

Chapter 1

Installation

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

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

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

If you're used to building websites with HTML, CSS, and perhaps even a smattering of JavaScript, you're probably familiar with uploading the files that make up your site to a certain location. It might be a web hosting service you've paid for, web space provided by your Internet Service Provider (ISP), or maybe a web server set

2 PHP & MySQL: Novice to Ninja

up by the IT department of the company that you work for. In any case, once you copy your files to any of these destinations, a software program called a **web server** is able to find and serve up copies of those files whenever they're requested by a web browser like Internet Explorer, Google Chrome, Safari, or Firefox. Common web server software programs you may have heard of include Apache HTTP Server (Apache) and Internet Information Services (IIS).

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

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

The goal of this first chapter is to set you up with a web server equipped with PHP and MySQL. I'll provide step-by-step instructions that work on recent Windows and Mac OS X, so no matter what flavor of computer you're using, the instructions you need should be right here.¹

Your Own Web Server

If you're lucky, your current web host's web server already has PHP and MySQL installed. Most do—that's one of the reasons why PHP and MySQL are so popular. If your web host is so equipped, the good news is that you'll be able to publish your

¹ Linux users, you'll find instructions in Appendix A, because I suspect that most of you will probably want to install it your own way, regardless of what I write here.

first database driven website without having to shop for a web host that supports the right technologies.

However, you're still going to need to install PHP and MySQL yourself. That's because you need your own PHP-and-MySQL-equipped web server on which to test your database driven website before you publish it for all the world to see.

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

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

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

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

Windows Installation

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

4 PHP & MySQL: Novice to Ninja

The easiest way to get a web server up and running on Windows is to use a free software package called XAMPP for Windows. This all-in-one program includes built-in copies of Apache, PHP, and MySQL. Let me take you through the process of installing it.



The Do-it-yourself Option

In past editions of this book, I recommended that you set up Apache, PHP, and MySQL individually, using the official installation packages for each. This is a good practice for beginners, I argued, because it gives you a strong sense of how these pieces all fit together.

Unfortunately, this meant that many readers spent their first few hours in “PHP Land” wrestling their way through a protracted sequence of detailed installation instructions. Worse still, sometimes the finer points of these became outdated due to some subtle change to one of the software packages.

Nowadays, I strongly believe that the best way to learn PHP and MySQL is to start *using* them right away. The quicker and more hassle-free the installation process, the better. That’s why I ask you to use XAMPP in this edition. In addition, there’s every chance you’re just dabbling in this stuff, so why junk up your computer with a bunch of separate but interdependent pieces of software that will be tricky to remove?

Nevertheless, if you’re a die-hard do-it-yourselfer, a tech-savvy power user, or if you simply reach the end of this book and wonder how the pros do it, I’ve included a detailed set of installation instructions for individual packages in Appendix A. Feel free to follow them instead of the instructions in this section if you’re that way inclined.

1. Download the latest version of XAMPP for Windows from the Apache Friends website² (you’ll need to scroll down to find the download links). Grab the Installer version that is recommended (as of this writing, XAMPP for Windows 1.7.7 is 81MB in size), then double-click the file to launch the installer, as shown in Figure 1.1.

² <http://www.apachefriends.org/en/xampp-windows.html>



Figure 1.1. The XAMPP Installer



User Account Control (UAC) warning

Depending on the version of Windows you're using and your exact system configuration, the XAMPP installer may display the warning message shown in Figure 1.2.

Although this message is a little alarming at first, be assured it's no big deal. It simply recommends not to install XAMPP in **C:\Program Files** as you do most programs due to problems this will cause with file permissions. The installer defaults to installing in **C:\xampp** anyway.

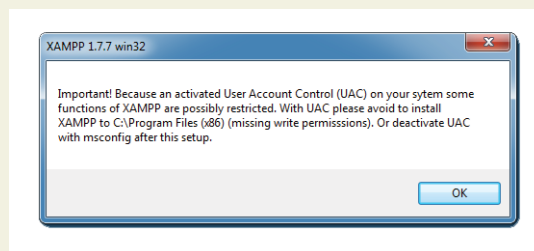


Figure 1.2. XAMPP may warn you about "User Account Control (UAC)"

6 PHP & MySQL: Novice to Ninja

2. The installer will prompt you for a location to install XAMPP. The default of **c:\xampp** shown in Figure 1.3 is an ideal choice, but if you have feel strongly about installing it elsewhere (such as on a different drive), go ahead and specify your preferred location. Just avoid the usual **C:\Program Files** (or similar) location, since XAMPP requires permissions that Windows restricts for files in that folder.

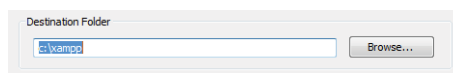


Figure 1.3. The default destination folder is a good choice

3. The installer will prompt you with a number of options. The default selections shown in Figure 1.4 are probably what you want at this stage. If you like to keep a clean desktop, you might want to uncheck the **Create a XAMPP desktop icon** checkbox. If you want your Apache and MySQL servers running at all times (rather than having to start them manually whenever you sit down to do some development), you can check the **Install Apache as service** and **Install MySQL as service** checkboxes. In the following instructions, though, I'll assume you haven't.

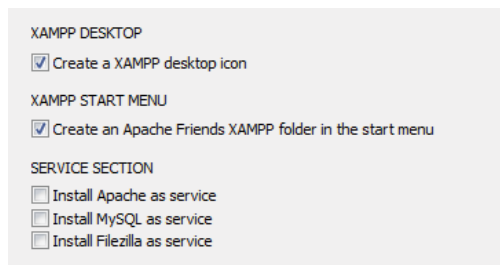


Figure 1.4. The default options are fine

4. Once the installer has completed, you'll be prompted to start the XAMPP Control Panel. Click **No** so that I can show you how to start it the conventional way. Once its work is done, the installer will quit.
5. At this point, I recommend shutting down and restarting your computer (even though the XAMPP installer won't ask you to). In my testing, the next steps failed to work until I restarted my system, and posts on the XAMPP support forum support this.³

³ <http://www.apachefriends.org/f/viewtopic.php?f=16&t=48484>

Once the installation is complete and your system has restarted, you can fire up the XAMPP Control Panel. You'll find it on the **Start** menu under **All Programs > Apache Friends > XAMPP > XAMPP Control Panel**. An orange XAMPP icon will appear in your Windows System Tray (although by default it will disappear after a few seconds), and the XAMPP Control Panel Application shown in Figure 1.5 will open.

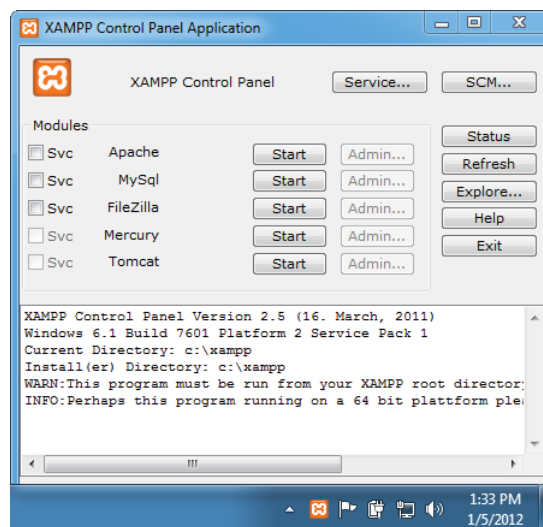


Figure 1.5. The XAMPP Control Panel

Click the **Start** buttons next to **Apache** and **MySQL** (sic) in the **Modules** list to launch the Apache and MySQL servers built into XAMPP. A green **Running** status indicator should appear next to each server in the list.

Depending on your Windows version and configuration, you'll probably receive a Windows Firewall alert for each server, like the one in Figure 1.6. This will happen when the servers attempt to start listening for browser requests from the outside world.

8 PHP & MySQL: Novice to Ninja



Figure 1.6. This security alert tells you Apache is doing its job

If you want to make absolutely sure that only you can access your development servers, click **Cancel**. You'll still be able to connect to the web server using a browser running on your own computer. In some cases, however, it can be handy to access your server from another computer on your network (such as from a co-worker's machine, to demonstrate the amazing website you have built); for this reason, I recommend selecting the **Private networks, such as my home or work network** option and clicking **Allow access**.



Why doesn't my server start?

If your Apache or MySQL server fails to start, there are a number of possible causes. By far the most common reason is that you already have a web server (be it another copy of Apache or Microsoft's Internet Information Services) or MySQL server running on your computer.

Look around your Start menu and the **Uninstall a program** section of your Windows Control Panel to see if you can spot another installation of Apache HTTP Server or MySQL in order to shut off or uninstall. There's another program similar to XAMPP called WampServer, which, if installed, could be the cause of the problem.

If you think you might have Microsoft's own web server—Internet Information Services (IIS)—running on your system, you can try following Microsoft's instructions for shutting it down.⁴

⁴ [http://technet.microsoft.com/en-us/library/cc732317\(WS.10\).aspx](http://technet.microsoft.com/en-us/library/cc732317(WS.10).aspx)

Still stuck? The advice in the XAMPP for Windows FAQ⁵ might help, especially if you're running Skype (as it can interfere with web servers in some network configurations).

Once both servers appear to be running smoothly, click the **Admin...** button next to **Apache**. Launch your web browser and load <http://localhost/xampp/>, the XAMPP for Windows admin page shown in Figure 1.7.

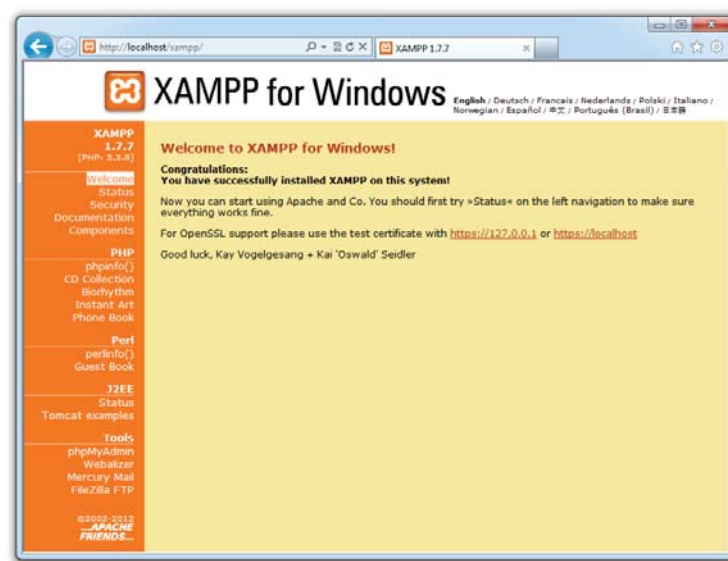


Figure 1.7. The admin page provided by XAMPP confirms your Apache web server is running

If you see this page it means your web server is up and running, because the page you're looking at was loaded from it! Notice that the URL in your browser's address bar starts with `http://localhost/` (some modern browsers will hide the protocol, "`http://`"); **localhost** is a special hostname that always points to your own computer. Throughout this book, whenever you want to load a web page from your own web server, you'll use a URL that starts with `http://localhost/`.

When you're done working with the XAMPP Control Panel, shut it down by clicking the **Exit** button. Alternatively, you can just close the window, which will leave the XAMPP icon in the Windows System Tray (if you have configured it to remain

⁵ <http://www.apachefriends.org/en/faq-xampp-windows.html#nostart>

visible). Clicking the icon will promptly launch the XAMPP Control Panel again when you need it.



XAMPP Control Panel Leaves the Lights On

When you exit the XAMPP Control Panel, the Apache and MySQL servers will keep running on your system. If you've finished coding for the day, I'd advise you to click the **Stop** button for each of these servers to shut them down before you quit the XAMPP Control Panel. There's no sense slowing down those Facebook games you play in the evening by running unnecessary servers!

Set the MySQL Root Password in XAMPP

Once you've set up your Windows computer with the proper servers, you now need to assign a **root password** for MySQL in XAMPP.

MySQL only allows authorized users to view and manipulate the information stored in its databases, so you'll need to tell MySQL who's authorized and who isn't. When MySQL is first installed, it's configured with a user named "root" that has access to do most tasks without entering a password. Therefore, your first task should be to assign a password to the root user so that unauthorized users are prohibited from tampering with your databases.



Why bother?

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

XAMPP makes it easy to resolve this and other configuration security issues with your new servers. With the Apache and MySQL servers running, open this address in your web browser: <http://localhost/security/>. Alternatively, you can click the **Security** link in the menu on the XAMPP administration page.

This page will list any security issues that XAMPP can identify with your current server configuration. Among them, you should see "The MySQL admin user root

has NO password.” Scroll down past the table and click the link that will fix the problems listed.

The very first section of the resulting form will prompt you to set a MySQL root user password. Go ahead and set one you’ll remember. Leave the **PhpMyAdmin authentication** (sic) set to **cookie**, and use the option to save the password to a file if you think you might forget it (but beware that the password will be saved where a person using your computer could find it). Click the **Password changing** button to change your password, then stop and start your MySQL server using the XAMPP Control Panel.

Seriously, don’t forget this password. It’s a pain to change it if you do, but I’ll show you how in Chapter 10. Here’s a spot for you to record your MySQL root password in case you need to:



My MySQL Root Password (Windows)

root user password: _____



XAMPP Directory Protection

XAMPP’s security page will also warn you that your web pages are accessible to anyone on your network. While this is technically true, I’m not too worried if a co-worker or family member could stumble on my work-in-progress website; furthermore, most home and office network configurations will prevent people outside your network from accessing the web server running on your computer.

That said, if you want to follow XAMPP’s advice to set a username and password that will be required to view pages on your web server, feel free to set one.

Mac OS X Installation

In this section, I’ll show you how to start running a PHP-and-MySQL-equipped web server on a Mac computer running Mac OS X version 10.5 (Leopard). If you’re not using a Mac, you can safely skip this section.

Mac OS X distinguishes itself by being the only consumer OS to install both Apache and PHP as components of every standard installation. (For that matter, it also comes

12 PHP & MySQL: Novice to Ninja

with Ruby, Python, and Perl—all of which are popular web programming languages.) That said, they take a few tweaks to switch on, and you will need a MySQL database server as well. The simplest way to handle it is to ignore the built-in software and install everything you need in a convenient, all-in-one package.

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



The Do-it-yourself Option

In past editions of this book, I recommended that you set up the built-in versions of Apache and PHP that come with Mac OS X, and install MySQL using its official installation package. This is a good practice for beginners, I argued, because it gives you a strong sense of how these pieces all fit together.

Unfortunately, this meant that many readers spent their first few hours in “PHP Land” wrestling their way through a protracted sequence of detailed installation instructions. Worse still, sometimes the finer points of these became outdated due to some subtle change to one of the software packages.

Nowadays, I strongly believe that the best way to learn PHP and MySQL is to start *using* them right away. The quicker and more hassle-free the installation process, the better. That’s why I ask you to use MAMP in this edition. There’s also every chance you’re just dabbling in this stuff, so why spend time tweaking the innards of your operating system when you can leave them safely set to the factory defaults?

Nevertheless, if you’re a die-hard do-it-yourselfer, a tech-savvy power user, or if you simply reach the end of this book and wonder how the pros do it, I’ve included a detailed set of installation instructions for the individual packages in Appendix A. Feel free to follow them instead of the instructions in this section if you’re that way inclined.

1. Download the latest version from the MAMP website⁶ (you want the free MAMP, not the commercial MAMP PRO). After downloading the file (as of this writing, MAMP 2.0.5 is about 116MB in size), double-click it to unzip the installer (**MAMP.pkg**). Then double-click it to launch the MAMP Installer, which is shown in Figure 1.8.

⁶ <http://www.mamp.info>

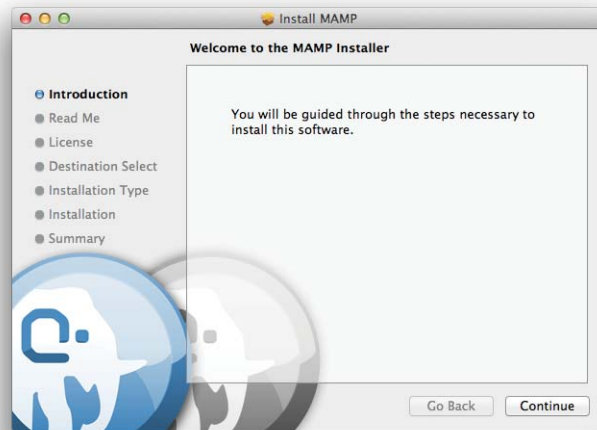


Figure 1.8. The MAMP package



Look Out Below!

The next step is a tricky one. Make sure you read on first before clicking blindly through the installer!

2. During the installation, you'll be prompted to choose whether or not to perform a standard installation. At this step, instead of clicking the **Install** button, click **Customize**. This will give you the opportunity to deselect **MAMP PRO** (which the installer will otherwise sneakily install in the hopes that you'll decide to buy it after all). This is especially important because the free MAMP will display a worrying warning message at startup if MAMP PRO is installed.



Miss this step?

If you missed this step and allowed the installer to put MAMP PRO on your system, it's easy enough to remove.

Open your **Applications** folder, double-click on the new **MAMP PRO** folder, and double-click to run the **MAMP PRO Uninstaller**. Click each checkbox in the **Uninstaller** window. Once they're all checked, click **Uninstall**. Quit the Uninstaller.

14 PHP & MySQL: Novice to Ninja

Browse to your **Applications** folder and find the new **MAMP** folder there. Open it, and double-click the **MAMP** icon inside to launch MAMP. As MAMP starts up, the following will happen. First, the MAMP window shown in Figure 1.9 will appear. The two status indicators will switch from red to green as the built-in Apache and MySQL servers start up. Next, MAMP will open your default web browser and load the MAMP welcome page, shown in Figure 1.10.



Figure 1.9. The MAMP window

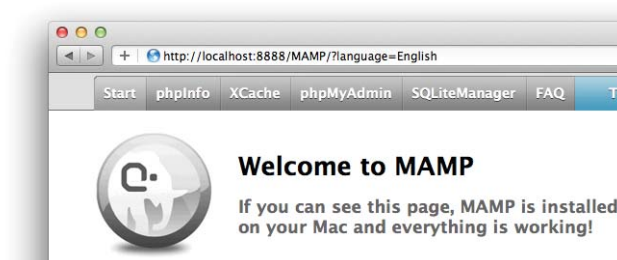


Figure 1.10. The MAMP welcome page confirms Apache, PHP, and MySQL are up and running

If you see this page it means your web server is up and running, because the page you're looking at was loaded from it! Notice that the URL in your browser's address bar starts with `http://localhost:8888/` (some modern browsers will hide the protocol,

“http://”); **localhost** is a special hostname that always points to your own computer. The “8888” is the **port number** that the browser is using to connect to your computer.

Every server running on a computer listens on a unique port number. Usually, websites are hosted on port 80, and browsers use that to connect when no port number is specified by the URL. By default, MAMP comes configured so that Apache will listen on port 8888 and MySQL will listen on port 8889. This ensures that MAMP will work even if your Mac already has a web server installed and listening on port 80, or a MySQL server listening on port 3306 (the standard MySQL server port).⁷

The code and instructions in the rest of this book will assume your web server is running on port 80 and your MySQL server is on port 3306. Now would be a good time to see if MAMP will run happily using these standard port numbers. Here’s how:

1. In the MAMP window, click **Stop Servers**. Wait for the indicators to turn red.
2. Click the **Preferences...** button and navigate to the **Ports** tab.
3. Click the **Set to default Apache and MySQL ports** button so that Apache will use port 80 and MySQL will use port 3306. Click **OK**.
4. Click **Start Servers**. MAMP will prompt you to enter your password, because running a server on an “official” Internet port number like 80 requires administrator privileges.

If both indicators turn green, click the **Open start page** button again, and verify that the MAMP welcome page shows up this time with a URL starting with `http://localhost/` (no port number). If so, you’re in good shape!

If one or both indicators don’t turn red in step 1, or if the welcome page fails to load correctly, in all likelihood you have yourself a port conflict. Somewhere on your Mac is another web or MySQL server that’s already using one or both of those ports. One place to check is the **Sharing** icon in System Preferences. If **Web Sharing** is en-

⁷ Of course, there are no guarantees that another application won’t be using port 8888 or 8889 on your system! I’ve had trouble with Playback by Yazsoft (an application for streaming media to game consoles like the Xbox 360 and PlayStation 3), which uses port 8888 when it is running. If in doubt, try a different port number!

abled, Mac OS X's built-in Apache server is running (normally on port 80). Another option is to try shutting down various applications. Under some conditions, Skype for Mac has prevented MAMP's MySQL server from launching for me, for example.

If, in the end, you're only able to make MAMP run happily on its default port numbers (8888 and 8889), go ahead and use them. Whenever this book mentions a URL starting with `http://localhost/`, you'll have to add the port number (`http://localhost:8888/`), and when the time comes to connect to MySQL, I'll tell you how to specify a nonstandard port number.

One last change to make to the default MAMP configuration is to switch on PHP error display. By default, when you make a serious mistake in your PHP code (and believe me, we all make plenty!), MAMP's Apache server will produce a blank web page. As a developer needing to figure out what you typed wrong, that's rather unhelpful; I'd much prefer to see a detailed error message in my browser window.

The reason why MAMP comes with the error display switched off is so that if you decide to host a publicly accessible website using it, visitors to the site won't see embarrassing error messages when you make a mistake. What's embarrassing on a public website, however, is practically *essential* in the development stage.

To switch on PHP error display, open the **MAMP** folder in your Mac's **Applications** folder. From there, drill down into **bin/php/**. This **php** folder will contain a subfolder for each version of PHP that comes with MAMP. You can double-check in MAMP's Preferences to be sure, but it's probably configured to run the most recent version, so open that folder (it's **php5.3.6** in my copy of MAMP 2.0.5), and then open the **conf** subfolder. Open the **php.ini** file in your favorite text editor (TextEdit will work fine), and look for these lines:

```
; Print out errors (as a part of the output). For production web
➤ sites,
; you're strongly encouraged to turn this feature off, and use
➤ error logging
; instead (see below). Keeping display_errors enabled on a
➤ production web site
; may reveal security information to end users, such as file paths
➤ on your Web
; server, your database schema or other information.
display_errors = Off
```


Change the **Off** in that last line to **On** and save the file. Now click **Stop Servers**, then **Start Servers** in MAMP to restart Apache with the new configuration. That's it—PHP will now display helpful (if a little soul-crushing) error messages.

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

Set the MySQL Root Password in MAMP

Once MAMP is up and running on your Mac with the relevant servers, your very next action should be to assign a **root password** for MySQL.

MySQL only allows authorized users to view and manipulate the information stored in its databases, so you'll need to tell MySQL who's authorized and who's not. When MAMP first installs MySQL, it's configured with a user named "root" that has access to perform most tasks. The password for this user is "root"—not exactly Fort Knox! Hence why your first task should be to assign a new password to the root user, preventing any tampering with your databases.



Why bother?

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

To set your MySQL root password, first make sure MAMP and its servers are running. Then open the Mac OS X **Terminal** application (found in the **Utilities** folder in the **Applications** folder) and type these commands (hitting **Enter** after each one):

1. `cd /Applications/MAMP/Library/bin/`

This navigates to the **Library/bin/** subfolder of your MAMP installation, which is where the Terminal utility programs are kept.

2. `./mysqladmin -u root -p password "newpassword"`

18 PHP & MySQL: Novice to Ninja

Replace *newpassword* with the new password you want to assign to your MySQL root user.

When you hit **Enter** you'll be prompted to enter the current password: **root**.

3. Quit Terminal.

Your password is now set, but this creates a new problem: MAMP itself needs unrestricted access to your MySQL server so that it can control it. If you click the **Open start page** button in MAMP at this point, you'll receive an error message: "Error: Could not connect to MySQL server!" Obviously, we need to tell MAMP what our new MySQL root password is.

You must edit several files in the MAMP folder to make it work again. You can open each of these files in TextEdit, or whichever text editor you prefer to use.



Editing PHP Scripts in Mac OS X with TextEdit

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

/Applications/MAMP/bin/mamp/index.php

Find the line that looks like this:

```
$link = @mysql_connect('/:Applications/MAMP/tmp/mysql/mysql.sock',  
    'root', 'root');
```

Replace the second 'root' with your new MySQL root password (that is, 'newpassword').

/Applications/MAMP/bin/phpMyAdmin/config.inc.php

This is a large file, so you may need to use your text editor's Find feature to locate these lines:

```
$cfg['Servers'][$i]['user']      = 'root';  
➤ // MySQL user  
$cfg['Servers'][$i]['password'] = 'root';
```

```
➡          // MySQL password (only needed
➡ // with 'config' auth_type)
```

Again, replace the second 'root' with your new MySQL root password (that's 'newpassword').

```
/Applications/MAMP/bin/checkMysql.sh
/Applications/MAMP/bin/quickCheckMysqlUpgrade.sh
/Applications/MAMP/bin/repairMysql.sh
/Applications/MAMP/bin/stopMysql.sh
/Applications/MAMP/bin/upgradeMysql.sh
```

The contents of each of these little files starts out looking a little like this (this is **checkMysql.sh**):

```
# /bin/sh
/Applications/MAMP/Library/bin/mysqlcheck --all-databases --check
➡ --check-upgrade -u root -proot
➡ --socket=/Applications/MAMP/tmp/mysql/mysql.sock
```

See that -proot? The p stands for “password” and the rest *is* the password. Change it to your new password (-*newpassword*).

Make the same change to each of these five files.

With all those changes made and saved, MAMP should work normally again, with your MySQL server nice and secure from outside intrusion!

Oh, and don't forget this password. It's kind of a pain to change it if you do (I'll show you how in Chapter 10). Here's a spot for you to record your MySQL root password in case you need to.



My MySQL Root Password (Mac)

root user password: _____

Linux Installation

These days, most people who run Linux as their operating system of choice are tech-savvy enough to know how to install software like Apache, PHP, and MySQL. Indeed, they probably feel strongly about *how* they should be installed, which would doubtlessly clash with any instructions I'd provide here.

If this describes you, go ahead and install the most recent versions of Apache, PHP, and MySQL that you're comfortable installing, using whichever package manager or build process pushes your buttons. Nothing in this book is going to be so advanced that the minutiae of how you configure these packages will matter.

That said, just in case you're one of the rare Linux users who could use some guidance on installing, I've included a detailed set of instructions for Linux users in Appendix A.

What to Ask Your Web Host


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

First, you'll need to know how to transfer files to your web host. You'll be uploading PHP scripts to your host the same way you normally send the HTML files, CSS files, and images that make up a static website; so if you already know how to do that, there's no need to bother your host. If you're just starting with a new host, however, you'll have to be aware of what file transfer protocol it supports (FTP or SFTP), as well as knowing what username and password to use when connecting with your (S)FTP program. You also must know what directory to put files into so that they're accessible to web browsers.

In addition, you'll require a few details about the MySQL server your host has set up for you. It's important to know the host name to use in order to connect to it (possibly `localhost`), and your MySQL username and password, which may or may not be the same as your (S)FTP credentials. Your web host will probably have provided an empty database for you to use, which prevents you from interfering

with other users' databases who may share the same MySQL server with you. If they have provided this, you should establish the name of that database.

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



My Hosting Details

File transfer protocol (circle one):	<input type="checkbox"/> FTP <input type="checkbox"/> SFTP
(S)FTP host name:	<input type="text"/>
(S)FTP username:	<input type="text"/>
(S)FTP password:	<input type="text"/>
MySQL host name:	<input type="text"/>
MySQL username:	<input type="text"/>
MySQL password:	<input type="text"/>
MySQL database name:	<input type="text"/>

Your First PHP Script

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

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

chapter1/today.php

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Today's Date</title>
  </head>
```

```
<body>
  <p>Today's date (according to this web server) is
    <?php

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

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



It's a Letter, Not a Number

The most important line of the code is this one:

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

Unfortunately, it's also the one most people type wrong when reading this book. See the character before the comma? It's not the number one (1), it's a lowercase L (l).



Editing PHP Scripts in Windows with Notepad

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



Editing PHP Scripts in Mac OS X with TextEdit

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

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



Try a Free IDE!

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

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

If you'd prefer not to type out all the code, you can download this file—along with the rest of the code in this book—from the code archive. See the Preface for details on how to download the code archive.

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



Where's my server's web root directory?

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

For the Apache server built into XAMPP, the web root directory is the **htdocs** directory within your XAMPP installation directory. You can reach it simply by choosing from the Start menu: **All Programs > Apache Friends > XAMPP > XAMPP htdocs folder**.

If the Apache server you're using is built into Mac OS X, the web root directory is **/Library/WebServer/Documents**. It can be easily accessed by clicking the **Open Computer Website Folder...** button under **Web Sharing** in the **Sharing** preference panel in System Preferences.

The Apache server built into MAMP has a web root directory in the **htdocs** folder inside the MAMP folder (**/Applications/MAMP/htdocs**). If you prefer using another folder as your web root, you can change it on the **Apache** tab of the MAMP application's Preferences. This facility makes it especially easy to switch between multiple web development projects by pointing MAMP at different folders.

Open your web browser of choice, and type `http://localhost/today.php` (or `http://localhost:port/today.php` if Apache is configured to run on a port other than the default of 80) into the address bar to view the file you just created.⁸



You Must Type the URL

You might be used to previewing your web pages by double-clicking on them, or by using the **File > Open...** feature of your browser. These methods tell your browser to load the file directly from your computer's hard drive, so they won't work with PHP files.

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

Figure 1.11 shows what the web page generated by your first PHP script should look like.

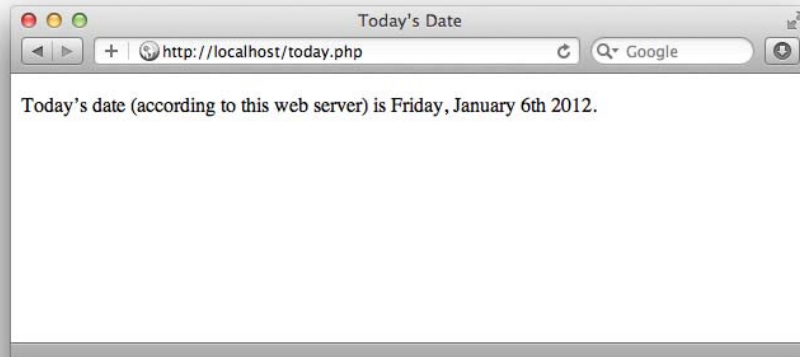


Figure 1.11. See your first PHP script in action!

Neat, huh? If you use the **View Source** feature in your browser, all you'll see is a regular HTML file with the date in it. The PHP code (everything between `<?php` and

⁸ If you installed Apache on Windows, you may have selected the option to run it on port 8080. If you're using MAMP, it's configured by default to run Apache on port 8888.

?> in the code above) was interpreted by the web server and converted to normal text before it was sent to your browser. The beauty of PHP, and other server-side scripting languages, is that the web browser can remain ignorant—the web server does all the work!

If you're worried that the code you typed made little sense to you, rest assured that you'll be up to speed on exactly how it works by the end of Chapter 3.

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

Full Toolbox, Dirty Hands

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

If the **today.php** script didn't work for you, drop by the SitePoint Forums⁹ and we'll be glad to help you figure out the problem.

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

⁹ <http://www.sitepoint.com/forums/>

Chapter 2

Introducing MySQL

In Chapter 1, we installed and set up two software programs: the Apache web server with PHP, and the MySQL database server. If you followed my recommendation, you would have set them up using an all-in-one package like XAMPP or MAMP, but don't let that diminish your sense of accomplishment!

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

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

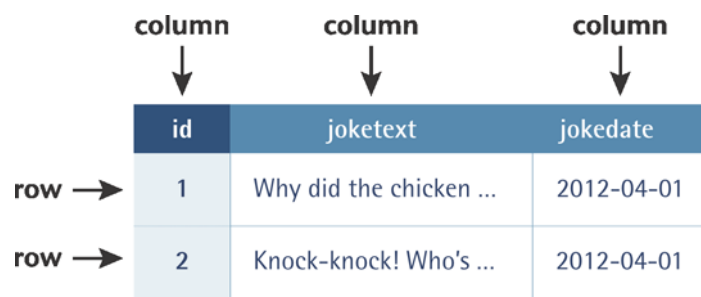
An Introduction to Databases

A database server (in our case, MySQL) is a program that can store large amounts of information in an organized format that's easily accessible through programming

languages like PHP. For example, you could tell PHP to look in the database for a list of jokes that you'd like to appear on your website.

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

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



The diagram shows a table with three columns and two rows. Above the table, three arrows labeled 'column' point down to the column headers: 'id', 'joketext', and 'jokedate'. To the left of the table, two arrows labeled 'row' point right to the first and second data rows. The table has a blue header row and two light blue data rows.

	column	column	column
	id	joketext	jokedate
row →	1	Why did the chicken ...	2012-04-01
row →	2	Knock-knock! Who's ...	2012-04-01

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

Notice that, in addition to columns for the joke text (`joketext`) and the date of the joke (`jokedate`), I've included a column named `id`. As a matter of good design, a database table should always provide a means by which we can identify each of its rows uniquely. Since it's possible that two identical jokes could be entered on the same date, we can't rely upon the `joketext` and `jokedate` columns to tell all the jokes apart. The function of the `id` column, therefore, is to assign a unique number

to each joke so that we have an easy way to refer to them and to keep track of which joke is which. We'll take a closer look at database design issues like this in Chapter 5.

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

Using phpMyAdmin to Run SQL Queries

Just as a web server is designed to respond to requests from a client (a web browser), the MySQL database server responds to requests from **client programs**. Later in this book, we'll write our own MySQL client programs in the form of PHP scripts, but for now we can use a client program that comes bundled with both XAMPP and MAMP: phpMyAdmin.

phpMyAdmin is itself a sophisticated web application written in PHP. Besides being included in XAMPP and MAMP, phpMyAdmin is provided by most commercial web hosts who offer PHP and MySQL as a tool for developers to manage their websites' MySQL databases. Much like PHP and MySQL, phpMyAdmin's ubiquity makes it an attractive tool for beginners to learn and use.

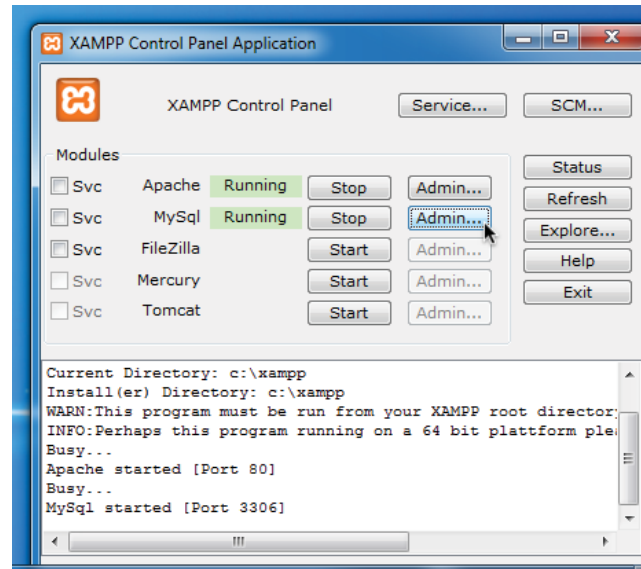


Don't have phpMyAdmin?

If you opted to follow the manual setup instructions in Appendix A rather than use the all-in-one package offered by XAMPP or MAMP to set up your web server, you probably don't have phpMyAdmin installed on your server. The good news is that you can download and install it from the phpMyAdmin website,¹ where instructions are provided.

If you're using XAMPP on Windows, you can access phpMyAdmin by clicking the **Admin...** button next to **MySQL** (sic) in the XAMPP Control Panel window, as shown in Figure 2.2.

¹ <http://www.phpmyadmin.net/>

Figure 2.2. Click the **Admin...** button to open phpMyAdmin

To access phpMyAdmin using MAMP on Mac OS X, click the **Open start page** button in the MAMP window. Then click the **phpMyAdmin** tab at the top of the screen, as shown in Figure 2.3.

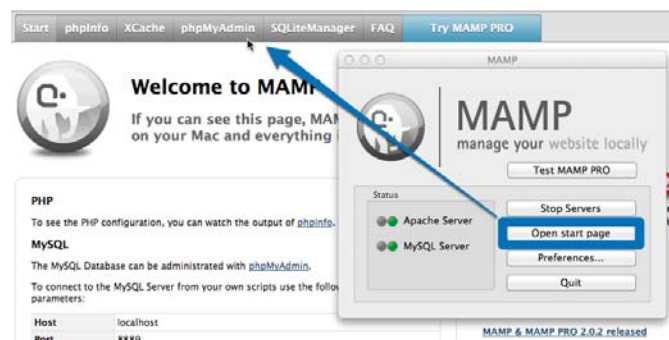


Figure 2.3. You can access phpMyAdmin from MAMP's start page

Either way, you should now have phpMyAdmin open in your default web browser, which should look like Figure 2.4. As of this writing, XAMPP includes the more recent (and better-looking) version 3.4 of phpMyAdmin, so I'll be showing screenshots of that. If you're using the older version 3.3, it won't look quite as nice, but it should work just the same.

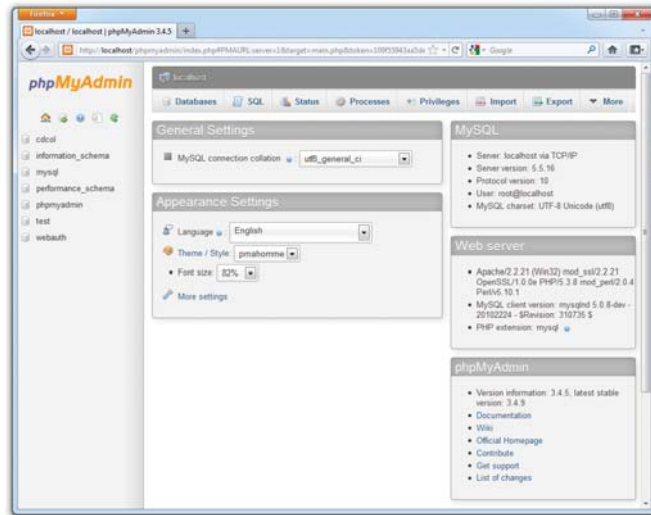


Figure 2.4. If you can see this, you have phpMyAdmin

If you go clicking around phpMyAdmin, you'll discover all the tools you need to manage every aspect of your MySQL server and the data it contains. For now, I'm going to ignore all of those features and focus on a particular one: the SQL query window.

See the row of buttons just beneath the phpMyAdmin logo? Clicking the second icon, indicated in Figure 2.5, opens the SQL query window shown in Figure 2.6.

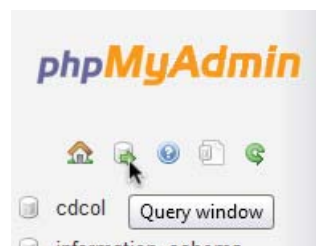


Figure 2.5. Click the second button ...

32 PHP & MySQL: Novice to Ninja

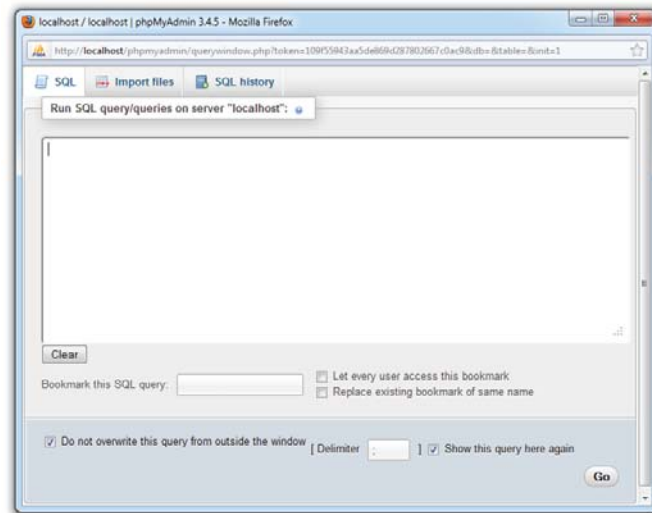


Figure 2.6. ... to open the SQL query window

Into that big, empty text box you can type commands to ask your database server questions or make it perform tasks. Let's try a few simple commands to take a look around your MySQL server.

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

Type this command into the SQL query window, then click **Go**:

SHOW DATABASES

You might think at first that nothing has happened, but you should now see the results in the main phpMyAdmin window, as shown in Figure 2.7.

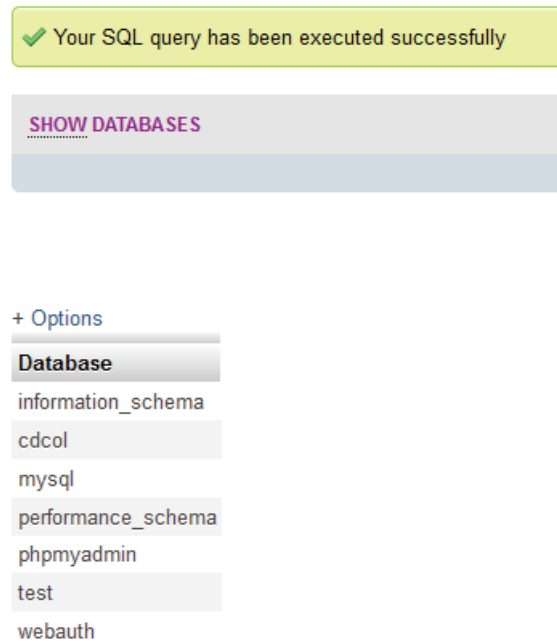


Figure 2.7. The query results are displayed in the main phpMyAdmin window

Your list of databases might be as long as the one shown in Figure 2.7, or if you're running MAMP it may only contain two critical databases. XAMPP uses additional databases to store configuration of its own, whereas MAMP is designed to avoid cluttering up your MySQL server with its own data. Either way, you will have databases named `information_schema` and `mysql`.

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

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

A third database, named `test`, is a sample database that's included with MySQL out of the box (again, MAMP does away with this database so you can start clean). If you see it in the list, you can delete the `test` database because you'll be creating your own database in a moment.

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

```
DROP DATABASE test
```

If you type this command into the SQL query window and click **Go**, phpMyAdmin will probably display an error message: **"DROP DATABASE" statements are disabled**. This message indicates that a safety feature built into phpMyAdmin is preventing you from running dangerous-looking queries like this one.

If you want to be able to drop databases (and this is probably a good ability to have, given the amount of experimentation I’m going to encourage you to do in this book), there is a way to do so tucked away in phpMyAdmin. In the main phpMyAdmin window, click the **Databases** tab (the leftmost tab at the top of the main window area). You’ll be presented with a list of databases on the server, with a checkbox next to each. Check the one you want to delete (**test** in this case); then click the **Drop** button at the bottom-right of the list as shown in Figure 2.8.

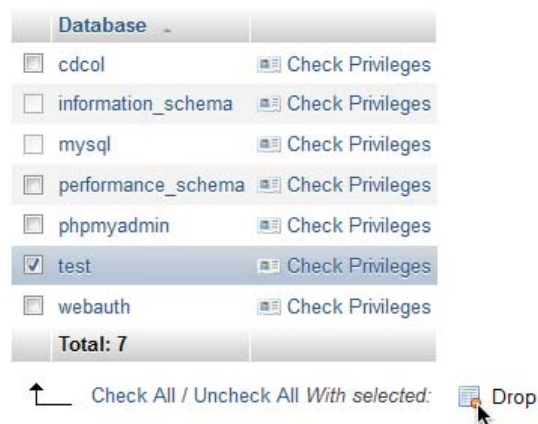


Figure 2.8. The ability to drop a database in phpMyAdmin is well hidden

phpMyAdmin presents one last prompt to make sure you mean to obliterate the database. If you confirm this, MySQL will obediently delete the database, and phpMyAdmin will display a message to verify it was successful.

Note that there are *other* potentially hazardous commands you can send to MySQL in addition to **DROP DATABASE**, but phpMyAdmin won’t always protect you if you

make a mistake. You have to be very careful to type your commands correctly in the SQL query window, otherwise you can destroy your entire database—along with all the information it contains—with a single command!

Structured Query Language

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

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



Learn SQL in Depth

In this book, I'll teach you the essentials of SQL that every PHP developer needs to know. If you decide to make a career out of building database driven websites, it pays to know some of the more advanced details of SQL, especially when it comes to making your sites run as quickly and smoothly as possible. To dive deeper into SQL, I highly recommend the book *Simply SQL*² by Rudy Limeback.

Creating a Database

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

It's just as easy to create a database as it is to delete one. Open the SQL query window again, and type this command:

```
CREATE DATABASE ijdbc
```

² <http://www.sitepoint.com/books/sql1/>

I chose to name the database `ijdb`, for Internet Joke Database,³ because that fits with the example I gave at the beginning of this chapter: a website that displays a database of jokes. Feel free to give the database any name you like, though.



Case Sensitivity in SQL Queries

Most MySQL commands are not case-sensitive, which means you can type `CREATE DATABASE`, `create database`, or even `CrEaTe DaTaBaSe`, and it will know what you mean. Database names and table names, however, are case-sensitive when the MySQL server is running on an operating system with a case-sensitive file system (such as Linux or Mac OS X, depending on your system configuration).

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

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

Now that you have a database, you need to tell phpMyAdmin that you want to use it. You've probably noticed by now that the left-hand sidebar in the main phpMyAdmin window contains a list of all the databases on your MySQL server. When you clicked **Go** to run your `CREATE DATABASE` command (you *did* click **Go**, didn't you?), this sidebar updated to show your new database's name in a drop-down menu, as shown in Figure 2.9.

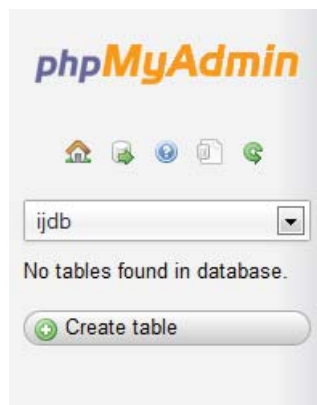


Figure 2.9. phpMyAdmin autoselects your new database for you

³ With a tip of the hat to the Internet Movie Database [<http://www.imdb.com>].

It's nice of phpMyAdmin to autoselect your new database for you, but you'll need to know how to select it yourself. Click the home button (the first in the row of icons beneath the phpMyAdmin logo) to go back to the home page of phpMyAdmin. The sidebar will once again display a list of all databases on your server.

To select a database to work with, just click its name in the sidebar. With your database selected, click the **Query window** button again to open a new SQL query window. This query window is slightly different from the last one: the caption for the text box now says **Run SQL query/queries on database ijdb**. Commands typed into this query window will run on your new database, instead of your MySQL server as a whole.



Figure 2.10. You must open a new query window to work with this database

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

Creating a Table

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

```
CREATE TABLE table_name (
  column1Name column1Type column1Details,
  column2Name column2Type column2Details,
  :
) DEFAULT CHARACTER SET charset ENGINE=InnoDB
```

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

```
CREATE TABLE joke (  
  id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,  
  joketext TEXT,  
  jokedate DATE NOT NULL  
) DEFAULT CHARACTER SET utf8 ENGINE=InnoDB
```

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

CREATE TABLE joke (

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

id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,

This second line says that we want a column called `id` that contains an integer (`INT`); that is, a whole number. The rest of this line deals with special details for the column:

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

joketext TEXT,

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

jokedate DATE NOT NULL

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

) DEFAULT CHARACTER SET utf8

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

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

ENGINE=InnoDB

This tells MySQL which **storage engine** to use to create this table.

Think of a storage engine as a file format. When building a website, you'll typically choose to use the JPEG format for the photos on your site, but stick with the PNG format for the images that make up your site design. Both formats are supported by browsers, but they each have strengths and weaknesses. Likewise, MySQL supports multiple formats for database tables.

The InnoDB format is by far the best choice for website databases like the one we'll build in this book. The older MyISAM format is the default, however, so we must tell MySQL that we want it to create an InnoDB table.

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

Now, if you type the above command correctly and click **Go**, the main phpMyAdmin window will confirm that the query executed successfully, and your first table will be created. If you made a typing mistake, phpMyAdmin will tell you there was a problem with the query you typed, and will try to indicate where it had trouble understanding what you meant.

Let's have a look at your new table to make sure it was created properly. Type the following command into the SQL query window, and click **Go**:

```
SHOW TABLES
```

phpMyAdmin should display the output shown in Figure 2.11.



Figure 2.11. phpMyAdmin lists the tables in the currently selected database

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

DESCRIBE joke

As you can see in Figure 2.12, there are three columns (or fields) in this table, which appear as the three rows in this table of results. The details are a little cryptic, but if you look at them closely, you should be able to figure out what they mean. It's nothing to be worried about, though. You have better things to do, like adding some jokes to your table!

Field	Type	Null	Key	Default	Extra
id	int(11)	NO	PRI	NULL	auto_increment
joketext	text	YES		NULL	
jokedate	date	NO		NULL	

Figure 2.12. phpMyAdmin lists the columns in the `joke` table as rows

We need to look at just one more task before we do that, though: deleting a table. This task is as frighteningly easy as deleting a database with a `DROP DATABASE` command—except that phpMyAdmin won't protect you here. *Don't* run this command with your `joke` table, unless you actually do want to be rid of it! If you really want to try it, be prepared to re-create your `joke` table from scratch:

DROP TABLE *tableName*

Inserting Data into a Table

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

```
INSERT INTO tableName SET
  column1Name = column1Value,
  column2Name = column2Value,
  :
```

```
INSERT INTO tableName
  (column1Name, column2Name, ...)
VALUES (column1Value, column2Value, ...)
```

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

```
INSERT INTO joke SET
joketext = "Why did the chicken cross the road? To get to the other
↳ side!",
jokedate = "2012-04-01"
```

```
INSERT INTO joke
(joketext, jokedate) VALUES (
  "Why did the chicken cross the road? To get to the other side!",
  "2012-04-01")
```

Note that in both forms of the `INSERT` command, the order in which you list the columns must match the order in which you list the values. Otherwise, the order of the columns isn't important. Go ahead and swap the order of the column and value pairs and try the query.

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

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

```
INSERT INTO joke SET
joketext = 'Why did the chicken cross the road? To get to the other
➤ side!',
jokedate = '2012-04-01'
```

You might be wondering what happens when there are quotes used within the joke's text. Well, if the text contains single quotes, you would surround it with double quotes. Conversely, if the text contains double quotes, surround it with single quotes.

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

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

```
INSERT INTO joke
(joketext, jokedate) VALUES (
'Knock-knock! Who\'s there? Boo! "Boo" who? Don\'t cry; it\'s only a
➤ joke!',
"2012-04-01")
```

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

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

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

Viewing Stored Data

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

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

```
SELECT * FROM joke
```

Read aloud, this command says “select everything from `joke`.” If you try this command, your results will resemble Figure 2.13.

id	joketext	jokedate
1	Why did the chicken cross the road? To get to the ...	2012-04-01

Figure 2.13. phpMyAdmin lists the full contents of the `joke` table

If you were doing serious work on such a database, you might be tempted to stop and read all the hilarious jokes in the database at this point. To save yourself the distraction, you might want to tell MySQL to omit the `joketext` column. The command for doing this is as follows:

```
SELECT id, jokedate FROM joke
```

This time, instead of telling it to “select everything,” we told it precisely which columns we wanted to see. The result should look like Figure 2.14.

id	jokedate
1	2012-04-01

Figure 2.14. You can select only what you need

What if we'd like to see *some* of the joke text? As well as being able to name specific columns that we want the `SELECT` command to show us, we can use functions to

modify each column's display. One function, called `LEFT`, enables us to tell MySQL to display a column's contents up to a specified number of characters. For example, let's say we wanted to see only the first 20 characters of the `joketext` column. Here's the command we'd use:

```
SELECT id, LEFT(joketext, 20), jokedate FROM joke
```

The results are shown in Figure 2.15.

id	LEFT(joketext, 20)	jokedate
1	Why did the chicken	2012-04-01

Figure 2.15. The `LEFT` function trims the text to a specified length

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

```
SELECT COUNT(*) FROM joke
```

As you can see in Figure 2.16, you have just one joke in your table.

COUNT(*)
1

Figure 2.16. The `COUNT` function counts the rows

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

```
SELECT COUNT(*) FROM joke WHERE jokedate >= "2012-01-01"
```

This query will count the number of jokes that have dates greater than or equal to January 1, 2012. In the case of dates, "greater than or equal to" means "on or after." Another variation on this theme lets you search for entries that contain a certain piece of text. Check out this query:

```
SELECT joketext FROM joke WHERE joketext LIKE "%chicken%"
```

This query displays the full text of all jokes containing the text “chicken” in their `joketext` column. The `LIKE` keyword tells MySQL that the named column must match the given pattern.⁴ In this case, the pattern we’ve used is `"%chicken%"`. The `%` signs indicate that the text “chicken” may be preceded and/or followed by any string of text.

Conditions may also be combined in the `WHERE` clause to further restrict results. For example, to display knock-knock jokes from April 2012 only, you could use the following query:

```
SELECT joketext FROM joke WHERE  
joketext LIKE "%knock%" AND  
jokedate >= "2012-04-01" AND  
jokedate < "2012-05-01"
```

Enter a few more jokes into the table (for example, the “Knock-Knock” joke mentioned earlier) and experiment with `SELECT` queries (for ideas, see Chapter 4).

You can do a lot with the `SELECT` command, so I’d encourage you to become quite familiar with it. We’ll look at some of its more advanced features later, when we need them.

Modifying Stored Data

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

```
UPDATE tableName SET  
  colName = newValue, ...  
WHERE conditions
```

⁴ In case you were curious, `LIKE` is case-insensitive, so this pattern will also match a joke that contains “Chicken,” or even “FuNkYcHiCkEn.”

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

```
UPDATE joke SET jokedate = "2013-04-01" WHERE id = "1"
```

Here's where that `id` column comes in handy, enabling you to easily single out a joke for changes. The `WHERE` clause used here works just as it did in the `SELECT` command. This next command, for example, changes the date of all entries that contain the word "chicken":

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

Deleting Stored Data

Deleting entries in SQL is dangerously easy, which you've probably noticed is a recurring theme. Here's the command syntax:

```
DELETE FROM tableName WHERE conditions
```

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

```
DELETE FROM joke WHERE joketext LIKE "%chicken%"
```



Careful with That Enter Key!

Believe it or not, the `WHERE` clause in the `DELETE` command is optional. Consequently, you should be very careful when typing this command! If you leave the `WHERE` clause out, the `DELETE` command will then apply to *all entries in the table*.

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

```
DELETE FROM joke
```

Scary, huh?

Let PHP Do the Typing

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

At this stage, you might be thinking that databases seem a little cumbersome. SQL can be tricky to type, as its commands tend to be long and verbose compared to other computer languages. You're probably dreading the thought of typing in a complete library of jokes in the form of `INSERT` commands.

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

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

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

In the meantime, we've accomplished our objective, and you can comfortably interact with MySQL using the phpMyAdmin query window. In Chapter 3, the fun continues as we delve into the PHP language, and use it to create several dynamically generated web pages.

If you like, you can practice with MySQL a little before you move on by creating a decent-sized joke table (for our purposes, five should be enough). This library of jokes will come in handy when you reach Chapter 4.

Chapter 3

Introducing PHP

PHP is a **server-side language**. This concept may be a little difficult to grasp, especially if you've only ever designed websites using client-side languages like HTML, CSS, and JavaScript.

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

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

Once the web server has executed the PHP code embedded in a web page, the result takes the place of the PHP code in the page. All the browser sees is standard HTML

code when it receives the page, hence the name “server-side language.” Let’s look back at the **today.php** example presented in Chapter 1:

chapter3/today.php

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Today's Date</title>
  </head>
  <body>
    <p>Today's date (according to this web server) is
      <?php

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

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

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

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Today's Date</title>
  </head>
  <body>
    <p>Today's date (according to this web server) is
      Sunday, April 1st 2012.
    </p>
  </body>
</html>
```

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

No browser compatibility issues

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

Access to server-side resources

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

Reduced load on the client

JavaScript can delay the display of a web page significantly (especially on mobile devices!), as the browser must run the script before it can display the web page. With server-side code this burden is passed to the web server, which you can make as beefy as your application requires (and your wallet can afford).

Basic Syntax and Statements

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

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

This is a typical PHP statement:

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

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

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

chapter3/echo.php

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Today&rsquo;s Date</title>
  </head>
  <body>
    <p><?php echo 'This is a <strong>test</strong>!'; ?></p>
  </body>
</html>
```

If you place this file on your web server and then request it using a web browser, your browser will receive this HTML code:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Today&rsquo;s Date</title>
  </head>
  <body>
    <p>This is a <strong>test</strong>!</p>
  </body>
</html>
```

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

chapter3/today.php (excerpt)

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

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

When you invoke a function in PHP—that is, ask it to do its job—you’re said to be **calling** that function. Most functions **return** a value when they’re called; PHP then behaves as if you’d actually just typed that returned value instead in your code. In this case, our `echo` statement contains a call to the `date` function, which returns the current date as a string of text (the format of which is specified by the text string in the function call). The `echo` statement therefore outputs the value returned by the function call.

You may wonder why we need to surround the string of text with both parentheses `((...))` and single quotes `('...')`. As in SQL, quotes are used in PHP to mark the beginning and end of strings of text, so it makes sense for them to be there. The parentheses serve two purposes. First, they indicate that `date` is a function that you want to call. Second, they mark the beginning and end of a list of **arguments** that you wish to provide, in order to tell the function what you want it to do.¹ In the case of the `date` function, you need to provide a string of text that describes the format in which you want the date to appear.² Later on, we’ll look at functions that take more than one argument, and we’ll separate those arguments with commas. We’ll also consider functions that take no arguments at all. These functions will still need the parentheses, even though there will be nothing to type between them.

Variables, Operators, and Comments

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

¹ I’m fairly sure they’re called arguments because that’s what often happens when you try to tell someone what to do.

² A full reference is available in the online documentation for the `date` function [<http://www.php.net/date/>].

```
$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 store different types of values over its lifetime. The following statement, if you were to type it after the aforementioned statement, assigns a new value to the existing `$testVariable`. Where it used to contain a number, it now contains a string of text:

```
$testVariable = 'Three';
```

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

```
$testVariable = 1 + 1; // assigns a value of 2
$testVariable = 1 - 1; // assigns a value of 0
$testVariable = 2 * 2; // assigns a value of 4
$testVariable = 2 / 2; // assigns a value of 1
```

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

Each arithmetic line ends with a **comment**. Comments enable you to describe what your code is doing. They insert explanatory text into your code—text that the PHP interpreter will ignore. Comments begin with `//` and they finish at the end of the same line. If you want a comment to span several lines, start it with `/*`, and end it with `*/`. The PHP interpreter will ignore everything between these two delimiters. I'll be using comments throughout the rest of this book to help explain some of the code I present.

Returning to the operators, one that sticks strings of text together is called the **string concatenation operator**:

```
$testVariable = 'Hi ' . 'there!'; // assigns a value of 'Hi there!'
```

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

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

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

Arrays

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

The simplest way to create an array in PHP is to use the array command:

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

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

To access a value stored in an array, you need to know its **index**. Typically, arrays use numbers as indices to point to the values they contain, starting with zero. That is, the first value (or element) of an array has index 0, the second has index 1, the third has index 2, and so on. Therefore, the index of the *n*th element of an array is

$n-1$. Once you know the index of the value you're interested in, you can retrieve that value by placing that index in square brackets after the array variable name:

```
echo $myArray[0];      // outputs 'one'
echo $myArray[1];      // outputs '2'
echo $myArray[2];      // outputs '3'
```

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

```
$myArray[1] = 'two';    // assign a new value
$myArray[3] = 'four';   // create a new element
```

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

```
$myArray[] = 'the fifth element';
echo $myArray[4];       // outputs 'the fifth element'
```

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

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

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

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

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

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


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

User Interaction and Forms

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

Veterans of JavaScript tend to think of interactivity in terms of event listeners, which let you react directly to the actions of the user; for example, the movement of the cursor over a link on the page. Server-side scripting languages such as PHP have a more limited scope when it comes to support for user interaction. As PHP code is only activated when a request is made to the server, user interaction occurs solely in a back-and-forth fashion: the user sends requests to the server, and the server replies with dynamically generated pages.³

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

Passing Variables in Links

The simplest way to send information along with a page request is to use the **URL query string**. If you've ever seen a URL containing a question mark that follows the filename, you've witnessed this technique in use. For example, if you search for "SitePoint" on Google, it will take you to a URL that looks like this one to see the search results:

```
http://www.google.com/search?hl=en&q=SitePoint
```

³ To some extent, the rise of Ajax techniques in the JavaScript world in recent years has changed this. It's now possible for JavaScript code—responding to a user action such as mouse movement—to send a request to the web server, invoking a PHP script. For the purposes of this book, however, we'll stick to non-Ajax applications. If you'd like to learn all about Ajax, check out *jQuery: Novice to Ninja* [<http://www.sitepoint.com/books/ajax1/>] by Earle Castledine and Craig Sharkie.

See the question mark in the URL? See how the text that follows the question mark contains your search query (SitePoint)? That information is being sent along with the request for `http://www.google.com/search`.

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

chapter3/links1/name.html (excerpt)

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

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

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

chapter3/links1/name.php (excerpt)

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

Now, put these two files (**name.html** and **name.php**) onto your web server, and load the first file in your browser (the URL should be like `http://localhost/name.html`, or `http://localhost:8888/name.html` if your web server is running on a port other than 80). Click the link in that first page to request the PHP script. The resulting page should say "Welcome to our website, Kevin!", as shown in Figure 3.1.

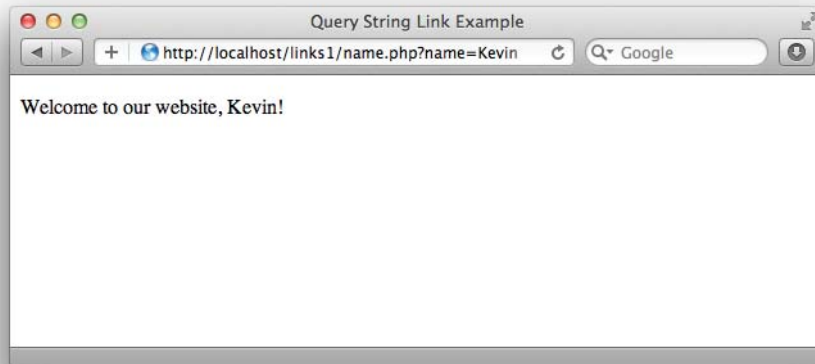


Figure 3.1. Greet users with a personalized welcome message

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

[chapter3/links1/name.php \(excerpt\)](#)

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

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

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

[chapter3/links1/name.php \(excerpt\)](#)

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

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

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

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

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

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

Edit the URL to insert a `` tag before the name, and a `` tag following the name:

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

Hit **Enter** to load this new URL, and note that the name in the page is now bold, as shown in Figure 3.2.⁴

⁴ You might notice that some browsers will automatically convert the `<` and `>` characters into URL escape sequences (`%3C` and `%3E`, respectively), but either way PHP will receive the same value.

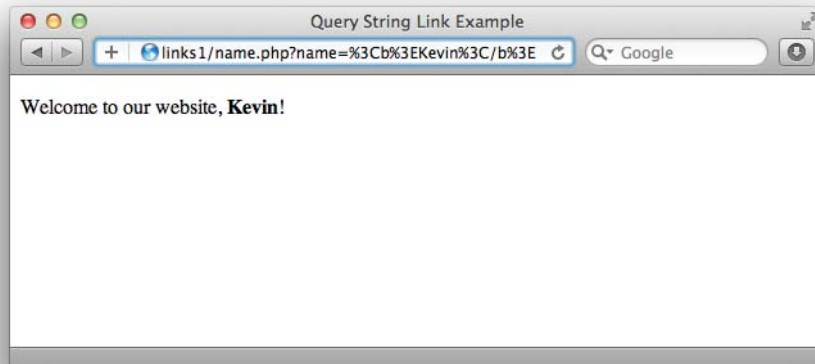


Figure 3.2. Easy exploitation will only embolden attackers!

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

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

So, how can we generate a page containing the user's name without opening it up to abuse by attackers? The solution is to treat the value supplied for the `$name` variable as plain text to be displayed on your page, rather than as HTML to be in-

cluded in the page's code. This is a subtle distinction, so let me show you what I mean.

Open up your **name.php** file again and edit the PHP code it contains so that it looks like this:⁵

chapter3/links2/name.php (excerpt)

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

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

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

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

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

⁵ In the code archive for this book, you'll find the updated files in the **links2** subfolder.

The first argument is the `$name` variable (the text to be converted). The second argument is the PHP constant⁶ `ENT_QUOTES`, which tells `htmlspecialchars` to convert single and double quotes in addition to other special characters. The third parameter is the string `'UTF-8'`, which tells PHP what character encoding to use to interpret the text you give it.



The Perks and Pitfalls of UTF-8 with PHP

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

```
<meta charset="utf-8">
```

This tag tells the browser receiving this page that the HTML code of the page is encoded as UTF-8 text.⁷

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

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

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

⁶ A PHP constant is like a variable whose value you're unable to change. Unlike variables, constants don't start with a dollar sign. PHP comes with a number of built-in constants like `ENT_QUOTES` that are used to control built-in functions like `htmlspecialchars`.

⁷ UTF-8 is one of many standards for representing text as a series of ones and zeros in computer memory, called character encodings. If you're curious to learn all about character encodings, check out *The Definitive Guide to Web Character Encoding* [<http://www.sitepoint.com/article/guide-web-character-encoding/>].

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

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

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

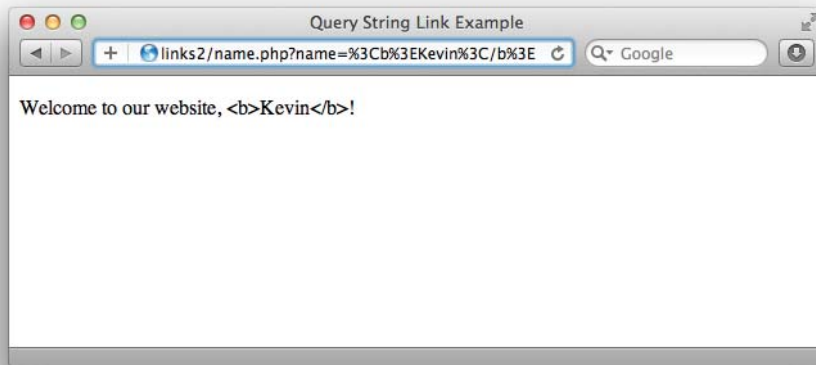


Figure 3.3. It sure is ugly, but it's secure!

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

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

Passing a single variable in the query string was nice, but it turns out you can pass *more* than one value if you want to! Let's look at a slightly more complex version of the previous example. Open up your **name.html** file again, and change the link to point to **name.php** with this more complicated query string:⁸

chapter3/links3/name.html (excerpt)

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

This time, our link passes two variables: `firstname` and `lastname`. The variables are separated in the query string by an ampersand (&, which must be written as `&` in HTML—yes, even in a link URL!). 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 **name.php** file:

chapter3/links3/name.php (excerpt)

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

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

The result is shown in Figure 3.4.

⁸ The updated version of the files may be found in the code archive in the **links3** subfolder.

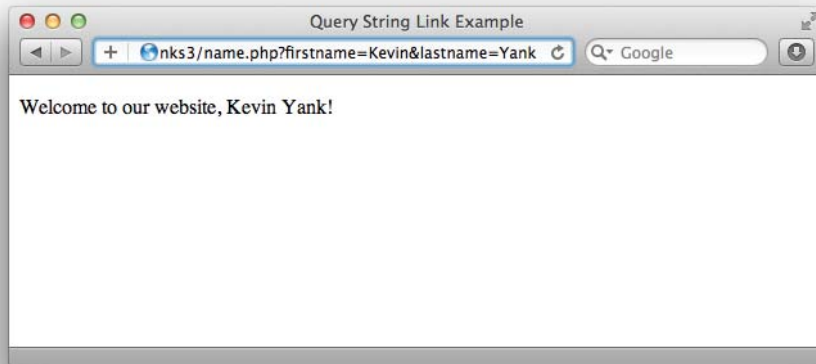


Figure 3.4. Create an even more personalized welcome message

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

Passing Variables in Forms

Rip the link out of **name.html** and replace it with this HTML code to create the form:⁹

chapter3/forms1/name.html (excerpt)

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

The form this code produces is shown in Figure 3.5.

⁹ The updated version of the files are in the **forms1** subfolder in the code archive.

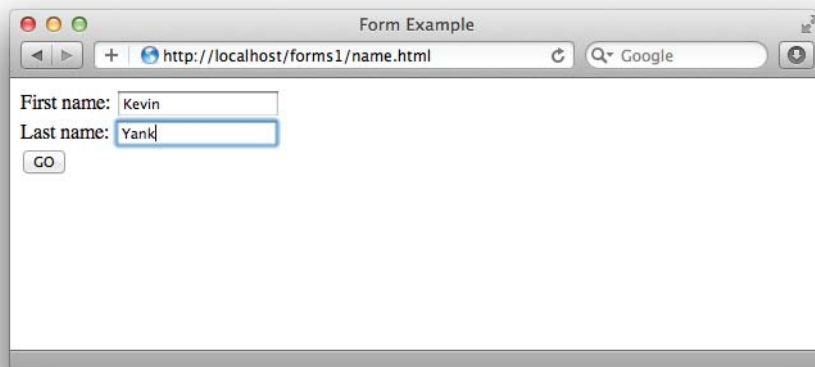


Figure 3.5. Make your own welcome message



Function Over Form

This form is quite plain looking, I'll grant you. Some judicious application of CSS would make this and all other examples in this book more attractive. Since this is a book about PHP and MySQL, however, I'm sticking with the plain look. Check out SitePoint's *The CSS3 Anthology*¹⁰ for advice on styling your forms with CSS.

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



Apostrophes in Form Fields

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

¹⁰ <http://www.sitepoint.com/books/cssant4/>

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

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

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

Edit your **name.html** file once more. Modify the form method by setting it to `post`:¹¹

chapter3/forms2/name.html (excerpt)

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

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

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

¹¹ The updated files are in **forms2** in the code archive.

```
chapter3/forms2/name.php (excerpt)

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

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

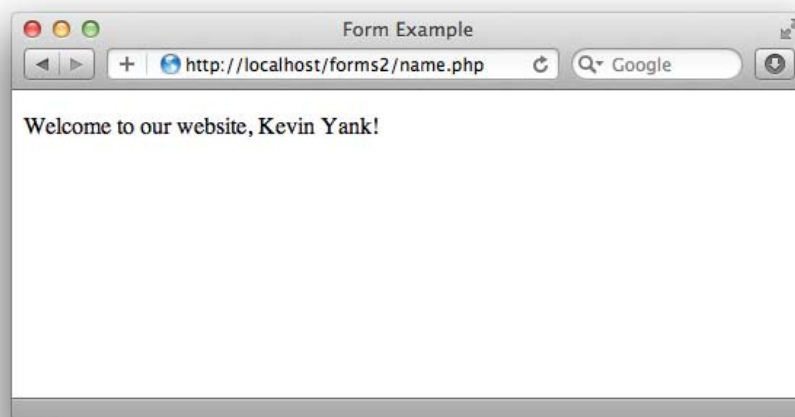


Figure 3.6. This personalized welcome is achieved without a query string

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

Sometimes, you want access to a variable without having to worry about whether it was sent as part of the query string or a form post. In cases like these, the special `$_REQUEST` array comes in handy. It contains all the variables that appear in both `$_GET` and `$_POST`. With this variable, we can modify our form processing script one more time so that it can receive the first and last names of the user from either source:¹²

chapter3/forms3/name.php (excerpt)

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

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

Control Structures

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

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

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

¹² The files in the code archive are located in the **forms3** subfolder.

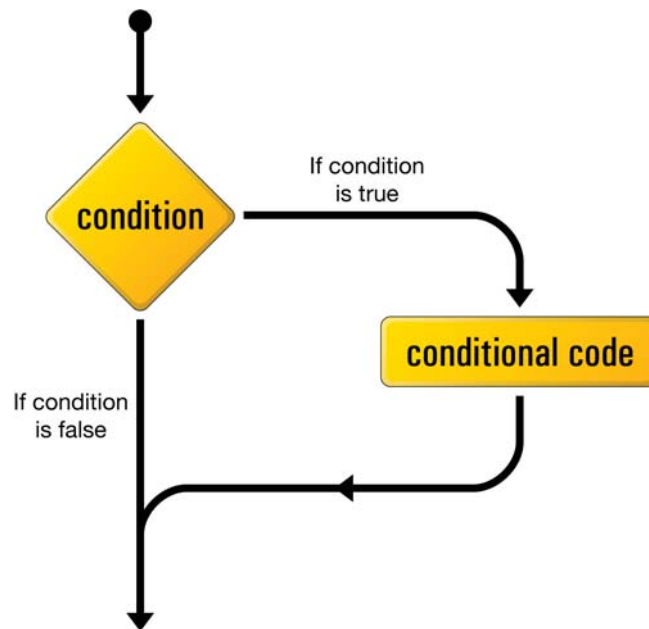


Figure 3.7. The logical flow of an `if` statement¹³

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

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

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

If you'll indulge my vanity for a moment, here's an example that shows a twist on the personalized welcome page example we created earlier. Start by opening up **name.html** for editing again. For simplicity, let's alter the form it contains so that it submits a single `name` variable to **name.php**:¹⁴

¹³ This diagram and several similar ones in this book were originally designed by Cameron Adams for the book, *Simply JavaScript* (Melbourne: SitePoint, 2006), which we wrote together. I have reused them here with his permission, and my thanks.

¹⁴ I've placed the updated versions of the files in the `if` subfolder in the code archive.

chapter3/if/name.html (excerpt)

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

Now edit **name.php**. Replace the PHP code it contains with the following:

chapter3/if/name.php (excerpt)

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

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

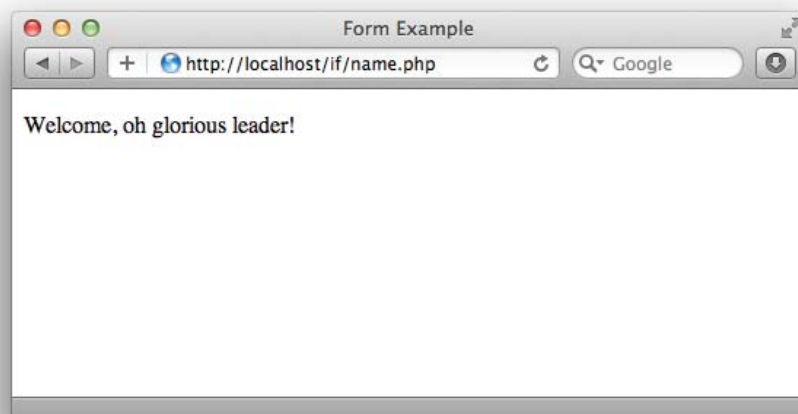


Figure 3.8. It's good to be the king

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

To offer a warmer welcome to all the plebs with names other than Kevin, we can use an **if-else statement** instead. The flow of an if-else statement is shown in Figure 3.9.

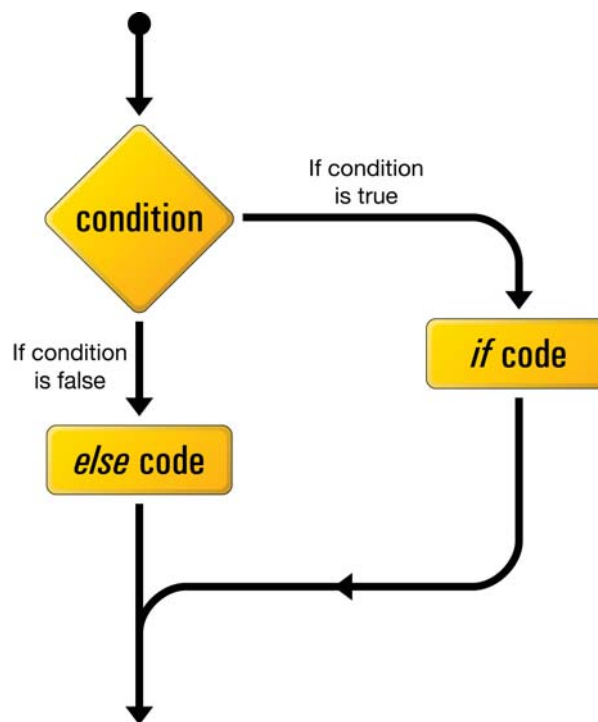


Figure 3.9. The logical flow of an if-else statement

The else portion of an if-else statement is tacked onto the end of the if portion:¹⁵

chapter3/ifelse1/name.php (excerpt)

```

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

```

¹⁵ This updated version of the example is located in the **ifelse1** subfolder in the code archive.

```
else
{
    echo 'Welcome to our website, ' .
        htmlspecialchars($name, ENT_QUOTES, 'UTF-8') . '!';
}
```

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

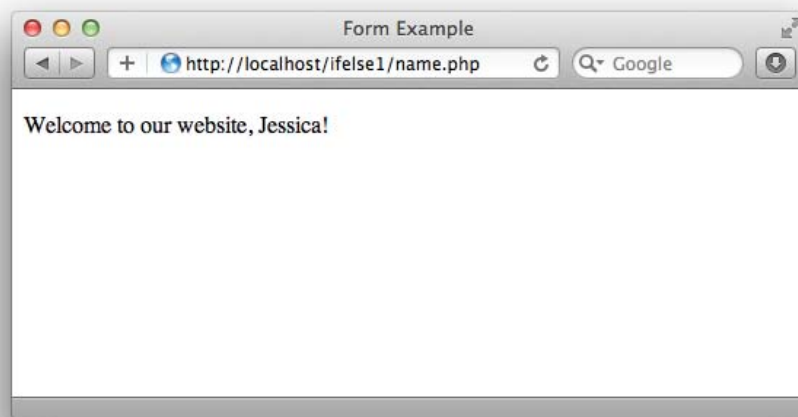


Figure 3.10. You gotta remember your peeps

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



Double Trouble

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

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

This condition is using the assignment operator (=) that I introduced back in the section called “Variables, Operators, and Comments”, instead of the equal operator

(==). Consequently, instead of comparing the value of `$name` to the string `'Kevin'`, it will actually *set* the value of `$name` to `'Kevin'`. Oops!

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

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

To do this, edit **name.html** back to the two-field version of the form:¹⁶

chapter3/ifelse2/name.html (excerpt)

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

Next, open up **name.php** and update the PHP code to match the following (I've highlighted the changes in bold):

chapter3/ifelse2/name.php (excerpt)

```
$firstName = $_REQUEST['firstname'];
$lastName = $_REQUEST['lastname'];
if ($firstName == 'Kevin' and $lastName == 'Yank')
{
    echo 'Welcome, oh glorious leader!';
}
else
{
    echo 'Welcome to our website, ' .
```

¹⁶ The updated files for this version of the example are in the **ifelse2** subfolder in the code archive.

```
    htmlspecialchars($firstName, ENT_QUOTES, 'UTF-8') . ' ' .  
    htmlspecialchars($lastName, ENT_QUOTES, 'UTF-8') . '!' ;  
}
```

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

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

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



Figure 3.11. Frankly, my dear ...

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 depend-

ing on some condition, the `while` loop allows us to use a condition to determine how many *times* we'll execute a set of statements repeatedly.

Figure 3.12 shows how a `while` loop operates.

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

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

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

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

chapter3/count10.php (excerpt)

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

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

`$count = 1;`

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

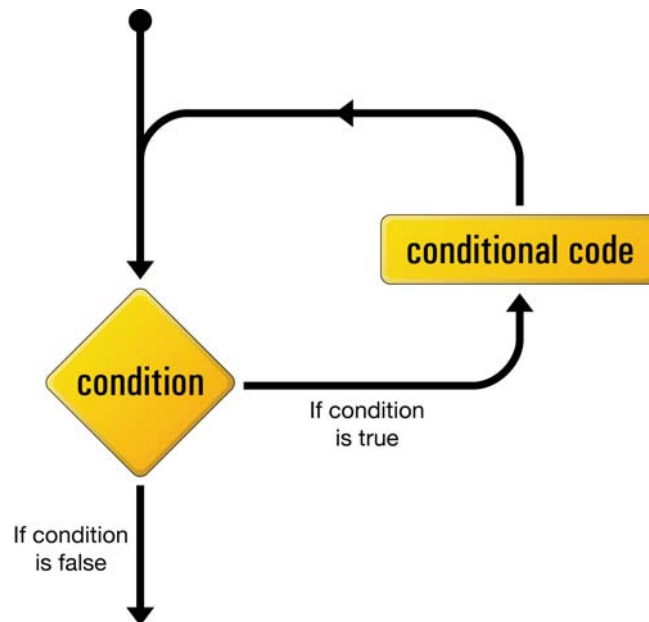


Figure 3.12. The logical flow of a while loop

```
while ($count <= 10)
```

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

```
{
```

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

```
echo "$count ";
```

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

```
++$count;
```

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

```
}
```

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

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

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



Figure 3.13. PHP demonstrates kindergarten-level math skills

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

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

Here's what it looks like in code:

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

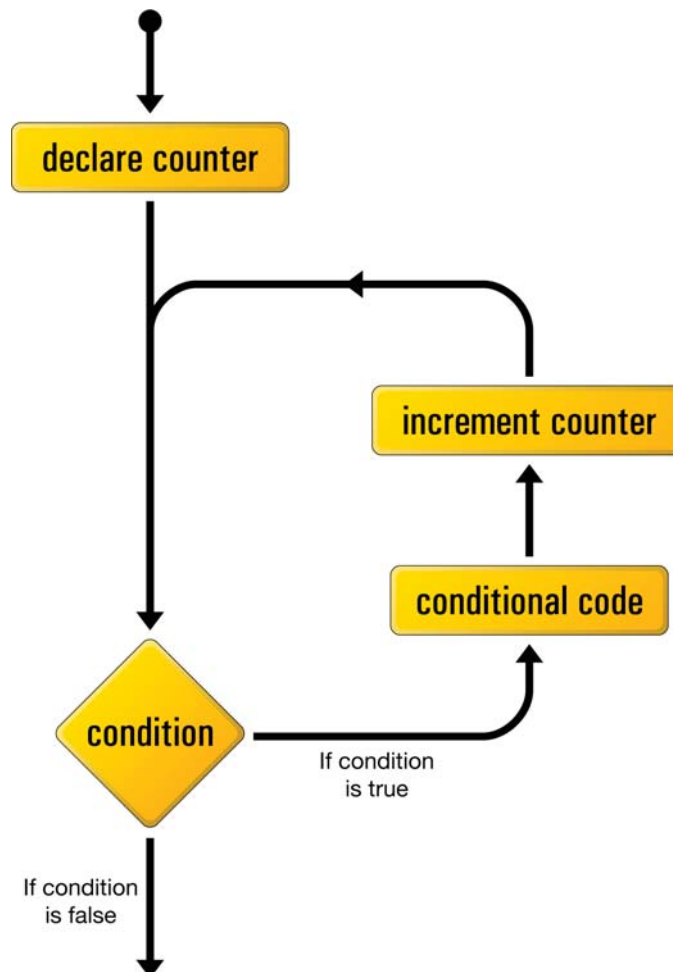


Figure 3.14. The logical flow of a for loop

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

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

chapter3/count10-for.php (excerpt)

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

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

Hiding the Seams

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

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

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

Avoid Advertising Your Technology Choices

The examples we've seen so far have contained a mixture of plain HTML files (with names ending in `.html`) and files that contain a mixture of HTML and PHP (with names ending in `.php`). Although this distinction between file types may be useful to you, the developer, there's no reason for your users to know which site pages rely on PHP code to generate them.

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

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

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

For example, take the **today.php** page that I introduced at the end of Chapter 1. Rename it from **today.php** to **index.php**. Then, instead of dropping it in the root of your web server, create a subdirectory named **today** and drop the **index.php** file in there. Now, load <http://localhost/today/> in your browser (or <http://localhost:8888/today/> or similar if you need to specify a port number for your server).

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

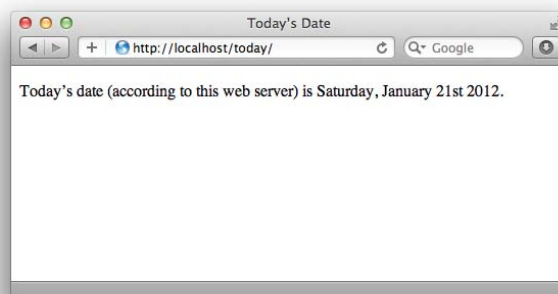


Figure 3.15. A more fashionable URL

Use PHP Templates

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

Particularly if you work in a team of not-so-savvy web designers, PHP-wise, having large blocks of cryptic PHP code intermingled with the HTML is a recipe for disaster. It's far too easy for designers to accidentally modify the PHP code, causing errors they'll be unable to fix.

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

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

Start by creating a new directory called **count10**, and create a file named **index.php** in this directory. Open the file for editing and type in this code:

chapter3/count10/index.php

```
<?php
$output = ''; ❶
for ($count = 1; $count <= 10; ++$count)
{
    $output .= "$count "; ❷
}

include 'count.html.php'; ❸
```

Yes, that's the *complete* code for this file. It contains no HTML code whatsoever. The for loop should be familiar to you by now, but let me point out the interesting parts of this code:

- ❶ Instead of echoing out the numbers 1 to 10, this script will add these numbers to a variable named `$output`. At the start of this script, therefore, we set this variable to contain an empty string.

- ❷ This line adds each number (followed by a space) to the end of the `$output` variable. The `.=` operator you see here is a shorthand way to add a value to the end of an existing string variable, by combining the assignment and string concatenation operators into one. The longhand version of this line is `$output = $output . "$count ";`, but the `.=` operator saves you some typing.
- ❸ This is an `include` statement, which instructs PHP to execute the contents of the `count.html.php` file at this location.¹⁷

Finally, you might have noticed that the file doesn't end with a `?>` to match the opening `<?php`. You can put it in if you really want to, but it's unnecessary. If a PHP file ends with PHP code, there's no need to indicate where that code ends—the end of the file does it for you. The big brains of the PHP world generally prefer to leave it off the end of files like this one that contain only PHP code.

Since the final line of this file includes the `count.html.php` file, you should create this next:

chapter3/count10/count.html.php

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Counting to Ten</title>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
  </body>
</html>
```

¹⁷ Outside of this book, you'll often see `includes` coded with parentheses surrounding the filename, as if `include` were a function like `date` or `htmlspecialchars`, which is far from the case. These parentheses, when used, only serve to complicate the filename expression, and are therefore avoided in this book. The same goes for `echo`, another popular one-liner.

This file is almost entirely plain HTML, except for the one line that outputs the value of the `$output` variable. This is the same `$output` variable that was created by the `index.php` file.

What we've created here is a **PHP template**: an HTML page with only very small snippets of PHP code that insert dynamically generated values into an otherwise static HTML page. Rather than embedding the complex PHP code that generates those values in the page, we put the code to generate the values in a separate PHP script—`index.php` in this case.

Using PHP templates like this enables you to hand over your templates to HTML-savvy designers without worrying about what they might do to your PHP code. It also lets you focus on your PHP code without being distracted by the surrounding HTML code.

I like to name my PHP template files so that they end with `.html.php`. As far as your web server is concerned, though, these are still `.php` files; the `.html.php` suffix serves as a useful reminder that these files contain both HTML and PHP code.

Many Templates, One Controller

What's nice about using `include` statements to load your PHP template files is that you can have *multiple* `include` statements in a single PHP script, as well as have it display different templates under various circumstances!

A PHP script that responds to a browser request by selecting one of several PHP templates to fill in and send back is commonly called a **controller**. A controller contains the logic that controls which template is sent to the browser.

Let's revisit one more example from earlier in this chapter: the welcome form that prompts a visitor for a first and last name.

We'll start with the PHP template for the form. For this, we can just reuse the `name.html` file we created earlier. Create a directory named `welcome` and save a copy of `name.html` called `form.html.php` into this directory. The only code you need to change in this file is the `action` attribute of the `<form>` tag:

chapter3/welcome/form.html.php

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Form Example</title>
  </head>
  <body>
    <form action="" method="post">
      <div><label for="firstname">First name:
        <input type="text" name="firstname" id="firstname"></label>
      </div>
      <div><label for="lastname">Last name:
        <input type="text" name="lastname" id="lastname"></label>
      </div>
      <div><input type="submit" value="GO"></div>
    </form>
  </body>
</html>

```

As you can see, we're leaving the `action` attribute blank. This tells the browser to submit the form back to the same URL it received it from: in this case, the URL of the controller that included this template file.

Let's take a look at the controller for this example. Create an **index.php** script in the **welcome** directory alongside your form template. Type the following code into this file:

chapter3/welcome/index.php

```

<?php
if (!isset($_REQUEST['firstname'])) ❶
{
  include 'form.html.php'; ❷
}
else ❸
{
  $firstName = $_REQUEST['firstname'];
  $lastName = $_REQUEST['lastname'];
  if ($firstName == 'Kevin' and $lastName == 'Yank')
  {
    $output = 'Welcome, oh glorious leader!'; ❹
  }
}

```

```

else
{
    $output = 'Welcome to our website, ' .
        htmlspecialchars($firstName, ENT_QUOTES, 'UTF-8') . ' ' .
        htmlspecialchars($lastName, ENT_QUOTES, 'UTF-8') . '!';
}

include 'welcome.html.php'; ❸
}

```

This code should look quite familiar at first glance; it's a lot like the **name.php** script we wrote earlier. Let me explain the differences:

- ❶ The controller's first task is to decide whether the current request is a submission of the form in **form.html.php** or not. You can do this by checking if the request contains a `firstname` variable. If it does, PHP will have stored the value in `$_REQUEST['firstname']`.

`isset` is a built-in PHP function that will tell you if a particular variable (or array element) has been assigned a value or not. If `$_REQUEST['firstname']` has a value, `isset($_REQUEST['firstname'])` will be true. If `$_REQUEST['firstname']` is unset, `isset($_REQUEST['firstname'])` will be false.

For the sake of readability, I like to put the code that sends the form in my controller first. We need this `if` statement to check if `$_REQUEST['firstname']` is *not* set. To do this, we use the **not operator** (`!`). By putting this operator before the name of a function, you reverse the value that function returns—from true to false, or from false to true.

Thus, if the request does *not* contain a `firstname` variable, then `!isset($_REQUEST['firstname'])` will return true, and the body of the `if` statement will be executed.

- ❷ If the request is not a form submission, the controller includes the **form.html.php** file to display the form.
- ❸ If the request *is* a form submission, the body of the `else` statement is executed instead.

This code pulls the `firstname` and `lastname` variables out of the `$_REQUEST` array, and then generates the appropriate welcome message for the name submitted.

- 4 Instead of echoing the welcome message, the controller stores the welcome message in a variable named `$output`.
- 5 After generating the appropriate welcome message, the controller includes the **welcome.html.php** template, which will display that welcome message.

All that's left is to write the **welcome.html.php** template. Here it is:

chapter3/welcome/welcome.html.php

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Form Example</title>
  </head>
  <body>
    <p>
      <?php echo $output; ?>
    </p>
  </body>
</html>
```

That's it! Fire up your browser and point it at `http://localhost/welcome/` (or `http://localhost:8888/welcome/` or similar if you need to specify a port number for your web server). You'll be prompted for your name, and when you submit the form, you'll see the appropriate welcome message. The URL should stay the same throughout this process.

One of the benefits of maintaining the same URL throughout this process of prompting the user for a name and displaying the welcome message is that the user can bookmark the page at any time during this process and gain a sensible result; whether it's the form page or the welcome message that's bookmarked, when the user returns, the form will be present once again. In the previous version of this example, where the welcome message had its own URL, returning to that URL without submitting the form would have generated a broken welcome message

("Welcome to our website, !"), or a PHP error message, depending on your server configuration.



Why so forgetful?

In Chapter 9, I'll show you how to remember the user's name between visits.

Bring on the Database

In this chapter, we've seen the PHP server-side scripting language in action as we've explored all the basic language features: statements, variables, operators, comments, and control structures. The sample applications we've seen have been reasonably simple, but we've still taken the time to ensure they have attractive URLs, and that the HTML templates for the pages they generate are uncluttered by the PHP code that controls them.

As you may have begun to suspect, the real power of PHP is in its hundreds (even thousands) of built-in functions that let you access data in a MySQL database, send email, dynamically generate images, and even create Adobe Acrobat PDF files on the fly.

In Chapter 4, we'll delve into the MySQL functions built into PHP, and see how to publish the joke database we created in Chapter 2 to the Web. This chapter will set the scene for the ultimate goal of this book: creating a complete content management system for your website in PHP and MySQL.

