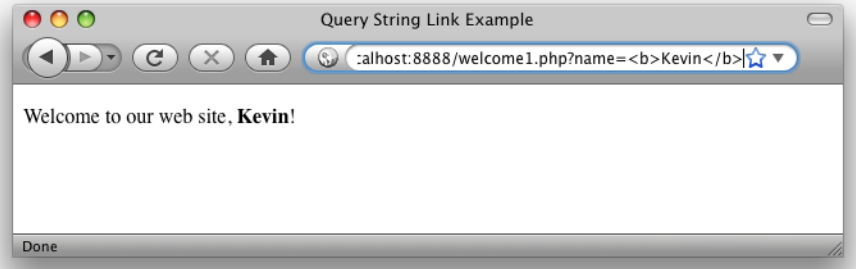


Figure 3.2. Easy exploitation will only embolden attackers!

See what's happening here? The user can type any HTML code into the URL, and your PHP script includes it in the code of the generated page without question. If the code is as innocuous as a `<b>` tag there's no problem, but a malicious user could include sophisticated JavaScript code that performed malicious actions like steal the user's password. All the attacker would have to do, then, would be to publish the modified link on some other site under the attacker's control, and then entice one of your users to click it. The attacker could even embed the link in an email and send it to your users. If one of your users clicked the link, the attacker's code would be included in your page and the trap would be sprung!

I hate to scare you with this talk of malicious hackers attacking your users by turning your own PHP code against you, particularly when you'e only just learning the language. The fact is, however, that PHP's biggest weakness as a language is how easy it is to introduce security issues like this. Some might say that most of the energy you spend learning to write PHP to a professional standard is spent on avoiding security issues. The sooner you're exposed to these issues, however, the sooner you become accustomed to avoiding them, and the less of a stumbling block they'll be for you going forward.

So, how can we generate a page containing the user's name without opening it up to abuse by attackers? The solution is to treat the value supplied for the `$name` variable as plain text to be displayed on your page, rather than as HTML to be included in the page's code. This is a subtle distinction, so let me show you what I mean.

Copy your **welcome1.html** file and rename it to **welcome2.html**. Edit the link it contains so that it points to **welcome2.php** instead of **welcome1.php**:

chapter3/welcome2.html *(excerpt)*

```
<a href="welcome2.php?name=Kevin">Hi, I&rsquo;m Kevin!</a>
```

Copy your **welcome1.php** file and rename it to **welcome2.php**. Edit the PHP code it contains so that it looks like this:

chapter3/welcome2.php *(excerpt)*

```
<?php
$name = $_GET['name'];
echo 'Welcome to our web site, ' .
    htmlspecialchars($name, ENT_QUOTES, 'UTF-8') . '!';
?>
```

There's a lot going on in this code, so let me break it down for you. The first line is the same as it was previously, assigning to `$name` the value of the `'name'` element from the `$_GET` array. The `echo` statement that follows it is drastically different, though. Whereas previously, we simply dumped the `$name` variable, naked, into the `echo` statement, this version of the code uses the built-in PHP function `htmlspecialchars` to perform a critical conversion.

Remember, the security hole comes from the fact that, in **welcome1.php**, HTML code in the `$name` variable is dumped directly into the code of the generated page, and can therefore do anything that HTML code can do. What `htmlspecialchars` does is convert "special HTML characters" like "<" and ">" into HTML character entities like `<` and `>`, which prevents them from being interpreted as HTML code by the browser. I'll demonstrate this for you in a moment.

First, let's take a closer look at this new code. The call to the `htmlspecialchars` function is the first example in this book of a PHP function that takes more than one parameter. Here's the function call all by itself:

```
htmlspecialchars($name, ENT_QUOTES, 'UTF-8')
```

The first parameter is the `$name` variable (the text to be converted). The second parameter is the PHP constant[3] `ENT_QUOTES`, which tells `htmlspecialchars` to convert single and double quotes in addition to other special characters. The third parameter is the string `'UTF-8'`, which tells PHP what character encoding to use to interpret the text you give it.

The Perks and Pitfalls of UTF-8 with PHP

You may have noticed that all of the example HTML pages in this book contain the following `<meta>` tag near the top:

```
<meta http-equiv="content-type"
    content="text/html; charset=utf-8"/>
```

This tag tells the browser that receives this page that the HTML code of the page is encoded as UTF-8 text.[4]

In a few pages, we'll reach the section on building HTML forms. By encoding your pages as UTF-8, your users can submit text containing thousands of foreign characters that your site would otherwise be unable to handle.

Unfortunately, many of PHP's built-in functions, such as `htmlspecialchars`, assume you're using the much simpler ISO-8859-1 character encoding by default. Therefore, you need to let them know you're using UTF-8 when you use these functions.

If you can, you should also tell your text editor to save your HTML and PHP files as UTF-8 encoded text, but this is only required if you want to type advanced characters (like curly quotes or dashes) or foreign characters (like "é") into your HTML or PHP code. The code in this book plays it safe and uses HTML character entities (for example, `&rsquo;` for a curly right quote), which will work regardless.

[3] A PHP constant is like a variable whose value you're unable to change. Unlike variables, constants don't start with a dollar sign. PHP comes with a number of built-in constants like `ENT_QUOTES` that are used to control built-in functions like `htmlspecialchars`.

[4] UTF-8 is one of many standards for representing text as a series of ones and zeros in computer memory, called character encodings. If you're curious to learn all about character encodings, check out *The Definitive Guide to Web Character Encoding* [http://www.sitepoint.com/article/guide-web-character-encoding/].

Open up **welcome2.html** in your browser and click the link that now points to **welcome2.php**. Once again, you'll see the welcome message "Welcome to our web site, Kevin!". As you did before, modify the URL to include `<b>` and `</b>` tags surrounding the name:

```
http://localhost/welcome2.php?name=<b>Kevin</b>
```

This time, when you hit **Enter**, instead of the name turning bold in the page, you should see the actual text that you typed, as shown in Figure 3.3.

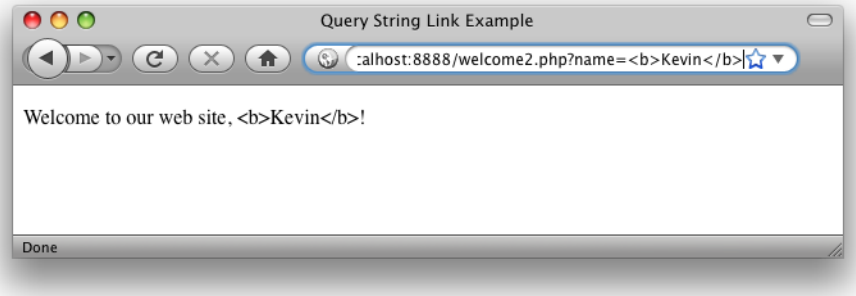


Figure 3.3. It sure is ugly, but it's secure!

If you view the source of the page, you can confirm that the `htmlspecialchars` function did its job and converted the "<" and ">" characters present in the provided name into the `&lt;` and `&gt;` HTML character entities, respectively. This prevents malicious users from injecting unwanted code into your site. If they try anything like that, the code is harmlessly displayed as plain text on the page.

We'll make extensive use of the `htmlspecialchars` function throughout this book to guard against this sort of security hole. No need to worry too much if you're having trouble grasping the details of how to use it for now. Before long, you'll find its use becomes second nature. For now, let's look at some more advanced ways of passing values to PHP scripts when we request them.

Passing a single variable in the query string was nice, but it turns out you can pass *more* than one value if you want to! Let's look at a slightly more complex version of the previous example. Save a copy of your **welcome2.html** file as **welcome3.html**, and change the link to point to **welcome3.php** with a query string as follows:

chapter3/welcome3.html *(excerpt)*

```
<a href="welcome3.php?firstname=Kevin&amp;lastname=Yank">Hi,
  I&rsquo;m Kevin Yank!</a>
```

This time, our link passes two variables: `firstname` and `lastname`. The variables are separated in the query string by an ampersand (`&`, which must be written as `&` in HTML). You can pass even more variables by separating each *`name=value`* pair from the next with an ampersand.

As before, we can use the two variable values in our **welcome3.php** file:

chapter3/welcome3.php *(excerpt)*

```
<?php
$firstname = $_GET['firstname'];
$lastname = $_GET['lastname'];
echo 'Welcome to our web site, ' .
    htmlspecialchars($firstname, ENT_QUOTES, 'UTF-8') . ' ' .
    htmlspecialchars($lastname, ENT_QUOTES, 'UTF-8') . '!';
?>
```

The `echo` statement is becoming quite sizable now, but it should still make sense to you. Using a series of string concatenations (`.`), it outputs "Welcome to our web site, " followed by the value of `$firstname` (made safe for display using `htmlspecialchars`), a space, the value of `$lastname` (again, treated with `htmlspecialchars`), and finally an exclamation mark.

The result is shown in Figure 3.4.

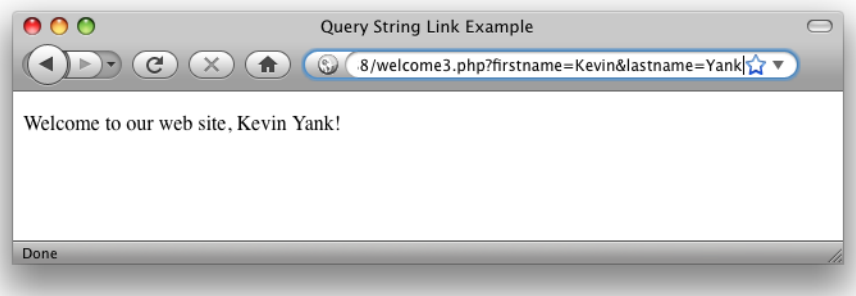


Figure 3.4. Create an even more personalized welcome message

This is all well and good, but we still have yet to achieve our goal of true user interaction, where the user can enter arbitrary information and have it processed by PHP. To continue with our example of a personalized welcome message, we'd like to invite the user to type his or her name and have it appear in the resulting page. To enable the user to type in a value, we'll need to use a HTML form.

Create a new HTML file named **welcome4.html** and type in this HTML code to create the form:

chapter3/welcome4.html *(excerpt)*

```
<form action="welcome4.php" method="get">
  <div><label for="firstname">First name:
    <input type="text" name="firstname" id="firstname"/></label>
  </div>
  <div><label for="lastname">Last name:
    <input type="text" name="lastname" id="lastname"/></label></div>
  <div><input type="submit" value="GO"/></div>
</form>
```

Self-closing Tags

The slashes that appear in some of these tags (such as `<input …/>`) are no cause for alarm. The XHTML standard for coding web pages calls for slashes to be used in any tag without a closing tag, which includes `<input/>` and `<meta/>` tags, among others.

Many developers prefer to code to the HTML standard instead of adopting XHTML and, in fact, this is a matter of some debate within web development circles. The upcoming HTML 5 standard leaves the choice up to the developer, so neither approach is strictly "more correct" than the other.

If you're curious about the factors to consider when making this decision for yourself, check out the relevant page of the SitePoint HTML Reference.[5]

The form this code produces is shown in Figure 3.5.[6]

[5] http://reference.sitepoint.com/html/html-vs-xhtml

[6] This form is quite plain-looking, I'll grant you. Some judicious application of CSS would make this—and all the other pages in this book—look more attractive. Since this is a book about PHP and MySQL, however, I've stuck with the plain look. Check out SitePoint books like *The Art & Science of CSS*

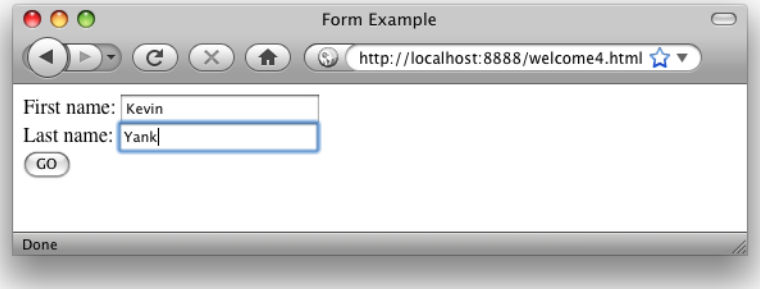


Figure 3.5. Make your own welcome message

Also make a copy of **welcome3.php** named **welcome4.php**. There's nothing that needs changing in this file.

This form has the exact same effect as the second link we looked at (with `first-name=Kevin&lastname=Yank` in the query string), except that you can now enter whatever names you like. When you click the submit button (which is labeled **GO**), the browser will load **welcome4.php** and add the variables and their values to the query string for you automatically. It retrieves the names of the variables from the `name` attributes of the `<input type="text" />` tags, and obtains the values from the text the user types into the text fields.

Apostrophes in Form Fields

If you are burdened with the swollen ego of most programmers (myself included), you probably took this opportunity to type your *own* name into this form. Who can blame you?

If your last name happens to include an apostrophe (for example, Molly O'Reilly), the welcome message you saw may have included a stray backslash before the apostrophe (that is, "Welcome to our web site, Molly O\'Reilly!").

This bothersome backslash is due to a PHP security feature called **magic quotes**, which we'll learn about in Chapter 4. Until then, please bear with me.

[http://www.sitepoint.com/books/cssdesign1/] (Melbourne: SitePoint, 2007) for advice on styling your forms with CSS.

The `method` attribute of the `<form>` tag is used to tell the browser how to send the variables and their values along with the request. A value of `get` (as used in **welcome4.html** above) causes them to be passed in the query string (and appear in PHP's `$_GET` array), but there is an alternative. It can be undesirable—or even technically unfeasible—to have the values appear in the query string. What if we included a `<textarea>` tag in the form, to let the user enter a large amount of text? A URL whose query string contained several paragraphs of text would be ridiculously long, and would possibly exceed the maximum length for a URL in today's browsers. The alternative is for the browser to pass the information invisibly, behind the scenes.

Make a copy of **welcome4.html** and name it **welcome5.html**. The code for the form in this new page is exactly the same, but where we set the form `method` to `get` in the last example, here we set it to `post`. Of course, we've also set the `action` attribute to point at `welcome5.php`:

chapter3/welcome5.html *(excerpt)*

```
<form action="welcome5.php" method="post">
  <div><label for="firstname">First name:
    <input type="text" name="firstname" id="firstname"/></label>
  </div>
  <div><label for="lastname">Last name:
    <input type="text" name="lastname" id="lastname"/></label></div>
  <div><input type="submit" value="GO"/></div>
</form>
```

This new value for the `method` attribute instructs the browser to send the form variables invisibly, as part of the page request, rather than embedding them in the query string of the URL.

Again, make a copy of **welcome4.php** and name it **welcome5.php**.

As we're no longer sending the variables as part of the query string, they stop appearing in PHP's `$_GET` array. Instead, they're placed in another array reserved especially for "posted" form variables: `$_POST`. We must therefore modify **welcome5.php** to retrieve the values from this new array:

chapter3/welcome5.php *(excerpt)*

```
<?php
$firstname = $_POST['firstname'];
$lastname = $_POST['lastname'];
echo 'Welcome to our web site, ' .
    htmlspecialchars($firstname, ENT_QUOTES, 'UTF-8') . ' ' .
    htmlspecialchars($lastname, ENT_QUOTES, 'UTF-8') . '!';
?>
```

Figure 3.6 shows what the resulting page looks like once this new form is submitted.

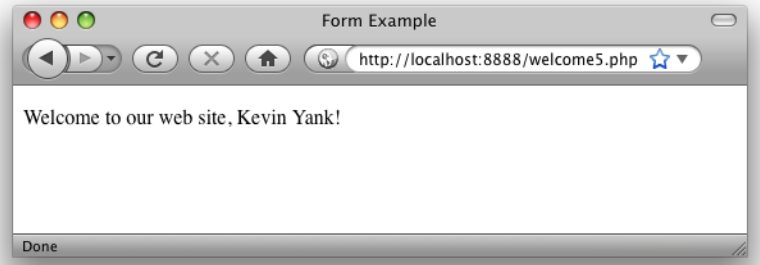


Figure 3.6. This personalized welcome is achieved without a query string

The form is functionally identical to the previous one; the only difference is that the URL of the page that's loaded when the user clicks the **GO** button will be without a query string. On the one hand, this lets you include large values, or sensitive values (like passwords), in the data that's submitted by the form, without their appearing in the query string. On the other hand, if the user bookmarks the page that results from the form's submission, that bookmark will be useless, as it lacks the submitted values. This, incidentally, is the main reason why search engines use the query string to submit search terms. If you bookmark a search results page on Google, you can use that bookmark to perform the same search again later, because the search terms are contained in the URL.

Sometimes, you want access to a variable without having to worry about whether it was sent as part of the query string or a form post. In cases like these, the special `$_REQUEST` array comes in handy. It contains all the variables that appear in both `$_GET` and `$_POST`. With this variable, we can modify our form processing script

one more time so that it can receive the first and last names of the user from either source:

chapter3/welcome6.php *(excerpt)*

```
<?php
$firstname = $_REQUEST['firstname'];
$lastname = $_REQUEST['lastname'];
echo 'Welcome to our web site, ' .
    htmlspecialchars($firstname, ENT_QUOTES, 'UTF-8') . ' ' .
    htmlspecialchars($lastname, ENT_QUOTES, 'UTF-8') . '!';
?>
```

That covers the basics of using forms to produce rudimentary user interaction with PHP. We'll look at more advanced issues and techniques in later examples.

Control Structures

All the examples of PHP code we've seen so far have been either one-statement scripts that output a string of text to the web page, or series of statements that were to be executed one after the other in order. If you've ever written programs in other languages (JavaScript, C, or BASIC) you already know that practical programs are rarely so simple.

PHP, just like any other programming language, provides facilities that enable you to affect the **flow of control**. That is, the language contains special statements that you can use to deviate from the one-after-another execution order that has dominated our examples so far. Such statements are called **control structures**. Don't understand? Don't worry! A few examples will illustrate perfectly.

The most basic, and most often used, control structure is the **`if` statement**. The flow of a program through an `if` statement can be visualized as in Figure 3.7.

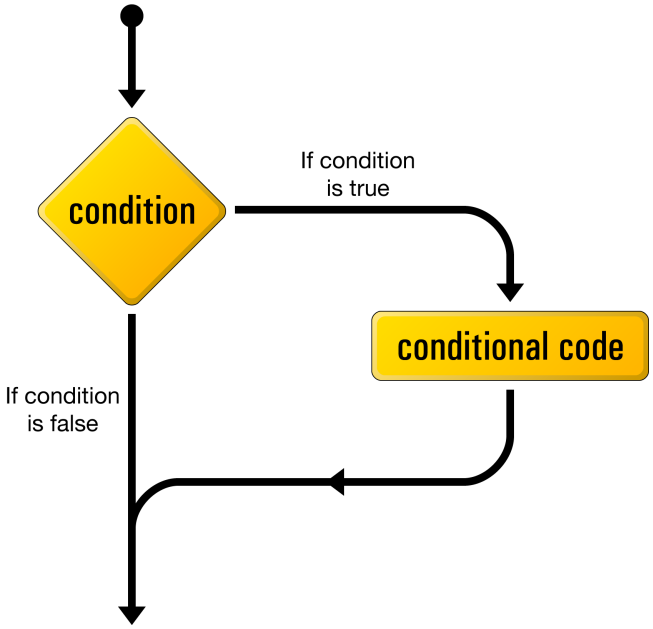


Figure 3.7. The logical flow of an `if` statement[7]

Here's what an `if` statement looks like in PHP code:

```
if (condition)
{
  // conditional code to be executed if condition is true
}
```

This control structure lets us tell PHP to execute a set of statements only if some condition is met.

If you'll indulge my vanity for a moment, here's an example that shows a twist on the personalized welcome page example we created earlier. Start by making a copy of **welcome6.html** called **welcome7.html**. For simplicity, let's alter the form it contains so that it submits a single `name` variable to **welcome7.php**:

[7] This diagram and several similar ones in this book were originally designed by Cameron Adams for the book, *Simply JavaScript* (Melbourne: SitePoint, 2006), which we wrote together. I have reused them here with his permission, and my thanks.

chapter3/welcome7.html *(excerpt)*

```
<form action="welcome7.php" method="post">
  <div><label for="name">Name:
    <input type="text" name="name" id="name"/></label></div>
  <div><input type="submit" value="GO"/></div>
</form>
```

Now make a copy of **welcome6.php** called **welcome7.php**. Replace the PHP code it contains with the following:

chapter3/welcome7.php *(excerpt)*

```
$name = $_REQUEST['name'];
if ($name == 'Kevin')
{
  echo 'Welcome, oh glorious leader!';
}
```

Now, if the `name` variable passed to the page has a value of `'Kevin'`, a special message will be displayed, as shown in Figure 3.8.

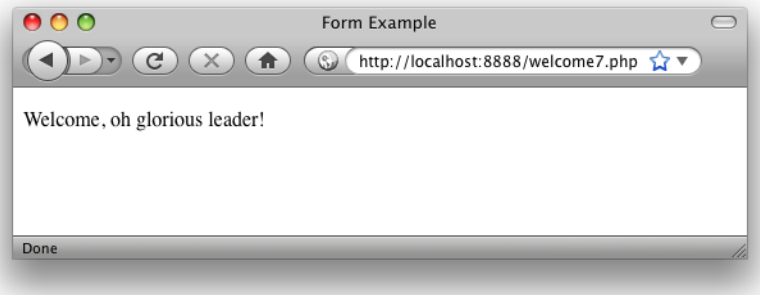


Figure 3.8. It's good to be the king

If a name other than Kevin is entered, this example becomes inhospitable—the conditional code within the `if` statement fails to execute, and the resulting page will be blank!

To offer an alternative to a blank page to all the plebs who have a different name to Kevin, we can use an **`if-else` statement** instead. The structure of an `if-else` statement is shown in Figure 3.9.

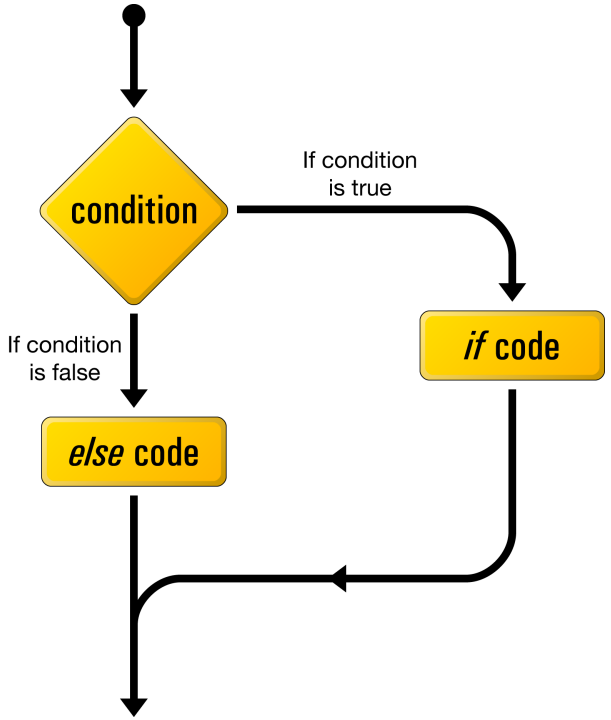


Figure 3.9. The logical flow of an `if-else` statement

The `else` portion of an `if-else` statement is tacked onto the end of the `if` portion, like this:

chapter3/welcome7.php *(excerpt)*

```
$name = $_REQUEST['name'];
if ($name == 'Kevin')
{
  echo 'Welcome, oh glorious leader!';
}
else
{
  echo 'Welcome to our web site, ' .
      htmlspecialchars($name, ENT_QUOTES, 'UTF-8') . '!';
}
```

Now if you submit a name other than Kevin, you should see the usual welcome message shown in Figure 3.10.

Figure 3.10. You gotta remember your peeps

The == used in the condition above is the **equal operator** that's used to compare two values to see whether they're equal.

Double Trouble

Remember to type the double-equals (==). A common mistake among beginning PHP programmers is to type a condition like this with a single equals sign:

```
if ($name = 'Kevin')     // Missing equals sign!
```

This condition is using the assignment operator (=) that I introduced back in the section called "Variables, Operators, and Comments", instead of the equal operator (==). Consequently, instead of comparing the value of `$name` to the string `'Kevin'`, it will actually *set* the value of `$name` to `'Kevin'`. Oops!

To make matters worse, the `if` statement will use this assignment operation as a condition, which it will consider to be true, so the conditional code within the `if` statement will always be executed, regardless of what the original value of `$name` happened to be.

Conditions can be more complex than a single check for equality. Recall that our form examples above would receive a first and last name. If we wanted to display a special message only for a particular person, we'd have to check the values of *both* names.

To do this, first make a copy of **welcome6.html** (which contains the two-field version of the form) called **welcome8.html**. Change the `action` attribute of the `<form>` tag to point to **welcome8.php**. Next, make a copy of **welcome7.php** called **welcome8.php**,

and update the PHP code to match the following (I've highlighted the changes in bold):

chapter3/welcome8.php *(excerpt)*

```
$firstname = $_REQUEST['firstname'];
$lastname = $_REQUEST['lastname'];
if ($firstname == 'Kevin' and $lastname == 'Yank')
{
  echo 'Welcome, oh glorious leader!';
}
else
{
  echo 'Welcome to our web site, ' .
      htmlspecialchars($firstname, ENT_QUOTES, 'UTF-8') . ' ' .
      htmlspecialchars($lastname, ENT_QUOTES, 'UTF-8') . '!';
}
```

This updated condition will be true if and only if `$firstname` has a value of `'Kevin'` and `$lastname` has a value of `'Yank'`. The **`and` operator** in the condition makes the whole condition true only if both of the comparisons are true. A similar operator is the **`or` operator**, which makes the whole condition true if one or both of two simple conditions are true. If you're more familiar with the JavaScript or C forms of these operators (`&&` and `||` for `and` and `or` respectively), that's fine—they work in PHP as well.

Figure 3.11 shows that having only one of the names right in this example fails to cut the mustard.

Figure 3.11. Frankly, my dear ...

We'll look at more complicated conditions as the need arises. For the time being, a general familiarity with `if-else` statements is sufficient.

Another often-used PHP control structure is the **`while` loop**. Where the `if-else` statement allowed us to choose whether or not to execute a set of statements depending on some condition, the `while` loop allows us to use a condition to determine how many times we'll execute a set of statements repeatedly.

Figure 3.12 shows how a `while` loop operates.

Here's what a `while` loop looks like in code:

```
while (condition)
{
  // statement(s) to execute repeatedly as long as condition is true
}
```

The `while` loop works very similarly to an `if` statement. The difference arises when the condition is true and the statement(s) are executed. Instead of continuing the execution with the statement that follows the closing brace (`}`), the condition is checked again. If the condition is still true, then the statement(s) are executed a second time, and a third, and will continue to be executed as long as the condition remains true. The first time the condition evaluates false (whether it's the first time it's checked, or the 101st), the execution jumps immediately to the statement that follows the `while` loop, after the closing brace.

Loops like these come in handy whenever you're working with long lists of items (such as jokes stored in a database … *hint, hint*), but for now I'll illustrate with a trivial example, counting to ten:

chapter3/count10.php *(excerpt)*

```
$count = 1;
while ($count <= 10)
{
  echo "$count ";
  ++$count;
}
```

This code may look a bit frightening, I know, but let me talk you through it line by line:

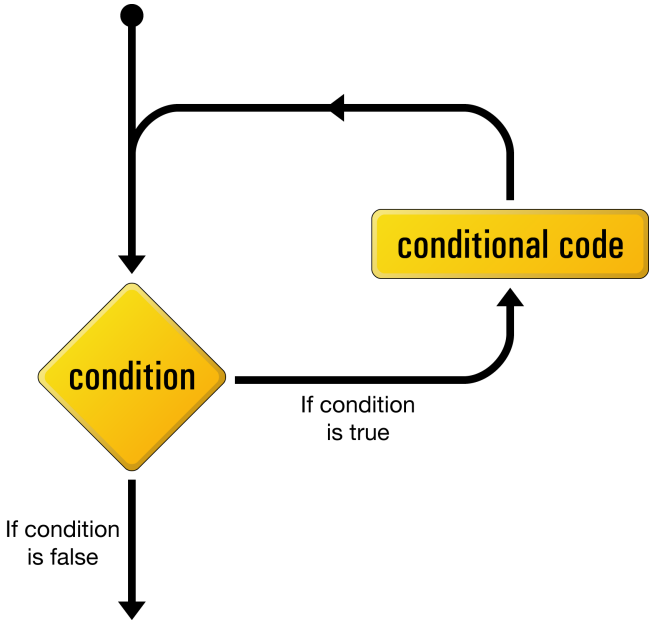


Figure 3.12. The logical flow of a `while` loop

`$count = 1;`

The first line creates a variable called `$count` and assigns it a value of `1`.

`while ($count <= 10)`

The second line is the start of a `while` loop, the condition for which is that the value of `$count` is less than or equal (`<=`) to 10.

`{`

The opening brace marks the beginning of the block of conditional code for the `while` loop. This conditional code is often called the **body** of the loop, and is executed over and over again, as long as the condition holds true.

`echo "$count ";`

This line simply outputs the value of `$count`, followed by a space. To make the code as readable as possible, I've used a double-quoted string to take advantage of variable interpolation (as explained in the section called "Variables, Operators, and Comments"), rather than use the string concatenation operator.

`++$count;`

The fourth line adds one to the value of `$count` (`++$count` is a shortcut for `$count = $count + 1`—either one would work).

`}`

The closing brace marks the end of the `while` loop's body.

So here's what happens when this piece of code is executed. The first time the condition is checked, the value of `$count` is `1`, so the condition is definitely true. The value of `$count` (`1`) is output, and `$count` is given a new value of `2`. The condition is still true the second time it's checked, so the value (`2`) is output and a new value (`3`) is assigned. This process continues, outputting the values `3`, `4`, `5`, `6`, `7`, `8`, `9`, and `10`. Finally, `$count` is given a value of `11`, and the condition is found to be false, which ends the loop.

The net result of the code is shown in Figure 3.13.

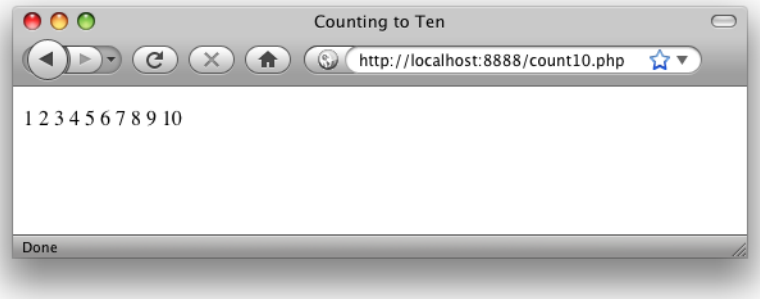


Figure 3.13. PHP demonstrates kindergarten-level math skills

The condition in this example used a new operator: `<=` (**less than or equal**). Other numerical comparison operators of this type include `>=` (**greater than or equal**), `<` (**less than**), `>` (**greater than**), and `!=` (**not equal**). That last one also works when comparing text strings, by the way.

Another type of loop that's designed specifically to handle examples like that above, in which we're counting through a series of values until some condition is met, is called a **`for` loop**. Figure 3.14 shows the structure of a `for` loop.

Here's what it looks like in code:

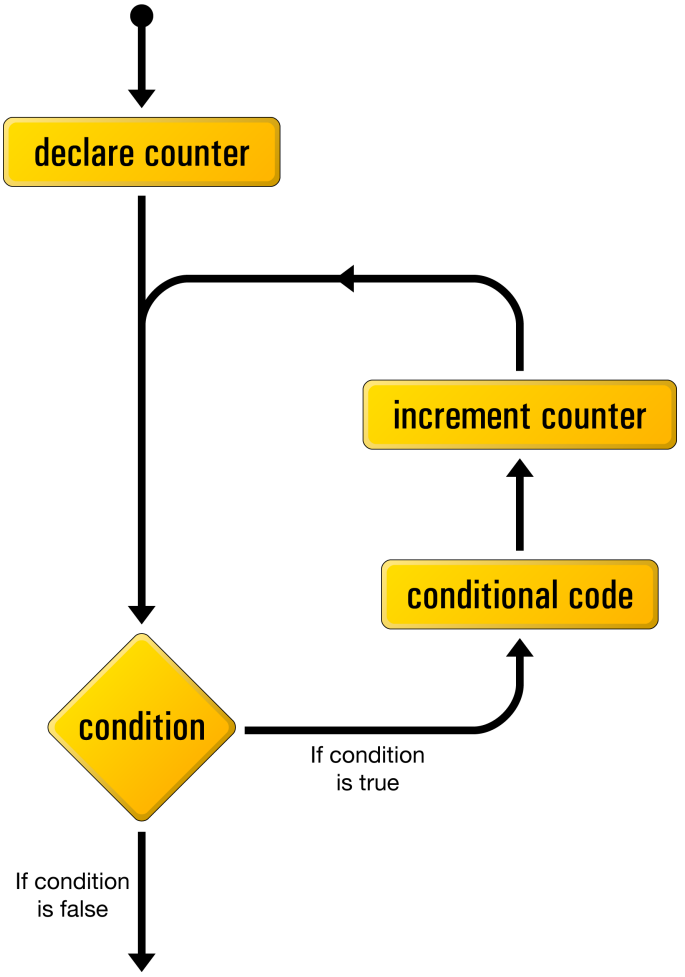


Figure 3.14. The logical flow of a `for` loop

```
for (declare counter; condition; increment counter)
{
  // statement(s) to execute repeatedly as long as condition is true
}
```

The *declare counter* statement is executed once at the start of the loop; the *condition* statement is checked each time through the loop, before the statements in the body are executed; the *increment counter* statement is executed each time through the loop, after the statements in the body.

Here's what the "counting to 10" example looks like when implemented with a `for` loop:

count10–for.php *(excerpt)*

```
for ($count = 1; $count <= 10; ++$count)
{
  echo "$count ";
}
```

As you can see, the statements that initialize and increment the `$count` variable join the condition on the first line of the `for` loop. Although, at first glance, the code seems a little more difficult to read, putting all the code that deals with controlling the loop in the same place actually makes it easier to understand once you're used to the syntax. Many of the examples in this book will use `for` loops, so you'll have plenty of opportunity to practice reading them.

Hiding the Seams

You're now armed with a working knowledge of the basic syntax of the PHP programming language. You understand that you can take any HTML web page, rename it with a **.php** file name extension, and inject PHP code into it to make it generate some or all of the page content on the fly. Not bad for a day's work!

Before we go any further, however, I want to stop and cast a critical eye over the examples we've discussed so far. Assuming your objective is to create database driven web sites that hold up to professional standards, there are a few unsightly blemishes we need to clean up.

The techniques in the rest of this chapter will add a coat of professional polish that can set your work apart from the crowd of amateur PHP developers out there. I'll rely on these techniques throughout the rest of this book to make sure that, no matter how simple the example, you can feel confident in the quality of the product you're delivering.

Avoid Advertising Your Technology Choices

The examples we've seen so far have contained a mixture of plain HTML files (with names ending in **.html**), and files that contain a mixture of HTML and PHP (with names ending in **.php**). Although this distinction between file types may be useful

to you, the developer, there is no reason your users need to be aware of which pages of your site rely on PHP code to generate them.

Furthermore, although PHP is a very strong choice of technology to build almost any database driven web site, the day may come when you want to switch from PHP to some new technology. When that day comes, do you really want all the URLs for dynamic pages on your site to become invalid as you change the file names to reflect your new language of choice?

These days, professional developers place a lot of importance on the URLs they put out into the world. In general, URLs should be as permanent as possible, so it makes no sense to embrittle them with little "advertisements" for the programming language you used to build each individual page.

An easy way to do away with the file name extensions in your URLs is to take advantage of directory indexes. When a URL points at a directory on your web server, instead of a particular file, the web server will look for a file named **index.html** or **index.php** inside that directory, and display that file in response to the request.

For example, take the **today.php** page that I introduced at the end of Chapter 1. Rename it from **today.php** to **index.php**. Then, instead of dropping it in the root of your web server, create a subdirectory name **today**, and drop the **index.php** file in there. Now, load http://localhost/today/ in your browser (or http://localhost:8080/today/, or similar if you need to specify a port number for your server).

Figure 3.15 shows the example with its new URL. This URL omits the unnecessary **.php** extension, and is shorter and more memorable—both desirable qualities when it comes to URLs today.

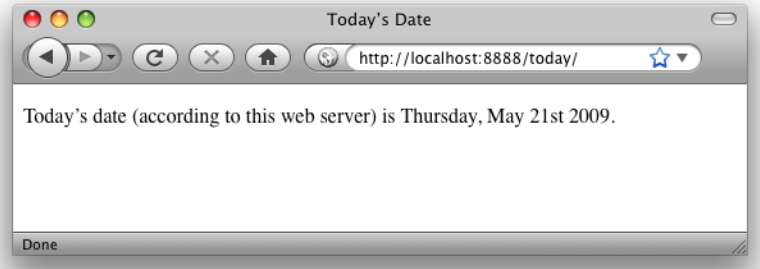


Figure 3.15. A more fashionable URL

Use PHP Templates

In the simple examples we've seen so far, inserting PHP code directly into your HTML pages has been a reasonable approach. As the amount of PHP code that goes into generating your average page grows, however, maintaining this mixture of HTML and PHP code can become unmanageable.

Particularly if you work in a team where the web designers are unsavvy, PHP-wise, having large blocks of cryptic PHP code intermingled with the HTML is a recipe for disaster. It's far too easy for designers to accidentally modify the PHP code, causing errors they'll be unable to fix.

A much more robust approach is to separate out the bulk of your PHP code, so that it resides in its own file, leaving the HTML largely unpolluted by PHP code.

The key to doing this is the PHP **`include` statement**. With an `include` statement, you can insert the contents of another file into your PHP code at the point of the statement. To show you how this works, let's rebuild the "count to ten" `for` loop example we looked at earlier.

Start by creating a new directory called **count10**, and create a file named **index.php** in this directory. Open the file for editing and type in this code:

chapter3/count10/index.php

```
<?php
$output = ''; (1)
for ($count = 1; $count <= 10; ++$count)
{
  $output .= "$count "; (2)
}

include 'count.html.php'; (3)
?>
```

Yes, that's the *complete* code for this file. It contains no HTML code whatsoever. The `for` loop should be familiar to you by now, but let me point out the interesting parts of this code:

(1) Instead of `echo`ing out the numbers 1 to 10, this script will add these numbers to a variable named `$output`. At the start of this script, therefore, we set this variable to contain an empty string.

(2) This line adds each number (followed by a space) to the end of the `$output` variable. The `.=` operator you see here is a shorthand way to add a value to the end of an existing string variable, by combining the assignment and string concatenation operators into one. The longhand version of this line is `$output = $output . "$count ";`, but the `.=` operator saves you some typing.

(3) This is an `include` statement, which instructs PHP to execute the contents of the **count.html.php** file at this location.[8]

[8] Outside of this book, you will often see includes coded with parentheses surrounding the filename, as if `include` were a function like `date` or `htmlspecialchars`, which is far from the case. These parentheses, when used, only serve to complicate the filename expression, and are therefore avoided in this book. The same goes for `echo`, another popular one-liner.

Since the final line of this file includes the **count.html.php** file, you should create this next:

chapter3/count10/count.html.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Counting to Ten</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
  </body>
</html>
```

This file is almost entirely plain HTML, except for the one line that outputs the value of the `$output` variable. This is the same `$output` variable that was created by the **index.php** file.

What we've created here is a **PHP template**—an HTML page with only very small snippets of PHP code that insert dynamically-generated values into an otherwise static HTML page. Rather than embedding the complex PHP code that generates those values in the page, we put the code to generate the values in a separate PHP script—**index.php** in this case.

Using PHP templates like this enables you to hand your templates over to HTML-savvy designers without worrying about what they might do to your PHP code. It also lets you focus on your PHP code without being distracted by the surrounding HTML code.

I like to name my PHP templates so that they end with **.html.php**. Although, as far as your web server is concerned, these are still **.php** files, the **.html.php** suffix serves as a useful reminder that these files contain both HTML and PHP code.

Many Templates, One Controller

What's nice about using `include` statements to load your PHP template files is that you can have *multiple* `include` statements in a single PHP script, and have it display different templates under different circumstances!

A PHP script that responds to a browser request by selecting one of several PHP templates to fill in and send back is commonly called a **controller**. A controller contains the logic that controls which template is sent to the browser.

Let's revisit one more example from earlier in this chapter: the welcome form that prompts a visitor for a first and last name.

We'll start with the PHP template for the form. For this, we can just reuse the **welcome8.html** file we created earlier. Create a directory named **welcome** and save a copy of **welcome8.html** called **form.html.php** into this directory. The only code you need to change in this file is the `action` attribute of the `<form>` tag:

chapter3/welcome/form.html.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Form Example</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <form action="" method="post">
      <div><label for="firstname">First name:
        <input type="text" name="firstname" id="firstname"/></label>
      </div>
      <div><label for="lastname">Last name:
        <input type="text" name="lastname" id="lastname"/></label>
      </div>
      <div><input type="submit" value="GO"/></div>
    </form>
  </body>
</html>
```

As you can see, we're leaving the `action` attribute blank. This tells the browser to submit the form back to the same URL from which it received the form—in this case, the URL of the controller that included this template file.

Let's take a look at the controller for this example. Create an **index.php** script in the **welcome** directory alongside your form template. Type the following code into this file:

chapter3/welcome/index.php

```
<?php
if (!isset($_REQUEST['firstname'])) 1
{
  include 'form.html.php'; 2
}
else 3
{
  $firstname = $_REQUEST['firstname'];
  $lastname = $_REQUEST['lastname'];
  if ($firstname == 'Kevin' and $lastname == 'Yank')
  {
    $output = 'Welcome, oh glorious leader!'; 4
  }
  else
  {
    $output = 'Welcome to our web site, ' .
        htmlspecialchars($firstname, ENT_QUOTES, 'UTF-8') . ' ' .
        htmlspecialchars($lastname, ENT_QUOTES, 'UTF-8') . '!';
  }

  include 'welcome.html.php'; 5
}
?>
```

This code should look fairly familiar at first glance; it's a lot like the **welcome8.php** script we wrote earlier. Let me explain the differences:

1. The first thing the controller needs to do is decide whether the current request is a submission of the form in **form.html.php** or not. You can do this by checking if the request contains a `firstname` variable. If it does, PHP will have stored the value in `$_REQUEST['firstname']`.

`isset` is a built-in PHP function that will tell you if a particular variable (or array element) has been assigned a value or not. If `$_REQUEST['firstname']` has a value, `isset($_REQUEST['firstname'])` will be true. If `$_REQUEST['firstname']` lacks a value, `isset($_REQUEST['firstname'])` will be false.

For the sake of readability, I like to put the code that sends the form first in my controller. What we need this `if` statement to check, therefore, is if `$_REQUEST['firstname']` is *not* set. To do this, we use the **not operator** (`!`). By putting this operator before the name of a function, you reverse the value that function returns from true to false, or from false to true.

Thus, if the request does *not* contain a `firstname` variable, then `!isset($_REQUEST['firstname'])` will return true, and the body of the `if` statement will be executed.

2. If the request is not a form submission, the controller includes the **form.html.php** file to display the form.

3. If the request *is* a form submission, the body of the `else` statement is executed instead.

   This code pulls the `firstname` and `lastname` variables out of the `$_REQUEST` array, and then generates the appropriate welcome message for the name submitted.

4. Instead of `echo`ing the welcome message, the controller stores the welcome message in a variable named `$output`.

5. After generating the appropriate welcome message, the controller includes the **welcome.html.php** template, which will display that welcome message.

All that's left is to write the **welcome.html.php** template. Here it is:

chapter3/welcome/welcome.html.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Form Example</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
  </body>
</html>
```

That's it! Fire up your browser and point it at http://localhost/welcome/ (or http://localhost:8080/welcome/ or similar if you need to specify a port number for your web server). You'll be prompted for your name, and when you submit the form, you'll see the appropriate welcome message. The URL should stay the same throughout this process.

One of the benefits of maintaining the same URL throughout the process of prompting the user for a name and displaying the welcome message is that the user can bookmark the page at any time during this process and gain a sensible result: when the user next returns, whether the form page or the welcome message was bookmarked, the form will be present itself once again. In the previous version of this example, where the welcome message had its own URL, returning to that URL without submitting the form would have generated a broken welcome message ("Welcome to our web site, !").

Why So Forgetful?

In Chapter 9 I'll show you how to remember the user's name between visits.

Bring On the Database

In this chapter, we've seen the PHP server-side scripting language in action as we've explored all the basic language features: statements, variables, operators, comments, and control structures. The sample applications we've seen have been reasonably simple, but despite this we've taken the time to ensure they have attractive URLs, and that the HTML templates for the pages they generate are uncluttered by the PHP code that controls them.

As you may have begun to suspect, the real power of PHP is in its hundreds (even thousands) of built-in functions that let you access data in a MySQL database, send email, dynamically generate images, and even create Adobe Acrobat PDF files on the fly.

In Chapter 4, we'll delve into the MySQL functions built into PHP, and see how to publish the joke database we created in Chapter 2 to the Web. This chapter will set the scene for the ultimate goal of this book—creating a complete content management system for your web site in PHP and MySQL.

Chapter 4

Publishing MySQL Data on the Web

This is it—the stuff you signed up for! In this chapter, you'll learn how to take information stored in a MySQL database and display it on a web page for all to see.

So far, you've installed and learned the basics of MySQL, a relational database engine, and PHP, a server-side scripting language. Now you're ready to learn how to use these new tools together to create a true database driven web site!

The Big Picture

Before we leap forward, it's worth taking a step back for a clear picture of our ultimate goal. We have two powerful tools at our disposal: the PHP scripting language and the MySQL database engine. It's important to understand how these will fit together.

The whole idea of a database driven web site is to allow the content of the site to reside in a database, and for that content to be pulled from the database dynamically to create web pages for people to view with a regular web browser. So, at one end of the system you have a visitor to your site who uses a web browser to request a page, and expects to receive a standard HTML document in return. At the other end

you have the content of your site, which sits in one or more tables in a MySQL database that understands only how to respond to SQL queries (commands).

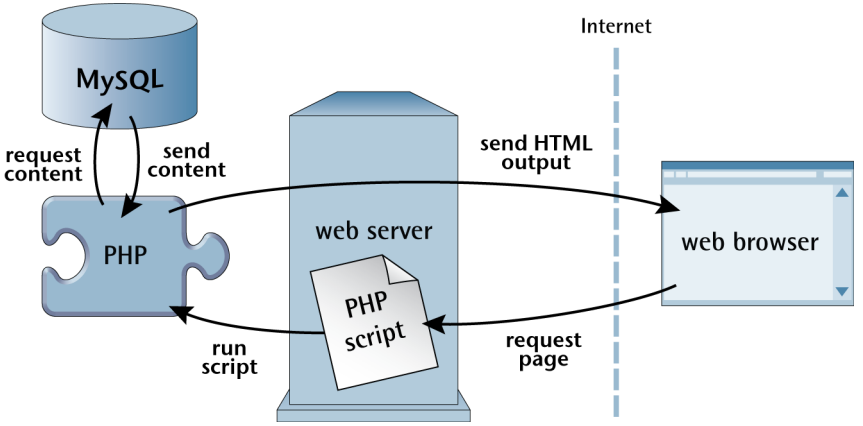


Figure 4.1. PHP retrieves MySQL data to produce web pages

As shown in Figure 4.1, the PHP scripting language is the go-between that speaks both languages. It processes the page request and fetches the data from the MySQL database (using SQL queries just like those you used to create a table of jokes in Chapter 2), then spits it out dynamically as the nicely formatted HTML page that the browser expects.

Just so it's clear and fresh in your mind, this is what will happen when a person visits a page on your database driven web site:

1. The visitor's web browser requests the web page using a standard URL.
2. The web server software (typically Apache) recognizes that the requested file is a PHP script, so the server fires up the PHP interpreter to execute the code contained in the file.
3. Certain PHP commands (which will be the focus of this chapter) connect to the MySQL database and request the content that belongs in the web page.
4. The MySQL database responds by sending the requested content to the PHP script.
5. The PHP script stores the content into one or more PHP variables, then uses `echo` statements to output the content as part of the web page.

6. The PHP interpreter finishes up by handing a copy of the HTML it has created to the web server.

7. The web server sends the HTML to the web browser as it would a plain HTML file, except that instead of coming directly from an HTML file, the page is the output provided by the PHP interpreter.

Connecting to MySQL with PHP

Before you can retrieve content out of your MySQL database for inclusion in a web page, you must know how to establish a connection to MySQL from inside a PHP script. Back in Chapter 2, you used a program called `mysql` that allowed you to make such a connection from the command prompt. Just as that program could connect directly to a running MySQL server, so too can the PHP interpreter; support for connecting to MySQL is built right into the language in the form of a library of built-in functions.

The built-in function `mysqli_connect` establishes a connection to a MySQL server:

```
mysqli_connect(hostname, username, password)
```

You may remember from Chapter 3 that PHP functions usually return a value when they're called. The `mysqli_connect` function, for example, returns a **link identifier** that identifies the connection that has been established. Since we intend to make use of the connection, we should hold onto this value. Here's an example of how we might connect to our MySQL server:

chapter4/connect/index.php *(excerpt)*

```
$link = mysqli_connect('localhost', 'root', 'password');
```

As described above, the values of the three function parameters may differ for your MySQL server; at the very least, you'll need to substitute in the root password you established for your MySQL server. What's important to see here is that the value returned by `mysqli_connect` is stored in a variable named `$link`.

As the MySQL server is a completely separate piece of software from the web server, we must consider the possibility that the server may be unavailable or inaccessible due to a network outage, or because the username/password combination you

provided is rejected by the server. In such cases, the `mysqli_connect` function returns `FALSE`, instead of a connection identifier, as no connection is established. This allows us to react to such failures using an `if` statement:

chapter4/connect/index.php *(excerpt)*

```
$link = mysqli_connect('localhost', 'root', 'password');
if (!$link)
{
  $output = 'Unable to connect to the database server.';
  include 'output.html.php';
  exit();
}
```

The condition in this `if` statement uses the not operator (`!`) to make the condition true when `$link` has a value of `false` (that is, when the connection attempt has failed). If the connection succeeds, `$link` will have a value that's considered true, which will make `!$link` false. In short, the body of the `if` statement is executed only if the connection fails.

Within the body of the `if` statement, we set the variable `$output` to contain a message about what went wrong. We then include the template **output.html.php**. This is a generic template that simply outputs the value of the `$output` variable:

chapter4/connect/output.html.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>PHP Output</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
  </body>
</html>
```

Finally, after outputting the message, the body of the `if` statement calls the built-in `exit` function.

`exit` is the first example in this book of a function that can be called with no parameters. When called this way, all this function does is cause PHP to stop executing the script at this point. This ensures that the rest of the code in our controller (which in most cases will depend on a successful database connection) will not be executed if the connection has failed.

Assuming the connection succeeds, however, you need to configure it before use. As I mentioned briefly in Chapter 3, you should use UTF-8 encoded text in your web sites to maximize the range of characters that your users will have at their disposal when filling in forms on your site. By default, when PHP connects to MySQL, it once again uses the simpler ISO-8859-1 encoding instead of UTF-8. You must therefore follow up your `mysqli_connect` code with a call to `mysqli_set_charset`—another built-in PHP function:

```
mysqli_set_charset($link, 'utf8')
```

Notice we use the `$link` variable that contains the MySQL link identifier to tell the function which database connection to use. This function returns `true` when it's successful and `false` if an error occurs. Once again, it's prudent to use an `if` statement to handle errors:

chapter4/connect/index.php *(excerpt)*

```
if (!mysqli_set_charset($link, 'utf8'))
{
  $output = 'Unable to set database connection encoding.';
  include 'output.html.php';
  exit();
}
```

Note that this time, instead of assigning the result of the function to a variable and then checking if the variable is true or false, I have simply used the function call itself as the condition. This may look a little strange, but it's a very commonly used shortcut. To check whether the condition is true or false, PHP executes the function and then checks its return value—exactly what we need to happen.

As in Chapter 2 when you connected to the MySQL server using the **mysql** program, once you've established a connection the usual next step is to select the database with which you want to work. Let's say you want to work with the joke database you created in Chapter 2. This database was called `ijdb`. Selecting that database in PHP is just a matter of another function call:

```
mysqli_select_db($link, 'ijdb');
```

`mysqli_select_db` simply sets the selected database (`'ijdb'`) for the specified database connection (`$link`). Yet again, it's best to guard against errors with an `if` statement:

chapter4/connect/index.php *(excerpt)*

```
if (!mysqli_select_db($link, 'ijdb'))
{
  $output = 'Unable to locate the joke database.';
  include 'output.html.php';
  exit();
}
```

To polish off this example, let's display a status message that indicates when everything has gone right. Here's the complete code of our controller:

chapter4/connect/index.php

```
<?php
$link = mysqli_connect('localhost', 'root', 'password');
if (!$link)
{
  $output = 'Unable to connect to the database server.';
  include 'output.html.php';
  exit();
}

if (!mysqli_set_charset($link, 'utf8'))
{
  $output = 'Unable to set database connection encoding.';
  include 'output.html.php';
  exit();
}
```

```
if (!mysqli_select_db($link, 'ijdb'))
{
  $output = 'Unable to locate the joke database.';
  include 'output.html.php';
  exit();
}

$output = 'Database connection established.';
include 'output.html.php';
?>
```

Fire up this example in your browser (if you put the **index.php** and **output.html.php** files in a directory named **connect** on your web server, the URL will be like http://localhost/connect/). If your MySQL server is up and running and everything works the way it should, you should see the message indicating success in Figure 4.2.

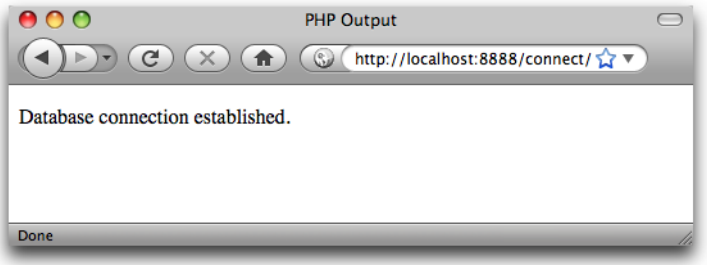


Figure 4.2. A successful connection

If PHP is unable to connect to your MySQL server, or if the username and password you provided are incorrect, you'll instead see a similar screen to that in Figure 4.3. To make sure your error handling code is working properly, you might want to misspell your password intentionally to test it out.

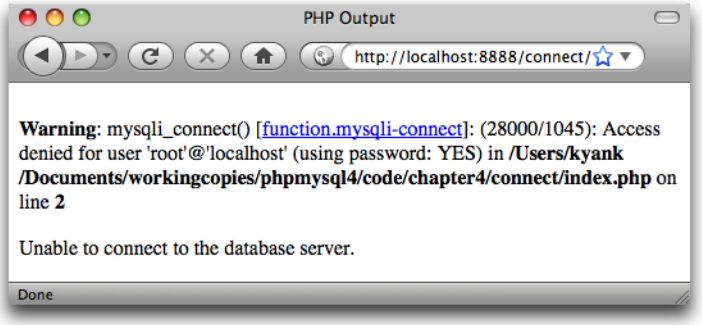


Figure 4.3. A connection failure

What PHP Error?

Depending on your web server's PHP configuration, you may or may not see the first paragraph shown in Figure 4.3. This warning message is automatically generated by PHP if it's configured to display errors. These detailed errors can be invaluable tools for diagnosing problems with your code during development. Since you'd probably prefer to keep this kind of technical information hidden once your site is live on the Web, it's common to switch off these errors on production servers.

If you installed Apache yourself, chances are this message will be displayed. If you're using a bundled Apache solution (like WampServer or MAMP), PHP error display may be switched off by default. To display these errors (they're especially helpful in development when you're trying to determine the cause of a problem), you need to open your server's **php.ini** file and set the `display_errors` option to `On`. You can access WampServer's **php.ini** file from the system tray menu. MAMP's **php.ini** file is in the **/Applications/MAMP/conf/php5** folder on your system.

If PHP connects to your MySQL server and then fails to find the `ijdb` database, you'll see a similar message to Figure 4.4. Once again, you should probably test your error handling code by intentionally misspelling your database name.

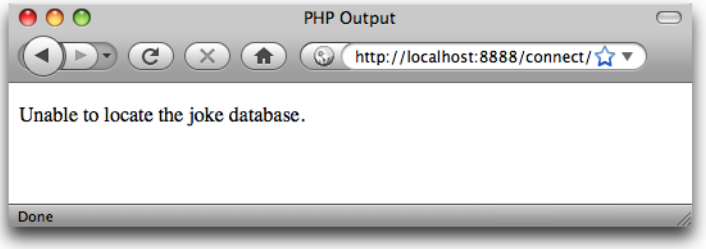


Figure 4.4. A connection failure

With a connection established and a database selected, you're ready to begin using the data stored in the database.

PHP Automatically Disconnects

You might be wondering what happens to the connection with the MySQL server after the script has finished executing. While PHP does have a function for disconnecting from the server (`mysqli_close`), PHP will automatically close any open database connections when they're no longer needed, so you can usually just let PHP clean up after you.

Sending SQL Queries with PHP

In Chapter 2, we connected to the MySQL database server using a program called `mysql` that allowed us to type SQL queries (commands) and view the results of those queries immediately. In PHP, a similar mechanism exists: the `mysqli_query` function.

```
mysqli_query(link, query)
```

Here *query* is a string that contains the SQL query you want to execute. As with `mysqli_select_db`, you must also provide the MySQL link identifier returned by `mysqli_connect`.

What this function returns will depend on the type of query being sent. For most SQL queries, `mysqli_query` returns either `true` or `false` to indicate success or failure respectively. Consider the following example, which attempts to create the `joke` table we created in Chapter 2:

chapter4/createtable/index.php *(excerpt)*

```
$sql = 'CREATE TABLE joke (
      id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
      joketext TEXT,
      jokedate DATE NOT NULL
    ) DEFAULT CHARACTER SET utf8';
if (!mysqli_query($link, $sql))
{
  $output = 'Error creating joke table: ' . mysqli_error($link);
  include 'output.html.php';
  exit();
}

$output = 'Joke table successfully created.';
include 'output.html.php';
```

Note once again we use the same `if` statement technique to handle possible errors produced by the query. This example also uses the `mysqli_error` function to retrieve a detailed error message from the MySQL server. Figure 4.5 shows the error that's displayed when the `joke` table already exists, for example.

Figure 4.5. The `CREATE TABLE` query fails because the table already exists

For `DELETE`, `INSERT`, and `UPDATE` queries (which serve to modify stored data), MySQL also keeps track of the number of table rows (entries) that were affected by the query. Consider the SQL command below, which we used in Chapter 2 to set the dates of all jokes that contained the word "chicken":

chapter4/updatechicken/index.php *(excerpt)*

```
$sql = 'UPDATE joke SET jokedate="2010-04-01"
    WHERE joketext LIKE "%chicken%"';
if (!mysqli_query($link, $sql))
{
  $output = 'Error performing update: ' . mysqli_error($link);
  include 'output.html.php';
  exit();
}
```

When we execute this query, we can use the `mysql_affected_rows` function to view the number of rows that were affected by this update:

chapter4/updatechicken/index.php *(excerpt)*

```
$output = 'Updated ' . mysqli_affected_rows($link) . ' rows.';
include 'output.html.php';
```

Figure 4.6 shows the output of this example, assuming you only have one "chicken" joke in your database.

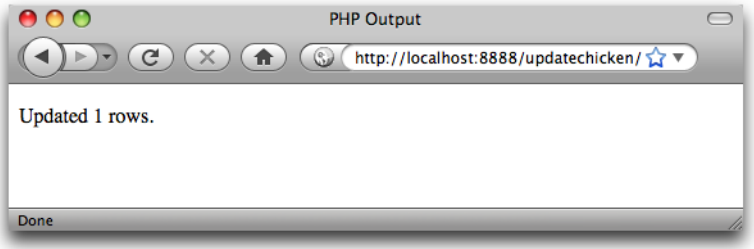


Figure 4.6. The number of database records updated is displayed

If you refresh the page to run the same query again, you should see the message change as shown in Figure 4.7 to indicate that no rows were updated, since the new date being applied to the jokes is the same as the existing date.

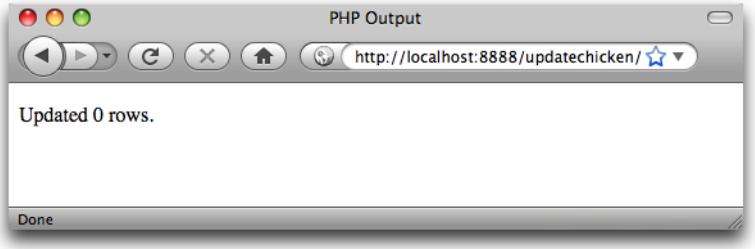


Figure 4.7. MySQL lets you know you're wasting its time

`SELECT` queries are treated a little differently as they can retrieve a lot of data, and PHP provides ways to handle that information.

Handling `SELECT` Result Sets

For most SQL queries, the `mysqli_query` function returns either `true` (success) or `false` (failure). For `SELECT` queries, more information is needed. You'll recall that `SELECT` queries are used to view stored data in the database. In addition to indicating whether the query succeeded or failed, PHP must also receive the results of the query. Thus, when it processes a `SELECT` query, `mysqli_query` returns a **result set**, which contains a list of all the rows (entries) returned from the query. `false` is still returned if the query fails for any reason:

chapter4/listjokes/index.php *(excerpt)*

```
$result = mysqli_query($link, 'SELECT joketext FROM joke');
if (!$result)
{
  $error = 'Error fetching jokes: ' . mysqli_error($link);
  include 'error.html.php';
  exit();
}
```

As before, errors are displayed using a very simple PHP template:

chapter4/listjokes/error.html.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>PHP Error</title>
```

```
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>
      <?php echo $error; ?>
    </p>
  </body>
</html>
```

Provided that no error was encountered in processing the query, the above code will store a result set into the variable `$result`. This result set contains the text of all the jokes stored in the `joke` table. As there's no practical limit on the number of jokes in the database, that result set can be quite big.

I mentioned back in Chapter 3 that the `while` loop is a useful control structure for dealing with large amounts of data. Here's an outline of the code that will process the rows in a result set one at a time:

```
while ($row = mysqli_fetch_array($result))
{
  // process the row…
}
```

The condition for the `while` loop is probably different to the conditions you're used to, so let me explain how it works. Consider the condition as a statement all by itself:

```
$row = mysqli_fetch_array($result);
```

The `mysqli_fetch_array` function accepts a result set as a parameter (stored in the `$result` variable in this case), and returns the next row in the result set as an array (we discussed arrays in Chapter 3). When there are no more rows in the result set, `mysqli_fetch_array` instead returns `false`.

Now, the above statement assigns a value to the `$row` variable, but, at the same time, the statement as a whole takes on that same value. This is what lets you use the statement as a condition in the `while` loop. Since a `while` loop will keep looping until its condition evaluates to `false`, this loop will occur as many times as there are rows in the result set, with `$row` taking on the value of the next row each time

the loop executes. All that's left to figure out is how to retrieve the values out of the `$row` variable each time the loop runs.

Rows of a result set returned by `mysqli_fetch_array` are represented as associative arrays. The indices are named after the table columns in the result set. If `$row` is a row in our result set, then `$row['joketext']` is the value in the `joketext` column of that row.

Our goal in this code is to store away the text of all the jokes so we can display them in a PHP template. The best way to do this is to store each joke as a new item in an array, `$jokes`:

chapter4/listjokes/index.php *(excerpt)*

```
while ($row = mysqli_fetch_array($result))
{
  $jokes[] = $row['joketext'];
}
```

With the jokes pulled out of the database, we can now pass them along to a PHP template (**jokes.html.php**) for display.

To summarize, here's the complete code of the controller for this example:

chapter4/listjokes/index.php

```
<?php
$link = mysqli_connect('localhost', 'root', 'password');
if (!$link)
{
  $error = 'Unable to connect to the database server.';
  include 'error.html.php';
  exit();
}

if (!mysqli_set_charset($link, 'utf8'))
{
  $output = 'Unable to set database connection encoding.';
  include 'output.html.php';
  exit();
}

if (!mysqli_select_db($link, 'ijdb'))
```

```
{
  $error = 'Unable to locate the joke database.';
  include 'error.html.php';
  exit();
}

$result = mysqli_query($link, 'SELECT joketext FROM joke');
if (!$result)
{
  $error = 'Error fetching jokes: ' . mysqli_error($link);
  include 'error.html.php';
  exit();
}

while ($row = mysqli_fetch_array($result))
{
  $jokes[] = $row['joketext'];
}

include 'jokes.html.php';
?>
```

All that's left to complete this example is to write the **jokes.html.php** template.

In this template, for the first time we need to display the contents of an array, rather than just a simple variable. The most common way to process an array in PHP is to use a loop. We have already seen `while` loops and `for` loops; another type of loop, which is particularly helpful for processing arrays, is the `foreach` loop:

```
foreach (array as $item)
{
  // process each $item
}
```

Instead of a condition, the parentheses at the top of a `foreach` loop contain an array, followed by the keyword `as`, and then the name of a new variable that will be used to store each item of the array in turn. The body of the loop is then executed once for each item in the array; each time, that item is stored in the specified variable so that the code can access it directly.

It's common to use a `foreach` loop in a PHP template to display each item of an array in turn. Here's how this might look for our `$jokes` array:

```
<?php
foreach ($jokes as $joke)
{
?>
  <!-- Code to output each $joke -->
<?php
}
?>
```

With this blend of PHP code to describe the loop and HTML code to display it, this code looks rather untidy. Because of this, it's common to use an alternative way of writing the `foreach` loop when it's used in a template:

```
foreach (array as $item):
  // process each $item
endforeach;
```

Here's how this form of the code looks in a template:

```
<?php foreach ($jokes as $joke): ?>
  <!-- Code to output each $joke -->
<?php endforeach; ?>
```

With this new tool in hand, we can write our template to display the list of jokes:

chapter4/listjokes/jokes.html.php

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>List of Jokes</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
  </head>
  <body>
    <p>Here are all the jokes in the database:</p>
    <?php foreach ($jokes as $joke): ?>
      <blockquote><p>
        <?php echo htmlspecialchars($joke, ENT_QUOTES, 'UTF-8'); ?>
      </p></blockquote>
```

```
    <?php endforeach; ?>
  </body>
</html>
```

Each joke is displayed in a paragraph (`<p>`) contained within a block quote (`<blockquote>`), since we're effectively quoting the author of each joke in this page.

Because jokes might conceivably contain characters that could be interpreted as HTML code (for example, <, >, or &), we must use `htmlspecialchars` to ensure that these are translated into HTML character entities (that is, `&lt;`, `&gt;`, and `&amp;`) so that they're displayed correctly.

Figure 4.8 shows what this page looks like once you've added a couple of jokes to the database.

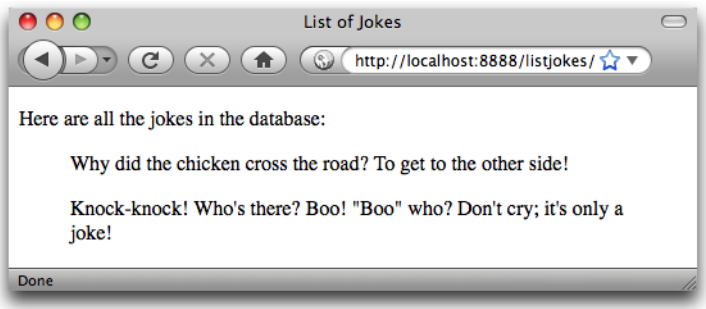


Figure 4.8. All my best material—in one place!

Inserting Data into the Database

In this section, I'll demonstrate how to use the tools at your disposal to enable site visitors to add their own jokes to the database.

If you want to let visitors to your site type in new jokes, you'll obviously need a form. Here's a template for a form that will fit the bill:

chapter4/addjoke/form.html.php *(excerpt)*

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
  <head>
    <title>Add Joke</title>
    <meta http-equiv="content-type"
        content="text/html; charset=utf-8"/>
    <style type="text/css">
    textarea {
      display: block;
      width: 100%;
    }
    </style>
  </head>
  <body>
    <form action="?" method="post">
      <div>
        <label for="joketext">Type your joke here:</label>
        <textarea id="joketext" name="joketext" rows="3" cols="40">
➥</textarea>
      </div>
      <div><input type="submit" value="Add"/></div>
    </form>
  </body>
</html>
```

As we've seen before, when submitted this form will request the same PHP script that generated the form—the controller script (**index.php**). You'll notice, however, that instead of leaving the `action` attribute empty (`""`), we set its value to `?`. As we'll see in a moment, the URL used to display the form in this example will feature a query string, and setting the `action` to `?` strips that query string off the URL when submitting the form.

Figure 4.9 shows what this form looks like in a browser.

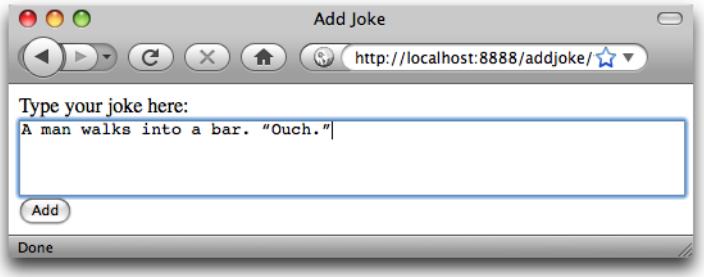


Figure 4.9. Another nugget of comic genius is added to the database

When this form is submitted, the request will include a variable, `joketext`, that contains the text of the joke as typed into the text area. This variable will then appear in the `$_POST` and `$_REQUEST` arrays created by PHP.

Let's tie this form into the preceding example, which displayed the list of jokes in the database. Add a link to the top of the list that invites the user to add a joke:

chapter4/addjoke/jokes.html.php *(excerpt)*

```
<body>
  <p><a href="?addjoke">Add your own joke</a></p>
  <p>Here are all the jokes in the database:</p>
```

Like the form, this link points back to the very same PHP script used to generate this page, but this time it adds a query string (`?addjoke`), indicating the user's intention to add a new joke. Our controller can detect this query string and use it as a signal to display the "Add Joke" form instead of the list of jokes.

Let's make the necessary changes to the controller now:

chapter4/addjoke/index.php *(excerpt)*

```
if (isset($_GET['addjoke']))
{
  include 'form.html.php';
  exit();
}
```

This opening `if` statement checks if the query string contains a variable named `addjoke`. This is how we detect that the user clicked the new link. Even though there is no value specified by the query string (`?addjoke`) for the `addjoke` variable, it does create it, which we can detect with `isset($_GET['addjoke'])`.

When we detect this variable, we display the form by including **form.html.php**, and then exit.

Once the user fills out the form and submits it, that form submission results in another request to this controller. This we detect by checking if `$_POST['joketext']` is set:

chapter4/addjoke/index.php *(excerpt)*

```
if (isset($_POST['joketext']))
{
```

To insert the submitted joke into the database, we must run an `INSERT` query using the value stored in `$_POST['joketext']` to fill in the `joketext` column of the `joke` table. This might lead you to write some code like this:

```
$sql = 'INSERT INTO joke SET
    joketext="' . $_POST['joketext'] . '",
    jokedate="today's date"';
```

There is a serious problem with this code, however: the contents of `$_POST['joketext']` are entirely under the control of the user who submitted the form. If a malicious user were to type just the right sort of SQL code into the form, this script would feed it to your MySQL server without question. This type of attack is called an **SQL injection attack**, and in the early days of PHP it was one of the most common security holes that hackers found and exploited in PHP-based web sites.

These attacks were so feared, in fact, that the team behind PHP added some built-in protection against SQL injections to the language that remains enabled by default in many PHP installations today. Called **magic quotes**, this protective feature of PHP automatically analyzes all values submitted by the browser and inserts backslashes (`\`) in front of any *dangerous* characters, like apostrophes—which can cause problems if they're included in an SQL query inadvertently.

The problem with the **magic quotes** feature is that it causes as many problems as it prevents. Firstly, the characters that it detects and the method it uses to sanitize them (prefixing them with a backslash) are only valid in some circumstances. Depending on the character encoding of your site, and the database server you're using, these measures may be completely ineffective.

Secondly, when a submitted value is used for some purpose *other* than creating an SQL query, those backslashes can be really bothersome. I mentioned this briefly in Chapter 2 when, in the welcome message example, the magic quotes feature would insert a spurious backslash into the user's last name if it contained an apostrophe.

In short, magic quotes was a bad idea, so much so that it's scheduled to be removed from PHP in version 6. In the meantime, however, you have to deal with the problems it creates in your code. The easiest way to do this is to detect if magic quotes is enabled on your web server and, if it is, to *undo* the modifications it has made to the submitted values.[1] Thankfully, the PHP Manual[2] provides a snippet of code that will do exactly this:

chapter4/addjoke/index.php *(excerpt)*

```
if (get_magic_quotes_gpc())
{
  function stripslashes_deep($value)
  {
    $value = is_array($value) ?
        array_map('stripslashes_deep', $value) :
        stripslashes($value);

    return $value;
  }

  $_POST = array_map('stripslashes_deep', $_POST);
  $_GET = array_map('stripslashes_deep', $_GET);
  $_COOKIE = array_map('stripslashes_deep', $_COOKIE);
  $_REQUEST = array_map('stripslashes_deep', $_REQUEST);
}
```

[1] You can disable magic quotes—and save your web server a lot of work—by setting the `magic_quotes_gpc` option in your **php.ini** file to `Off`. To make sure your code still functions if this setting is changed, however, you should still deal with magic quotes in your code when it's enabled.
[2] http://www.php.net/manual/en/security.magicquotes.disabling.php

Avoid wasting time trying to understand the inner workings of this code; to keep the code short, it uses several advanced PHP features that we've yet to see—and one or two others that are beyond the scope of this book. Rather, just drop this code into the top of your controller—and indeed any other PHP script that will receive user input in the form of query variables or a form submission (or, as we'll learn in Chapter 9, browser cookies). And be assured; from this point forward, I'll remind you whenever this code is required by an example.[3]

With the damage done by magic quotes reversed, you must now prepare those values that you *do* intend to use in your SQL query. Just as it provides `htmlspecialchars` for outputting user-submitted values into HTML code, PHP provides a function that prepares a user-submitted value so that you can use it safely in your SQL query: `mysqli_real_escape_string`. Not the most elegant name, but it does the trick. Here's how you use it:

```
$joketext = mysqli_real_escape_string($link, $_POST['joketext']);
$sql = 'INSERT INTO joke SET
    joketext="' . $joketext . '",
    jokedate="today's date"';
```

This code first uses `mysqli_real_escape_string` to store a "query safe" version of the contents of `$_POST['joketext']` in the new variable `$joketext`. It then uses this variable to insert the submitted value into the `INSERT` query as the value of the `joketext` column.

The lingering question in this code is how to assign today's date to the `jokedate` field. We *could* write some fancy PHP code to generate today's date in the *`YYYY-MM-DD`* form that MySQL requires, but it turns out MySQL itself has a function to do this: `CURDATE`:

```
$joketext = mysqli_real_escape_string($link, $_POST['joketext']);
$sql = 'INSERT INTO joke SET
    joketext="' . $joketext . '",
    jokedate=CURDATE()';
```

[3] In Chapter 6 I'll show you how to manage the burden of repeatedly including this code snippet in your controller code.

The MySQL function `CURDATE` is used here to assign the current date as the value of the `jokedate` column. MySQL actually has dozens of these functions, but we'll introduce them only as required. Appendix B provides a reference that describes all commonly used MySQL functions.

Now that we have our query, we can complete the `if` statement we started above to handle submissions of the "Add Joke" form. We can execute our `INSERT` query by using the `mysqli_query` function:

chapter4/addjoke/index.php *(excerpt)*

```
if (isset($_POST['joketext']))
{
  $joketext = mysqli_real_escape_string($link, $_POST['joketext']);
  $sql = 'INSERT INTO joke SET
      joketext="' . $joketext . '",
      jokedate=CURDATE()';
  if (!mysqli_query($link, $sql))
  {
    $error = 'Error adding submitted joke: ' . mysqli_error($link);
    include 'error.html.php';
    exit();
  }

  header('Location: .');
  exit();
}
```

But wait! This `if` statement has one more new trick up its sleeve. Once we've added the new joke to the database, instead of displaying the PHP template as previously, we want to redirect the user's browser back to the list of jokes. That way they are able to see the newly added joke among them. That's what the two lines highlighted in bold at the end of the `if` statement above do.

Your first instinct in order to achieve the desired result might be to allow the controller, after adding the new joke to the database, simply to fetch the list of jokes from the database and display the list using the **jokes.html.php** template as usual. The problem with doing this is that the resulting page, from the browser's perspective, would be the effect of having submitted the "Add Joke" form. If the user were then to refresh the page, the browser would resubmit that form, causing another copy of the new joke to be added to the database! This is rarely the desired behaviour.

Instead, we want the browser to treat the updated list of jokes as a normal web page, able to be reloaded without resubmitting the form. The way to do this is to answer the browser's form submission with an **HTTP redirect**[4]—a special response that tells the browser "the page you're looking for is over *here*."

The PHP `header` function provides the means of sending special server responses like this one, by letting you insert special **headers** into the response sent to the server. In order to signal a redirect, you must send a `Location` header with the URL of the page to which you wish to direct the browser:

```
header('Location: URL');
```

In this case, we want to send the browser back to the very same page—our controller. We're asking the browser to submit another request—this time, without a form submission attached to it—rather than sending the browser to another location. Since we want to point the browser at our controller (**index.php**) using the URL of the parent directory, we can simply tell the browser to reload the current directory, which is expressed as a period (`.`).

Thus, the two lines that redirect the browser back to our controller after adding the new joke to the database:

chapter4/addjoke/index.php *(excerpt)*

```
  header('Location: .');
  exit();
}
```

[4] HTTP stands for HyperText Transfer Protocol, and is the language that describes the request/response communications that are exchanged between the visitor's web browser and your web server.

$_SERVER['PHP_SELF'] is the URL of the current page

Another common means of obtaining the URL of the current page in PHP is with `$_SERVER['PHP_SELF']`.

Like `$_GET`, `$_POST`, and `$_REQUEST`, `$_SERVER` is an array variable that is automatically created by PHP. `$_SERVER` contains a whole bunch of information supplied by your web server. In particular, `$_SERVER['PHP_SELF']` will always be set to the URL of the PHP script that your web server used to generate the current page.

Unfortunately, because the web server automatically translates a request for http://localhost/addjoke/ to a request for http://localhost/addjoke/index.php, `$_SERVER['PHP_SELF']` will contain the latter URL. Redirecting the browser to . lets us preserve the shorter, more memorable form of the URL.

For this reason, I have avoided using `$_SERVER['PHP_SELF']` in this book. Since it's so commonly used in basic PHP examples around the Web, however, I thought you might like to know what it does.

The rest of the controller is responsible for displaying the list of jokes as before. Here's the complete code of the controller:

chapter4/addjoke/index.php

```
<?php
if (get_magic_quotes_gpc())
{
  function stripslashes_deep($value)
  {
    $value = is_array($value) ?
        array_map('stripslashes_deep', $value) :
        stripslashes($value);

    return $value;
  }

  $_POST = array_map('stripslashes_deep', $_POST);
  $_GET = array_map('stripslashes_deep', $_GET);
  $_COOKIE = array_map('stripslashes_deep', $_COOKIE);
  $_REQUEST = array_map('stripslashes_deep', $_REQUEST);
}
```

```
if (isset($_GET['addjoke']))
{
  include 'form.html.php';
  exit();
}

$link = mysqli_connect('localhost', 'root', 'password');
if (!$link)
{
  $error = 'Unable to connect to the database server.';
  include 'error.html.php';
  exit();
}

if (!mysqli_set_charset($link, 'utf8'))
{
  $output = 'Unable to set database connection encoding.';
  include 'output.html.php';
  exit();
}

if (!mysqli_select_db($link, 'ijdb'))
{
  $error = 'Unable to locate the joke database.';
  include 'error.html.php';
  exit();
}

if (isset($_POST['joketext']))
{
  $joketext = mysqli_real_escape_string($link, $_POST['joketext']);
  $sql = 'INSERT INTO joke SET
      joketext="' . $joketext . '",
      jokedate=CURDATE()';
  if (!mysqli_query($link, $sql))
  {
    $error = 'Error adding submitted joke: ' . mysqli_error($link);
    include 'error.html.php';
    exit();
  }

  header('Location: .');
  exit();
}
```

```
$result = mysqli_query($link, 'SELECT joketext FROM joke');
if (!$result)
{
  $error = 'Error fetching jokes: ' . mysqli_error($link);
  include 'error.html.php';
  exit();
}

while ($row = mysqli_fetch_array($result))
{
  $jokes[] = $row['joketext'];
}

include 'jokes.html.php';
?>
```

As you review this code to make sure it all makes sense to you, note that the calls to `mysqli_connect` and `mysqli_select_db` must come before any of the code that runs database queries. A database connection is unnecessary to display the "Add Joke" form, however, so that code can come at the very top of the controller script.

Load this up and add a new joke or two to the database via your browser. The resulting page should look like Figure 4.10.

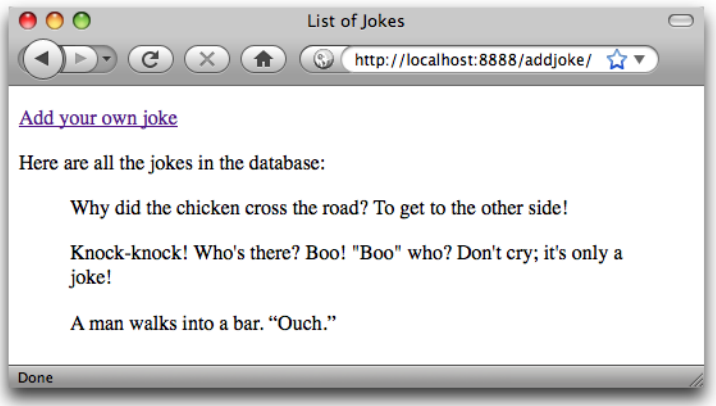


Figure 4.10. Look, Ma! No SQL!

There you have it! With a single controller (**index.php**) pulling the strings, you're able to view existing jokes in, and add new jokes to, your MySQL database.

Deleting Data from the Database

In this section, we'll make one final enhancement to our joke database site. We'll place next to each joke on the page a button labeled **Delete** that, when clicked, will remove that joke from the database and display the updated joke list.

If you like a challenge, you might want to take a stab at writing this feature yourself before you read on to see my solution. Although we're implementing a brand new feature, we'll mainly be using the same tools that we have for the previous examples in this chapter. Here are a few hints to start you off:

- You'll still be able to do it all with a single controller script (**index.php**).
- You'll need to use the SQL `DELETE` command, which I introduced in Chapter 2.
- To delete a particular joke in your controller, you'll need to identify it uniquely. The `id` column in the `joke` table was created to serve this purpose. You're going to have to pass the ID of the joke to be deleted with the request to delete a joke. The easiest way to do this is to use a hidden form field.

At the very least, take a few moments to think about how you would approach this. When you're ready to see the solution, read on!

To begin with, we need to modify the `SELECT` query that fetches the list of jokes from the database. In addition to the `joketext` column, we must also fetch the `id` column, so we can identify each joke uniquely:

chapter4/deletejoke/index.php *(excerpt)*

```
$result = mysqli_query($link, 'SELECT id, joketext FROM joke');
if (!$result)
{
  $error = 'Error fetching jokes: ' . mysqli_error($link);
  include 'error.html.php';
  exit();
}
```

We must also modify the `while` loop that stores the database results in the `$jokes` array. Instead of simply storing the text of each joke as an item in the array, we must store both the ID and text of each joke. One way to do this is to make each item in the `$jokes` array an array in its own right:

chapter4/deletejoke/index.php *(excerpt)*

```
while ($row = mysqli_fetch_array($result))
{
  $jokes[] = array('id' => $row['id'], 'text' => $row['joketext']);
}
```

Once this `while` loop runs its course, we'll have the `$jokes` array, each item of which is an associative array with two items: the ID of the joke and its text. For each joke (`$jokes[`*n*`]`), we can therefore retrieve its ID (`$jokes[`*n*`]['id']`) and its text (`$jokes[`*n*`]['text']`).

Our next step, then, should be to update the **jokes.html.php** template to retrieve each joke's text from this new array structure, and also to provide a **Delete** button for each joke:

chapter4/deletejoke/jokes.html.php *(excerpt)*

```
<?php foreach ($jokes as $joke): ?>
  <form action="?deletejoke" method="post"> (1)
    <blockquote>
      <p>
        <?php echo htmlspecialchars($joke['text'], ENT_QUOTES, (2)
            'UTF-8'); ?>
        <input type="hidden" name="id" value="<?php
            echo $joke['id']; ?>"/> (3)
        <input type="submit" value="Delete"/> (4)
      </p>
    </blockquote>
  </form> (5)
<?php endforeach; ?>
```

Here are the highlights of this updated code:

(1) Each joke will be displayed in a form, which, if submitted, will delete that joke. We signal this to our controller using the `?deletejoke` query string in the `action` attribute.

(2) Since each joke in the `$jokes` array is now represented by a two-item array instead of a simple string, we must update this line to retrieve the text of the joke. We do this using `$joke['text']` instead of just `$joke`.

3. When we submit the form to delete this joke, we wish to send along the ID of the joke to be deleted. To do this, we need a form field containing the joke's ID, but this is a field we'd prefer to keep hidden from the user. We therefore using a hidden form field (`<input type="hidden"/>`). The `name` of this field is `id`, and its `value` is the ID of the joke to be deleted (`$joke['id']`).

   Unlike the text of the joke, the ID is not a user-submitted value, so there's no need to worry about making it HTML-safe with `htmlspecialchars`. We can rest assured it will be a number, since it's automatically generated by MySQL for the `id` column when the joke is added to the database.

4. This submit button (`<input type="submit"/>`) submits the form when clicked. Its `value` attribute gives it a label of `Delete`.

5. Finally, we close the form for this joke.

This Markup Could Be Better

If you know your HTML, you're probably thinking those `<input/>` tags belong outside of the `blockquote` element, since they aren't a part of the quoted text (the joke).

Strictly speaking, that's true: the form and its `input`s should really be either before or after the `blockquote`. Unfortunately, to make that tag structure display clearly requires a little Cascading Style Sheets (CSS) code that's really beyond the scope of this book.

Rather than attempt to teach you CSS layout techniques in a book about PHP and MySQL, I've decided to go with this imperfect markup. If you plan to use this code in the real world, you should invest some time into learning CSS (or securing the services of a person who does) so that you can take complete control of your HTML markup without worrying about the CSS code required to make it look nice.

Figure 4.11 shows what the joke list looks like with the **Delete** buttons added.

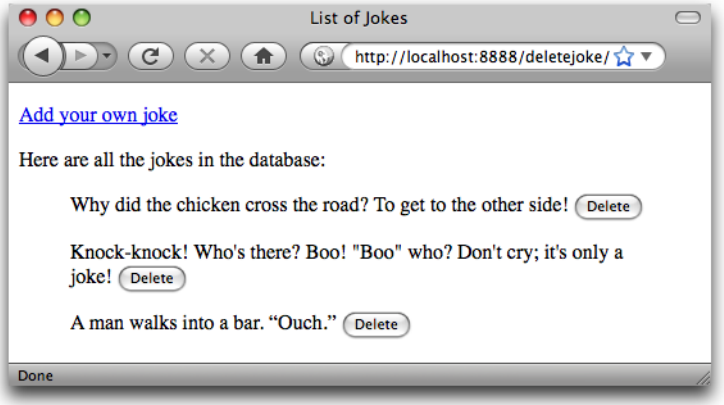


Figure 4.11. Each button will delete its respective joke

All that remains to make this new feature work is to update the controller so that it can process the form submission that results from clicking one of our new **Delete** buttons:

chapter4/deletejoke/index.php *(excerpt)*

```
if (isset($_GET['deletejoke']))
{
  $id = mysqli_real_escape_string($link, $_POST['id']);
  $sql = "DELETE FROM joke WHERE id='$id'";
  if (!mysqli_query($link, $sql))
  {
    $error = 'Error deleting joke: ' . mysqli_error($link);
    include 'error.html.php';
    exit();
  }

  header('Location: .');
  exit();
}
```

This chunk of code works exactly like the one we added to process the "Add Joke" code earlier in this chapter. We start by using `mysqli_real_escape_string` to sanitize the submitted value of `$_POST['id']` before using it in a database query[5]—this time, a `DELETE` query. Once that query is executed, we use the PHP

[5] You might think it's unnecessary to sanitize this value, since it's produced by a hidden form field that the user is unable to see. In fact, however, *all* form fields—even hidden ones—are ultimately under the

`header` function to ask the browser to send a new request to view the updated list of jokes.

Why Not a Link?

If you tackled this example yourself, your first instinct might have been to provide a **Delete** hyperlink for each joke, instead of going to the trouble of writing an entire HTML form containing a **Delete** button for each joke on the page. Indeed, the code for such a link would be much simpler:

```
<?php foreach ($jokes as $joke): ?>
  <blockquote>
    <p>
      <?php echo htmlspecialchars($joke['text'], ENT_QUOTES,
          'UTF-8'); ?>
      <a href="?deletejoke&amp;id=<?php echo $joke['id'];
          ?>">Delete</a>
    </p>
  </blockquote>
<?php endforeach; ?>
```

In short, hyperlinks should never be used to perform *actions* (like deleting a joke); hyperlinks should only be used to provide a link to some related content. The same goes for forms with `method="get"`, which should only be used to perform queries of existing data. Actions should only ever be performed as a result of a form with `method="post"` being submitted.

The reason is that forms with `method="post"` are treated differently by browsers and related software. If you submit a form with `method="post"` and then click the **Refresh** button in your browser, for example, the browser will ask if you're certain you wish to resubmit the form. Browsers have no similar protection against resubmission when it comes to links and forms with `method="get"`.

Similarly, web accelerator software (and some modern browsers) will automatically follow hyperlinks present on a page in the background, so that the target pages will be available for immediate display if the user clicks one of those links. If your site deleted a joke as a result of a hyperlink being followed, you could find your jokes getting deleted automatically by your users' browsers!

---

user's control. There are widely distributed browser add-ons, for example, that will make hidden form fields visible and available for editing by the user. Remember: any value submitted by the browser is ultimately suspect when it comes to protecting your site's security.

Here's the complete code of the finished controller. If you have any questions, make sure to post them in the SitePoint Forums![6]

chapter4/deletejoke/index.php

```
<?php
if (get_magic_quotes_gpc())
{
  function stripslashes_deep($value)
  {
    $value = is_array($value) ?
        array_map('stripslashes_deep', $value) :
        stripslashes($value);

    return $value;
  }

  $_POST = array_map('stripslashes_deep', $_POST);
  $_GET = array_map('stripslashes_deep', $_GET);
  $_COOKIE = array_map('stripslashes_deep', $_COOKIE);
  $_REQUEST = array_map('stripslashes_deep', $_REQUEST);
}

if (isset($_GET['addjoke']))
{
  include 'form.html.php';
  exit();
}

$link = mysqli_connect('localhost', 'root', 'password');
if (!$link)
{
  $error = 'Unable to connect to the database server.';
  include 'error.html.php';
  exit();
}

if (!mysqli_set_charset($link, 'utf8'))
{
  $output = 'Unable to set database connection encoding.';
  include 'output.html.php';
  exit();
}
```

[6] http://www.sitepoint.com/forums/

```
if (!mysqli_select_db($link, 'ijdb'))
{
  $error = 'Unable to locate the joke database.';
  include 'error.html.php';
  exit();
}

if (isset($_POST['joketext']))
{
  $joketext = mysqli_real_escape_string($link, $_POST['joketext']);
  $sql = 'INSERT INTO joke SET
      joketext="' . $joketext . '",
      jokedate=CURDATE()';
  if (!mysqli_query($link, $sql))
  {
    $error = 'Error adding submitted joke: ' . mysqli_error($link);
    include 'error.html.php';
    exit();
  }

  header('Location: .');
  exit();
}

if (isset($_GET['deletejoke']))
{
  $id = mysqli_real_escape_string($link, $_POST['id']);
  $sql = "DELETE FROM joke WHERE id='$id'";
  if (!mysqli_query($link, $sql))
  {
    $error = 'Error deleting joke: ' . mysqli_error($link);
    include 'error.html.php';
    exit();
  }

  header('Location: .');
  exit();
}

$result = mysqli_query($link, 'SELECT id, joketext FROM joke');
if (!$result)
{
  $error = 'Error fetching jokes: ' . mysqli_error($link);
  include 'error.html.php';
```

```
  exit();
}

while ($row = mysqli_fetch_array($result))
{
  $jokes[] = array('id' => $row['id'], 'text' => $row['joketext']);
}

include 'jokes.html.php';
?>
```

Mission Accomplished

In this chapter, you learned some new PHP functions that allow you to interface with a MySQL database server. Using these functions, you built your first database driven web site, which published the `ijdb` database online, and allowed visitors to add jokes to it and delete jokes from it.

In a way, you could say this chapter achieved the stated mission of this book, to teach you how to build a database driven web site. Of course, the example in this chapter contains only the bare essentials. In the rest of this book, I'll show you how to flesh out the skeleton you learned to build in this chapter.

In Chapter 5, we go back to the MySQL command line. We'll learn how to use relational database principles and advanced SQL queries to represent more complex types of information, and give our visitors credit for the jokes they add!

What's Next?

You've only seen a small part of what this book has to offer. As you move through the book you'll quickly notice that it's written in a clear tutorial format that's easy to understand, and illustrated with plenty of screenshots and diagrams, providing quick visual cues. If you hate wading through dry academic-style "how to" texts, this book will be a breath of fresh air to you.

If you've never built a database driven web site and you're looking to go beyond the limitations of a static site, this book will start you off in no time. If you've created database-driven web sites before, the extensive PHP and MySQL reference guides included will ensure this book remains an extremely handy desk reference.

To find out more and to order your copy, visit http://www.sitepoint.com/launch/3eb28e.

100% Satisfaction Guarantee

Oh—by the way, since we want you to feel as confident as we do that this book is an essential PHP and MySQL learning text, you have a full 30 days to read and use it. If, in that time, you feel ill-equipped in the knowledge to start your own PHP and MySQL project, then simply send the book back and we'll give you a prompt refund of the full purchase price, minus shipping and handling.

So, for the cost of a new T-shirt, learn how to make your own database driven web site today!

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z