SitePoint

Build Your Own

Database Driven Website

Using PHP & MySQL

by Kevin Yank

www.sitepoint.com

A Practical Step-by-Step Guide

Build Your Own Database Driven Website Using PHP & MySQL (First 4 Chapters)

Thank you for downloading the first four chapters of Kevin Yank's *"Build Your Own Database Driven Website Using PHP & MySQL"*.

This excerpt encapsulates the Summary of Contents, Information About the Author and SitePoint.com, Table of Contents, Introduction, and the first four chapters of the book.

We hope you find this information useful in evaluating the book.

For more information visit SitePoint.com

Summary of Contents of this Excerpt

Summary of Additional Contents of the Book

Build Your Own Database Driven Website Using PHP & MySQL
by Kevin Yank

Editor: Georgina Laidlaw
Cover Design: Alex Walker
Printing History: August 2001 First Edition
October 2001 Minor fixes

Published by SitePoint Pty. Ltd.
Suite 6, 50 Regent Street Richmond
VIC Australia 3121.

Web: www.sitepoint.com
Email: editor@sitepoint.com

ISBN 0-9579218-0-2
Printed and bound in the United States of America

About the Author

Kevin Yank is the Technical Content Director for SitePoint.com, author of many well received tutorials and articles, and editor of the SitePoint Tech Times, an extremely popular technically-oriented email newsletter for Web Developers.

Before graduating from McGill University in Montreal with a Bachelor of Computer Engineering, Kevin was not only a budding Web Developer himself, but also an active advisor for the Sausage Software Web Development Forums and writer of several practical guides on advanced HTML and JavaScript.

These days, when he's not discovering new technologies, writing books, or catching up on sleep, Kevin can be found helping other up-and-coming Web Developers in the SitePoint Forums (www.sitepointforums.com).

About SitePoint.com

SitePoint.com is an online community and content-based resource site for Web Developers. At the time of printing this book, SitePoint.com registered over 2.6 million page views per month, boasted more than 80,000 newsletter subscribers and over 11,000 active Community Forum members.

Providing regular cutting-edge articles, in-depth product reviews, step-by-step tutorials, practical advice and a vibrant community, SitePoint.com is one of the most comprehensive sites in the world for resources related to building and growing a successful Website.

Visit SitePoint at www.sitepoint.com.

Table of Contents

Introduction

On the Web today, content is king. Once you've mastered HTML and learned a few neat tricks in JavaScript and Dynamic HTML, you can probably build a pretty impressive-looking Web site design. But then comes the time to fill that fancy page layout with some real information. Any site that successfully attracts repeat visitors has to have fresh and constantly-updated content. In the world of traditional site building, that means HTML files—and lots of 'em.

The problem is that, more often than not, the people who provide the content for a site are not the same people who handle its design. Oftentimes, the content provider doesn't even know HTML. How, then, is the content to get from the provider onto the Web site? Not every company can afford to staff a full-time Webmaster, and most Webmasters have better things to do than copying Word files into HTML templates anyway.

Maintenance of a content-driven site can be a real pain, too. Many sites (perhaps yours?) feel locked into a dry, outdated design because rewriting those hundreds of HTML files to reflect a new design would take forever. Server-side includes (SSI's) can help alleviate the burden a little, but you still end up with hundreds of files that need to be maintained should you wish to make a fundamental change to your site.

The solution to these headaches is database-driven site design. By achieving complete separation between your site's design and the content you want to present, you can work with each without disturbing the other. Instead of writing an HTML file for every page of your site, you only need to write a page for each *kind* of information you want to be able to present. Instead of endlessly pasting new content into your tired page layouts, create a simple content management system that allows the writers to post new content themselves without a lick of HTML!

In this book, I'll provide you with a hands-on look at what's involved in building a database-driven Web site. We'll use two tools for this, both of which may be new to you: the PHP scripting language and the MySQL relational database management system. If your Web host provides PHP and MySQL support, you're in great shape. If not, we'll be looking at the set-up procedures under Linux and Windows, so don't sweat it.

Who Should Read this Book

This book is aimed at intermediate or 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 be making use of it without much in the way of explanation. A teensy bit of JavaScript may serve us well at some point as well, but I'll be sure to keep it simple for the uninitiated.

By the end of this book, you can expect to have a grasp of what's involved in setting up and 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 allows you to do a lot more than access a database easily) and Structured Query Language (SQL – the standard language for interacting with relational databases) as supported by MySQL, one of the most popular free database engines available today. Most importantly, you'll come away with everything you need to get started on your very own database-driven site in no time!

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 need a refresher on a particular topic.

The chapters contained in this document are:

Chapter 1: Installation

Before you can start building your database-driven Web presence, 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

set-up procedures on both Windows and Linux, and show you how to test that PHP is operational on your Web server.

Chapter 2: Getting Started with MySQL

Although I'm sure you'll be anxious to get started 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've never worked with a relational database before, this should definitely be an enlightening chapter that will whet your appetite for things to come! In the process, we'll build up a simple database to be used in later chapters.

Chapter 3: Getting Started with PHP

Here's where the fun really starts. In this chapter, I'll introduce you to the PHP scripting language, which can be used to easily build dynamic Web pages that present up-to-the-moment information to your visitors. Readers with previous programming experience will probably be able to get away with 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 we bring together PHP and MySQL, which you'll have seen separately in the previous two chapters, to create some of your first database-driven Web pages. We'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.

The other chapters included in the full print version of the book include:

Chapter 5: Relational Database Design

Although we'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 we'll have dealt with so far. 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, I'll teach the essential principles of good database design, emphasizing the importance of data normalization. If you don't know what that means, then this is definitely an important chapter for you to read!

Chapter 6: 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 Website, and with a few modifications you should be able to build a Web 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 7: Content Formatting and Submission

Just because you're implementing a nice, easy tool to allow site administrators to add content to your site without their knowing HTML, doesn't mean you have to restrict that content to 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 things. I'll also show you a simple way to safely make a content submission form directly available to your content providers, so that they can submit new content directly into your system for publication, pending an administrator's approval.

Chapter 8: MySQL Administration

While MySQL is a good, simple database solution for those who don't need 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 9: Advanced SQL

In chapter 8 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 get you efficiently juggling complex data like a pro.

Chapter 10: Advanced PHP

PHP lets you do a lot more than just retrieve, display, insert, and update information stored in a MySQL database. In this chapter, I'll give you a peek at some other interesting things you can do with PHP, such as server-side includes, handling file uploads, and sending email. Of course, as we'll see, these features are really useful for improving the performance and security of your database-driven site, as well as sending feedback to your visitors.

Chapter 11: Storing Binary Data in MySQL

Some of the most interesting applications of database-driven Web design include some juggling of binary files. Online file storage services like the now-defunct *iDrive*, are prime examples, but a system as simple as a personal photo gallery can benefit from storing binary files (e.g. pictures) in a database for retrieval and management on the fly. In this chapter, we develop a very simple online file storage and viewing system and learn the ins and outs of working with binary data in MySQL.

Chapter 12: Cookies and Sessions in PHP

One of the most hyped new features in PHP4 is built-in support for sessions. But what are sessions? 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 exploring the link between the two. At the end of this chapter, we'll develop a simple shopping cart system to demonstrate their use.

Code Archive

As you progress through the text, you'll note a number of references to the code archive. Located at `http://sitepoint.com/books/?bookid=More`, the Web site for this book contains not only an archive of all the code presented within this text, but also errata, updates, and information about other SitePoint publications.

The Book

The four chapters contained in this document are only the first part of my book "Build Your Own Database Driven Website Using PHP & MySQL".

The book contains eight more chapters that cover advanced database concepts, the design of a complete content management system, MySQL server administration, and much, much more!

It also includes a complete set of appendices, plus a FREE download of the code archive for all the examples demonstrated.

Put simply, this is the best desk reference for PHP and MySQL Web development currently available…

For more information visit SitePoint.com

Your Feedback

If you have a question about any of the information covered in this book, your best chance of a quick response is to post your query in the SitePoint.com Forums (www.sitepointforums.com).

And so, without further ado, let's get started!

1 Installation

Welcome to the Show

Hi there and welcome to the first book in SitePoint.com's *Practical Guide* series! Over the course of this book, it will be my job to guide you as you take your first steps beyond the HTML-and-JavaScript world of client-side site design. Together we'll explore what it takes to build the kind of large, content-driven sites that are so successful today, but which can be a real headache to maintain if they aren't done right.

Before we get started, you need to gather together the tools you'll need for the job. In this first chapter, I'll guide you as you download and set up the two software packages you'll need: PHP and MySQL.

PHP is a server-side scripting language. You can think of it as a "plug-in" for your Web server that will allow it to do more than just send plain Web pages when browsers request them. With PHP installed, your Web server will be able to read a new kind of file (called a **PHP script**) that can do things like retrieve up-to-the-minute information from a database and insert it into a Web page before sending it to the browser that requested it. PHP is completely free to download and use.

To retrieve information from a database, you first need to *have* a database. That's where MySQL comes in. MySQL is a relational database management system, or RDBMS. Exactly what role it plays and how it works we'll get into later, but basically it's a software package that is very good at the organization and management of large amounts of information. MySQL also makes that information really easy to access with server-side scripting languages like PHP. MySQL is released under the GNU General Public License (GPL), and is thus free for most uses on all of the platforms it supports. This includes most Unix-based platforms, like Linux and even Mac OS X, as well as Windows 9x / ME / NT / 2000.

If you're lucky, your current Web host may already have installed MySQL and PHP on your Web server for you. If that's the case, much of this chapter will not apply to you, and you can skip straight to the section entitled "If Your Web Host Provides PHP and MySQL" to make sure your setup is ship shape.

Everything we'll discuss in this book may be done on a Windows- or Unix-based server. The installation procedure will differ in accordance with the type of server you have at your disposal. The following two sections deal with installation on a Windows-based Web server, and installation under Linux (and other Unix-based platforms), respectively. Unless you're especially curious, you should only need to read the section that applies to you.

Windows Installation

Installing MySQL

As I mentioned above, MySQL may be downloaded free of charge. Simply proceed to `http://www.mysql.com/downloads/` and choose the recommended stable release (as of this writing, it is MySQL 3.23). Under the heading of *Standard binary (tarball) distributions* (which basically means the program doesn't need to be compiled, and is ready to run once you download it), find and click *Windows 95/98/NT/2000 (Intel)*. If you're on a high-speed connection, you'll probably want to check out one of the download mirrors listed at `http://www.mysql.com/downloads/mirrors.html` to get a reasonable download speed. After downloading the file (it's about 12MB as of this writing), unzip it and run the setup.exe program contained therein.

Once installed, MySQL is ready to roll (barring a couple of configuration tasks that we'll look at shortly), except for one minor issue that only affects you if you're running Windows NT / 2000 / XP. If you use any of those operating systems, find a file called `my-example.cnf` in the directory to which you just installed MySQL. Copy it to the root of your C: drive and rename it to `my.cnf`. If you don't like the idea of a MySQL configuration file sitting in the root of your C: drive, you can instead name it `my.ini` and put it in your Windows directory (e.g. `D:\WINNT` if Windows 2000 is installed on drive D:). Whichever you choose, open the file in WordPad (Notepad is likely to display it incorrectly) and look for the following line:

```
# basedir = d:/mysql/
```

Uncomment this line by removing the '#' symbol at the start, and change the path to point to your MySQL installation directory, using slashes (/) instead of backslashes (\). For instance, I changed the line on my system to read as follows:

```
basedir = d:/Program Files/mysql/
```

With that change made, save the file and close WordPad. MySQL will now run on your Windows NT/2000/XP system! If you're using Windows 95/98/ME, this step is not necessary – MySQL will run just fine as-installed.

Just like your Web server, MySQL is a program that should be run in the background so that it may respond to requests for information at any time. The server program may be found in the "bin" subfolder of the folder into which you installed MySQL. To make things complicated, however, there are actually several versions of the MySQL server to choose from:

- `mysqld.exe` This is the basic version of MySQL if you run Windows 95, 98, or ME. It includes support for all advanced features, and includes debug code to provide additional information in the case of a crash (if your system is set up to debug programs). As a result of this code, however, the server might run a little slow, and I've generally found that MySQL is so stable that crashes aren't really a concern.
- `mysqld-opt.exe` This version of the server lacks a few of the advanced features of the basic server, and does not include the debug code. It's optimized to run quickly on today's processors. For beginners, the advanced features are not a big concern. You certainly won't be using them while you complete the tasks in this book. This is the version of choice for beginners running Windows 95, 98, or ME.
- `mysqld-nt.exe` This version of the server is compiled and optimized like `mysqld-opt`, but is designed to run under Windows NT/2000/XP as a service. If you're using any of those operating systems, this is probably the server for you.
- `mysqld-max.exe` This version is like `mysqld-opt`, but contains advanced features that support transactions.
- `mysqld-max-nt.exe` This version's similar to `mysqld-nt`, but has advanced features that support transactions.

All these versions were installed for you in the `bin` directory. If you're running on Win98x/ME I recommend sticking with `mysql-opt` for now - move to `mysqld-max` if you ever need the advanced features. On Windows NT/2000/XP, `mysqld-nt` is my recommendation. Upgrade to `mysqld-max-nt` when you need more advanced features.

Starting MySQL is also a little different under WinNT/2000/XP, but this time let's start with the procedure for Win95/98/ME. Open an MS-DOS Command Prompt and proceed to the MySQL `bin` directory, and run your chosen server program:

```
C:\mysql\bin> mysqld-opt
```

Don't be surprised when you receive another command prompt. This command launches the server program so that it runs in the background, even after you close the command prompt. If you press Ctrl-Alt-Del to pull up the task list, you should see the MySQL server listed as one of the tasks that's active on your system.

To ensure that the server is started whenever Windows starts, you might want to create a shortcut to the program and put it in your Startup folder. This is just like creating a shortcut to any other program on your system.

On WinNT/2000/XP, you must install MySQL as a system service. Fortunately, this is very easy to do. Simply open a *Command Prompt* and run your chosen server program with the `-install` option:

```
C:\mysql\bin> mysqld-nt –install
```

This will install MySQL as a service that will be started the next time you reboot Windows. To manually start MySQL without having to reboot, just type this command (which can be run from any directory):

```
C:\> net start mysql
```

To verify that the MySQL server is running properly, press Ctrl-Alt-Del and open the *Task List*. If all is well, the server program should be listed on the *Processes* tab.

Installing PHP

The next step is to install PHP. At the time of this writing, PHP 4.x has become well-established as the version of choice; however, some old servers still use PHP 3.x (usually because nobody has bothered to update it). I'll cover the installation of PHP4 here, so be aware that if you're still working with PHP3 there may be some minor differences.

Download PHP for free from `http://www.php.net/` (or one of its mirrors listed at `http://www.php.net/mirrors.php`). You'll want the *Win32 Binaries* package, and be sure to grab the version that includes both the *CGI binary* and the *server API versions* if you have a choice.

In addition to PHP itself, you will need a **Web server** such as Internet Information Services (IIS), Apache, Sambar or OmniHTTPD. PHP was designed to run as a plug-in for existing Web server software. To test dynamic Web pages with PHP, you'll need to equip your own computer with Web server software, so that PHP has something to plug into. If you have Windows 2000/XP, then install IIS (if it's not already on your system): open *Control Panel*, *Add/Remove Programs*, *Add/Remove Windows Components*, and select IIS from the list of components. If you're not lucky enough to have IIS at your disposal, you can instead use a free Web server like Apache. I'll give instructions for both options in detail.

First, **whether you have IIS or not**, complete these steps:

- Unzip the file you downloaded into a directory of your choice. I recommend `C:\PHP` and will refer to this directory from here onward, but feel free to choose another directory if you like.
- Find the file called `php4ts.dll` in the PHP folder and copy it to the `System32` subfolder of your Windows folder (e.g. `C:\Windows\System32`).
- Find the file called `php.ini-dist` in the PHP folder and copy it to your Windows folder. Once there, rename it to `php.ini`.
- Open the `php.ini` file in your favorite text editor (use WordPad if Notepad doesn't display the file properly). It's a large file with a lot of confusing options, but look for a line that begins with `extension_dir` and set it so that it points to the `extensions` subfolder of your PHP folder:

  ```
  extension_dir = C:\PHP\extensions
  ```

  A little further down, look for a line that starts with `session.save_path` and set it to your Windows `TEMP` folder:

  ```
  session.save_path = C:\WINDOWS\TEMP
  ```

  Save the changes you made and close your text editor.

Now, **if you have IIS**, follow these instructions:

- In the Windows *Control Panel*, open *Administrative Tools* | *Internet Information Services*.
- In the tree view, expand the entry labeled *local computer*, then under *Web Sites* look for *Default Web Site* (unless you have virtual hosts set up, in which case, choose the site you want to add PHP support to). Right-click on the *Web Site* and choose *Properties*.
- Click the *ISAPI Filters* tab, and click *Add...*. In the *Filter Name* field, type `PHP,` and in the *Executable* field, browse for the file called `php4isapi.dll` in the `sapi` subfolder of your PHP folder (e.g. `C:\PHP\sapi\php4isapi.dll`). Click OK.
- Click the *Home Directory* tab, and click the *Configuration…* button. On the *Mappings* tab click *Add*. Again choose your `php4isapi.dll` file as the executable and type `.php` in the extension box (including the '`.`'). Leave everything else unchanged and click *OK*. If you want your Web server to treat other file extensions as PHP files (`.php3`, `.php4`, and `.phtml` are common choices), repeat this step for each extension. Click *OK* to close the *Application Configuration* window.
- Click *OK* to close the *Web Site Properties* window. Close the *Internet Information Services* window.

- Again, in the *Control Panel* under *Administrative Tools*, open *Services*. Look for the *World Wide Web Publishing* service near the bottom of the list. Right-click on it and choose *Restart* to restart IIS with the new configuration options. Close the *Services* window.
- You're done! PHP is installed!

If you don't have IIS, you'll first need to install some other Web server. For our purposes I'll assume you have downloaded and installed Apache server from http://httpd.apache.org/; however, PHP can also be installed on Sambar Server (http://www.sambar.com/), OmniHTTPD (http://www.omnicron.ab.ca/httpd/), and others.

Once you've downloaded and installed Apache according to the instructions included with it, open http://localhost/ in your Web browser, to make sure it works properly. If you don't see a Web page explaining that Apache was successfully installed, then either you haven't run Apache yet, or your installation is faulty. Check the documentation and make sure Apache is running properly before you install PHP.

If you've made sure Apache is up and running, you can add PHP support:

- On your *Start Menu*, choose *Programs*, *Apache httpd Server*, *Configure Apache Server*, *Edit Configuration*. This will open the `httpd.conf` file in *NotePad*.
- All of the 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 three lines to the very bottom of the file:

```
LoadModule php4_module c:/php/sapi/php4apache.dll
AddType application/x-httpd-php .php .php3 .phtml
AddType application/x-httpd-php-source .phps
```

  Be sure the `LoadModule` line points to the `php4apache.dll` file on your system, and note the use of slashes instead of backslashes.
- Save your changes and close Notepad.
- Restart Apache by choosing *Programs*, *Apache httpd Server*, *Control Apache Server*, *Restart* on the *Start menu*. If all is well, Apache will start up again without complaint.
- You're done! PHP is installed!

With MySQL and PHP installed, you're ready to proceed to the Post-Installation Setup Tasks section on page 12.

Installing under Linux

This section covers the procedure for installing PHP and MySQL under most current distributions of Linux. These instructions were tested under the latest versions of RedHat Linux and Mandrake Linux; however, they should work on other distributions such as Debian without much trouble. The steps involved will be very similar, if not identical.

As a user of one of the handful of Linux distributions available, you may be tempted to download and install the RPM distributions of PHP and MySQL. RPM's are nice, pre-packaged versions of software that are really easy to install. Unfortunately, they also limit the software configuration options available to you. If you already have MySQL and PHP installed in RPM form, then feel free to proceed with those versions, and skip forward to the "Post-Installation Setup Tasks" section. If you encounter any problems, you can always return here to uninstall the RPM versions and reinstall PHP and MySQL by hand.

Since many Linux distributions will automatically install PHP and MySQL for you, your first step should be to remove any old RPM versions of PHP and MySQL from your system. If one exists, use your distribution's graphical software manager to remove all packages with 'php' or 'mysql' in their names (`mod_php` is one that is often missed).

If your distribution doesn't have a graphical software manager, or if you didn't install a graphical user interface for your server, you can remove these from the command line. You'll need to be logged in as the root user to issue the commands to do this. Note that in the following commands, "%" represents the shell prompt, and doesn't to be typed in.

```
% rpm –e mysql
% rpm –e mod_php
% rpm –e php
```

If any of these commands tell you that the package in question is not installed, don't worry about it unless you know for a fact that it is. In such cases, it will be necessary for you to remove the offending item by hand. Seek help from an experienced user if you don't know how. If the second command runs successfully (i.e. no message is displayed), then you did indeed have an RPM version of PHP installed, and you'll need to do one more thing to get rid of it entirely. Open your Apache configuration file (usually `/etc/httpd/conf/httpd.conf`) in your favorite text editor and look for the two lines shown here. They usually appear in separate sections of the file, so don't worry if they're not together. The path of the libphp4.so file may also be slightly different (e.g. `extramodules` instead of just `modules`). If you can't find them, don't worry – it just means that the RPM uninstaller was smart enough to remove them for you.

```
LoadModule php4_module modules/libphp4.so
AddModule mod_php4.c
```

These lines are responsible for telling Apache to load PHP as a plug-in module. Since you just uninstalled that module, you'll need to get rid of these lines to make sure Apache keeps working properly. You can comment out these lines by adding a hash (#) at the beginning of both lines.

To make sure Apache is still in working order, you should now restart it without the PHP plug-in:

```
% /etc/rc.d/init.d/httpd restart
```

With everything neat and tidy, you're ready to download and install MySQL and PHP.

Installing MySQL

MySQL is freely available for Linux from `http://www.mysql.com` (or one of its mirrors listed at `http://www.mysql.com/downloads/mirrors.html`). Download the latest stable release (listed as *recommended* on the download page). You should grab the *Tarball* version under *Source downloads*, with filename `mysql-3.xx.xx.tar.gz`.

With the program downloaded (it was about 10.5MB as of this writing), you should make sure you're logged in as root before proceeding with the installation, unless you only want to install MySQL in your own home directory. To begin, unpack the downloaded file and move into the directory that is created:

```
% tar xfz mysql-3.xx.xx.tar.gz
% cd mysql-version
```

Next, you need to configure the MySQL install. Unless you really know what you're doing, all you should have to do is tell it where to install. I recommend `/usr/local/mysql`:

```
% ./configure --prefix=/usr/local/mysql
```

After sitting through the screens and screens of configuration tests, you'll eventually get back to a command prompt. Be sure the configuration completed successfully. If you see an error message just before the configuration quit, you'll need to address the problem it's complaining about. On Mandrake 8.0, for example, it complained of *"No curses/termcap library found"*. I simply installed the *libncurses5-devel* package in Mandrake's Software Manager, and the configuration worked fine on a second attempt. Once configuration is complete, you're ready to compile MySQL:

```
% make
```

After even more screens of compilation (this could take as long as a half an hour on some systems), you'll again be returned to the command prompt. You're now ready to install your newly compiled program:

```
% make install
```

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

```
% scripts/mysql_install_db
```

With that done, you can delete the directory you've been working in, which just contains all the source files and temporary installation files. If you ever need to reinstall, you can simply re-extract the `mysql-version.tar.gz` file.

With MySQL installed and ready to store information, all that's left is to get the server running on your computer. While you can run the server as the root user, or even as yourself (if, for example, you installed the server in your own home directory), the best idea is to set up on the system a special user whose sole purpose is to run the MySQL server. This will remove any possibility of someone using the MySQL server as a way to break into the rest of your system. To create a special MySQL user, you'll need to log in as root and type the following commands:

```
% /usr/sbin/groupadd mysqlgrp
% /usr/sbin/useradd –g mysqlgrp mysqlusr
```

By default, MySQL stores all database information in the `var` subdirectory of the directory to which it was installed. We want to make it so that nobody can access that directory except our new MySQL user. Assuming you installed MySQL to the `/usr/local/mysql` directory, you can use these commands:

```
% cd /usr/local/mysql
% chown –R mysqlusr.mysqlgrp var
% chmod –R go-rwx var
```

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

```
% bin/safe_mysqld --user=mysqlusr &
```

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 hostname) in MySQL's `var` 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:

```
% bin/mysqladmin -u root status
```

A little blurb with some statistics about the MySQL server should be displayed. If you receive an error message, something has gone wrong. Again, check the `hostname.err` file to see if the MySQL server output an error message while starting up. If you retrace your steps to make sure you followed the process described above, and this doesn't solve the problem, a post to the SitePoint.com Forums will help you pin it down in no time.

If you want your MySQL server to run automatically whenever the system is running (just like your Web server probably does), you'll have to set it up to do so. In the `share/mysql` 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.

First of all, assuming you've set up a special MySQL user to run the MySQL server, you'll need to tell the MySQL server to start as that user by default. To do this, create in your system's `/etc` directory a file called `my.cnf` that contains these two lines:

```
[mysqld]
user=mysqlusr
```

Now, when you run `safe_mysqld` or `mysql.server` to start the MySQL server, it will launch as `mysqlusr` without your having to tell it to. All that's left to do is to set up your system to run `mysql.server` automatically at startup.

Setting up your system to run the script at startup is a highly operating system-dependant task. If you're not sure of how to do this, you'd be best to ask someone who knows. However the following commands (starting in the MySQL directory) will do the trick for most versions of Linux[1]:

```
% cp share/mysql/mysql.server /etc/rc.d/init.d/
% cd /etc/rc.d/init.d
% chmod 500 mysql.server
% cd /etc/rc.d/rc3.d
% ln –s ../init.d/mysql.server S99mysql
```

[1] Under the current SUSE distribution of Linux, there is no `init.d` directory, just a symbolic link pointing back to `/etc/rc.d`; symbolic links for startup files should thus be directed to the `/etc/rc.d` directory (e.g. `ln –s ../mysql.server S99mysql`).

```
% cd /etc/rc.d/rc5.d
% ln –s ../init.d/mysql.server S99mysql
```

That's it! To test that this works, reboot your system and request the status of the server as before.

One final thing you might like to do for convenience's sake 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:

```
% ln –s /usr/local/mysql/bin/mysql /usr/local/bin/mysql
% ln –s /usr/local/mysql/bin/mysqladmin /usr/local/bin/mysqladmin
% ln –s /usr/local/mysql/bin/mysqldump /usr/local/bin/mysqldump
```

Installing PHP

As mentioned above, PHP is not really a program in and of itself. Instead, it's a plug-in module for your Web server (probably Apache). There are actually three ways to install the PHP plug-in 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 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 since you're likely to have Apache installed already, you'd probably prefer to avoid having to download, recompile, and reinstall it from scratch. For this reason, we'll use the third option.

To start, download the PHP Source Code package from `http://www.php.net` (or one of its mirrors listed at `http://www.php.net/mirrors.php`). At the time of this writing, PHP 4.x has become well-established as the version of choice; however, some old servers still use PHP 3.x (usually because nobody has bothered to update it). I'll be covering the installation of PHP4 here, so be aware that if you still work with PHP3 there may be some minor differences.

The file you downloaded should be called `php-version.tar.gz`. To begin, we'll extract the files it contains:

```
% tar xfz php-version.tar.gz
% cd php-version
```

To install PHP as a loadable Apache module, you'll need the Apache `apxs` program. This comes with most versions of Apache, but if you're using the copy that was installed with your distribution of Linux, you may need to install the Apache development RPM package to access Apache `apxs`. You should be able to install this package by whatever means your software distribution provides. By default, RedHat and Mandrake will install the program as `/usr/sbin/apxs`, so if you see this file, you know it's installed.

For the rest of the install procedure, you'll need to be logged in as the root user so you can make changes to the Apache configuration files.

The next 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 (like Apache and MySQL). Unless you know exactly what you're doing, simply type the command like this (all on one line):

```
% ./configure
        --prefix=/usr/local/php
        --with-config-file-path=/usr/local/php
        --with-apxs=/usr/sbin/apxs
        --enable-track-vars
        --enable-magic-quotes
        --enable-debugger
```

Again, check for any error messages and install any files it identifies as missing. On Mandrake 8.0, for example, it complained that the 'lex' command wasn't found. I searched for 'lex' in the Mandrake package list and it came up with 'flex', which it described as a program for matching patterns of text used in many programs' build processes. Once that was installed, the configuration process went without a hitch. After

you watch several screens of tests scroll by, you'll be returned to the command prompt. The following two commands will compile and then install PHP. Take a coffee break: this will take some time.

```
% make
% make install
```

PHP is now installed in `/usr/local/php` (unless you specified a different directory with the `--prefix` option of the `configure` script above) and it'll expect to find its configuration file, named `php.ini`, in the `lib` subdirectory (unless you specified a different directory with the `--with-config-file-path` option of the `configure` script above). PHP comes with two sample `php.ini` files called `php.ini-dist` and `php.ini-optimized`. Copy these files from your installation work directory to the directory in which PHP expects to find its `php.ini` file, then make a copy of the `php.ini-dist` file and call it `php.ini`:

```
% cp php.ini* /usr/local/php/lib/
% cd /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 make sure Apache knows where to find PHP, so that it can load the program when it starts up. Open your Apache `httpd.conf` configuration file (usually `/etc/httpd/conf/httpd.conf` if you're using your Linux distribution's copy of Apache) in your favorite text editor. Look for a line that looks like this:

```
LoadModule php4_module lib/apache/libphp4.so
```

Find the new, uncommented line (no # at the start of the line), not the old line that you may have commented out earlier. It may not appear along with the other `LoadModule` lines in the file. Once you find it, you might need to change the path to match all the other `LoadModule` lines in the file. Under RedHat Linux, this means you'll have to change the line to make it look like this:

```
LoadModule php4_module modules/libphp4.so
```

PHP will probably run correctly without this change, but on older versions of RedHat, Apache won't be able to find the libphp4.so file until you make this change. If you prefer, leave the line alone for now - you can come back and change it if you run into trouble.

Next, look for the line that begins with `DirectoryIndex`. In recent distributions, this may be in a separate file called `commonhttpd.conf`. This line tells Apache what filenames to use when it looks for the default page for a given directory. You'll see the usual index.html and so forth, but you need to add `index.php`, `index.php3`, and `index.phtml` to that list if they're not there already:

```
DirectoryIndex index.html ... index.php index.phtml index.php3
```

Finally, go right to the bottom of the file (again, this should go in `commonhttpd.conf` if you have such a file) and add these lines, to tell Apache which file extensions should be seen as PHP files:

```
AddType application/x-httpd-php .php .php3 .phtml
AddType application/x-httpd-php-source .phps
```

That should do it! Save your changes and restart your Apache server. If all things go to plan, Apache should start up without any error messages. If you run into any trouble, the helpful folks in the SitePoint.com Forums (myself included) will be happy to help.

Post-Installation Setup Tasks

No matter which operating system you're running, once PHP is installed and the MySQL server is in operation, the very first thing you need to do is assign a "root password" for MySQL. MySQL only lets authorized users view and manipulate the information stored in its databases, so you'll need to tell MySQL who is an authorized user, and who isn't. When MySQL is first installed, it's configured with a user named `root` that has access to do pretty much any task without even entering a password. Your first task should be to assign a password to the root user so that unauthorized users can't mess around in your databases.

It's important to realize that MySQL, just like a Web server or an FTP server, can be accessed from any computer on the same network. If you're working on a computer connected to the Internet that means anyone in the world could try to connect to your MySQL server! The need to pick a hard-to-guess password should be immediately obvious!

To set a root password for MySQL, type the following command in the bin directory of your MySQL installation (include the quotes):

```
mysqladmin –u root password "your new password"
```

To make sure MySQL has registered this change, you should tell it to reload its list of authorized users and passwords:

```
mysqladmin –u root reload
```

If this command returns an error message to tell you that access was denied, don't worry: this just means the password has already taken effect.

To try out your new password, request that the MySQL server tell you its current status:

```
mysqladmin –u root –p status
```

Enter your password when prompted. You should see a brief message that provides information about the server and its current status. The `-u root` argument tells the program that you want to be identified as the MySQL user called `root`. The `-p` argument tells the program to prompt you for your password before it tries to connect. The `status` argument just tells it that you're interested in viewing the system status.

If at any time you want to shut down the MySQL server, you can use the command below. Notice the usage of the same `-u root` and `-p` arguments as before:

```
mysqladmin –u root –p shutdown
```

With your MySQL database system safe from intrusion, all that's left is to configure PHP. To do this, we'll use a text file called `php.ini`. If you installed PHP under Windows, you should already have copied `php.ini` into your Windows directory. If you installed PHP under Linux using the instructions above, you should already have copied `php.ini` into the PHP `lib` folder (`/usr/local/php/lib`), or wherever you chose to put it.

Open `php.ini` in your favorite text editor and have a glance through it. Most of the settings are pretty well explained, and most of the default settings are just fine for our purposes. Just check to make sure that your settings match these:

```
register_globals = On
magic_quotes_gpc = On
doc_root = the document root folder of your Web server
extension_dir = the PHP install directory
```

Save the changes to `php.ini`, and then restart your Web server. To restart Apache under Linux, log in as root and type this command:

```
/etc/rc.d/init.d/httpd restart
```

You're done! Now you just need to test to make sure everything's working okay (see "Your First PHP Script" below).

If Your Web Host Provides PHP and MySQL

If the host that provides you with Web space has already installed and set up MySQL and PHP for you and you just want to learn how to use them, there really isn't a lot you need to do. Now would be a good time to get in touch with your host and request any information you may need to access these services.

Specifically, you'll need a username and password to access the MySQL server they've set up for you. They'll probably have provided an empty database for you to use as well (which prevents you from messing with the databases of other users who share the same MySQL server), and you'll want to know the name of your database.

There are two ways you can access the MySQL server directly. Firstly, you can use telnet or secure shell (SSH) to log in to the host. You can then use the MySQL client programs (`mysql`, `mysqladmin`, `mysqldump`) installed there to interact with the MySQL server directly. The second method is to install those client programs onto your own computer, and have them connect to the MySQL server. Your Web host may support one or both of these methods, so you'll need to ask which.

If your host allows you to log in by telnet or SSH to do your work, you'll need a username and password for the login, in addition to those you'll use to access the MySQL server (they can be different). Be sure to ask for both sets of information.

If they support remote access to the MySQL server, you'll want to download a program that lets you connect to, and interact with, the server. This book assumes you've downloaded from http://www.mysql.com/ a binary distribution of MySQL that includes the three client programs (`mysql`, `mysqladmin`, and `mysqldump`). Free packages are available for Windows, Linux and other operating systems. Installation basically consists of finding the three programs and putting them in a convenient place. The rest of the package, which includes the MySQL server, can be freely discarded. If you prefer a more graphical interface, download something like MySQL GUI (also available from http://www.mysql.com). I'd really recommend getting comfortable with the basic client programs first, though, as the commands you use with them will be similar to those you'll include in your PHP scripts to access MySQL databases.

Some less expensive Web hosts these days support neither telnet/SSH access, nor direct access to their MySQL servers. Instead, they provide a management console that allows you to browse and edit your database through your Web browser. Although this is a fairly convenient and not overly restrictive solution, it doesn't help you learn. Instead, I'd recommend the installation of a MySQL server on your own system to help, especially in the next chapter. Once you're comfortable working with your learning server, you can start using the server provided by your Web host with their management console. See the previous sections for instructions on installing MySQL under Windows and Linux.

Your First PHP Script

It would be unfair of me to help you get everything installed and not even give you a taste of what a PHP-driven Web page looks like until Chapter 3, so here's a little something to whet your appetite.

Open up your favorite text or HTML editor and create a new file called `today.php`. 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; otherwise, Notepad will helpfully save the file as `today.php.txt`. Type this into the file:

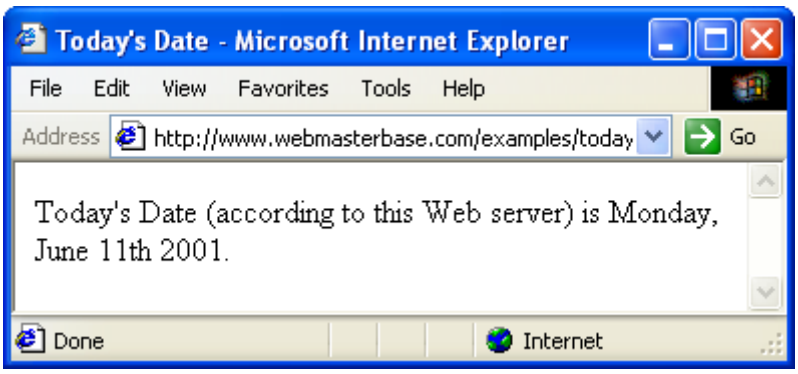
```
<html>
<head>
<title>Today's Date</title>
</head>
<body>
<p>Today's Date (according to this Web server) is
<?php

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

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

If you prefer, you can download this file along with the rest of the code in this book in the code archive at http://sitepoint.com/books/?bookid=More.

Save this and place it on your Web site as you would any regular HTML file, then view it in your browser. Note that if you view the file on your own machine, you *cannot* use the *File*, *Open* feature of your browser, because your Web server must intervene to interpret the PHP code in the file. Instead, you must move the file into the **document root** folder of your Web server software (e.g. `C:\inetpub\wwwroot\` in IIS, or `C:\Apache Group\Apache\htdocs\` in Apache for Windows), then load it into your browser by typing http://localhost/today.php. This allows the Web server to run the PHP code in the file and replace it with the date before it's sent to the Web browser. Here's what the output should look like:



Pretty 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) has been interpreted by the Web server and converted to normal text before it's sent to your browser. The beauty of PHP (and other server-side scripting languages) is that the Web browser doesn't have to know anything about it – the Web server does all the work!

And don't worry too much about the exact code I used in this example. Before too long you'll know it like the back of your hand.

If you *don't* see the date, then something is wrong with the PHP support in your Web server. Use View Source in your browser to look at the code of the page. You'll probably see the PHP code there in the page. Since the browser doesn't understand PHP, it just sees `<?php ... ?>` as one long, invalid HTML tag, which it ignores. Make sure that PHP support has been properly installed on your Web server, either in accordance with the instructions provided in previous sections of this chapter, or by your Web host.

Summary

You should now have everything you need to get MySQL and PHP installed on your Web Server. If the little example above didn't work right (for example, if the raw PHP code appeared instead of the date), something went wrong with your setup procedure. Drop by the SitePoint.com Forums (`http://www.sitepointforums.com/`) and we'll be glad to help you figure out the problem!

In Chapter 2, you'll learn the basics of relational databases and get started working with MySQL. If you've never even touched a database before, I promise you it'll be a real eye opener!

2 Getting Started with MySQL

In Chapter 1, we installed and set up two software programs: PHP and MySQL. In this chapter, we'll learn how to work with MySQL databases using Structured Query Language (SQL).

An Introduction to Databases

As I've already explained, PHP is a server-side scripting language that lets you insert into your Web pages instructions that your Web server software (be it Apache, IIS, or whatever) 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 get interesting when a database is added to the mix. 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 scripting 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 advantage of this would be twofold. First, instead of having to write an HTML file for each of your jokes, you could write a single PHP file that was designed to fetch any joke out of the database and display it. Second, to add 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 **things**. For our joke database, we'd probably start with a table called "jokes" which 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 "jokes" table might have columns for the text of the jokes, and the dates on which the jokes were added to the database. Each joke that we stored in this table would then be said to be a **row** in the table. These rows and columns form a table that looks like this:

	Column ↓	Column ↓	Column ↓
	ID	**JokeText**	**JokeDate**
Row →	1	Why did the chicken…?	2000-04-01
Row →	2	"Knock-knock!" "Who's there?"	2000-02-22

Notice that, in addition to columns for the joke text ("JokeText") and the date of the joke ("JokeDate"), I included a column named "ID". As a matter of good design, a database table should always provide a way to uniquely identify each of its rows. 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 we have an easy way to refer to them, and to keep track of which joke is which. Such database design issues will be covered with greater depth in Chapter **5**.

So, to review, the above is a three-column table with two rows (or entries). Each row in the table contains a joke's ID, its text, and the date of the joke. With this basic terminology under our belts, we're ready to get started with MySQL.

Logging On to MySQL

The standard interface for working with MySQL databases is to connect to the MySQL server software (which you set up in Chapter 1) and type commands one at a time. To make this connection to the server, you'll need the MySQL client program. If you installed the MySQL server software yourself either under Windows or under some brand of UNIX, you already have this program installed in the same location as the server program. Under Linux, for example, the program is called `mysql` and is located by default in the `/usr/local/mysql/bin` directory. Under Windows, the program is called `mysql.exe` and is located by default in the `C:\mysql\bin` directory.

If you didn't set up the MySQL server yourself (if, for example, you'll be working on your Web host's MySQL server), there are two ways to connect to the MySQL server. The first is to use Telnet or a Secure Shell (SSH) connection to log into your Web host's server, and then run `mysql` from there. The second is to download and install the MySQL software from http://www.mysql.com/ (available free for Windows and Linux) on your own computer, and use it to connect to the MySQL server over the Internet. Both ways work fine, and your Web host may support one, the other, or both (you'll need to ask).

Whichever method and operating system you use, you'll end up at a command line, ready to run the MySQL client program and connect to your MySQL server. Here's what you should type:

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

You need to replace `hostname` with the host name or IP address of the computer on which the MySQL server is running. If the client program is run on the same computer as the server, you can actually leave off the `-h hostname` part of the command instead of typing `-h localhost` or `–h 127.0.0.1`. `username` should be your MySQL user name. If you installed the MySQL server yourself, this will just be `root`. If you're using your Web host's MySQL server, this should be the MySQL user name they assigned you.

The `-p` argument tells the program to prompt you for your password, which it should do as soon as you enter the command above. If you set up the MySQL server yourself, this password is the root password you chose in Chapter 1. If you're using your Web host's MySQL server, this should be the MySQL password they gave you.

If you typed everything properly, the MySQL client program will introduce itself and then dump you on the MySQL command line:

```
mysql>
```

Now, 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 several of its subscribers to use, for example), so your next step should be to pick a database to work with. First, let's retrieve a list of databases on the current server. Type this command (don't forget the semicolon!), and hit ENTER.

```
mysql> SHOW DATABASES;
```

MySQL will show you a list of the databases on the server. If this is a brand new server (i.e. if you installed this server yourself in Chapter 1), the list should look like this:

```
+----------+
| Database |
+----------+
| mysql    |
| test     |
+----------+
2 rows in set (0.11 sec)
```

The MySQL server uses the first database, called `mysql`, to keep track of users, their passwords, and what they're allowed to do. We'll steer clear of this database for the time being, and come back to it in Chapter 8 when we discuss MySQL Administration. The second database, called `test`, is a sample database. You can actually get rid of this database. I won't be referring to it in this book (and we'll create our own example database momentarily). Deleting something in MySQL is called "dropping" it, and the command for doing so is appropriately named:

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

If you type this command and press Enter, MySQL will obediently delete the database, saying "Query OK" in confirmation. Notice that you're not prompted with any kind of "are you sure?" message. You have to be

very careful to type your commands correctly in MySQL because, as this example shows, you can obliterate your entire database—along with all the information it contains—with one single command!

Before we go any further, let's learn a couple of things about the MySQL command line. As you may have noticed, all commands in MySQL are terminated by a semicolon (`;`). If you forget the semicolon, MySQL will think you haven't finished typing your command, and will let you continue to type on another line:

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

MySQL shows you that it's waiting for you to type more of your command by changing the prompt from `mysql>` to `->`. For long commands, this can be handy, as it allows you to spread your commands out over several lines.

If you get halfway through a command and realize 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 completely ignore the command you had begun to type, and will go back to the prompt to wait for another command.

Finally, if at any time you want to exit the MySQL client program, just type "quit" or "exit" (either one will work). This is the only command that doesn't need a semicolon, but you can use one if you want to.

```
mysql> quit
Bye
```

So what's SQL?

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

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.

Creating a Database

Those of you working on your Web host's MySQL server have probably already been assigned a database to work with. Sit tight, we'll get back to you in a moment. Those of you running a MySQL server that you installed yourselves will need to create your own database. It’s just as easy to create a database as it is to delete one:

```
mysql> CREATE DATABASE jokes;
```

I chose to name the database "jokes", since that fits with the example we're working with. Feel free to give the database any name you like, though. Those of you working on your Web host's MySQL server will probably have no choice in what to name your database, since it will usually already have been created for you.

Now that we have a database, we need to tell MySQL that we want to use it. Again, the command isn't too hard to remember:

```
mysql> USE jokes;
```

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 our jokes.

Creating a Table

The SQL commands we've encountered so far have been pretty simple, but since 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 (
    ->  column_1_name column_1_type column_1_details,
```

```
    ->  column_2_name column_2_type column_2_details,
    ->  ...
    -> );
```

Let's return to our example "Jokes" table. Recall that it had three columns: ID (a number), JokeText (the text of the joke), and JokeDate (the date the joke was entered). The command to create this table looks like this:

```
mysql> CREATE TABLE Jokes (
    ->  ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->  JokeText TEXT,
    ->  JokeDate DATE NOT NULL
    -> );
```

It looks pretty scary, huh? Let's break it down:

- The first line is pretty simple: it says that we want to create a new table called `Jokes`.
- The 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 this column. First, this column is not allowed to be left blank (`NOT NULL`). Next, if we don't specify any value in particular when we add a new entry to the table, we want MySQL to pick a value that is one more than the highest value in the table so far (`AUTO_INCREMENT`). Finally, this column is to act as a unique identifier for the entries in this table, so all values in this column must be unique (`PRIMARY KEY`).
- The third line is super-simple; it says that we want a column called `JokeText,` which will contain text (`TEXT`).
- The fourth line defines our last column, called `JokeDate`, which will contain data of type `DATE`, and which cannot be left blank (`NOT NULL`).

Note that, while you're free to type your SQL commands in upper or lower case, a MySQL server running on a UNIX-based system will be case-sensitive when it comes to database and table names, as these correspond to directories and files in the MySQL data directory. Otherwise, MySQL is completely case–insensitive, but for one exception: table, column, and other names must be spelled exactly the same when they’re used more than once in the same command.

Note also that we assigned a specific type of data to each column we created. `ID` will contain integers, `JokeText` will contain text, and `JokeDate` will contain dates. MySQL requires you to specify a data type for each column in advance. Not only does this help keep your data organized, but it 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 give you some indication of where it had trouble understanding what you meant.

For such a complicated command, `Query OK` is pretty a pretty boring 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 jokes |
+-----------------+
| Jokes           |
+-----------------+
1 row in set
```

This is a list of all the tables in our database (which I named `jokes` above). The list contains only one table: the `Jokes` table we just created. So far everything looks good. Let's have a closer look at the `Jokes` table itself:

```
mysql> DESCRIBE Jokes;
+----------+---------+------+-----+------------+---------
| Field    | Type    | Null | Key | Default    | Extra
+----------+---------+------+-----+------------+---------
| ID       | int(11) |      | PRI | 0          | auto_inc
| JokeText | text    | YES  |     | NULL       |
```

```
| JokeDate | date    |      |     | 0000-00-00 |
+----------+---------+------+-----+------------+---------
3 rows in set
```

This provides a list of the columns (or fields) in the table. As we can see, there are three columns in this table, which appear as the 3 rows in this table of results. The details are somewhat cryptic, but if you look at them closely for a while you should be able to figure out what most of them mean. Don't worry about it too much, though. We've got better things to do, like adding some jokes to our table!

We need to look at just one more thing before we get to that, though: deleting a table. This is just as frighteningly easy to do as it is to delete a database. In fact, the command is almost identical:

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

Inserting Data into a Table

Our database is created and our table is built; all that's left is to put some actual jokes into our database. The command for inserting data into our database is called (appropriately enough) `INSERT`. There are two basic forms of this command:

```
mysql> INSERT INTO table_name SET
    ->  columnName1 = value1,
    ->  columnName2 = value2,
    ->  ...
    -> ;

mysql> INSERT INTO table_name
    -> (columnName1, columnName2, ...)
    -> VALUES (value1, value2, ...);
```

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

```
mysql> INSERT INTO Jokes SET
    -> JokeText = "Why did the chicken cross the road? To get to the other
side!",
    -> JokeDate = "2000-04-01";

mysql> INSERT INTO Jokes
    -> (JokeText, JokeDate) VALUES (
    -> "Why did the chicken cross the road? To get to the other side!",
    -> "2000-04-01"
    -> );
```

Note that in the second form 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 doesn't matter, as long as you give values for all required fields.

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 your 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 and presentation. As, at this point in our experience with databases, we only need fairly simple lists of results, we'll consider only the simpler forms of the `SELECT` command.

This command will list everything stored in the `Jokes` table:

```
mysql> SELECT * FROM Jokes;
```

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

```
+----+-------------------------------------------------------------+---------
---+
| ID | JokeText                                                    | JokeDate
|
+----+-------------------------------------------------------------+---------
---+
|  1 | Why did the chicken cross the road? To get to the other side! | 2000-04-
01 |
```

```
+----+-------------------------------------------------------------+---------
---+
1 row in set (0.05 sec)
```

It looks a little messed up, because the text in the `JokeText` column is too long for the table to fit properly on the screen. 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 Jokes;
```

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 | 2000-04-01 |
+----+------------+
1 row in set (0.00 sec)
```

Not bad, but we'd like to see at least some of the joke text, wouldn't we? In addition to listing the columns that we want the `SELECT` command to show us, we can modify those columns with **functions**. One function, called `LEFT`, lets us tell MySQL to display up to a specified maximum number of characters when it displays a column. For example, let's say we wanted to see only the first 20 characters of the `JokeText` column:

```
mysql> SELECT ID, LEFT(JokeText,20), JokeDate FROM Jokes;
+----+----------------------+------------+
| ID | LEFT(JokeText,20)    | JokeDate   |
+----+----------------------+------------+
|  1 | Why did the chicken  | 2000-04-01 |
+----+----------------------+------------+
1 row in set (0.05 sec)
```

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

```
mysql> SELECT COUNT(*) FROM Jokes;
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
1 row in set (0.06 sec)
```

As we can see, we only have one joke in our table.

So far, all our examples have fetched all the entries in the table. But if we add what's called a `WHERE` **clause** (for reasons that will become obvious in a moment) to a `SELECT` command, we can limit which entries are returned as results. Consider this example:

```
mysql> SELECT COUNT(*) FROM Jokes WHERE JokeDate >= "2000-01-01";
```

This query will count the number of jokes that have dates "greater than or equal to" January 1st, 2000. "Greater than or equal to", when dealing with dates, 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 Jokes WHERE JokeText LIKE "%chicken%";
```

This query displays the text of all jokes that contain the word "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 here indicate that the word "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 2000 only, we could use the following query:

```
mysql> SELECT JokeText FROM Jokes WHERE
    -> JokeText LIKE "%knock%" AND
```

```
    -> JokeDate >= "2000-04-01" AND
    -> JokeDate < "2000-05-01";
```

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

There's a lot more you can do with the SELECT statement, but we'll save looking at some of its more advanced features for later, when we need them.

Modifying Stored Data

Once you've entered your data into a database table, you might like to change it at some point. 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 INSERT command (that set column values) and of the SELECT command (that pick out entries to modify). The general form of the UPDATE command is as follows:

```
mysql> UPDATE table_name SET
    -> col_name = new_value, ...
    -> 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 Jokes SET JokeDate="1990-04-01" WHERE ID=1;
```

Here's where that ID column comes in handy. It allows us to easily single out a joke for changes. The WHERE clause here works just like it does in the SELECT command. This next command, for example, changes the date of all entries that contain the word "chicken":

```
mysql> UPDATE Jokes SET JokeDate="1990-04-01"
    -> WHERE JokeText LIKE "%chicken%";
```

Deleting Stored Data

The deletion of entries in SQL is dangerously easy (if you can't tell by now, this is a recurring theme). Here's the command syntax:

```
mysql> DELETE FROM table_name WHERE conditons;
```

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

```
mysql> DELETE FROM Jokes WHERE JokeText LIKE "%chicken%";
```

One thing to note is that the WHERE clause is actually optional. However, you should be very careful if you leave it off, as the DELETE command will then apply to all entries in the table. This command will empty the Jokes table in one fell swoop:

```
mysql> DELETE FROM Jokes;
```

Scary, huh?

Summary

There's a lot more to the MySQL database system and the SQL language than the few basic commands we've looked at here, but these commands are by far the most commonly used. So far we've only worked with a single table. To realize the true power of a relational database, we'll also need to learn how to use multiple tables together to represent potentially complex relationships between database entities.

All this and more will be covered in Chapter **5**, where we'll discuss
database design principles, and look at some more advanced examples. For now, though, we've hopefully accomplished our objective, and you can comfortably interact with MySQL using the command line interface.

In Chapter 3, the fun continues as we delve into the PHP server-side scripting language, and use it to create dynamic Web pages. If you like, you can practice with MySQL a little before you move on, by creating a decent-sized Jokes table - this will come in handy in Chapter 4!

3 Getting Started with PHP

In Chapter 2, we learned how to use the MySQL database engine to store a list of jokes in a simple database (composed of a single table named `Jokes`). To do so, we used the MySQL command-line client to enter SQL commands (queries). In this chapter, we'll introduce the PHP server-side scripting language. In addition to the basic features we'll explore here, this language has full support for communication with MySQL databases.

Introducing PHP

As we've discussed previously, PHP is a server-side scripting language. This concept is not obvious, especially if you're used to designing pages with just HTML and JavaScript. A server-side scripting language is similar to JavaScript in many ways, as they both allow you to embed little programs (scripts) into the HTML of a Web page. When executed, such scripts allow you to control what will actually appear in the browser window with more flexibility than is possible using straight HTML.

The key difference between JavaScript and PHP is simple. JavaScript is interpreted by the Web browser once the Web page that contains the script has been downloaded. Meanwhile, server-side scripting languages like PHP are interpreted by the Web server before the page is even sent to the browser. And, once it's interpreted, the results of the script replace the PHP code in the Web page itself, so all the browser sees is a standard HTML file. The script is processed entirely by the server, hence the designation: server-side scripting language.

Let's look back at the `today.php` example presented in Chapter 1:

```
<html>
<head>
<title>Today's Date</title>
</head>
<body>
<p>Today's Date (according to this Web server) is
<?php

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

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

Most of this is plain HTML. The line between `<?php` and `?>`, however, is written in PHP. `<?php` means "begin PHP code", and `?>` means "end PHP code". 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 something like this:

```
<html>
<head>
<title>Today's Date</title>
</head>
<body>
<p>Today's Date (according to this Web server) is
Wednesday, May 30th 2001.</p>
</body>
</html>
```

Notice that all signs of the PHP code have disappeared. In their place, the output of the script has appeared, and 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 and nothing else, so you don't have to worry about whether the language you're using will be supported by your visitors' browsers.

- **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 would only be able to display the date according to the computer on which the Web browser was running. Now, while this isn't an especially impressive example of the exploitation of server-side resources, we could just as easily have inserted some other information that would only be available to a script running on the Web server, for example, information stored in a MySQL database that runs on the Web server computer.
- **Reduced load on the client.** JavaScript can significantly slow down the display of a Web page on slower computers, as the browser must run the script before it can display the Web page. With server-side scripting, this becomes the burden of the Web server machine.

Basic Syntax and Commands

PHP syntax will be very familiar to anyone with an understanding of C, C++, Java, JavaScript, Perl, or any other C-derived language. A PHP script consists of a series of commands, or **statements**, each of which is an instruction that the Web server must follow 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 <b>test</b>!" );
```

This statement invokes a **built-in function** called `echo` and passes it a string of text: `This is a <b>test</b>!` Built-in functions can be thought of as things that PHP knows how to do without us having to spell out the details. PHP has a lot of built-in functions that let us do everything from sending email, to working with information that's stored in various types of databases. The `echo` function, however, simply takes the text that it's given, and places it into the HTML code of the page at the current location. Consider the following (`echo.php` in the code package):

```
<html>
<head>
<title> Simple PHP Example </title>
</head>
<body>
<p><?php echo("This is a <b>test</b>!"); ?></p>
</body>
</html>
```

If you paste this code into a file called `echo.php` (or `echo.php3`, if your Web host has not configured `.php` files to be recognized as PHP scripts) and place it on your Web server, a browser that views the page will see this:

```
<html>
<head>
<title> Simple PHP Example </title>
</head>
<body>
<p>This is a <b>test</b>!</p>
</body>
</html>
```

Notice that the string of text contained HTML tags (`<b>` and `</b>`), which is perfectly acceptable.

You may wonder why we need to surround the string of text with both parentheses and quotes. Quotes are used to mark the beginning and end of strings of text in PHP, so their presence is fully justified. The parentheses serve a dual purpose. First, they indicate that `echo` is a function that you want to call. Second, they mark the beginning and end of a list of "parameters" that you wish to provide, in order to tell the function what to do. In the case of the `echo` function, you only need to provide the string of text that you want to appear on the page. Later on, we'll look at functions that take more than one parameter (and we'll separate those parameters with commas), and we'll consider functions that take no parameters at all (for which we'll still need the parentheses, though we won't type anything between them).

Variables and Operators

Variables in PHP are identical to variables in most other programming languages. For the uninitiated, a variable is a name given to an imaginary box into which any 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 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 it appears after the statement above, assigns a new value to our 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 is 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.
```

The lines above each end with a comment. Comments are a way to describe what your code is doing - they insert explanatory text into your code, and tell the PHP interpreter to ignore it. Comments begin with `//` and they finish at the end of the same line. You might be familiar with `/* */` style comments in other languages - these work in PHP as well. I'll be using comments throughout the rest of this book to help explain what the code I present is doing.

Now, to get back to the four statements above, the operators we used here allow you to **add**, **subtract**, **multiply**, and **divide** numbers. Among others, there is also an operator that sticks strings of text together, called the **concatenation operator**:

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

Variables may be used almost anywhere that you use an actual value. Consider these examples:

```
$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 especially. 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. This process of converting variable names to their values is known in technical circles as **variable interpolation**. However, as the last line demonstrates, a string surrounded with single quotes will not interpolate variable names within the string.

User Interaction and Forms

For many applications of PHP, the ability to interact with users who view the Web page is essential. Veterans of JavaScript tend to think in terms of event handlers, which let you react directly to the actions of the user - for example, the movement of the mouse over a link on the page. Server-side scripting languages such as PHP have a more limited scope when it comes to user interaction. As PHP code is activated when a page is requested from the server, user interaction can only occur in a back-and-forth fashion: the user sends requests to the server, and the server replies with dynamically generated pages.

The key to creating interactivity with PHP is to understand the techniques we can use to send information about a users' interaction along with their request for a new Web page. PHP makes this fairly easy, as we'll now see.

The simplest method we can use to send information along with a page request uses the **URL query string**. If you've ever seen a URL with a question mark following the filename, you've witnessed this technique in use. Let's look at an easy example. Create a regular HTML file called `welcome.html` (no `.php` file extension is required, since there will be no PHP code in this file) and insert this link:

```
<a href="welcome.php?name=Kevin"> Hi, I'm Kevin! </a>
```

This is a link to a file called `welcome.php`, but as well as linking to the file, we'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, we have created a link that loads `welcome.php`, and informs the PHP code contained in the file that `name` equals `Kevin`.

To really understand the results of this, we need to look at `welcome.php`. Create it as a new HTML file, but this time note the `.php` extension - this tells the Web server that it can expect to interpret some PHP code in the file. If your Web server is not configured to accept `.php` as a file extension for PHP files, you may have to call it `welcome.php3` instead (in which case you'll also want to adjust the link above accordingly). In the body of this new file, type:

```
<?php
  echo( "Welcome to our Web site, $name!" );
?>
```

Now, if you use the link in the first file to load this second file, you'll see that the page says "Welcome to our Web site, Kevin!" The value of the variable passed in the query string of the URL was automatically placed into a PHP variable called `$name`, which we used to display the value passed as part of a text string.

You can pass more than one value in the query string if you want to. Let's look at a slightly more complex version of the same example. Change the link in the HTML file to read as follows (this is `welcome2.html` in the code archive located at http://sitepoint.com/books/?bookid=More):

```
<a href="welcome.php?firstname=Kevin&lastname=Yank"> Hi, I'm Kevin Yank! </a>
```

This time, we'll pass two variables: `firstname` and `lastname`. The variables are separated in the query string by an ampersand (&). 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 `welcome.php` file (this is `welcome2.php` in the code archive located at http://sitepoint.com/books/?bookid=More):

```
<?php
  echo( "Welcome to my Website, $firstname $lastname!" );
?>
```

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

Here's the code (`welcome3.html`):

```
<form action="welcome.php" method="get">
First Name: <input type="text" name="firstname" /><br />
Last Name: <input type="text" name="lastname" /><br />
<input type="submit" value="GO" />
</form>
```

Don't be alarmed at the slashes that appear in some of these tags (e.g. `<br />`). The new XHTML standard for coding Web pages, calls for these in any tag that does not have a closing tag, which includes `<input>` and `<br>` tags, among others. Current browsers do not require you to use the slashes, of course, but for the sake of standards-compliance, the HTML code in this book will observe this recommendation. Feel free to leave the slashes out if you prefer (I agree that they're not especially nice to look at).

This form has the exact same effect as the second link we looked at (with `firstname=Kevin&lastname=Yank` in the query string), except that you can enter whatever names you like. When you click the submit button (which has a label of "GO"), the browser will load `welcome.php` and automatically add the variables and their values to the query string for you. It retrieves the names of the

variables from the `name` attributes of the `input type="text"` tags, and it obtains the values from the information the user typed into the text fields.

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 above) causes them to be passed in the query string, but there is an alternative. It's not always desirable – or even technically feasible – 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 that contained several paragraphs of text in the query string would be ridiculously long, and would exceed by far the maximum length of the URL in today's browsers. The alternative is for the browser to pass the information invisibly, behind the scenes. The code for this looks exactly the same, but where we set the form method to `get` in the last example, here we set it to `post` (`welcome4.html`):

```
<form action="welcome.php" method="post">
First Name: <input type="text" name="firstname" /><br />
Last Name: <input type="text" name="lastname" /><br />
<input type="submit" value="GO" />
</form>
```

This 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 not have 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 their submission of the form, that bookmark will be useless, as it doesn't contain the submitted values. This, incidentally, is the main reason that search engines like AltaVista use the query string to submit search terms. If you bookmark a search results page on AltaVista, you can use that bookmark to perform the same search again later, because the search terms are contained in the URL.

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

Control Structures

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

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

The most basic, and most often-used, control structure is the **`if-else` statement**. Here's what it looks like:

```
if ( condition ) {
  // Statement(s) to be executed if
  // condition is true.
} else {
  // (Optional) Statement(s) to be
  // executed if condition is false.
}
```

This control structure lets us tell PHP to execute one set of statements or another, depending on whether some condition is true or false. If you'll indulge my vanity for a moment, here's an example that shows a twist on the `welcome.php` file we created earlier:

```
if ( $name == "Kevin" ) {
  echo( "Welcome, oh glorious leader!" );
} else {
  echo( "Welcome, $name!" );
}
```

Now, if the name variable passed to the page has a value of `Kevin`, a special message will be displayed. Otherwise, the normal message will be displayed and will contain the name that the user entered.

As indicated in the code structure above, the **`else` clause** (that part of the `if-else` statement that says what to do if the condition is false) is optional. Let's say you wanted to display the special message above

only if the appropriate name was entered, but otherwise, you didn't want to display any message. Here's how the code would look:

```
if ( $name == "Kevin" ) {
  echo( "Welcome, oh glorious leader!" );
}
```

The `==` used in the condition above is the PHP operator that's used to compare two values to see whether they're equal. It's important to remember to type the double-equals, because if you were to use a single equals sign you'd be using the assignment operator discussed above. So, instead of comparing the variable to the designated value, instead, you'd assign a new value to the variable (an operation which, incidentally, evaluates as true). This would not only cause the condition to always be true, but might change the value in the variable you're checking, which could cause all sorts of problems.

Conditions can be more complex than a single comparison for equality. Recall that we modified `welcome.php` to take 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 (`welcome3.php`):

```
if ( "Kevin" == $firstname and "Yank" == $lastname ) {
  echo( "Welcome, oh glorious leader!" );
} else {
  echo( "Welcome to my Website, $firstname $lastname!" );
}
```

This condition will be true if and only if `$firstname` has a value of `Kevin` and `$lastname` has a value of `Yank`. The word `and` in the above condition makes the whole condition true only if both of the comparisons evaluate to true. Another such operator is `or`, 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), they work in PHP as well.

We'll look at more complicated comparisons as the need arises. For the time being, a general familiarity with the `if-else` statement 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 to repeatedly execute a set of statements. Here's what a `while` loop looks like:

```
while ( condition ) {
  // statement(s) to execute over
  // and over as long as condition
  // remains true
}
```

This works very similarly to an `if-else` statement without an `else` clause. 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 one-hundred-and-first), execution jumps immediately to the next statement that follows the while loop (after the closing brace).

Loops like these come in handy whenever you're working with long lists of things (such as jokes stored in a database... hint-hint!), but for now we'll illustrate with a trivial example: counting to ten. This script is available as `count10.php` in the code archive (located at http://sitepoint.com/books/?bookid=More).

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

It looks kind of scary, I know, but let me talk you through it line by line. The first line creates a variable called `$count` and assigns it a value of `1`. The second line is the start of a `while` loop, the condition for which is that the value of `$count` is less than or equal (`<=`) to `10`. The third and fourth lines make up the body of the `while` loop, and will be executed over and over, as long as that condition holds true. The third

line simply outputs the value of `$count` followed by a space. The fourth line adds one to the value of `$count` (`$count++` is a shortcut for `$count = $count + 1` -- both will work).

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 `$count` is given a new value of `2`. The condition is still true the second time it is checked, so the value (`2`) is output and a new value (`3`) is assigned. This process continues, outputting the values `3`, `4`, `5`, `6`, `7`, `8`, `9`, and `10`. Finally, `$count` is given a value of `11`, and the condition is false, which ends the loop. The net result of the code is to output the string "`1 2 3 4 5 6 7 8 9 10 `".

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 is designed specifically to handle examples like that above, where we are counting through a series of values until some condition is met, is called a **`for` loop**. Here's what they look like

```
for ( initialize; condition; update ) {
  // statement(s) to execute over
  // and over as long as condition
  // remains true after each update
}
```

Here's what the above `while` loop example looks like when implemented as a `for` loop:

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

Multi-Purpose Pages

Let's say you wanted to construct your site so that it showed the visitor's name at the top of every page. With our custom welcome message example above, we're halfway there already. Here are the problems we'll need to overcome to extend the example into what we need:

- We need the name on every page of the site, not just one.
- We have no control over which page of our site users will view first.

The first problem isn't too hard to overcome. Once we have the user's name in a variable on one page, we can pass it with any request to another page by adding the name to the query string of all links:

```
<a href="newpage.php?name=<?php echo(urlencode($name)); ?>"> A link </a>
```

Notice that we've embedded PHP code right in the middle of an HTML tag. This is perfectly legal, and will work just fine. A shortcut exists for those times when you simply want to echo a PHP value in the middle of your HTML code. The shortcut looks like this:

```
<a href="newpage.php?name=<?=urlencode($name)?>"> A link </a>
```

The tags `<?= ... ?>` perform the same function as the much longer code `<?php echo( ... ); ?>`. This is a handy shortcut that I'll use several times through the rest of this book.

You're familiar with the `echo` function, but `urlencode` is probably new to you. This function takes any special characters in the string (for example, spaces) and converts them into the special codes they need to be in order to appear in the query string. For example, if the `$name` variable had a value of `"Kevin Yank"`, then as spaces are not allowed in the query string, the output of `urlencode` (and thus the string output by `echo`) would be `"Kevin+Yank"`. This would then be automatically converted back by PHP when it created the `$name` variable in `newpage.php`.

Okay, so we've got the user's name being passed with every link in our site. Now all we need is to get that name in the first place. In our welcome message example, we had a special HTML page with a form in it that prompted the user for his or her name. The problem with this (identified by the second point above) is that we couldn't -- nor would we wish to -- force the user to enter our Web site by that page every time he or she visited our site.

The solution is to have every page of our site check to see if a name has been specified, and prompt the user for a name if necessary. This means that every page of our site will either display its content, or a prompt the

user to enter a name, depending on whether the `$name` variable is found to have a value. If this is beginning to sound to you like a good place for an `if-else` statement, you're a quick study!

We'll refer to pages that are capable of displaying completely different content depending on some condition, as "multi-purpose pages". The code of a multi-purpose page looks something like this:

```
<html>
<head>
<title> Multi-Purpose Page Outline </title>
</head>
<body>

<?php if (condition) { ?>

<!-- HTML content to display if condition is true -->

<?php } else { ?>

<!-- HTML content to display if condition is false -->

<?php } ?>

</body>
</html>
```

This may confuse you at first, but in fact this is just a normal `if-else` statement with HTML code sections that depend on the condition, instead of PHP statements. This example illustrates one of the big selling points of PHP: that you can switch in and out of "PHP mode" whenever you like. If you think of `<?php` as the command to switch into "PHP mode", and `?>` as the command to go back into "normal HTML mode", the above example should make perfect sense.

There's an alternate form of the `if-else` statement that can make your code more readable in situations like this. Here's the outline for a multi-purpose page using the alternate `if-else` form:

```
<html>
<head>
<title> Multi-Purpose Page Outline </title>
</head>
<body>

<?php if (condition): ?>

<!-- HTML content to display if condition is true -->

<?php else: ?>

<!-- HTML content to display if condition is false -->

<?php endif; ?>

</body>
</html>
```

Okay, now that we have all the tools we need in hand, let's look at a sample page of our site (`samplepage.php` in the code archive located at http://sitepoint.com/books/?bookid=More):

```
<html>
<head>
<title> Sample Page </title>
</head>
<body>

<?php if ( !isset($name) ): ?>

  <!-- No name has been provided, so we
       prompt the user for one.         -->

  <form action="<?=$PHP_SELF?>" method="get">
  Please enter your name: <input type="text" name="name" />
  <input type="submit" value="GO" />
  </form>
```

```
<?php else: ?>

  <p>Your name: <?=$name?></p>

  <p>This paragraph contains a <a
href="newpage.php?name=<?=urlencode($name)?>">link</a> that passes the name
variable on to the next document.</p>

<?php endif; ?>

</body>
</html>
```

There are two new tricks in the above code, but overall you should be pretty comfortable with the way it works. First of all, we're using a new function called `isset` in the condition. This function returns (outputs) a value of true if the variable it is given has been assigned a value (i.e. if a name has been provided), and false if the variable does not exist (i.e. if a name has not yet been given). The exclamation mark (also known as the **negation operator**, or the **not operator**) that appears before the name of the function reverses the returned value from true to false or vice-versa. Thus, the form is displayed when the $name variable is *not* set. The second new trick is the use of the variable `$PHP_SELF` to specify the `action` attribute of the `form` tag. This variable is one of several that PHP always gives a value to automatically. In particular, `$PHP_SELF` will always be set to the URL of the current page. This gives us an easy way to create a form that, when submitted, will load the very same page, but this time with the `$name` variable specified.

If we structure all the pages on our site in this way, visitors will be prompted for their name by the first page they attempt to view, whichever page this happens to be. Once they enter their name and click "GO", they'll be presented with the exact page they requested. The name they entered is then passed in the query string of every link from that point onward, ensuring that they are prompted only the once.

Summary

In this chapter, we've had a taste of the PHP server-side scripting language by exploring all the basic language features: statements, variables, operators, and control structures. The sample applications we've seen have been pretty simple, but don't let that dissuade you. The real power of PHP is in the hundreds of built-in functions that let you do everything: access data in a MySQL database, send e-mail, dynamically generate images, and even create Adobe Acrobat PDF files on the fly.

In Chapter 4, we'll delve into the MySQL functions in PHP, to show how to publish the joke database that we created in Chapter 2 on the Web. This will set the scene for the ultimate goal of this series – creating a complete content management system for your Web site in PHP and MySQL.

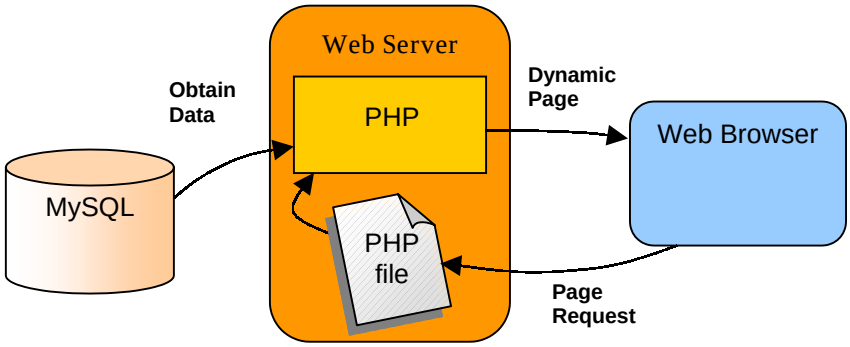
4 Publishing MySQL Data on the Web

This is it -- the stuff you signed up for! In this chapter, you'll learn how to take information stored in a database and display it on a Web page for all to see. So far you have installed and learned the basics of MySQL, a relational database engine, and PHP, a server-side scripting language. Now you'll see how to use these two new tools together to create a true database-driven Web site!

A Look Back at First Principles

Before we leap forward, it's worth a brief look back to remind ourselves of the goal we're working toward. We have two powerful, new tools at our disposal: the PHP scripting language, and the MySQL database engine. It's important to understand how these two 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 dynamically pulled from the database to create Web pages for people to view with a regular Web browser. So on one end of the system you have a visitor to your site who uses a Web browser to load `http://www.yoursite.com/`, and expects to view a standard HTML Web page. On the other end you have the content of your site, which sits in one or more tables in a MySQL database that only understands how to respond to SQL queries (commands).



As shown in the diagram above, 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, then spits it out dynamically as the nicely-formatted HTML page that the browser expects. With PHP, you can write the presentation aspects of your site (the fancy graphics and page layouts) as "templates" in regular HTML. Where the content belongs in those templates, you use some PHP code to connect to the MySQL database and -- using SQL queries just like those you used to create a table of jokes in Chapter 2 -- retrieve and display some content in its place.

Just so it's clear and fresh in your mind, this is what will happen when someone visits a page on our database-driven Web site:

- The visitor's Web browser requests the Web page using a standard URL.
- The Web server software (Apache, IIS, or whatever) recognizes that the requested file is a PHP script, and so the server interprets the file using its PHP plug-in, before responding to the page request.
- Certain PHP commands (which we have yet to learn) connect to the MySQL database and request the content that belongs in the Web page.
- The MySQL database responds by sending the requested content to the PHP script.
- The PHP script stores the content into one or more PHP variables, and then uses the now-familiar `echo` function to output the content as part of the Web page.

- The PHP plug-in finishes up by handing a copy of the HTML it has created to the Web server.
- 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 plug-in.

Connecting to MySQL with PHP

Before you can get content out of your MySQL database for inclusion in a Web page, you must first 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. PHP has no need of any special program, however; support for connecting to MySQL is built right into the language. The following PHP function call establishes the connection:

```
mysql_connect(address, username, password);
```

Here, `address` is the IP address or hostname of the computer on which the MySQL server software is running (`"localhost"` if it's running on the same computer as the Web server software), and `username` and `password` are the same MySQL user name and password you used to connect to the MySQL server in Chapter 2.

You may or may not remember that functions in PHP usually return (output) a value when they are called. Don't worry if this doesn't ring any bells for you -- it's a detail that I glossed over when I first discussed functions. In addition to doing something useful when they are called, most functions output a value, and that value may be stored in a variable for later use. The `mysql_connect` function shown above, for example, returns a number 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.

```
$dbcnx = mysql_connect("localhost", "root", "mypasswd");
```

As described above, the values of the three function parameters may differ for your MySQL server. What's important to see here is that the value returned by `mysql_connect` (which we'll call a connection identifier) is stored in a variable named `$dbcnx`.

Since the MySQL server is a completely separate piece of software, we must consider the possibility that the server is unavailable, or inaccessible due to a network outage, or because the username / password combination you provided is not accepted by the server. In such cases, the `mysql_connect` function doesn't return a connection identifier (since no connection is established). Instead, it returns false. This allows us to react to such failures using an `if` statement:

```
$dbcnx = @mysql_connect("localhost", "root", "mypasswd");
if (!$dbcnx) {
  echo( "<p>Unable to connect to the " .
        "database server at this time.</p>" );
  exit();
}
```

There are three new tricks in the above code fragment. First, we have placed an `@` symbol in front of the `mysql_connect` function. Many functions, including `mysql_connect`, automatically display ugly error messages when they fail. Placing an `@` symbol in front of the function name tells the function to fail silently, allowing us to display our own, friendlier error message.

Next, we put an exclamation point in front of the `$dbcnx` variable in the condition of the `if` statement. The exclamation point is the PHP **negation operator**, which basically flips a false value to true, or a true value to false. Thus, if the connection fails and `mysql_connect` returns false, `!$dbcnx` will evaluate to true, and cause the statements in the body of our `if` statement to be executed. Alternatively, if a connection was made, the connection identifier stored in `$dbcnx` will evaluate to true (any number other than zero is considered "true" in PHP), so `!$dbcnx` will evaluate to false, and the statements in the `if` statement will not be executed.

The last new trick is the `exit` function, which is the first example that we've encountered of a function that takes no parameters. All this function does is cause PHP to stop reading the page at this point. This is a good response to a failed database connection, because in most cases the page will be unable to display any useful information without that connection.

As in Chapter 2, the next step, once a connection is established, is to select the database you want to work with. Let's say we want to work with the joke database we created in Chapter 2. The database we created was called `jokes`. Selecting that database in PHP is just a matter of another function call:

```
mysql_select_db("jokes", $dbcnx);
```

Notice we use the `$dbcnx` variable that contains the database connection identifier to tell the function which database connection to use. This parameter is actually optional. When it's omitted, the function will automatically use the link identifier for the last connection opened. 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:

```
if (! @mysql_select_db("jokes") ) {
  echo( "<p>Unable to locate the joke " .
        "database at this time.</p>" );
  exit();
}
```

Notice 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 if the condition is true or false, PHP executes the function and then checks its return value – exactly what we need to happen.

With a connection established and a database selected, we are now ready to begin using the data stored in the database.

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 `mysql_query` function.

```
mysql_query(query, connection_id);
```

Here `query` is a string that contains the SQL command we want to execute. As with `mysql_select_db`, the connection identifier parameter is optional.

What this function returns will depend on the type of query being sent. For most SQL commands, `mysql_query` returns either true or false to indicate success or failure respectively. Consider the following example, which attempts to create the `Jokes` table we created in Chapter 2:

```
$sql = "CREATE TABLE Jokes (
          ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
          JokeText TEXT,
          JokeDate DATE NOT NULL
        )";
if ( @mysql_query($sql) ) {
  echo("<p>Jokes table successfully created!</p>");
} else {
  echo("<p>Error creating Jokes table: " .
       mysql_error() . "</p>");
}
```

Again, we use the `@` trick to suppress any error messages produced by `mysql_query`, and instead print out a friendlier error message of our own. The `mysql_error` function used here returns a string of text that describes the last error message that was sent by the MySQL server.

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":

```
$sql = "UPDATE Jokes SET JokeDate='1990-04-01'
        WHERE JokeText LIKE '%chicken%'";
```

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:

```
if ( @mysql_query($sql) ) {
  echo("<p>Update affected " . mysql_affected_rows() .
       " rows.</p>");
} else {
```

```
  echo("<p>Error performing update: " . mysql_error() .
       "</p>");
}
```

SELECT queries are treated a little differently, since they can retrieve a lot of data, and PHP must provide ways to handle that information.

Handling SELECT Result Sets

For most SQL queries, the mysql_query function returns either true (success) or false (failure). For SELECT queries this just isn't enough. 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. As a result, when it processes a SELECT query, mysql_query returns a number that identifies 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.

```
$result = @mysql_query("SELECT JokeText FROM Jokes");
if (!$result) {
  echo("<p>Error performing query: " . mysql_error() .
       "</p>");
  exit();
}
```

Provided no error was encountered in processing the query, the above code will place a result set that contains the text of all the jokes stored in the Jokes table into the variable $result. As there's no practical limit on the number of jokes in the database, that result set can be pretty big.

We mentioned before that the while loop is a useful control structure for dealing with large amounts of data. Here's an outline of the code to process the rows in a result set one at a time:

```
while ( $row = mysql_fetch_array($result) ) {
  // process the row...
}
```

The condition for the while loop probably doesn't much resemble the conditions you're used to, so let me explain how it works. Consider the condition as a statement all by itself:

```
$row = mysql_fetch_array($result);
```

The mysql_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. If you're not familiar with the concept of arrays, don't worry: we'll discuss it in a moment. When there are no more rows in the result set, mysql_fetch_array instead returns false.

Now, the above statement assigns a value to the $row variable, but at the same time the whole statement itself takes on that same value. This is what lets us use the statement as a condition in our while loop. Since while loops keep looping until their condition evaluates to false, the 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 is to figure out how to get the values out of the $row variable each time the loop runs.

Rows of a result set are represented as 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. In the case of our database row, the compartments are named after the table columns in our result set. If $row is a row in our result set, then $row["JokeText"] is the value in the JokeText column of that row. So here's what our while loop should look like if we want to print the text of all the jokes in our database:

```
while ( $row = mysql_fetch_array($result) ) {
  echo("<p>" . $row["JokeText"] . "</p>");
}
```

To summarize, here's the complete code of a PHP Web page that will connect to our database, fetch the text of all the jokes in the database, and display them in HTML paragraphs. The code of this example is available as jokelist.php in the code archive (located at http://sitepoint.com/books/?bookid=More).

```
<html>
<head>
<title> Our List of Jokes </title>
<head>
```

```
<body>
<?php

  // Connect to the database server
  $dbcnx = @mysql_connect("localhost", "root",
                          "mypasswd");
  if (!$dbcnx) {
    echo( "<p>Unable to connect to the " .
          "database server at this time.</p>" );
    exit();
  }

  // Select the jokes database
  if (! @mysql_select_db("jokes") ) {
    echo( "<p>Unable to locate the joke " .
          "database at this time.</p>" );
    exit();
  }

?>
<p> Here are all the jokes in our database: </p>
<blockquote>

<?php

  // Request the text of all the jokes
  $result = @mysql_query("SELECT JokeText FROM Jokes");
  if (!$result) {
    echo("<p>Error performing query: " . mysql_error() .
         "</p>");
    exit();
  }

  // Display the text of each joke in a paragraph
  while ( $row = mysql_fetch_array($result) ) {
    echo("<p>" . $row["JokeText"] . "</p>");
  }


?>

</blockquote>
</body>
</html>
```

Inserting Data into the Database

In this section, we'll see how we can use all the tools at our disposal to allow visitors to our site to add their own jokes to the database. If you enjoy a challenge, you might want to try to figure this out on your own before you read any further. There is precious little new material in this section. It's mostly just a sample application of everything we've learned so far.

If you want to let visitors to your site type in new jokes, you'll obviously need a form. Here's the code for a form that will fit the bill:

```
<form action="<?=$PHP_SELF?>" method="post">
<p>Type your joke here:<br />
<textarea name="joketext" rows="10" cols="40" wrap>
</textarea><br />
<input type="submit" name="submitjoke" value="SUBMIT" />
</p>
</form>
```

As we've seen before, this form, when submitted, will load the very same page (due to the use of the `$PHP_SELF` variable for the form's `ACTION` attribute), but with two variables attached to the request. The first, `$joketext`, will contain the text of the joke as typed into the text area. The second, `$submitjoke`, will always contain the value `"SUBMIT"`, which can be used as a sign that a joke has been submitted.

To insert the submitted joke into the database, we just use `mysql_query` to run an `INSERT` query, using the `$joketext` variable for the value to be submitted:

```
if ($submitjoke == "SUBMIT") {
  $sql = "INSERT INTO Jokes SET
          JokeText='$joketext',
          JokeDate=CURDATE()";
  if (@mysql_query($sql)) {
    echo("<p>Your joke has been added.</p>");
  } else {
    echo("<p>Error adding submitted joke: " .
         mysql_error() . "</p>");
  }
}
```

The one new trick in this whole example appears in the SQL code here. Note the use of the MySQL function `CURDATE()` to assign the current date as the value of the `JokeDate` column to be inserted into the database. MySQL actually has dozens of these functions, but we'll only introduce them as required. For a complete function reference, refer to Appendix **B**.

We now have the code that will allow a user to type a joke and add it to our database. All that remains is to slot it into our existing joke viewing page in a useful fashion. Since most users will only want to view our jokes, we don't want to mar our page with a big, ugly form unless the user expresses an interest in adding a new joke. For this reason, our application is well suited for implementation as a multi-purpose page. Here's the code (available as `jokes.php` in the code archive located at http://sitepoint.com/books/?bookid=More):

```
<html>
<head>
<title> The Internet Joke Database </title>
</head>
<body>
<?php
  if (isset($addjoke)): // If the user wants to add a joke
?>

<form action="<?=$PHP_SELF?>" method="post">
<p>Type your joke here:<br />
<textarea name="joketext" rows="10" cols="40" wrap>
</textarea><br />
<input type="submit" name="submitjoke" value="SUBMIT" />
</p>
</form>

<?php
  else: // Default page display

    // Connect to the database server
    $dbcnx = @mysql_connect("localhost", "root",
                            "mypasswd");
    if (!$dbcnx) {
      echo( "<p>Unable to connect to the " .
            "database server at this time.</p>" );
      exit();
    }

    // Select the jokes database
    if (! @mysql_select_db("jokes") ) {
      echo( "<p>Unable to locate the joke " .
            "database at this time.</p>" );
      exit();
    }

    // If a joke has been submitted,
    // add it to the database.
    if ($submitjoke == "SUBMIT") {
      $sql = "INSERT INTO Jokes SET
              JokeText='$joketext',
              JokeDate=CURDATE()";
      if (@mysql_query($sql)) {
        echo("<p>Your joke has been added.</p>");
      } else {
        echo("<p>Error adding submitted joke: " .
             mysql_error() . "</p>");
```

```
    }
  }

  echo("<p> Here are all the jokes in our database:" .
       "</p>");

  // Request the text of all the jokes
  $result = @mysql_query("SELECT JokeText FROM Jokes");
  if (!$result) {
    echo("<p>Error performing query: " .
         mysql_error() . "</p>");
    exit();
  }

  // Display the text of each joke in a paragraph
  while ( $row = mysql_fetch_array($result) ) {
    echo("<p>" . $row["JokeText"] . "</p>");
  }

  // When clicked, this link will load this page
  // with the joke submission form displayed.
  echo("<p><a href='$PHP_SELF?addjoke=1'>" .
       "Add a Joke!</a></p>");

endif;

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

There we go! With a single file that contains a little PHP code we're able to view existing jokes, and add jokes to, our MySQL database.

A Challenge

As homework, see if you can figure out how to put a link labeled "Delete this Joke" next to each joke on the page that, when clicked, will remove that joke from the database and display the updated joke list. Here are a few hints to get you started:

- You'll still be able to do it all in a single multi-purpose page.
- You'll need to use the SQL `DELETE` command, which we learned about in Chapter 2.
- This is the tough one. To delete a particular joke, you'll need to be able to uniquely identify it. The `ID` column in the `Jokes` table was designed 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 query string of the "Delete this Joke" link is a perfect place to put this value.

If you think you have the answer, or if you'd just like to see the solution, turn the page. Good luck!

Summary

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 `Jokes` database online, and allowed visitors to add jokes of their own to it.

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!

"Homework" Solution

Here's the solution to the "homework" challenge posed above. These changes were required to insert a "Delete this Joke" link next to each joke:

- Previously, we passed an `$addjoke` variable with our "Add a Joke!" link at the bottom of the page to signal that our script should display the joke entry form, instead of the usual list of jokes. In a similar fashion, we pass a `$deletejoke` variable with our "Delete this Joke" link to indicate our desire to have a joke deleted.
- For each joke, we fetch the `ID` column from the database, along with the `JokeText` column, so that we know which ID is associated with each joke in the database.
- We set the value of the `$deletejoke` variable to the ID of the joke that we're deleting. To do this, we insert the ID value fetched from the database into the HTML code for the "Delete this Joke" link of each joke.
- Using an `if` statement, we watch to see if `$deletejoke` is set to a particular value (through the `isset` function) when the page loads. If it is, we use the value to which it is set (the ID of the joke to be deleted) in an SQL `DELETE` statement that deletes the joke in question.

Here's the complete code, which is also available as `challege.php` in the code archive (located at http://sitepoint.com/books/?bookid=More). If you have any questions, don't hesitate to post them in the SitePoint.com forums!

```
<html>
<head>
<title> The Internet Joke Database </title>
</head>
<body>
<?php
  if (isset($addjoke)): // The user wants to add a joke
?>

<form action="<?=$PHP_SELF?>" method="post">
<p>Type your joke here:<br />
<textarea name="joketext" rows="10" cols="40" wrap>
</textarea><br />
<input type="submit" name="submitjoke" value="SUBMIT" />
</p>
</form>

<?php
  else:

    // Connect to the database server
    $dbcnx = @mysql_connect("localhost", "root",
                            "mypasswd");
    if (!$dbcnx) {
      echo( "<p>Unable to connect to the " .
            "database server at this time.</p>" );
      exit();
    }

    // Select the jokes database
    if (! @mysql_select_db("jokes") ) {
      echo( "<p>Unable to locate the joke " .
            "database at this time.</p >" );
      exit();
    }

    // If a joke has been submitted,
    // add it to the database.
    if ($submitjoke == "SUBMIT") {
      $sql = "INSERT INTO Jokes SET
              JokeText='$joketext',
              JokeDate=CURDATE()";
      if (@mysql_query($sql)) {
        echo("<P>Your joke has been added.</P>");
      } else {
```

```
        echo("<P>Error adding submitted joke: " .
             mysql_error() . "</P>");
      }
    }

    // If a joke has been deleted,
    // remove it from the database.
    if (isset($deletejoke)) {
      $sql = "DELETE FROM Jokes
              WHERE ID=$deletejoke";
      if (@mysql_query($sql)) {
        echo("<p>The joke has been deleted.</p>");
      } else {
        echo("<p>Error deleting joke: " .
             mysql_error() . "</p>");
      }
    }

    echo("<p> Here are all the jokes in our database: " .
         "</p>");

    // Request the ID and text of all the jokes
    $result = @mysql_query(
                        "SELECT ID, JokeText FROM Jokes");
    if (!$result) {
      echo("<p>Error performing query: " .
           mysql_error() . "</p>");
      exit();
    }

    // Display the text of each joke in a paragraph
    // with a "Delete this Joke" link next to each.
    while ( $row = mysql_fetch_array($result) ) {
      $jokeid = $row["ID"];
      $joketext = $row["JokeText"];
      echo("<p>$joketext " .
           "<a href='$PHP_SELF?deletejoke=$jokeid'>" .
           "Delete this Joke</a></p>");
    }

    // When clicked, this link will load this page
    // with the joke submission form displayed.
    echo("<p><a href='$PHP_SELF?addjoke=1'>Add a ".
         "Joke!</a></p>");

  endif;

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

What's Next?

If you've enjoyed the first four chapters of *Build Your Own Database Driven Website Using PHP & MySQL*, why not keep reading?

Explore the storage of binary data in MySQL, learn about cookies and sessions in PHP, and benefit from a handy set of PHP and MySQL reference tables that include PHP and MySQL syntax, functions, column types and more.

In the next 8 Chapters you'll learn how to:

- Build a Working Content Management System
- Build an ecommerce shopping cart
- Automatically send email on triggered events
- Build a Web-based file repository or photo gallery
- Utilize sessions and cookies to track site visitors.
- And a whole lot more...

You shouldn't be without this amazing hands-on desk reference!

Pick up your copy from SitePoint.com today!

"Kevin Yank has the extraordinary ability to introduce a new level of clarity and simplicity to a powerful technology like MySQL. This book is a must-have for all Webmasters"

Matt Wagner, MySQL.com