Building a Database-Driven Web Site Using PHP and MySQL

Kevin Yank

Introduction

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

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

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 feel locked into a dry, outdated design because rewriting those hundreds of HTML files to reflect a new design would take forever.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

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

In this 10-part weekly series of articles, I'll provide a hands-on look at what's involved in building a database-driven Web site. We'll be using two new tools for this: the PHP scripting language and the MySQL relational database. If your Web host provides PHP/MySQL support, you're in great shape. If not, we'll be looking at the set-up procedures under Unix and Windows, so don't sweat it.

These articles are aimed at intermediate or advanced Web designers looking to make the leap into server-side programming. You'll be expected to be comfortable with HTML, as I'll be making use of it without explanation. A teensy bit of JavaScript may serve us well at some point, but I'll be sure to keep it simple for the uninitiated.

By the end of this series, you can expect to have a grasp of what's involved in setting up and building a database-driven Web site. If you follow along with the examples, you'll also learn the basics of PHP (a server-side scripting language that allows you to do a lot more than access a database easily) and Structured Query Language (SQL -- the standard language for interacting with relational databases). Most importantly, you'll come away with everything you need to get started on your very own database-driven site in no time!

Part 1: Installation

Welcome to the Show

Hi there, and welcome to the first in SitePoint.com's ten-part series on building a database-driven Web site! For the next few months, it will be my job to guide you as you take your first steps beyond the HTML-and-JavaScript world of client-side site design. Together we'll learn everything that's needed to build the kind of large, content-driven sites that are so successful today, but which can be a real headache to maintain if they aren't done right.

Before we get started, we need to gather together the tools we'll need for the job. In this first article, we'll download and set up the two software packages we'll be using: PHP and MySQL.

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

To retrieve information from a database, you first need to have a database. That's where MySQL comes in. MySQL is a relational database management system, or RDBMS. Exactly what role it plays and how it works we'll get into later, but basically it's a software package that is very good at organizing and managing large amounts of information. MySQL also makes that information really easy to get at using server-side scripting languages like PHP. MySQL is free for non-commercial use on most Unix-based platforms, like Linux. MySQL for Windows 9x/NT/2000 costs about US$200 to buy, but you can download an older version for free if you just want to try it out. For our purposes, the older version will serve just fine, but if you find MySQL for Windows useful and you decide to use it on one of your own sites, you should pay for it.

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

Everything we'll discuss in this article series may be done on a Windows- or Unix-based server. Depending on which type of server you'll be using, the installation procedure will be different. The next section deals with installation on a Windows-based Web server. The section after that deals with installation under Linux (and other Unix-based platforms). Unless you're especially curious, you should only need to read the section that applies to you.

Installation under Windows

As I mentioned above, MySQL for Windows costs about US$200 to buy. For those of us who just want to try it out and see what it can do, T.c.X. (the company that develops MySQL) provides an older version that can be downloaded for free. It can be found by going to http://www.mysql.com/ (or one of its mirrors listed at http://www.mysql.com/mirrors.html) and selecting "Register and download shareware version of MySQL-Win32" in the "Downloads" section under "Downloads for Windows MySQL related software". After downloading the file, unzip it and run the setup.exe program contained therein.

Once installed, MySQL is ready to roll (barring a couple of configuration tasks that we'll look at shortly). Just like your Web server, MySQL is a server that should be run in the background so that it may respond to requests for information at any time. The server program may be found in the "bin" subfolder of the folder where you installed MySQL. If you are using the shareware version of MySQL, the server is called `mysqld-shareware.exe`. Before proceeding, rename this file to `mysqld.exe`. From the MS-DOS Prompt, start the server:

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

To ensure that the server is started whenever Windows starts, you might want to create a shortcut to the program and put it in your Startup folder. If you decide to buy MySQL, it will come with a version that can be installed as a Windows NT/2000 service with the following command:

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

If you have trouble running the shareware version under Windows NT/2000, you can try running the server as a standalone program:

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

The next step is to install PHP. At the time of this writing, PHP 4.0 was available as "Release Candidate 2"--or "almost ready but not quite". Personally I use PHP 4.0-RC2 and don't have any trouble with it. Since the final version is slated for release "real soon now" (likely before this series of articles is even finished), I'd recommend you install the latest version of 4.0 so you don't have to change anything when the final version is released.

PHP may be downloaded for free from http://www.php.net/ (or one of its mirrors listed at http://www.php.net/mirrors.php). You want the "binaries for Win32" package. Don't worry about grabbing any of the add-ons; we don't need them. A good installation guide for PHP 3.0 for Windows is available at the following URL: http://www.umesd.k12.or.us/php/win32install.html. It'll probably be updated with instructions for PHP 4.0 when it is finally released, but since installation of 4.0 is pretty much identical to installation of 3.0, you shouldn't have any trouble following the instructions with either version.

Don't worry about any of the optional steps (like choosing extension modules)- we'll work through those things together in a little bit. If you have any trouble following the instructions, feel free to post your question to the SitePoint.com Forums. I will be glad to help if the other helpful people there don't beat me to it!

With MySQL and PHP installed, you're ready to proceed to Post-Installation Setup Tasks.

Installing under Linux

This section covers the exact procedure for installing PHP and MySQL under RedHat Linux 5 or later. If you're using a different flavor of Linux, or another Unix-based operating system, the steps involved will be very similar, if not identical.

As a user of RedHat Linux, you may be tempted to download and install the RPM distributions of PHP and MySQL. RPM's are nice, pre-packaged versions of software that are really easy to install. Unfortunately, they also limit the options you have in choosing how the software is configured. For this reason, I consider the RPM versions of PHP and MySQL to be more trouble than they are worth.

Since a few of the default RedHat Linux install configurations will automatically install PHP for you, your first step should be to remove any old versions of PHP and MySQL from your system. You'll need to be logged in as the root user to issue the commands to do this. Note that in the following commands, "%" represents the shell prompt, and is not something that needs to be typed.

```
% rpm -e mysql
% rpm -e php
```

If either or both of these commands tell you that the program in question is not installed, don't worry about it. If the second command runs successfully (i.e. no message is displayed), then you did indeed have an older version of PHP installed, and you'll need to do one more thing to get rid of it entirely. Open your Apache configuration file (usually `/etc/httpd/conf/httpd.conf`) in your favorite text editor and look for the two lines shown here. They usually appear in separate sections of the file, so don't worry if they're not together.

```
LoadModule php3_module modules/libphp3.so
AddModule mod_php3.c
```

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

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

```
% /etc/rc.d/init.d/httpd stop
% /etc/rc.d/init.d/httpd start
```

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

Installing MySQL under Linux

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

With the program downloaded, you should make sure you're logged in as root before proceeding with the installation, unless you only want to install MySQL in your own home directory. Begin by unpacking the downloaded file and moving into the directory that is created:

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

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

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

After sitting through the screens and screens of configuration tests, you'll eventually get back to a command prompt. You're ready to compile MySQL:

```
% make
```

After even more screens of compilation, you'll again be returned to the command prompt. You're now ready to install your newly compiled program:

```
% make install
```

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

```
% scripts/mysql_install_db
```

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

With MySQL installed and ready to store information, all that's left is to get the server running on your computer. While you can run the server as the root user, or even as yourself (if, for example, you installed the server in your own home directory), the best idea is to set up a special user on the system that can do nothing but run the MySQL server. This will remove any possibility of someone using the MySQL server as a way to break into the rest of your system. To create a special MySQL user, you'll need to log in as root and type the following commands:

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

By default, MySQL stores all database information in the `var` subdirectory of the directory to which it was installed. We want to make it so that nobody can access that directory except our new MySQL user. The following commands will do this (I'm assuming you installed MySQL to the `/usr/local/mysql` directory):

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

Everything's set for you to try launching the MySQL server for the first time. From the MySQL directory, type the following command:

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

The MySQL server has now been launched by the MySQL user and will stay running (just like your Web or FTP server) until your computer is shut down. To test that the server is running properly, type the following command:

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

A little blurb with some statistics about the MySQL server should be displayed. If you get an error message, something has gone wrong. If retracing your steps to make sure you did everything described above doesn't solve the problem, a post to the SitePoint.com Forums will probably help you pin it down in no time.

If you want to set up your MySQL server to run automatically whenever the system is running (just like your Web server probably does), you'll have to set it up to do so. In the `share/mysql` subdirectory of the MySQL directory, you'll find a script called `mysql.server` that can be added to your system startup routines to do this.

Assuming you've set up a special MySQL user to run the MySQL server, you'll need to edit the `mysql.server` script before you use it. Open it in your favorite text editor and change the `mysql_daemon_user` setting to refer to the user you created above:

```
mysql_daemon_user=mysqlusr
```

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

```
% cp share/mysql/mysql.server /etc/rc.d/init.d/
% cd /etc/rc.d/init.d
% chmod 500 mysql.server
% cd /etc/rc.d/rc3.d
% ln -s ../init.d/mysql.server S99mysql
% cd /etc/rc.d/rc5.d
% ln -s ../init.d/mysql.server S99mysql
```

That's it! To test that this works, you can reboot your system and request the status of the server as before to make sure it runs properly at startup.

Installing PHP under Linux

As mentioned above, PHP is not really a program in and of itself. Rather, it is a plug-in module for your Web server (probably Apache). There are actually three ways you can install the PHP plug-in for Apache:

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

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

The second and third options are pretty much identical in terms of performance, but since you likely already have Apache installed, you'd probably prefer to avoid downloading, recompiling, and reinstalling it from scratch. For this reason, we'll be using the third option.

Start by downloading the PHP Source package from http://www.php.net/ (or one of its mirrors listed at http://www.php.net/mirrors.php). At the time of this writing, PHP 4.0 was available as "Release Candidate 2"-or "almost ready but not quite". Personally I use PHP 4.0-RC2 and don't have any trouble with it. Since the final version will be out "real soon now" (likely before this series of articles is even finished), I'd recommend you install the latest version of 4.0 so you don't have to change anything when the final version is released. In case you do decide to stick with 3.0, however, I'll be sure to point out any spots in the installation procedure that would differ between the two.

The file you downloaded should be called `php-version.tar.gz`. We'll start by extracting the files it contains:

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

To install PHP as a loadable Apache module, you'll need the Apache `apxs` program. This comes with most versions of Apache, but if you're using the copy that was installed by RedHat Linux, you'll need to install the Apache development RPM package to get it. You'll find this package on your RedHat CD or you can download it from http://www.redhat.com/. By default, RedHat will install the program as `/usr/sbin/apxs`. If you see that file, you know it's installed.

For the rest of this install procedure, you'll need to be logged in as the root user, because it involves making changes to the Apache configuration files.

The next step is to configure the PHP installation program by letting it know what options you want to have enabled and where it should find the programs it needs to know about (like Apache and MySQL). Unless you know what you're doing, you should just type the command like this (all on one line):

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

If you are installing PHP 3.0 (and not 4.0 or later), you'll also need to tell it where to find MySQL on your system with the following additional parameter:

```
  --with-mysql=/usr/local/mysql/
```

After watching several screens of tests scroll by, you'll be returned to the command prompt. The following two commands will compile and then install PHP:

```
% make
% make install
```

PHP is now installed in `/usr/local/php` (unless you specified a different directory with the `--prefix` option of `./configure` above), and expects to find its configuration file, named `php.ini`, in the same directory (unless you specified a different directory with the `--with-config-file-path` option of `./configure` above). PHP comes with a sample `php.ini` file called `php.ini-optimized` (`php.ini-dist` for PHP 3.0). Copy this file from your installation work directory to where it belongs:

```
% cp php.ini-optimized /usr/local/php/php.ini
```

Or for PHP 3.0:

```
% cp php.ini-dist /usr/local/php/php.ini
```

We'll worry about fine-tuning `php.ini` shortly. For now, we need to make sure Apache knows where to find PHP so that it can load it when starting up. Open your Apache `httpd.conf` configuration file (`/etc/httpd/conf/httpd.conf` on RedHat Linux) in your favorite text editor. Look for a line like the following:

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

If you installed PHP 3.0, the line will read `php3` instead of `php4`. You're looking for a new, uncommented line (no `#` at the start of the line), not the old line that we commented out earlier. Chances are it will not appear along with the other `LoadModule` lines in the file. Once you find it, you need to change the path so that it matches all the other `LoadModule` lines in the file. Under RedHat Linux, this means changing the line so that it looks like this:

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

Next, look for the line starting with `DirectoryIndex`. This line tells Apache what filenames to use when looking for the default page for a given directory. You'll see the usual `index.html` and so forth, but you need to add `index.php` and `index.php3` to that list:

```
DirectoryIndex index.html index.cgi ... index.php index.php3
```

Finally, go right to the bottom of the file and add the following line to tell Apache what file extensions should be seen as PHP files:

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

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

Post-Installation Setup Tasks

Once PHP is installed and the MySQL server is running, whether you're running under Windows or Linux or some other operating system, the very first thing to be done is to assign a "root password" for MySQL. MySQL only lets authorized users view and manipulate the information stored in its databases, and it's up to you to make sure that MySQL knows who is an authorized user and who isn't. When MySQL is first installed, it is configured with a user named "root" that has access to do pretty much anything without even entering a password. Your first task should be to assign a password to the root user so that not just anyone can go messing around in your databases.

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

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

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

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

```
mysqladmin -u root reload
```

If this command gives you an error message telling you that access was denied, don't worry. It just means the password has already taken effect.

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

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

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

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

```
mysqladmin -u root -p shutdown
```

With your MySQL database system safe from intrusion, all that's left is to configure PHP. PHP is configured using a text file called `php.ini`. If you installed PHP under Windows you should already have copied `php.ini` into your Windows directory. If you installed PHP under Linux using the instructions above, you should already have copied `php.ini` into the PHP installation folder (`/usr/local/php`).

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

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

If you're running PHP version 4.0, you'll also need to check the following line:

```
register_globals = On
```

And if you're running PHP version 3.0 under Windows, uncomment the following line by removing the semicolon at the start of it (PHP 4.0 doesn't need this):

```
extension=php_mysql.dll
```

Save the changes to `php.ini`, then restart your Web server. Under Linux, you can restart Apache if you're logged in as root by typing:

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

You're done! Now all that's left is to test to make sure everything's working okay (see Your First PHP Script).

If Your Web Host Provides PHP and MySQL

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

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

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

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

If they support remote access to the MySQL server, you'll want to download a program for connecting to and interacting with the server. This article series will assume you've downloaded the set of MySQL client programs from http://www.mysql.com/. Packages are available for Windows or Unix, and are free. Install instructions are fairly simple and are included with the packages. If you prefer something more graphical, you can download something like MySQLWinAdmin for Windows (also available from http://www.mysql.com/). I'd really recommend getting comfortable with the basic client programs first, though, since the commands you use with them will be similar to those you include in your PHP scripts to access MySQL databases.

Your First PHP Script

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

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

```
<HTML>
<HEAD>
<TITLE>Today's Date</TITLE>
</HEAD>
<BODY>
<P>Today's Date (according to this Web server) is
<?php
  echo( date("l, F dS Y.") );
?>
</BODY>
</HTML>
```

Save it and place it on your Web site as you would any regular HTML file, then see what it looks like when you view it in your browser. If you haven't yet had time to set up PHP on your Web server, click here to see the results on our server.

Pretty neat, huh? If you use the view source feature in your browser, all you'll see is a regular HTML file with the date in it. The PHP code (everything between `<?php` and `?>` in the code above) has been interpreted by the Web server and converted to normal text before sending it to your browser. The beauty of PHP (and other server-side scripting languages) is that the Web browser doesn't have to know anything about it!

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

Wrap-up

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

In the next section, we'll learn the basics of relational databases and get started working with MySQL. If you've never even touched a database before, I promise you it'll be a real eye opener! Meanwhile, I'd love to hear what you thought of the first installment in this series of articles. Drop me a line at kevin@sitepoint.com, or stop by the SitePoint.com Forums to speak your mind.

Part 2: Getting Started with MySQL

Hi there, and welcome back! Last week, we went through the process of installing and setting up two software programs: PHP and MySQL. This week, we'll be concentrating on the latter by learning how to work with MySQL databases using Structured Query Language (SQL).

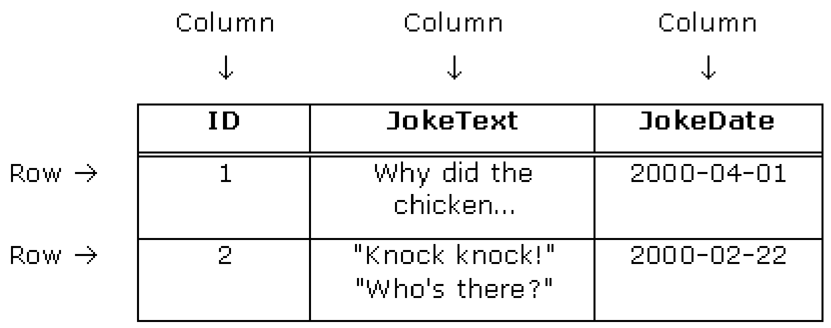
An Introduction to Databases

As I explained briefly last week, PHP is a server-side scripting language that lets you insert instructions into your Web pages that your Web server software (be it Apache, Personal Web Server, or whatever) will execute before sending those pages to a browser that requests them. In a brief example, I showed how it was possible to insert the current date into a Web page every time it was requested.

Now that’s all well and good, but things really get interesting when a database is added to the mix. A database server (in our case, MySQL) is a program that can store large amounts of information in an organized format that is easily accessible from scripting languages like PHP. For example, you could tell PHP to look in the database for a list of jokes that you’d like to appear on your Web site.

In this example, the jokes would be stored entirely in the database. The advantage of this would be twofold. First, instead of having to write an HTML file for each of your jokes, you could write a single PHP file designed to fetch any joke out of the database and display it. Second, to add a joke to your Web site would just be a matter of adding the joke to the database. The PHP code would take care of the rest by automatically displaying the new joke along with the rest when it fetched the list of jokes from the database.

Let’s run with this example as we look at how data is stored in a database. A database is composed of one or more 'tables', each of which contains a list of 'things'. For our joke database, we would probably start with a table called "jokes" which would contain a list of jokes. Each table in a database has one or more columns, or fields. Each column holds a certain piece of information about each "thing" in the database. Returning to our example, our "jokes" table might have columns for the text of the jokes and the dates the jokes were added to the database. Each joke that we stored in this table would then be said to be a 'row' in the table. To see where all this terminology comes from, have a look at what this table actually looks like:



Column
Column
Column
Row →
Row →

ID	JokeText	JokeDate
1	Why did the chicken...	2000-04-01
2	"Knock knock!" "Who's there?"	2000-02-22

Notice that, in addition to columns for the joke text ("JokeText") and the date of the joke ("JokeDate"), I included a column named "ID". The function of this column is to assign a unique number to each joke so we have an easy way to refer to them and to keep track of which joke is which.

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

Logging onto MySQL

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

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

Whatever method you choose, whatever operating system you're using, you'll end up at a command line ready to run the MySQL client program to connect to your MySQL server. Here's what you should type:

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

You need to replace `<hostname>` by the host name or IP address of the computer on which the MySQL server is running. If you're running the client program on the same computer as the server, you can actually leave off the `-h <hostname>` part of the command instead of typing `-h localhost`, for example. `<username>` should be your MySQL user name. If you installed the MySQL server yourself, this will just be `root`. If you're using your Web host's MySQL server, this should be the MySQL user name they assigned you.
The "-p" argument tells the program to prompt you for your password, which it

should do as soon as you enter the command above. If you set up the MySQL server yourself, this password is the root password you chose in Part I. If you're using your Web host's MySQL server, this should be the MySQL password they gave you.

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

```
mysql>
```

Now, the MySQL server can actually keep track of more than one database (this allows a Web host to set up a single MySQL server for several of its subscribers to use, for example), so your next step should be to pick a database to work with. First, let's get a list of databases on the current server. Type the following command (don't forget the semicolon!), then ENTER.

```
mysql> SHOW DATABASES;
```

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

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

The MySQL server uses the first database, called `mysql`, to keep track of users, their passwords, and what they're allowed to do. We'll steer clear of this database for the time being and come back to it in a later article. The second database, called `test` is a sample database. We can actually get rid of this database, since we won't be using it in this series of articles (and we'll be building plenty of samples ourselves). Deleting something in MySQL is called "dropping" it, and the command for doing so is appropriately named:

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

If you type this command and press Enter, MySQL will obediently delete the database, saying `Query OK` as confirmation. Notice you are not prompted with any kind of "are you sure?" message. You have to be very careful to type your commands correctly in MySQL because, as this example shows, you can obliterate your entire database--along with all the information it contains--with one single command!

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

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

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

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

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

MySQL will completely ignore the command you had begun typing, and will go back to the prompt to wait for another command.

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

```
mysql> quit
Bye
```

So what's SQL?

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

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

Creating a Database

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

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

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

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

```
mysql> USE jokes;
```

You're now ready to start using your database. Since a database is empty until you add some tables to it, creating a table to hold our jokes will be our first order of business.

Creating A Table

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

```
mysql> CREATE TABLE <table name> (
    ->  <column 1 name> <col. 1 type> <col. 1 details>,
    ->  <column 2 name> <col. 2 type> <col. 2 details>,
    ->  ...
    -> );
```

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

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

Pretty scary-looking, huh? Let's break it down:

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

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

Note also that we assigned a specific type of data to each column we created. `ID` will contain integers, `JokeText` will contain text, and `JokeDate` will contain dates. MySQL requires you to specify a data type for each column in advance. Not only does this help keep your data organized, but it allows you to compare the values in a column in powerful ways (as we'll see later). For a complete list of supported MySQL data types, see the MySQL Reference Manual.

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

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

```
mysql> SHOW TABLES;
```

The response should look like this:

```
+-----------------+
| Tables in jokes |
+-----------------+
| Jokes           |
+-----------------+
1 row in set
```

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

```
mysql> DESCRIBE Jokes;
+----------+---------+------+-----+------------+- -
| Field    | Type    | Null | Key | Default    | ...
+----------+---------+------+-----+------------+- -
| ID       | int(11) |      | PRI | 0          | ...
| JokeText | text    | YES  |     | NULL       |
| JokeDate | date    |      |     | 0000-00-00 |
+----------+---------+------+-----+------------+- -
3 rows in set
```

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

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

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

Inserting Data into a Table

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

```
mysql> INSERT INTO <table name> SET
    ->  columnName1 = value1,
    ->  columnName2 = value2,
    ->  ...
    -> ;

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

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

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

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

Note that in the second form of the `INSERT` command, the order you list the columns in must match with the order you list the values in. Otherwise, the order of the columns doesn't matter, as long as you give values for all required fields.

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

Viewing Stored Data

The command for viewing data stored in your database tables, `SELECT`, is easily 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 retrieving and presenting data. Since at this point in our experience with databases we only have need of fairly simple lists of results, we'll limit ourselves to considering only the simpler forms of the select command.

The following command will list everything stored in the `Jokes` table:

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

Read aloud, this command says "select everything from Jokes". If you try this command, you'll see something resembling the following:

```
+----+---------------------------------------
-----------------------+------------+
| ID | JokeText
                        | JokeDate   |
+----+---------------------------------------
-----------------------+------------+
|  1 | Why did the chicken cross the road? To
 get to the other side! | 2000-04-01 |
+----+---------------------------------------
-----------------------+------------+
1 row in set (0.05 sec)
```

It looks a little messed up, because the text in the JokeText column is too long for the table to fit properly on the screen. For this reason, you might want to tell MySQL to leave out the `JokeText` column. The command for doing this is as follows:

```
mysql> SELECT ID, JokeDate FROM Jokes;
```

This time instead of telling it to "select everything", we told it precisely which columns we were interested in seeing. The results look like this:

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

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

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

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

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

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

So far, all of our examples have fetched all the entries in the table. By adding what's called a "`WHERE` clause" (for reasons that will become obvious in a moment) to a `SELECT` command, we can limit what entries are returned as results. Take the following example:

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

This query will count the number of jokes that have dates "greater than or equal to" January 1st, 2000. "Greater than or equal to" when dealing with dates means "on or after".

Another variation on this theme lets you search for entries containing a certain piece of text. Check out this query:

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

This query displays the text of all jokes that contain the word `chicken` in their `JokeText` column. The `LIKE` keyword tells MySQL that the named column must match the given pattern. In this case, the pattern we've used is `"%chicken%"`. The `%` signs here indicate that the word `chicken` may be preceded and/or followed by any string of text.

Conditions may also be combined in the `WHERE` clause to further restrict results. For example, to display knock-knock jokes from April 2000 only, we could use the following query:

```
mysql> SELECT JokeText FROM Jokes WHERE
    -> JokeText LIKE "%knock knock%" AND
    -> JokeDate >= "2000-04-01" AND
    -> JokeDate < "2000-05-01";
```

Enter a few more jokes into the table and experiment with SELECT statements a little. A good familiarity with the SELECT statement will come in handy later in this series.
There's a lot more you can do with the SELECT statement, but we'll save looking at some of its more advanced features for when we need them. If you're too curious to wait, the MySQL Reference Manual has got plenty of information on the subject.

Modifying Stored Data

Once you've entered some data into a database table, you might like to change it at some point. Whether you're correcting a spelling mistake, or changing the date attached to a joke, such changes are made using the UPDATE command. This command contains elements of the INSERT command (for setting column values) and of the SELECT command (for picking out entries to modify). The general form of the UPDATE command is as follows:

```
mysql> UPDATE <tableName> SET
    -> <col_name>=<new_value>, ...
    -> WHERE <where clause>;
```

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

```
mysql> UPDATE Jokes SET JokeDate="1990-04-01" WHERE ID=1;
```

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

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

Deleting Stored Data

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

```
mysql> DELETE FROM <tableName> WHERE <where clause>;
```

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

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

One thing to note is that the WHERE clause is actually optional, but you should be very careful to know what you're doing if you leave it off because then the DELETE command applies to all entries in the table. The following command will empty the Jokes table in one fell swoop:

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

Scary, huh?

Wrap-up

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

All this and more will be covered in Part Four of this series, where we'll be discussing database design principles and looking at some more advanced examples. For now, though, we've hopefully accomplished our objective of getting you comfortably interacting with MySQL using the command line interface.

In Part Three, the fun continues as we delve into the PHP server-side scripting language and learn how to use it to create dynamic Web pages. In the meantime, you can practice with MySQL by creating a decent-sized `Jokes` table, as it'll come in handy two weeks from now! Until then, your questions and comments are welcome in the SitePoint.com Forums.

Part 3: Getting Started with PHP

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

Presenting PHP

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

The key difference between JavaScript and PHP is that, while the Web browser interprets JavaScript once the Web page containing the script has been downloaded, server-side scripting languages like PHP are interpreted by the Web server before the page is even sent to the browser. Once interpreted, the PHP code is replaced in the Web page by the results of the script, so all the browser sees is a standard HTML file. The script is processed entirely by the server. Thus the designation: server-side scripting language.

Let's look back at the `today.php` example presented in Part One:

```
<HTML>
<HEAD>
<TITLE>Today's Date</TITLE>
</HEAD>
<BODY>
<P>Today's Date (according to this Web server) is
<?php
  echo( date("l, F dS Y.") );
?>
</BODY>
</HTML>
```

Most of this is plain HTML. The line between `<?php` and `?>`, however, is written in PHP. `<?php` means "begin PHP code", and `?>` means "end PHP code". The Web server is asked to interpret everything between these two delimiters and convert it to regular HTML code before sending the Web page to a browser that requests it. The browser is presented with something like this:

```
<HTML>
<HEAD>
<TITLE>Today's Date</TITLE>
</HEAD>
<BODY>
<P>Today's Date (according to this Web server) is
Wednesday, June 7th 2000.</BODY>
</HTML>
```

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

- No browser compatibility issues. PHP scripts are interpreted by the Web server and nothing else, so you don't have to worry about whether the language you're using will be supported by your visitors' browsers.
- Access to server-side resources. In the above example, we place the date according to the Web server into the Web page. If we had inserted the date using JavaScript, we would only be able to display the date according to the computer on which the Web browser was running. Now while this isn't an especially impressive example of exploiting server-side resources, we could have just as easily inserted some other information that would only be available to a script running on the Web server—information stored in a MySQL database running on the Web server computer, for example.
- Reduced load on the client. JavaScript can significantly slow down the display of a Web page on slower computers, as the browser must run the script before it can display the Web page. With server-side scripting, this becomes the burden of the Web server machine to bear.

Basic Syntax and Commands

PHP syntax will be very familiar to anyone with an understanding of C, C++, Java, JavaScript, Perl, or any other C-derived language. A PHP script consists of a series of commands, or "statements", each of which is an instruction that the Web server must follow before proceeding to the next. PHP statements, like those in the above-mentioned languages, are always terminated by a semicolon (`;`).

The following is a typical PHP statement:

```
echo( "This is a <B>test</B>!" );
```

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

```
<HTML>
<HEAD>
<TITLE> Simple PHP Example </TITLE>
</HEAD>
<BODY>
<P><?php echo("This is a <B>test</B>!"); ?></P>
</BODY>
```

```
</HTML>
```

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

```
<HTML>
<HEAD>
<TITLE> Simple PHP Example </TITLE>
</HEAD>
<BODY>
<P>This is a <B>test</B>!</P>
</BODY>
</HTML>
```

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

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

Variables and Operators

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

```
$testvariable = 3;
```

PHP is a "loosely typed" language, which means that a single variable may contain any type of data (be it a number, a string of text, or some other kind of value), and may change types over its lifetime. So the following statement, if written after the statement above, assigns a new value to our existing `$testvariable`. In the process, the variable changes from containing a number to containing a string of text:

```
$testvariable = "Three";
```

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

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

The lines above each end with a comment. Comments are a way to describe what your code is doing by inserting explanatory text into your code and telling the PHP interpreter to ignore it. Comments begin with `//` and end at the end of the same line. If you're familiar with `/* */` style comments in other languages, these work in PHP as well. I'll be using comments throughout the rest of this series to help explain what the code I present is doing.

Getting back to the four statements above, the operators used allow you to add, subtract, multiply, and divide numbers. Among others, there is also an operator for sticking strings of text together:

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

Variables may be used pretty much anywhere an actual value can be. Consider the following example:

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

Notice the last two lines especially. You can include the name of a variable right inside a text string and have the value inserted in its place if you surround the string with double quotes. As the last line demonstrates, however, a string surrounded with single quotes will not convert variable names to their values.

User Interaction and Forms

For many of the interesting applications of PHP, the ability to interact with the user viewing the Web page is essential. Veterans of JavaScript will be used to thinking in terms of event handlers, which allow you to react directly to many user actions, such as moving the mouse over a link on the page. Server-side scripting languages such as PHP have a more limited scope when it comes to user interaction. Since the only time PHP code is actually run is when a page is requested from the server, user interaction can only occur in a back-and-forth fashion, with the user sending requests to the server and the server replying with dynamically generated pages.

The key to user interaction with PHP is to understand the techniques that exist for sending information along with a user’s request for a new Web page. PHP makes this fairly easy, as we’ll now see.

The simplest method for sending information along with a page request is using the "URL query string". If you’ve ever seen a URL with a question mark following the filename, you’ve seen this technique in use. Let’s look at an easy example. Create a regular HTML file (no `.php` file extension is required, since there will be no PHP code in this file) and insert the following link:

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

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

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

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

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

You can pass more than one value in the query string if you want to. Let's look at a slightly more complex version of the same example. Change the link in the HTML file to read as follows:

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

This time, we are passing two variables: `firstname` and `lastname`. The variables are separated in the query string by an ampersand (`&`). You can pass even more variables if you want by separating each `name=value` pair from the next with an ampersand.

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

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

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

```
<FORM ACTION="welcome.php" METHOD=GET>
First Name: <INPUT TYPE=TEXT NAME="firstname"><BR>
Last Name: <INPUT TYPE=TEXT NAME="lastname">
<INPUT TYPE=SUBMIT VALUE="GO">
</FORM>
```

This form has the exact same effect as the second link we looked at (with `firstname=Kevin&lastname=Yank` in the query string), except you can type whatever names you like. When you click the submit button (which has a label of "GO"), the browser will load `welcome.php` and automatically add the variables and their values to the query string for you. It gets the names of the variables from the `NAME` attributes of the `INPUT TYPE=TEXT` tags and it gets the values from whatever the user types into the text fields.

The `METHOD` attribute of the `FORM` tag is used to tell the browser how to send the variables and their values along with the request. A value of `GET` (as used above) causes them to be passed in the query string, but there is another alternative. It is not always desirable -- or even technically feasible -- to have the values appear in the query string. What if we included a `TEXTAREA` tag in your form to let the user enter a large amount of text? A URL containing several paragraphs of text in the query string would be ridiculously long, and would exceed by far the maximum length of the URL in today's browsers. The alternative is for the browser to pass the information invisibly, behind the scenes. The code for this looks exactly the same, but instead of setting the form method to `GET`, we set it to `POST`:

```
<FORM ACTION="welcome.php" METHOD=POST>
First Name: <INPUT TYPE=TEXT NAME="firstname"><BR>
Last Name: <INPUT TYPE=TEXT NAME="lastname">
<INPUT TYPE=SUBMIT VALUE="GO">
</FORM>
```

This form is functionally identical to the previous one. The only difference is that the URL of the page loaded when the user clicks the "GO" button will not have a query string. On the one hand, this lets you include large values, or sensitive values (like passwords) in the data submitted by the form without them appearing in the query string. On the other, if the user bookmarks the page resulting from the submission of the form, that bookmark will be useless, since it does not contain the submitted values. This, incidentally, is the main reason that search engines like AltaVista use the query string to submit search terms. If you bookmark a search results page on AltaVista, you can use that bookmark to perform the same search again later, since the search terms are contained in the URL.

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

Control Structures

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

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

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

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

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

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

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

As indicated in the code structure above, the "`else` clause" (that part of the `if-else` statement that says what to do if the condition is false) is optional. Let's say you wanted to display the special message above if the appropriate name was entered, but otherwise not display anything. Here's how the code would look:

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

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

A safeguard against making this common mistake is to swap the positions of the variable and the constant value in the comparison as follows:

```
if ( "Kevin" == $name ) {
```

This has exactly the same effect, but look what happens if you mistakenly use a single equals sign. PHP will attempt to assign the value of the variable (`$name`) to the constant value (`"Kevin"`). Since you can't change the value of a constant, PHP will choke and display an error message, immediately drawing your attention to the fact that you forgot the second equals sign!

Conditions can be more complex than a single comparison for equality. Recall that we modified `welcome.php3` to take a first and last name. If we wanted to display a special message only for a particular person, we'd have to check the values of both names:

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

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

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

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

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

This works very similarly to an `if-else` statement without an else clause. The difference arises when the condition is true and the statement(s) are executed. Instead of continuing execution with the next statement following the closing brace (`}`), the condition is checked again. If the condition is still true, then the statement(s) are executed a second time, and a third... and will continue to be executed as long as the condition remains true. The first time the condition evaluates false (whether it's the first time it's checked or the one-hundred-and-first), execution jumps immediately to the next statement following the while loop (after the closing brace).

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

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

Kind of scary-looking, I know, but let me talk you through it line by line. The first line creates a variable called `$count` and assigns it a value of `1`. The second line is the beginning of a `while` loop, the condition for which is that the value of `$count` is less than or equal (`<=`) to `10`. The third and fourth lines make up the body of the `while` loop, and will be executed over and over as long as that condition holds true. The third line simply outputs the value of `$count` followed by a space. The fourth line adds one to the value of `$count` (`$count++` is a shortcut for `$count = $count + 1` -- both will work).

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

The condition we used 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.

Multi-Purpose Pages

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

- We need the name on every page of the site, not just one.

- We have no control over which page of our site users will view first.

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

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

Notice that we've embedded PHP code right in the middle of an HTML tag. This is perfectly legal, and will work just fine. We're familiar with the `echo` function, but `urlencode` is new. What this does is take any special characters in the string (spaces, for example) and converts them to the special codes needed for them to appear in the query string. For example, if the `$name` variable had a value of `"Kevin Yank"`, then since spaces are not allowed in the query string, the output of `urlencode` (and thus the string output by echo) would be `"Kevin+Yank"`, which would then be automatically converted back when creating the `$name` variable in `newpage.php`.

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

The solution is to have every page of our site check to see if a name has been specified, and prompt the user for a name if necessary. This means that every page of our site will either display its content or a prompt to enter a name depending on whether the `$name` variable is found to have a value. If this is beginning to sound to you like a good place for an `if-else` statement, you're a quick study!

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

```
<HTML>
<HEAD>
<TITLE> Multi-Purpose Page Outline </TITLE>
</HEAD>
<BODY>
<?php if (<condition>) { ?>
<!-- HTML content to display if <condition> is true -->
<?php } else { ?>
<!-- HTML content to display if <condition> is false -->
<?php } ?>
</BODY>
</HTML>
```

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

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

```
<HTML>
```

```
<HEAD>
<TITLE> Multi-Purpose Page Outline </TITLE>
</HEAD>
<BODY>
<?php if (<condition>): ?>
<!-- HTML content to display if <condition> is true -->
<?php else: ?>
<!-- HTML content to display if <condition> is false -->
<?php endif; ?>
</BODY>
</HTML>
```

Okay, so with all the tools we need in hand, let's look at a sample page of our site:

```
<HTML>
<HEAD>
<TITLE> Sample Page </TITLE>
</HEAD>
<BODY>
<?php if ( isset($name) ): ?>
  <P>Your name: <?php echo($name); ?></P>
  <P>This paragraph contains a
<A HREF="newpage.php?name=<?php echo(urlencode
($name)); ?>">link</A> that passes the
name variable on to the next document.</P>
<?php else: ?>
  <!-- No name has been provided, so we
       prompt the user for one.         -->
  <FORM ACTION=<?php echo($PHP_SELF); ?> METHOD=GET>
  Please enter your name: <INPUT TYPE=TEXT NAME="name">
  <INPUT TYPE=SUBMIT VALUE="GO">
  </FORM>
<?php endif; ?>
</BODY>
</HTML>
```

There are two new tricks in the above code, but overall you should be pretty comfortable with the way it works. First of all, we are using a new function called `isset` in the condition. This function returns (outputs) a value of true if the variable it is given has been assigned a value (i.e. if a name has been provided), and false if the variable does not exist (i.e. if a name has not yet been given). The second new trick is the use of the variable `$PHP_SELF` to specify the `ACTION` attribute of the `FORM` tag. This variable is one of several that PHP always gives a value to automatically. In particular, `$PHP_SELF` will always be set to the URL of the current page. This gives us an easy way to create a form that, when submitted, will load the very same page, but this time with the `$name` variable specified.

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

Wrap-up

This week, we've gotten a taste of the PHP server-side scripting language by exploring all the basic language features: statements, variables, operators, and control structures. The sample applications we've seen have been pretty simple, but don't let that dissuade you. The real power of PHP is in the hundreds of built-in functions that let you do everything from accessing data in a MySQL database to sending e-mail, and from dynamically generating images to creating Adobe Acrobat PDF files on the fly.

In Part Four, we'll delve into the MySQL functions to publish the joke database that we created last week on the Web!

Part 4: Publishing MySQL Data on the Web

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

A Look Back at First Principles

Before we leap forward, it's worth a brief look back to remind ourselves of the goal we are working toward. We have two powerful, new tools at our disposal: the PHP scripting language, and the MySQL database engine. It's important to understand how these two are going to fit together.

The whole idea of a database-driven Web site is to allow the content of the site to reside in a database, and for that content to be dynamically pulled from the database to create Web pages featuring it for people using a regular Web browser to view. So on one end of the system you have a visitor to your site using a Web browser, loading http://www.yoursite.com/, and expecting to view a standard HTML Web page. On the other end you have the content of your site sitting in one or more tables in a MySQL database that only understands how to respond to SQL queries (commands).

The PHP scripting language is the go-between that speaks both languages. Using PHP, you can write the presentation aspects of your site (the fancy graphics and page layouts) as "templates" in regular HTML. Where the content belongs in those templates, you use some PHP code to connect to the MySQL database and -- using SQL queries just like those you used to create a table of jokes in Part Two -- retrieve and display some content in its place.

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

- The visitor's Web browser asks for the Web page using a standard URL.
- The Web server software (Apache, IIS, or whatever) recognizes that the requested file is a PHP script, and so interprets it using its PHP plug-in before responding to the page request.
- Some PHP commands (which we have yet to learn) connect to the MySQL database and request the content that belongs in the Web page.
- The MySQL database responds by sending the requested content to the PHP script.
- The PHP script stores the content into one or more PHP variables, then uses the now-familiar `echo` function to output it as part of the Web page.
- The PHP plug-in finishes up by handing a copy of the HTML it has created to the Web server.
- The Web server sends the HTML to the Web browser as it would a plain HTML file, except instead of coming directly from an HTML file, the page is the output provided by the PHP plug-in.

Connecting to MySQL with PHP

Before we can get content out of our MySQL database for inclusion in our Web page, we must first know how to establish a connection to MySQL. Back in Part Two, we used a program called `mysql` that allowed us to make such a connection. PHP has no need of any special program, however; support for connecting to MySQL is built right into the language. The following PHP function call establishes the connection:

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

Where `<address>` is the IP address or hostname of the computer on which the MySQL server software is running (`"localhost"` if running on the same computer as the Web server software), and `<username>` and `<password>` are the same MySQL user name and password you used to connect to the MySQL server in Part Two.

You may or may not remember that functions in PHP usually return (output) a value when they are called. Don't worry if this doesn't ring any bells for you -- it's a detail that we glossed over when originally discussing functions. In addition to doing something useful when they are called, most functions output a value, and that value may be stored in a variable for later use. The `mysql_connect` function shown above, for example, returns a number that identifies the connection that has been established. Since we intend to make use of the connection, we should hold onto this value. Here's an example of how we might connect to our MySQL server.

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

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

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

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

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

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

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

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

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

Notice we use the $dbcnx variable containing the database connection identifier to tell the function what database connection to use. This parameter is actually optional. When it is omitted, the function will automatically use the link identifier for the last connection opened. This function returns true when successful and false if an error occurs. Once again, it is prudent to use an if statement to handle errors:

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

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

Performing SQL Queries with PHP

In Part Two, we connected to the MySQL database server using a program called mysql that allowed us to type SQL queries (commands) and view the results of those queries immediately. In PHP, a similar mechanism exists: the mysql_query function.

```
mysql_query(<query>, <connection id>);
```

Where <query> is a string containing the SQL command to be executed. As with mysql_select_db, the connection identifier parameter is optional.

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

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

The mysql_error function used here returns a string of text describing the last error message that was sent by the MySQL server.

For DELETE, INSERT, and UPDATE queries (which serve to modify stored data), MySQL also keeps track of the number of table rows (entries) that were affected by the query. Consider the following SQL command, which we used in Part Two to set the dates of all jokes containing the word "chicken":

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

When executing this query, we can use the mysql_affected_rows function to view the number of rows that were affected by this update:

```
if ( mysql_query($sql) ) {
  echo("<P>Update affected " .
       mysql_affected_rows() . " rows.</P>");
} else {
  echo("<P>Error performing update: " .
       mysql_error() . "</P>");
}
```

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

Handling SELECT Result Sets

For most SQL queries, the mysql_query function returns either true (success) or false (failure). For SELECT queries this just isn't enough. You'll recall that SELECT queries are used to view stored data in the database. In addition to indicating whether the query succeeded or failed, PHP must also receive the results of the query. As a result, when processing a SELECT query, mysql_query returns a number that identifies a "result set", containing a list of all the rows (entries) returned from the query. False is still returned if the query fails for whatever reason.

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

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

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

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

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

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

The `mysql_fetch_array` function accepts a result set as a parameter (stored in the `$result` variable in this case), and returns the next row in the result set as an array. If you're not familiar with the concept of arrays, don't worry; we'll discuss it in a moment. When there are no more rows in the result set, `mysql_fetch_array` instead returns false.

Now, the above statement assigns a value to the `$row` variable, but at the same time the whole statement itself takes on that same value. This is what lets us use the statement as a condition in our `while` loop. Since `while` loops keep looping until their condition evaluates to false, the loop will occur as many times as there are rows in the result set, with `$row` taking on the value of the next row each time through the loop. All that's left is to figure out how to get the values out of the `$row` variable each time through the loop.

Rows of a result set are represented as arrays. An array is a special kind of variable that contains multiple values. If you think of a variable as a box containing a value, then an array can be thought of as a box with compartments, with each compartment able to store an individual value. In the case of our database row, the compartments are named after the table columns in our result set. If `$row` is a row in our result set, then `$row["JokeText"]` is the value in the `JokeText` column of that row. So here's what our `while` loop should look like if we want to print the text of all the jokes in our database:

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

To summarize, here's the complete code of a PHP Web page that will connect to our database, fetch the text of all the jokes in the database, and display them in HTML paragraphs:

```
<HTML>
<HEAD>
<TITLE> Our List of Jokes </TITLE>
<HEAD>
<BODY>
<?php
  // Connect to the database server
  $dbcnx = @mysql_connect("localhost",
           "root", "mypasswd");
  if (!$dbcnx) {
    echo( "<P>Unable to connect to the " .
          "database server at this time.</P>" );
    exit();
  }
  // Select the jokes database
  if (! @mysql_select_db("jokes") ) {
    echo( "<P>Unable to locate the joke " .
          "database at this time.</P>" );
    exit();
  }
?>
<P> Here are all the jokes in our database: </P>
<BLOCKQUOTE>
<?php

  // Request the text of all the jokes
  $result = mysql_query(
            "SELECT JokeText FROM Jokes");
  if (!$result) {
    echo("<P>Error performing query: " .
         mysql_error() . "</P>");
    exit();
  }
  // Display the text of each joke in a paragraph
  while ( $row = mysql_fetch_array($result) ) {
    echo("<P>" . $row["JokeText"] . "</P>");
  }
?>
</BLOCKQUOTE>
</BODY>
</HTML>
```

Inserting Data into the Database

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

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

```
<FORM ACTION="<?php echo($PHP_SELF); ?>" METHOD=POST>
<P>Type your joke here:<BR>
<TEXTAREA NAME="joketext" ROWS=10 COLS=40 WRAP></TEXTAREA><BR>
<INPUT TYPE=SUBMIT NAME="submitjoke" VALUE="SUBMIT">
</FORM>
```

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

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

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

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

We now have the code to allow a user to type a joke and add it to our database. All that remains is to slot it into our existing joke viewing page in a useful fashion. Since most users will only want to view our jokes, we don't want to mar our page with a big, ugly form unless the user expresses an interest in adding a new joke.

For this reason, our application is well suited for implementation as a multi-purpose page. Here's the code:

```
<HTML>
...
<BODY>
<?php
  // If the user wants to add a joke
  if (isset($addjoke)):
?>
<FORM ACTION="<?php echo($PHP_SELF); ?>" METHOD=POST>
<P>Type your joke here:<BR>
<TEXTAREA NAME="joketext" ROWS=10 COLS=40 WRAP>
</TEXTAREA><BR>
<INPUT TYPE=SUBMIT NAME="submitjoke" VALUE="SUBMIT">
</FORM>
<?php
  else:
    // Connect to the database server
    $dbcnx = @mysql_connect("localhost",
             "root", "mypasswd");
    if (!$dbcnx) {
      echo( "<P>Unable to connect to the " .
            "database server at this time.</P>" );
      exit();
    }
    // Select the jokes database
    if (! @mysql_select_db("jokes") ) {
      echo( "<P>Unable to locate the joke " .
            "database at this time.</P>" );
      exit();
    }
    // If a joke has been submitted,
    // add it to the database.
    if ("SUBMIT" == $submitjoke) {
      $sql = "INSERT INTO Jokes SET " .
             "JokeText='$joketext', " .
             "JokeDate=CURDATE()";
      if (mysql_query($sql)) {
        echo("<P>Your joke has been added.</P>");
      } else {
        echo("<P>Error adding submitted joke: " .
             mysql_error() . "</P>");
      }
    }

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

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

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

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

  endif;

?>
</BODY>
</HTML>
```

There we go! With a single file containing a little PHP code we are able to view jokes in and add jokes to our MySQL database.

A Challenge

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

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

If you think you have the answer or if you would just like to see the solution, advance to the next page. Good luck!

Wrap-up

This week, we learned some new PHP functions that allow us to interface with a MySQL database server. Using these functions, we built our first database-driven Web site by publishing our Jokes database online and allowing visitors to add jokes of their own to it.

In Part Five, we go back to the MySQL command line and learn how to use relational database principles and some more advanced SQL queries to represent more complex types of information, and give our visitors credit for the jokes they add!

Challenge Solution

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

- Previously, we passed an `$addjoke` variable with our "Add a Joke!" link at the bottom of the page to signal that our script should display the joke entry form instead of the usual list of jokes. In similar fashion, we pass a `$deletejoke` variable with our "Delete this Joke" link to indicate our desire to have a joke deleted.
- We fetch the `ID` column from the database for each joke along with the `JokeText` column, so that we have the ID associated with each joke in the

database.
- We set the value of the `$deletejoke` variable to the ID of the joke being deleted. This was done by inserting the ID value fetched from the database into the HTML code for the "Delete this Joke" link of each joke.
- Using an `if` statement, we watch to see if `$deletejoke` is set to some value (using the `isset` function) when loading the page. If it is, we use the value that it is set to (the ID of the joke to be deleted) in an SQL `DELETE` statement to delete the joke in question.

Here's the complete code. If you have any questions, don't hesitate to post them in the SitePoint.com forums!

```
<HTML>
...
<BODY>
<?php
  // If the user wants to add a joke
  if (isset($addjoke)):
?>
<FORM ACTION="<?php echo($PHP_SELF); ?>" METHOD=POST>
<P>Type your joke here:<BR>
<TEXTAREA NAME="joketext" ROWS=10 COLS=40 WRAP>
</TEXTAREA><BR>
<INPUT TYPE=SUBMIT NAME="submitjoke" VALUE="SUBMIT">
</FORM>
<?php
  else:
    // Connect to the database server
    $dbcnx = @mysql_connect(
               "localhost", "root", "mypasswd");
    if (!$dbcnx) {
      echo( "<P>Unable to connect to the " .
            "database server at this time.</P>" );
      exit();
    }
    // Select the jokes database
    if (! @mysql_select_db("jokes") ) {
      echo( "<P>Unable to locate the joke " .
            "database at this time.</P>" );
      exit();
    }
    // If a joke has been submitted,
    // add it to the database.
    if ("SUBMIT" == $submitjoke) {
      $sql = "INSERT INTO Jokes SET " .
             "JokeText='$joketext', " .
             "JokeDate=CURDATE()";
      if (mysql_query($sql)) {
        echo("<P>Your joke has been added.</P>");
      } else {
        echo("<P>Error adding submitted joke: " .
             mysql_error() . "</P>");
      }
    }
    // If a joke has been deleted,
    // remove it from the database.
    if (isset($deletejoke)) {
      $sql = "DELETE FROM Jokes " .
             "WHERE ID=$deletejoke";
      if (mysql_query($sql)) {
        echo("<P>The joke has been deleted.</P>");
      } else {
        echo("<P>Error deleting joke: " .
             mysql_error() . "</P>");
      }
    }

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

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

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

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

  endif;

?>
</BODY>
</HTML>
```

Part 5: Relational Database Design

Since Part Two of this series, we've been working with a very simple database of jokes, composed of a single table named, appropriately enough, `Jokes`. While this database has served us well as an introduction to using MySQL databases, there is a lot more to relational database design than this simple example illustrates. This week, we'll expand on our example and learn about a few new features of MySQL in an effort to realize and appreciate what relational databases have to offer.

Be forewarned that many topics will be covered only in an informal, hands-on (i.e. non-rigorous) sort of way. As any computer science major will tell you, database design is a serious area of research with tested and mathematically provable principles that, while useful, are beyond the scope of this article. For more information, I would recommend stopping by http://www.datamodel.org/ for a list of good books, as well as several useful resources on the subject.

Giving Credit where Credit is Due

To start things off, let's recall the structure of our `Jokes` table. It contains three columns: `ID`, `JokeText`, and `JokeDate`. Together, these columns allow us to identify jokes (`ID`), keep track of their text (`JokeText`) as well as the date they were entered (`JokeDate`).

Now let's say we wanted to track another piece of information about our jokes: the names of the people who submitted them. It would seem natural to want to add a new column to our `Jokes` table for this. The SQL `ALTER` command (which we have not seen before) lets us do exactly what we need. Log into your MySQL server using the `mysql` command-line program as in Part Two, select your database (jokes if you used the name suggested in Part Two) then type the following command:

```
mysql> ALTER TABLE Jokes ADD COLUMN
    -> AuthorName VARCHAR(100);
```

This adds a column called `AuthorName` to our table. The type declared is a variable-length character string of up to 100 characters in length (plenty for even very esoteric names). Let's also add a column for the author's e-mail address:

```
mysql> ALTER TABLE Jokes ADD COLUMN
    -> AuthorEMail VARCHAR(100);
```

For more information about the `ALTER` command, see the MySQL Reference Manual. Just to make sure the two columns were added properly, we should ask MySQL to describe the table to us:

```
mysql> DESCRIBE Jokes;
+-------------+--------------+------+-----+-- -  -
| Field       | Type         | Null | Key | Def...
+-------------+--------------+------+-----+-- -  -
| ID          | int(11)      |      | PRI | ...
| JokeText    | text         | YES  |     | ...
| JokeDate    | date         |      |     | ...
| AuthorName  | varchar(100) | YES  |     | ...
| AuthorEMail | varchar(100) | YES  |     | ...
+-------------+--------------+------+-----+-- -  -
5 rows in set (0.01 sec)
```

Looks good. Obviously, we would need to make changes to the HTML and PHP form code we created in Part Four for adding new jokes to the database, but I'll leave figuring out those details to you as an exercise. Using `UPDATE` queries, we could now add author details to all the jokes in the table. Before getting carried away with this, however, we need to stop and consider if we made the right design choice here. In this case, it turns out that we did not.

Rule of Thumb: Keep Things Separate

As your knowledge of database-driven Web sites continues to grow, you decide that a personal joke list isn't enough. In fact, you begin to get more submitted jokes than you have original jokes of your own. You decide to launch a Web site where people from all over the world can share jokes with each other. You've heard of the Internet Movie Database (IMDB)? You decide to open the Internet Joke Database (IJDB)! Adding the author's name and e-mail address to each joke certainly makes a lot of sense, but the way we did it above leads to several potential problems:

- What if a frequent contributor to your site named Joan Smith changed her email address? She might begin submitting new jokes using the new address, but all the old jokes would still have the old address attached to them. Looking at your database, you might just think there were two different people named Joan Smith submitting to your database. If she were especially thoughtful, she might inform you of the change of address, and you might try to update all the old jokes with the new address, but if you missed just one joke your database would still have incorrect information stored in it. Database design experts refer to this sort of problem as an "update anomaly".

- It would be natural for you to rely on your database to provide a list of all the people who've ever submitted jokes to your site. In fact, you could easily obtain a mailing list using the following query:

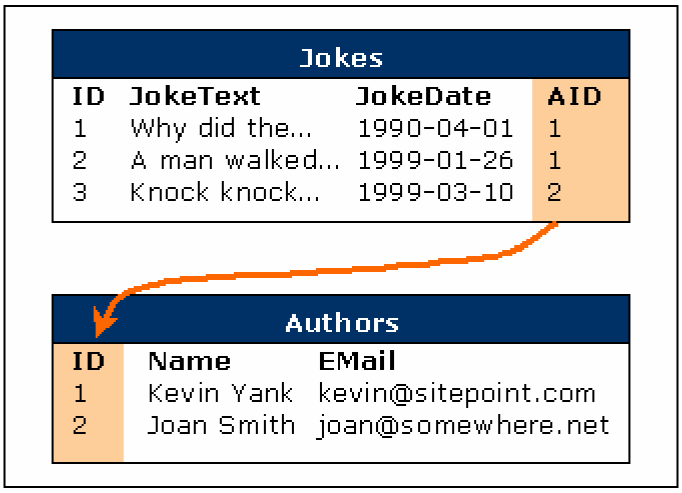
```
mysql> SELECT DISTINCT AuthorName, AuthorEMail
    -> FROM Jokes;
```

The word `DISTINCT` in the above query tells MySQL not to output duplicate result rows. For example, if Joan Smith submitted 20 jokes to your site, her name and email address would appear 20 times in the list instead of just once if you failed to use the `DISTINCT` option.

- If for some reason you decided to remove all the jokes that a particular author had submitted to your site, doing so would also remove all record of this person from the database, and you wouldn't be able to email them with information about your site anymore! Since your mailing list might be a major source of income for your site, you wouldn't want to go throwing away people's email addresses just because you didn't like the jokes they submitted to your site. Database design experts call this a "delete anomaly".

- You have no guarantee that Joan Smith would not enter her name as "Joan Smith" one day, as "J. Smith" the next, and as "Smith, Joan" on yet another occasion. This would make keeping track of a particular author exceedingly difficult (especially if Joan Smith had several email addresses she liked to use, too).

These problems and more can be very quickly dealt with. Instead of storing the information for the authors in the `Jokes` table, let us create an entirely new table for our list of authors. Since we used a column called `ID` in the `Jokes` table to uniquely identify each of our jokes with a number, we'll use an identically named column in our new table to identify our authors. We can then use those "author ID's" in our `Jokes` table to associate authors with their jokes. The complete database layout is shown here:



Jokes

ID	JokeText	JokeDate	AID
1	Why did the...	1990-04-01	1
2	A man walked...	1999-01-26	1
3	Knock knock...	1999-03-10	2

Authors

ID	Name	EMail
1	Kevin Yank	kevin@sitepoint.com
2	Joan Smith	joan@somewhere.net

What the above two tables show are three jokes and two authors. The `AID` column (short for "Author ID") of the `Jokes` table provides a relationship between the two tables (indicating that Kevin Yank submitted jokes 1 and 2 and Joan Smith submitted joke 3). Notice also that, since each author now only appears once in the database, and appears independently of the jokes he or she has submitted, we have avoided all the problems outlined above.

The most important characteristic of this database design, however, is that, since we are storing information about two types of "things" (jokes and authors), it is most appropriate to have two tables. This is a rule of thumb that you should always keep in mind when designing a database: each type of entity (or "thing") that you want to be able to store information about should be given its own table.

Setting up the above database from scratch is fairly simple (involving just two CREATE TABLE queries), but since we'd like to make these changes in a non-destructive manner (i.e. without losing any of our precious knock-knock jokes), we'll use the ALTER command again. First, we get rid of the author-related columns in the Jokes table:

```
mysql> ALTER TABLE Jokes DROP COLUMN AuthorName;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
mysql> ALTER TABLE Jokes DROP COLUMN AuthorEMail;
Query OK, 0 rows affected (0.00 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Now we create our new table:

```
mysql> CREATE TABLE Authors (
    -> ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> Name VARCHAR(100),
    -> EMail VARCHAR(100)
    -> );
```

Finally, we add the AID column to our Jokes table:

```
mysql> ALTER TABLE Jokes ADD COLUMN AID INT;
```

All that's left is to add some authors to the new table, and assign authors to all the existing jokes in the database by filling in the AID column.

Dealing with Multiple Tables

With our data now separated into two tables, it may seem like we are making the process of retrieving that data more complicated for ourselves. Consider, for example, our original goal: to display a list of jokes with the name and email address of the author next to each joke. In our single-table solution, we could get all the information we needed to produce such a list using a single SELECT statement in our PHP code:

```
$jokelist = mysql_query(
  "SELECT JokeText, AuthorName, AuthorEMail ".
  "FROM Jokes");
while ($joke = mysql_fetch_array($jokelist)) {
  $joketext = $joke["JokeText"];
  $name = $joke["AuthorName"];
  $email = $joke["AuthorEMail"];
  // Display the joke with author information
  echo( "<P>$joketext<BR>" .
        "(by <A HREF='mailto:$email'>$name</A>)</P>" );
}
```

In our new system, this would at first no longer seem possible. Since the details about the author of each joke aren't stored in the Jokes table, one might think we would have to fetch those details individually for each joke we wanted to display. The code to do so would look something like this:

```
// Get the list of jokes
$jokelist = mysql_query(
  "SELECT JokeText, AID FROM Jokes");
while ($joke = mysql_fetch_array($jokelist)) {
  // Get the text and Author ID for the joke
  $joketext = $joke["JokeText"];
  $aid = $joke["AID"];

  // Get the author details for the joke
  $authordetails = mysql_query(
    "SELECT Name, Email FROM Authors WHERE ID=$aid");
  $author = mysql_fetch_array($authordetails);
  $name = $author["Name"];
  $email = $author["EMail"];
  // Display the joke with author information
  echo( "<P>$joketext<BR>" .
        "(by <A HREF='mailto:$email'>$name</A>)</P>" );
}
```

Pretty messy, and it involves a query to the database for every single joke to be displayed, which could slow down the display of our page considerably. With all this taken into account, it would seem like the "old way" was actually the better solution, despite its weaknesses.

Fortunately, relational databases are designed to make working with data stored in multiple tables easy! Using a new form of the `SELECT` statement, called a "join", we can have the best of both worlds. Joins allow us to treat related data in multiple tables as if they were stored in a single table. Here's what the syntax of a join looks like:

```
mysql> SELECT <columns> FROM <tables>
    -> WHERE <condition(s) for data to be related>;
```

In our case, the columns we're interested in are `JokeText` in the `Jokes` table, and `Name` and `Email` in the `Authors` table. The condition for an entry in the `Jokes` table to be related to an entry in the `Authors` table is that the value of the `AID` column in the `Jokes` table is equal to the value of the `ID` column in the `Authors` table. Here's an example of a join (the first two queries are just to show you what's contained in the two tables -- they are not needed):

```
mysql> SELECT LEFT(JokeText,20), AID FROM Jokes;
+---------------------+------+
| LEFT(JokeText,20)   | AID  |
+---------------------+------+
| Why did the chicken  |    1 |
| A man walked into a  |    1 |
| Knock knock. Who's t |    2 |
+---------------------+------+
3 rows in set (0.00 sec)
mysql> SELECT * FROM Authors;
+----+------------+--------------------+
| ID | Name       | EMail              |
+----+------------+--------------------+
|  1 | Kevin Yank | kyank@attglobal.net |
|  2 | Joan Smith | joan@somewhere.net  |
+----+------------+--------------------+
2 rows in set (0.00 sec)
mysql> SELECT LEFT(JokeText,15), Name, Email
    -> FROM Jokes, Authors WHERE AID = Authors.ID;
+-------------------+------------+--------- -- -
| LEFT(JokeText,15) | Name       | EMail
+-------------------+------------+--------- -- -
| Why did the chi   | Kevin Yank | kyank@attg...
| A man walked in   | Kevin Yank | kyank@attg...
| Knock knock. Wh   | Joan Smith | joan@somew...
+-------------------+------------+--------- -- -
3 rows in set (0.00 sec)
```

See? The results of the third `SELECT`, which is a join, group the values stored in the two tables into a single table of results, with related data correctly grouped together. Even though our data is stored in two tables, we can still get all the information we need to produce the joke list on our Web page with a single database query.

Note in the query that, since there are columns named `ID` in both tables, we had to specify the name of the table when referring to the `ID` column in the `Authors` table (`Authors.ID`). If we had not specified the table name, MySQL wouldn't have known which `ID` we were referring to, and would have produced the following error:

```
mysql> SELECT LEFT(JokeText,20), Name, Email
    -> FROM Jokes, Authors WHERE AID = ID;
ERROR 1052: Column: 'ID' in where clause is ambiguous
```

Now that we know how to efficiently get at the data stored in our two tables, we can rewrite the code for our joke list to take advantage of joins:

```
$jokelist = mysql_query(
  "SELECT JokeText, Name, EMail " .
  "FROM Jokes, Authors WHERE AID=Authors.ID");
while ($joke = mysql_fetch_array($jokelist)) {
  $joketext = $joke["JokeText"];
  $name = $joke["Name"];
  $email = $joke["EMail"];
  // Display the joke with author information
  echo( "<P>$joketext<BR>" .
        "(by <A HREF='mailto:$email'>$name</A>)</P>" );
}
```

The more you work with databases, the more you'll come to realize how powerful this simple ability to combine data in separate tables into a single table of results really is. Consider, for example, the following query, which displays a list of all jokes written by Joan Smith:

```
mysql> SELECT JokeText FROM Jokes, Authors WHERE
    -> Name="Joan Smith" AND AID=Authors.ID;
```

The results that are output from the above query come only from the `Jokes` table, but we use a join to let us search for jokes based on a value stored in the `Authors` table. There will be plenty more examples of clever queries like this throughout this series, but hopefully this example alone illustrates that the practical applications of joins are many and varied, and in almost all cases can save you a lot of work!

Simple Data Relationships

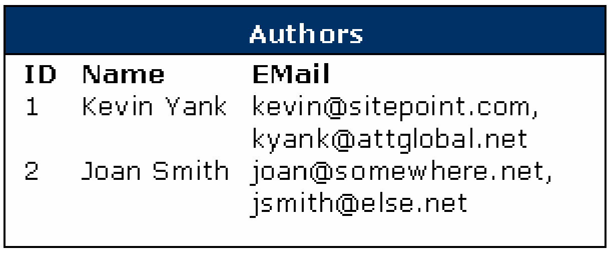
The best type of database layout for a given situation is usually dictated by the type of relationship that exists between the pieces of data you are working with. In this section, we shall examine the typical relationship types and how best to represent them in a relational database.

In the case of a simple one-to-one relationship, a single table is all that is needed. An example of a one-to-one relationship that we have seen is the email address of each author in our joke database. Since there will be one e-mail address for each author, and one author for each e-mail address, there is no reason to split the addresses off into a separate table.

A many-to-one relationship is a little more complicated, but we have seen one of these already as well. Each joke in our database is associated with just one author, but many jokes may have been written by that one author. This joke-author relationship is many-to-one. We have already covered the problems that result from storing the information associated with a joke's author in the same table as the joke itself. In brief, it results in potentially many copies of the same data that are difficult to keep synchronized and that waste space. By splitting the data into two tables and using an `ID` column to link the two together (making joins possible as shown above), all of these problems disappear.

We have yet to see a one-to-many relationship, but thinking of such an example is not difficult. In our database so far, we have assumed that each author only has one email address. While this may not always be the case, this is a reasonable limitation to impose since we only really need one e-mail address to get in touch with an author. We simply trust that he or she would enter their most-used email address -- or at least one that is checked regularly -- when adding him or herself to the database. If we did, however, want to support multiple email addresses, we would be faced with a one-to-many relationship (one author may have many email addresses, but each email address belongs to exactly one author).

When someone inexperienced in database design approaches a one-to-many relationship like this one, his or her first approach is often to try to store multiple values in a single database field, like this:

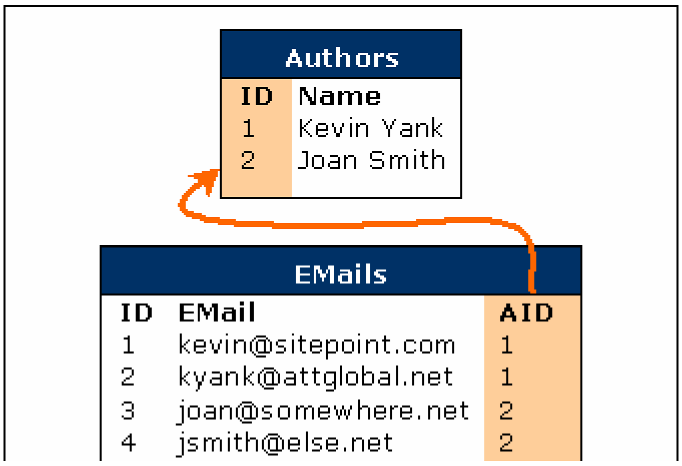


Authors

ID	Name	EMail
1	Kevin Yank	kevin@sitepoint.com, kyank@attglobal.net
2	Joan Smith	joan@somewhere.net, jsmith@else.net

While this would work, retrieving a single email address from the database would involve breaking up the string by searching for commas (or whatever special character you chose to use as a separator) -- a not-so-simple and potentially time-consuming operation. Imagine the PHP code necessary for removing one particular email address from one particular author! In addition, you'd need to allow for much longer values in the `EMail` column, which could result in wasted disk space because the majority of authors would only have one email address.

The solution for a one-to-many relationship such as this is very similar to the solution we saw for a many-to-one relationship above. As one might expect, the pattern is simply reversed. We would just break the `Authors` table into two tables, `Authors` and `EMails`, and then associate the email addresses with their authors using an Author ID (`AID`) column in the `EMails` table:



Authors

ID	Name
1	Kevin Yank
2	Joan Smith

EMails

ID	EMail	AID
1	kevin@sitepoint.com	1
2	kyank@attglobal.net	1
3	joan@somewhere.net	2
4	jsmith@else.net	2

Using a join, it is trivial to list the email addresses associated with a particular author:

```
mysql> SELECT EMail FROM Authors, EMails WHERE
    -> Name="Kevin Yank" AND AID=Authors.ID;
+---------------------+
| EMail               |
+---------------------+
| kevin@sitepoint.com |
| kyank@attglobal.net |
+---------------------+
2 rows in set (0.00 sec)
```

Many-to-Many Relationships

Okay, you've now got a steadily growing database of jokes published on your Web site. It's growing so quickly, in fact, that the number of jokes is becoming unmanageable! People visiting your site are faced with a mammoth page containing hundreds of jokes listed with no structure whatsoever. Something has to change.

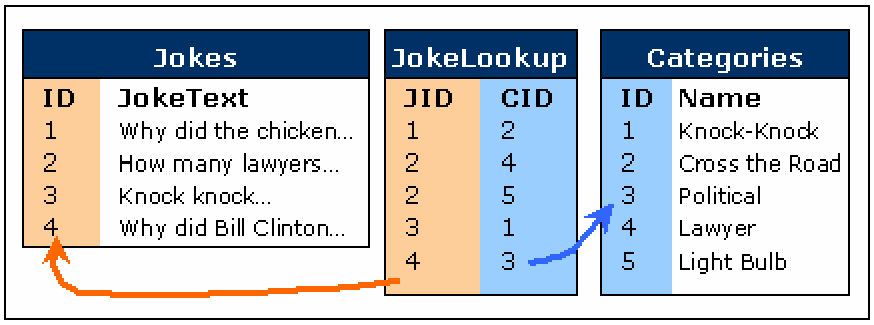
You decide to place your jokes into categories such as "Knock-Knock Jokes", "Crossing the Road Jokes", "Lawyer Jokes", and "Political Jokes". Remembering our rule of thumb from earlier, you identify joke categories as a different type of "thing", and create a new table for them:

```
mysql> CREATE TABLE Categories (
    -> ID INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> Name VARCHAR(100),
    -> Description TEXT
    -> );
Query OK, 0 rows affected (0.00 sec)
```

Now comes the daunting task of assigning categories to your jokes. It occurs to you that a "political" joke might also be a "crossing the road" joke, and a "knock-knock" joke might also be a "lawyer" joke. A single joke might belong to many categories, and each category will contain many jokes. This is a many-to-many relationship.

Once again, many inexperienced developers begin to think of ways to store several values into a single column, because the obvious solution is to add a `Categories` column to the `Jokes` table and use it to list the `ID`'s of the categories to which each joke belongs. A second rule of thumb would be useful here: if you need to store multiple values in a single column, chances are your design is flawed.

The correct way to represent a many-to-many relationship is to use a "lookup table". This is a table that contains no actual data, but which serves to define pairs of entries that are related. Here's what the database design would look like for our joke categories:



Jokes	
ID	JokeText
1	Why did the chicken...
2	How many lawyers...
3	Knock knock...
4	Why did Bill Clinton...

JokeLookup	
JID	CID
1	2
2	4
2	5
3	1
4	3

Categories	
ID	Name
1	Knock-Knock
2	Cross the Road
3	Political
4	Lawyer
5	Light Bulb

The `JokeLookup` table associates joke ID's (`JID`) with category ID's (`CID`). Looking at the above example, we can see that the joke beginning with "How many lawyers..." belongs to both the "Lawyer" and "Light Bulb" categories.

Creating a lookup table is done in much the same way as creating any other table. The difference lies in the choice of the primary key. Every table we have created so far has had a column named `ID` that was designated the `PRIMARY KEY` when the table was created. Designating a column as a primary key tells MySQL not to allow two entries to have the same value in that column. It also speeds up join operations based on that column.

In the case of a lookup table, there is no single column that we want to force to have unique values. Each joke ID may appear more than once, since a joke may belong to more than one category, and each category ID may appear more than once, since a category may contain many jokes. What we don't want to allow is the same pair of values appearing in the table twice. And since the sole purpose of this table is to facilitate joins, the speed benefits offered by a primary key would come in very handy. For this reason, we usually create lookup tables with a multi-column primary key as follows:

```
mysql> CREATE TABLE JokeLookup (
    -> JID INT NOT NULL,
    -> CID INT NOT NULL,
    -> PRIMARY KEY(JID,CID)
    -> );
```

This creates our table with the `JID` and `CID` columns together forming the primary key. This enforces the uniqueness that is appropriate to a lookup table (preventing a particular joke from being assigned to a particular category more than once), and speeds up joins that make use of this table.

With our lookup table in place and containing some category assignments, we can use joins to create several interesting and very practical queries. The following query lists all jokes in the "Knock-Knock" category:

```
mysql> SELECT JokeText
    -> FROM Jokes, Categories, JokeLookup
    -> WHERE Name="Knock-Knock" AND
    -> CID=Categories.ID AND JID=Jokes.ID;
```

The following query lists the categories that jokes beginning with "How many lawyers..." belong to:

```
mysql> SELECT Categories.Name
    -> FROM Jokes, Categories, JokeLookup
    -> WHERE JokeText LIKE "How many lawyers%"
    -> AND CID=Categories.ID AND JID=Jokes.ID;
```

The following query, which also makes use of our `Authors` table to form a join of four tables (!!!), lists the names of all authors that have written Knock-Knock jokes:

```
mysql> SELECT Authors.Name
    -> FROM Jokes, Authors, Categories, JokeLookup
    -> WHERE Categories.Name="Knock-Knock"
    -> AND CID=Categories.ID AND JID=Jokes.ID
    -> AND AID=Authors.ID;
```

Wrap-up

This week, we looked at the fundamentals of good database design, and learned how MySQL (and, for that matter, all relational database management systems) provide support for representing different types of relationships between entities. From our meager understanding of one-to-one relationships, we have expanded our knowledge to include many-to-one, one-to-many, and many-to-many relationships.

In the process, we learned a few new features of some common SQL commands. In particular, we learned how to use a `SELECT` to join data in multiple tables into a single set of results.

In Part Six, we use all the knowledge we have gained so far, plus a few new tricks, to build a basic content management system in PHP. The aim of such a system is to provide a customized, secure, Web-based interface for managing the contents of the database instead of having to type everything in by hand on the MySQL command line.

Part 6: A Content Management System

So far, we've seen several examples of database-driven Web pages: pages that display information culled from a MySQL database when the page is requested. Until now, however, we haven't seen a solution that would be much more manageable than raw HTML files when scaled up to encompass a Web site as large and complex as SitePoint.com. Sure, our Internet Joke Database was nice, but when it came to managing categories and authors, we would always be returning to the MySQL command line, trying to remember complicated `SELECT` and `INSERT` statements, as well as table and column names, to accomplish the most menial of tasks.

To make the leap from a Web page that displays information stored in a database to a complete database-driven Web site, we need to add a content management system. Such a system usually takes the form of a series of Web pages, access to which is restricted to users authorized to make changes to the Web site. These pages provide a database administration interface, allowing a user to easily view and change the information stored in the database without messing with the mundane details of SQL syntax.

The beginnings of a content management system were seen at the end of Part Four, where we allowed site visitors to add jokes to and (if you worked through the challenge) delete jokes from the database using a Web-based form and a "delete this joke" link, respectively. While impressive, these are not features you would normally include in the interface presented to casual site visitors. For example, you don't want someone to be able to add offensive material to your Web site without your knowledge. And you definitely don't want just anyone to be able to delete jokes from your site.

By relegating those "dangerous" features to the restricted-access site administration pages, you avoid the risk of exposing your data to just anyone while maintaining the power of being able to manage the contents of your database without having to memorize SQL queries. In this part of the series, we'll expand on the capabilities of our joke management system to take advantage of some of the enhancements we made to our database in Part Five. Specifically, we'll allow a site administrator to manage authors and categories, and assign these to corresponding jokes.

As explained above, these administration pages must be protected by an appropriate access restriction scheme. Placing the corresponding PHP files into a directory protected by an Apache-style .htaccess file listing authorized users would be one way of doing this. Consult your Web server's documentation for information on restricting access to Web pages.

Since we'll be working with some fairly large PHP files in this part, it will be necessary to gloss over some of the details due to space constraints. The complete code of all the files discussed in this part of the series, which together form the complete joke management system, is provided in a source code archive, which may be downloaded by clicking here (code.zip - 9.8KB).

The Front Page

As of the end of Part Five, our database contained tables for three types of entities: jokes, authors, and joke categories. Note that we are sticking with our original assumption of one email address per author, so we won't have a separate table for email addresses. The front page of our content management system, therefore, will contain links to pages for managing each of these three things:

```
<!-- admin.html -->
<HTML>
<HEAD>
<TITLE>JMS</TITLE>
</HEAD>
<BODY>
<H1>Joke Management System</H1>
<UL>
  <LI><A HREF="jokes.php">Manage Jokes</A>
  <LI><A HREF="authors.php">Manage Authors</A>
  <LI><A HREF="cats.php">Manage Joke Categories</A>
</UL>
</BODY>
</HTML>
```

Managing Authors

Let us begin with `authors.php`, the file responsible for letting administrators add new authors, and delete and edit existing ones. If you're comfortable with the idea of multipurpose pages, you'll probably want to place the code for all of this into the single file, `authors.php`. Since the code for this file would be fairly long, we'll be using separate files in our examples to break up the code a little.

The first thing we wish to present to an administrator wishing to manage authors is a listing of all authors currently stored in the database. Code-wise, this is the same thing as listing the jokes in the database. Since we want to allow deleting and editing existing authors, we'll include links for these functions next to each author's name. Just like the "Delete this Joke" links in the challenge at the end of Part Four, these links will have the ID of the author attached to them so that the target document knows which author you wish to edit or delete. Finally, we shall provide a "Create New Author" link that leads to a form similar in operation to the "Add a Joke" link we created in Part Four.

```
<!-- authors.php -->
<HTML>
<HEAD>
```

```
<TITLE> Manage Authors </TITLE>
</HEAD>
<BODY>
<H1>Manage Authors</H1>
<P ALIGN=CENTER><A HREF="newauthor.php">
Create New Author</A></P>
<UL>
<?php
$cnx = mysql_connect('localhost','user','password');
mysql_select_db('jokes');
$authors = mysql_query("SELECT ID, Name FROM Authors");
if (!$authors) {
  echo("<P>Error retrieving authors from database!<BR>".
       "Error: " . mysql_error());
  exit();
}
while ($author = mysql_fetch_array($authors)) {
  $id   = $author["ID"];
  $name = $author["Name"];
  echo("<LI>$name ".
       "[<A HREF='editauthor.php?id=$id'>Edit</A>|".
       "<A HREF='deleteauthor.php?id=$id'>Delete</A>]");
}
?>
</UL>
<P ALIGN=CENTER><A HREF="admin.html">
Return to Front Page</A></P>
</BODY>
</HTML>
```

Deleting Authors

`deleteauthor.php` is responsible for removing an author from the database given its ID. As we have seen before, this is frighteningly easy to do with a `DELETE` query, but there is added complexity here. Remember that our `Jokes` table has an `AID` column that indicates the author responsible for a given joke. When removing an author from the database, we must also get rid of any references to that author in other tables. If we didn't, then the next author to be added to the database might get the same ID as the now-deleted author, and the jokes belonging to the deleted author would then incorrectly point to the new author.

We have two choices in handling this situation:

- When deleting an author, also delete any jokes attributed to the author.
- When deleting an author, set the `AID` of any jokes attributed to the author to `NULL`, indicating that they have no author.

Since most authors would not like us using their jokes without giving them credit, we'll opt for the first option. This also saves us from having to handle jokes with `NULL` values in their `AID` column when displaying our library of jokes.

```
<!-- deleteauthor.php -->
<HTML>
<HEAD>
<TITLE> Delete Author </TITLE>
</HEAD>
<BODY>
<?php
$cnx = mysql_connect('localhost','user','password');
mysql_select_db('jokes');
// Delete all jokes belonging to the author
// along with the entry for the author.
```

```
$ok1 = mysql_query("DELETE FROM Jokes WHERE AID=$id");
$ok2 = mysql_query("DELETE FROM Authors WHERE ID=$id");
if ($ok1 and $ok2) {
  echo("<P>Author deleted successfully!</P>");
} else {
  echo("<P>Error deleting author from database!<BR>".
       "Error: " . mysql_error());
}
?>
<P><A HREF="authors.php">Return to Authors list</A></P>
</BODY>
</HTML>
```

Adding Authors

Next comes `newauthor.php`, which allows administrators to add new authors to the database. Again, this is just like adding new jokes, which we tackled in Part Four.

```
<!-- newauthor.php -->
<HTML>
<HEAD>
<TITLE> Add New Author </TITLE>
</HEAD>
<BODY>
<?php
if ($submit): // A new author has been entered
              // using the form below.
  $dbcnx = @mysql_connect(
             "localhost", "root", "mypasswd");
  mysql_select_db("jokes");
  $sql = "INSERT INTO Authors SET " .
         "Name='$name', " .
         "EMail='$email'";
  if (mysql_query($sql)) {
    echo("<P>New author added</P>");
  } else {
    echo("<P>Error adding new author: " .
         mysql_error() . "</P>");
  }
?>
<P><A HREF="<?php echo($PHP_SELF); ?>">
Add another Author</A></P>
<P><A HREF="authors.php">Return to Authors list</A></P>
<?php
  else: // Allow the user to enter a new author
?>
<FORM ACTION="<?php echo($PHP_SELF); ?>" METHOD=POST>
<P>Enter the new author:<BR>
Name: <INPUT TYPE=TEXT NAME="name" SIZE=20
MAXLENGTH=100><BR>
eMail: <INPUT TYPE=TEXT NAME="email" SIZE=20
MAXLENGTH=100><BR>
<INPUT TYPE=SUBMIT NAME="submit" VALUE="SUBMIT"></P>
</FORM>
<?php endif; ?>
</BODY>
</HTML>
```

Editing Authors

All that's left is `editauthor.php`, which must provide an interface for editing an existing author's details. This page will actually be very similar to `newauthor.php`, except the form fields will initially contain the values stored in the database, and an `UPDATE` query will be used instead of an `INSERT` query when the form is submitted.

One minor complication comes into play, here. To initialize the form fields with the values stored in the database, the page will obviously use the `$id` variable passed from `authors.php` to retrieve the values and store them in PHP variables (say, `$name` and `$email`). The code for our form should then look like this:

```
<FORM ACTION="<?php echo($PHP_SELF); ?>" METHOD=POST>
<P>Edit the author:<BR>
Name: <INPUT TYPE=TEXT NAME="name" VALUE="<?php echo($name); ?>" SIZE=20
MAXLENGTH=100><BR>
eMail: <INPUT TYPE=TEXT NAME="email" VALUE="<?php echo($email); ?>"
SIZE=20 MAXLENGTH=100><BR>
<INPUT TYPE=HIDDEN NAME="id" VALUE="<?php echo($id); ?>">
<INPUT TYPE=SUBMIT NAME="submit" VALUE="SUBMIT"></P>
</FORM>
```

As an aside, notice the hidden form field, which we use to pass along the author's ID with the updated values when the form is submitted.

But consider what happens if the author's name was `"The Jokester"` (with the quotes). The input tag produced by the PHP script would look like this:

```
<INPUT TYPE=TEXT NAME="name" VALUE=""The Jokester"" SIZE=20
MAXLENGTH=100>
```

Obviously, this is invalid HTML. We need to 'escape' the quotes in the name with backslashes so that Web browsers know that the quotes are part of the value, and do not represent the beginning or end of an attribute value:

```
<INPUT TYPE=TEXT NAME="name" VALUE="\"The Jokester\"" SIZE=20
MAXLENGTH=100>
```

PHP provides a function called `addslashes` that automatically adds backslashes where they are needed (specifically, in front of special characters like quotes and apostrophes). Using this function on the values of all variables that you retrieve from a database and plan to use within an HTML attribute value will let you avoid problems with quotes in your text strings.

```
$name = addslashes($name);
```

It may occur to you that we've used variable values in SQL queries before. Consider the following SQL `INSERT` command:

```
mysql> INSERT INTO Authors SET
    -> Name='Jennifer O'Reilly',
    -> eMail='jen@hotmail.com';
```

Obviously, the apostrophe in this author's surname will cause problems here. Why haven't we worried about this problem before now? PHP has a nifty little feature called "magic quotes", which we enabled with the following setting in your `php.ini` (or `php3.ini`) file:

```
magic_quotes_gpc = On
```

This setting basically tells PHP to automatically use the `addslashes` function on any variables that are passed with the request for the page. The `gpc` stands for "get, post, cookies", which are the three methods by which information may be passed with a request for a Web page. Now, since all the values we've inserted into our database up until now have been passed as part of a form submission, the Magic Quotes feature of PHP has automatically added slashes to them every time. Values retrieved from a MySQL database, however, do not benefit from the Magic Quotes feature, and so must have slashes added before they are used in any situation where quotes, apostrophes and other special characters may be a problem.

While we're on the subject of troublesome special characters, there is another set of characters that can cause problems. Specifically, HTML tag delimiters such as < and > can wreak havoc when they appear in a piece of text to be output to a Web page. Again, a function is provided for converting these to a 'safe' format. The function is called `htmlspecialchars`, and is used in exactly the same way as `addslashes`:

```
$text = htmlspecialchars("<HTML> can be dangerous!");
echo($text); // output: <HTML> can be dangerous!
```

With these issues in mind, we can now create `editauthor.php`, the complete code for which is provided in the source code archive (code.zip - 9.8KB).

Managing Categories

When you compare the roles of authors and joke categories in the database, they are really very similar. They both reside in tables of their own, and they both serve to group jokes together in some way. As a result, categories can be handled with almost the exact same code as we have developed for authors, with one important exception.

When deleting a category, we cannot also delete any jokes belonging to that category, since those jokes may also belong to other categories. We could check each joke to see if it belonged to any other categories and only delete those that did not, but rather than engage in such a time-consuming process, let's allow for the possibility of having jokes in our database that don't belong to any category at all. These jokes would be invisible to visitors to our site, but would remain in the database in case we wanted to assign them to a category later on.

Thus, to delete a category, we also need to delete any entries in the `JokeLookup` table that refer to that category:

```
<!-- deletecat.php -->
...
// Delete all joke lookup entries for the
// category along with the entry for the category.
$ok1 = mysql_query(
         "DELETE FROM JokeLookup WHERE CID=$id");
$ok2 = mysql_query(
         "DELETE FROM Categories WHERE ID=$id");
...
```

Other than this one detail, category management is functionally identical to author management. The code for `cats.php`, `newcat.php`, `deletecat.php`, and `editcat.php` is provided in the source code archive (code.zip - 9.8KB) if you need it.

Managing Jokes

In addition to adding, deleting, and modifying jokes in our database, we also need to support assigning categories and authors to our jokes. Furthermore, we are likely to have many more jokes than authors or categories. As a result, displaying a complete list of jokes as we did for the authors and categories could result in an unmanageably long list with no easy way to spot the one joke we are after. We need to provide a more intelligent method of browsing our library of jokes.

Since at different times we may know the category, author, or some of the text in a joke we wish to work with, let us support all of these methods for locating jokes in our database. The form that will prompt the administrator for information about the desired joke must present lists of categories and authors. The code for this is as follows:

```
<!-- jokes.php -->
<HTML>
<HEAD>
<TITLE> Manage Jokes </TITLE>
</HEAD>
<BODY>
<H1>Manage Jokes</H1>
<P><A HREF="newjoke.php">Create New Joke</A></P>
<?php

$dbcnx = @mysql_connect(
         "localhost", "root", "mypasswd");
mysql_select_db("jokes");
$authors = mysql_query(
             "SELECT ID, Name FROM Authors");
$cats    = mysql_query(
             "SELECT ID, Name FROM Categories");
?>
<FORM ACTION="jokelist.php" METHOD=POST>
<P>View jokes satisfying the following criteria:<BR>
By Author:
<SELECT NAME="aid" SIZE=1>
  <OPTION SELECTED VALUE="">Any Author
<?php
  while ($author = mysql_fetch_array($authors)) {
    $aid = $author["ID"];
    $aname = $author["Name"];
    echo("<OPTION VALUE='$aid'>$aname\n");
  }
?>
</SELECT><BR>
By Category:
<SELECT NAME="cid" SIZE=1>
  <OPTION SELECTED VALUE="">Any Category
<?php
  while ($cat = mysql_fetch_array($cats)) {
    $cid = $cat["ID"];
    $cname = $cat["Name"];
    echo("<OPTION VALUE='$cid'>$cname\n");
  }
?>
</SELECT><BR>
Containing Text:
<INPUT TYPE=TEXT NAME="searchtext"><BR>
<INPUT TYPE=SUBMIT NAME="submit" VALUE="Search">
</FORM>
<P ALIGN=CENTER><A HREF="admin.html">
Return to Front Page</A></P>
</BODY>
</HTML>
```

Note that the `\n` at the end of the strings being output using the `echo` function is the special code for a new line, which serves to make the HTML code output by this script more readable.

Managing Jokes (cont'd.)

It's up to `jokelist.php` to use the values submitted in the form to build a list of jokes satisfying the criteria specified. Obviously, this will be done with a `SELECT` query, but the exact form of that query will depend on what was entered using the form we just defined. Since building this `SELECT` statement is a fairly complicated process, let's work through `jokelist.php` a little at a time.

First, we get the preliminaries out of the way:

```
<!-- jokelist.php -->
<HTML>
<HEAD>
<TITLE> Manage Jokes </TITLE>
</HEAD>
<BODY>
<H1>Manage Jokes</H1>
<P><A HREF="jokes.php">New Search</A></P>
<?php

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

Now, we start by defining a few strings that, when strung together, form the `SELECT` query that we would need if no constraints were selected in the form:

```
// The basic SELECT statement
$select = "SELECT DISTINCT ID, JokeText";
$from = " FROM Jokes";
$where = " WHERE ID > 0";
```

The `WHERE` clause in the above might be somewhat confusing. The idea here is for us to be able to build on this basic `SELECT` statement depending on what constraints were selected in the form. Such constraints will require us to add to the `FROM` and `WHERE` clauses (portions) of the `SELECT` statement. But if there were no constraints specified (i.e. the administrator wanted a list of all jokes in the database), there would be no need for a `WHERE` clause at all! Since it's difficult to add to a `WHERE` clause that doesn't exist, we needed to come up with a "do nothing" `WHERE` clause. Requiring `Jokes.ID` to be greater than zero fits the bill nicely, since the `AUTO_INCREMENT` feature of MySQL (which is responsible for assigning the values in this column) will always choose integers greater than zero.

Our next task is to check each of the possible constraints (author, category, and search text) that may have been set in the form, and adjust the SQL accordingly. First, we deal with the possibility that an author was specified. The "Any Author" option in the form was given a value of "" (the empty string), so if the value of that form field (stored in `$aid`) is not equal to `""`, then an author was specified and we adjust our query:

```
if ($aid != "") { // An author is selected
  $where .= " AND AID=$aid";
}
```

The `.=` operator is used to tack a new string onto the end of an existing one. In this case, we are adding to the `WHERE` clause the condition that the `AID` in the `Jokes` table must match the author ID selected in the form (`$aid`).

Next, we handle a joke category being specified:

```
if ($cid != "") { // A category is selected
  $from  .= ", JokeLookup";
  $where .= " AND ID=JID AND CID=$cid";
}
```

Since the categories associated with a particular joke are stored in the `JokeLookup` table, we need to add this table to the query to create a join by tacking the name of the table onto the end of the `$from` variable. To complete the join, we also specify that the `ID` column (in the `Jokes` table) must match the `JID` column (in `JokeLookup`), by adding this condition to the `$where` variable. Finally, we require the `CID` column (in `JokeLookup`) to match the category ID selected in the form (`$cid`).

Handling search text is fairly simple. We just use the `LIKE` SQL operator that we learned way back in Part Two:

```
if ($searchtext != "") { // Search text specified
  $where .= " AND JokeText LIKE '%$searchtext%'";
}
```

With our SQL query built, we can use it to retrieve our jokes and display them, along with links for editing and deleting them (just like we did for authors and joke categories). For readability, we display our jokes in an HTML table:

```
?>
<TABLE BORDER=1>
<TR><TH>Joke Text</TH><TH>Options</TH></TR>
<?php
$jokes = mysql_query($select . $from . $where);
if (!$jokes) {
  echo("</TABLE>");
  echo("<P>Error retrieving jokes from database!<BR>".
       "Error: " . mysql_error());
  exit();
}
while ($joke = mysql_fetch_array($jokes)) {
  echo("<TR>\n");
  $id       = $joke["ID"];
  $joketext = $joke["JokeText"];
  echo("<TD>$joketext</TD>\n");
  echo("<TD>[<A HREF='editjoke.php?id=$id'>".
       "Edit</A>|".
       "<A HREF='deletejoke.php?id=$id'>".
       "Delete</A>]</TD>\n");
  echo("</TR>\n");
}
?>
</TABLE>
</BODY>
</HTML>
```

Managing Jokes (cont'd.)

With `jokelist.php` out of the way, let's tackle `newjoke.php`, which is linked-to at the top of `jokes.php`. This page will be very similar to `newauthor.php` and `newcat.php`. In addition to specifying the joke text, however, this page must allow an administrator to assign an author and categories to a joke. These features make the code of this file worth some examination.

We know from seeing the code of `newauthor.php` that the PHP code for processing the form submission comes before the form code itself (it doesn't have to, but this is the layout we've used so far). Let's begin by looking at the form code, however, so that the code for handling submission of the form makes more sense.

We begin by fetching lists of all the authors and categories in the database:

```
<?php
  else:
  $dbcnx = @mysql_connect(
             "localhost", "root", "mypasswd");
  mysql_select_db("jokes");
  $authors = mysql_query(
               "SELECT ID, Name FROM Authors");
  $cats    = mysql_query(
               "SELECT ID, Name FROM Categories");
?>
```

Next, we create our form. We begin with a standard text area for typing in the text of the joke:

```
<FORM ACTION="<?php echo($PHP_SELF); ?>" METHOD=POST>
<P>Enter the new joke:<BR>
<TEXTAREA NAME="joketext" ROWS=15 COLS=45 WRAP>
</TEXTAREA>
```

We'll prompt the administrator to select an author using a drop-down list of the authors we found in the database:

```
<P>Author:
<SELECT NAME="aid" SIZE=1>
  <OPTION SELECTED VALUE="">Select One
  <OPTION VALUE="">---------
<?php
  while ($author = mysql_fetch_array($authors)) {
    $aid = $author["ID"];
    $aname = $author["Name"];
    echo("<OPTION VALUE='$aid'>$aname\n");
  }
?>
</SELECT></P>
```

A drop-down list will not suffice for selecting categories, since we want to allow the administrator to select multiple categories. Thus, we'll use a series of checkboxes -- one for each category. The checkboxes are given the names `cat1`, `cat2`, `cat3`, and so on using the category ID to which each checkbox refers. The checkboxes are labeled with the names of the categories.

```
<P>Place in categories:<BR>
<?php
  while ($cat = mysql_fetch_array($cats)) {
    $cid = $cat["ID"];
    $cname = $cat["Name"];
    echo("<INPUT TYPE=CHECKBOX NAME='cat$cid'>".
         "$cname<BR>\n");
  }
?>
</P>
```

And we finish off our form as usual:

```
<P><INPUT TYPE=SUBMIT NAME="submit" VALUE="SUBMIT"></P>
</FORM>
```

```
<?php endif; ?>
```

Processing this form is not totally straightforward, so we'll examine the code responsible for that as well. It starts off pretty simply, as we add the joke to the `Jokes` table. Since an author is required, we make sure that `$aid` contains a value. This prevents the administrator from choosing the "Select One" option in the author select list, since that choice has a value of `""` (the empty string).

```
<?php
if ($submit): // A new joke has been entered
              // using the form.
  if ($aid == "") {
    echo("<P>You must choose an author " .
         "for this joke. Click 'Back' " .
         "and try again.</P>");
    exit();
  }
  $dbcnx = @mysql_connect(
             "localhost", "root", "mypasswd");
  mysql_select_db("jokes");
  $sql = "INSERT INTO Jokes SET " .
         "JokeText='$joketext', " .
         "AID='$aid'";
  if (mysql_query($sql)) {
    echo("<P>New joke added</P>");
  } else {
    echo("<P>Error adding new joke: " .
         mysql_error() . "</P>");
  }
  $jid = mysql_insert_id();
```

The last line in the above code uses a function that we have not seen before: `mysql_insert_id`. This function returns the number assigned to the last-inserted entry by the `AUTO_INCREMENT` feature in MySQL. In other words, it retrieves the ID of the newly inserted joke, which we'll need later.

Managing Jokes (cont'd.)

The code for adding the entries to `JokeLookup` based on which checkboxes were checked is not so simple. First of all, we have never seen how a checkbox passes its value to a PHP variable before. Also, we need to deal with the fact that we don't know in advance how many checkboxes there were in the form (since the number of categories in the database is not fixed).

A checkbox will pass its value to a PHP variable if it is checked, and will do nothing when it is unchecked. In our form above, we didn't assign values to any of the checkboxes. Checkboxes without assigned values pass `"on"` as the value of their corresponding variables when they are checked. Since PHP considers any string value "true" when it is used as a condition in an `if` statement, and an empty (or unassigned) variable "false", we can just use the checkbox variables as they are to test if the checkboxes were checked or not.

As for handling the issue of an unknown number of checkboxes, it's best to explain how the code works line by line. First, we retrieve a list of all categories in the database, along with their ID's:

```
  $cats = mysql_query(
            "SELECT ID, Name FROM Categories");
```

Since this same list was used to create the checkboxes, it makes sense that we would use it here to process them. We use a `while` loop to step through this list as usual:

```
  while ($cat = mysql_fetch_array($cats)) {
    $cid = $cat["ID"];
    $cname = $cat["Name"];
```

For each category in the list, we want to use the corresponding checkbox variable to determine whether or not to add the new joke to that category. The problem here is that the name of that checkbox variable depends on the ID of the category to which it refers. We must therefore construct the name of our variable using the category ID that we are interested in (`$cid`). Here's the code:

```
    $var = "cat$cid"; // The name of the variable
    if ($$var) { // The checkbox is checked
```

That double dollar sign is not a typo. The variable named `$var` will have a value of `"cat#"`, where `#` is the ID of the current category. The value of `$$var`, then, will be the value of the variable named `$cat#`. This is a pretty obscure feature of PHP called "variable variables", which is only ever really useful in situations like these. Anyway, we use `$$var` in the `if` statement above to insert an entry into `JokeLookup` when the checkbox has been checked:

```
      $sql = "INSERT IGNORE INTO JokeLookup " .
             "SET JID=$jid, CID=$cid";
      $ok = mysql_query($sql);
      if ($ok) {
        echo("<P>Joke added to category: $cname</P>");
      } else {
        echo("<P>Error inserting joke ".
             "into category $cname:" .
             mysql_error() . "</P>");
      }
    } // end of if ($$var)
  } // end of while loop
?>
<P><A HREF="<?php echo($PHP_SELF); ?>">
Add another Joke</A></P>
<P><A HREF="jokes.php">Return to Joke Search</A></P>
```

The word `IGNORE` in the `INSERT` query used here is a precaution only. Recall that when we defined the `JokeLookup` table we set the `JID` and `CID` columns to be the primary key for the table. If somehow the `JID`/`CID` pair that is being inserted already exists in the table, attempting to insert it again would normally cause an error. By adding `IGNORE` to the command, a re-insert of the same pair is simply ignored by MySQL and no error occurs. This situation should never actually happen, but it's better to be safe than sorry.

The remaining two files, `editjoke.php` and `deletejoke.php` mirror their author and category counterparts, with minor adjustments. `editjoke.php` must make provide the same author select box and category checkboxes as `addjoke.php`, except this time they must be initialized to reflect the values stored in the database for the joke selected. `deletejoke.php`, meanwhile, must not only delete the selected joke from the `Jokes` table, but must also remove any entries in the `JokeLookup` table for that joke. The code for both of these files is provided in the source code archive (code.zip - 9.8KB), but we will not spend time examining the details, since these files are just an application of skills that should be fairly familiar to you by now.

Wrap-up

There are a few minor things that our content management system is still not able to do. For example, it is currently not able to provide a listing of all jokes that do not belong to any category -- something that could come in very handy as the jokes in the database grow in number. You might also like to sort joke listings by various criteria. These particular capabilities require a few more advanced SQL tricks that we will see later.

Ignoring these little details for the moment, we now have a system that allows someone with no SQL or database knowledge to administer our database of jokes with ease! Together with a set of PHP-powered pages for regular site visitors to view the jokes, this content management system allows us to set up a complete database-driven Web site that can be maintained by someone with absolutely no database knowledge. If that sounds like a valuable commodity to businesses looking to get on the Web today, you're right!

In fact, only one aspect of our site requires special knowledge (beyond the use of a Web browser) to use: content formatting. It would not be surprising, for example, for someone to want to enter a joke that contained more than one paragraph of text. In our current system, this could be accomplished by entering the HTML code for the joke directly into the "Create New Joke" form. Why is this unacceptable?

As we stated way back in the introduction to this series, one of the most desirable features of a database-driven Web site is that the people responsible for adding content to the site need not be familiar with HTML. If we require knowledge of HTML for something as simple as dividing a joke into paragraphs, we have failed to reach our goal.

In Part Seven, we'll see how we can make use of some features of PHP to provide a simpler means of formatting content without requiring site administrators to know the ins and outs of HTML. We'll also bring back the "Submit Your Own Joke" link to our site by finding out how we can safely accept content submissions from casual site visitors.

Part 7: Content Formatting and Submission

We're almost there. We've designed a database for storing jokes, organizing them into categories, and tracking their authors. We've learned how to create a Web page that displays this library of jokes to site visitors. We've even developed a set of Web pages that a site administrator can use to manage the joke library without having to know anything about databases.

In so doing, we've removed the headaches of continually plugging new content into a tired HTML page template, and creating an unmanageable mass of HTML files. The HTML is now kept completely separate from the data it displays. If you want to redesign the site, you just have to make the changes to the HTML contained in the PHP files that site visitors see. A change to one file (e.g. changing a font) is immediately reflected in the page layouts of all jokes, because all jokes are displayed using that single PHP file. Only one task still requires HTML to enter into the equation for managing the content of the Web site: content formatting.

On any but the simplest of Web sites, it will be necessary to allow content (in this case, jokes) to have formatting applied to them. In the simple case, this may just be the ability to break text into paragraphs. Often, however, content providers will expect facilities such as boldfaced or italicized text, hyperlinks, etc.

Our current database and site design supports all of this, since a site administrator can include HTML tags in the text of a joke, and these will have their usual effects when the joke text is inserted into the page that a site visitor's browser requests. However, to achieve our goal of eliminating HTML from the system entirely, we must provide some other way of formatting text.

In this part of our series, we'll learn some new PHP functions that will enable us to provide basic text formatting without the use of HTML. In so doing, we'll have completed a content management system easy enough for anyone with a Web browser to use. We'll then take full advantage of this ease of use by allowing site visitors to once again submit their own jokes -- this time without the risk of our site becoming filled with obscene or otherwise inappropriate material.

Out with the Old...

Before we provide a new method of formatting text, we should first disable the old one. Someone with no knowledge of HTML might unknowingly include HTML syntax (however invalid) in a plain text document that could produce unexpected results, or even mess up your finely tuned page layout. Consider the following sentence:

```
The gunman drew his weapon. <BANG!>
```

Someone entering the above text into the database might be surprised to see the last word (`<BANG!>`) missing from the Web page displaying the content. Anyone with a basic knowledge of HTML would know that the Web browser has discarded that segment of text as an invalid HTML tag, but we're trying to cater to users with no knowledge of HTML whatsoever.

In Part Five, we saw a PHP function that solved this problem quite neatly: `htmlspecialchars`. This function, if applied to the text of our joke before it was inserted into a Web page, would convert the string above into the following "HTML safe" version:

```
The gunman drew his weapon. &lt;BANG!&gt;
```

When interpreted by the site visitor's Web browser, this would produce the desired result. As a first step, therefore, we must modify the PHP file on our Web site responsible for displaying the text of jokes so that it uses `htmlspecialchars` on that text before outputting it to the Web:

```
<!-- joke.php -->
...
// Get the joke text from the database
$joke = mysql_query("SELECT JokeText FROM Jokes ".
                    "WHERE ID=$id");
$joke = mysql_fetch_array($joke);
$joketext = $joke["JokeText"];
// Filter out HTML code
$joketext = htmlspecialchars($joketext);
echo( $joketext );
...
```

We have now neutralized any HTML code appearing in the site content. With this clean slate, we are ready to implement our own content formatting method by implementing a markup language of our very own.

Regular Expressions

Implementing our own markup language will involve spotting our custom tags in the text of jokes and replacing them with their HTML equivalents before outputting the joke text to the user's browser. Anyone with experience using regular expressions knows that they are ideal for this sort of work.

A regular expression is a string if text containing special codes that allow it to be used with a few PHP functions for searching and manipulating text. The following, for example, is a regular expression that searches for the text "PHP" (without the quotes):

```
PHP
```

Not much to it, is there? To use a regular expression, you must be familiar with the regular expression functions available in PHP. `ereg` is the most basic, and can be used to determine whether a regular expression is "satisfied" by a particular text string. Consider the following code:

```
$text = "PHP rules!";
if (ereg("PHP", $text)) {
  echo( '$text contains the string "PHP".' );
} else {
  echo( '$text does not contain the string "PHP".' );
}
```

In this example, the regular expression is satisfied because the string stored in variable `$text` contains "PHP". The above code will thus output the following (note that the single quotes prevent PHP from filling in the value of the variable `$text`):

```
$text contains the string "PHP".
```

`eregi` is a function that behaves almost identically to `ereg`, except it ignores the case of text when looking for matches:

```
$text = "What is Php?";
if (eregi("PHP", $text)) {
  echo( '$text contains the string "PHP".' );
} else {
  echo( '$text does not contain the string "PHP".' );
}
```

Again, this outputs the same message:

```
$text contains the string "PHP".
```

As was mentioned above, there are special codes that may be used in regular expressions. Some of these can be downright confusing and difficult to remember, so if you intend to make extensive use of them you may wish to find a good reference for yourself. A tutorial-style reference to standard regular expression syntax may be found at http://www.delorie.com/gnu/docs/rx/rx_toc.html, and the book Professional PHP Programming by Wrox Press contains a regular expression syntax reference in its appendices. Let's work our way through a few examples to learn the basic regular expression syntax.

First of all, a caret (`^`) may be used to indicate the beginning of the string, while a dollar sign (`$`) is used to indicate the end:

```
PHP         // Matches "What is PHP?"
^PHP        // Matches "PHP rules!" but not "What is PHP?"
```

```
PHP$        // Matches "I love PHP" but not "What is PHP?"
^PHP$       // Matches "PHP" but nothing else
```

Obviously, you may sometimes want to use `^`, `$`, or other special characters to represent the corresponding character in the search string rather than the special meaning implied by regular expression syntax. To remove the special meaning of a character, prefix it with a backslash:

```
\$\$\$      // Matches "Show me the $$$!"
```

Square brackets may be used to define a set of characters that may match. For example, the following regular expression will match any digit from 1 to 5 inclusive.

```
[12345]     // Matches "1" and "3", but not "a" or "12"
```

Ranges of numbers and letters may also be specified.

```
[1-5]       // Same as previous
[a-z]       // Matches any lowercase letter
[0-9a-zA-Z] // Matches any letter or digit
```

The characters `?`, `+`, and `*` also have special meanings. Specifically, `?` means "the preceding character is optional", `+` means "one or more of the previous character", and `*` means "zero or more of the previous character".

```
bana?na     // Matches "banana" and "banna",
            // but not "banaana".
bana+na     // Matches "banana" and "banaana",
            // but not "banna".
bana*na     // Matches "banna", "banana", and "banaaana",
            // but not "bnana".
^[a-zA-z]+$ // Matches any string of one or more
            // letters and nothing else.
```

Parentheses may be used to group strings together to apply `?`, `+`, or `*` to them as a whole.

```
ba(na)+na   // Matches "banana" and "banananana",
            // but not "bana" or "banaana".
```

Here are a few codes for matching special characters in regular expressions:

```
\n          // Matches a newline character
.           // Matches any character except a newline
\r          // Matches a carriage return character
\t          // Matches a tab character
```

There are more special codes and syntax tricks for regular expressions, all of which should be covered in any reference (such as those mentioned above). For now, we have more than enough for our purposes.

String Replacement with Regular Expressions

Using `ereg` or `eregi` with the regular expression syntax we have just learned, we can easily detect the presence of tags in a given text string. What we need to do, however, is to pinpoint those tags and replace them with appropriate HTML tags. To do this, we need to look at a couple more regular expression functions offered by PHP: `ereg_replace` and `eregi_replace`.

`ereg_replace`, like `ereg`, accepts a regular expression and a string of text and attempts to match the regular expression in the string. In addition, however, `ereg_replace` takes a second string of text, and replaces every match of the regular expression with that string.

The syntax for `ereg_replace` is as follows:

```
$newstring = ereg_replace(<regexp>, <replacewith>, <oldstring>);
```

Where `<regexp>` is the regular expression, and `<replacewith>` is the string that will replace matches to `<regexp>` in `<oldstring>`. The function returns the new string that is the outcome of the replacement operation. In the above, this gets stored in `$newstring`.

`eregi_replace`, as expected, is identical to `ereg_replace`, except the case of letters is not considered when searching for matches.

We're now ready to start building our custom markup language.

Boldface and Italicized Text

Let's start by implementing tags for creating boldfaced and italicized text. Let's say we want `[B]` to begin bold text and `[EB]` to end bold text. Obviously, we must replace `[B]` with `<B>` and `[EB]` with `</B>`. Doing this is a simple application of `eregi_replace`:

```
$joketext = eregi_replace("\[b]","<B>",$joketext);
$joketext = eregi_replace("\[eb]","</B>",$joketext);
```

Notice that since `[` normally indicates the beginning of a set of acceptable characters in a regular expression, we put a backslash before it to remove its special meaning. Without a matching `[`, the `]` loses its special meaning and doesn't need a backslash, although you could put a backslash in front of it as well if you wanted to be thorough.

Also notice that, since we are using `eregi_replace`, which is case insensitive, both `[B]` and `[b]` will work as tags in our custom markup language.

Italicized text can be done the same way:

```
$joketext = eregi_replace("\[i]","<I>",$joketext);
$joketext = eregi_replace("\[ei]","</I>",$joketext);
```

Paragraphs

While we could create tags for paragraphs just as we did for boldface and italicized text above, a simpler approach makes even more sense. Since the user will be typing the content into a form field that allows them to format text using the enter key, we shall take a single linefeed (`\n`) to indicate a line break (`<BR>`) and a double linefeed (`\n\n`) to indicate a new paragraph (`<P>`). Of course, since PC's represent new lines as a linefeed-carriage return pair (`\n\r`), we must strip out carriage returns first. The code for all this is as follows:

```
// Strip out carriage returns
$joketext = ereg_replace("\r","",$joketext);
// Handle paragraphs
$joketext = ereg_replace("\n\n","<P>",$joketext);
// Handle line breaks
$joketext = ereg_replace("\n","<BR>",$joketext);
```

That's it! The text will now appear in paragraphs as the user expects, and they don't need to learn any custom tags to do it.

Hyperlinks

While it may seem silly to support hyperlinks in the text of jokes, this feature makes plenty of sense in other applications. Hyperlinks are a little more complicated than simply converting some code to an HTML tag. We need to be able to output a URL as well as the text that should appear as the link.

Another feature of `ereg_replace` and `eregi_replace` comes into play here. By surrounding a portion of the regular expression with parentheses, you can "capture" the corresponding portion of the matched text and use it in the replace string with the code `\\n`, where `n` is 1 for the first parenthesized portion of the regular expression, 2 for the second, up to 9 for the 9th. Consider the following example:

```
$text = "banana";
$text = eregi_replace("(.*)(nana)", "\\2\\1", $text);
echo($text); // outputs "nanaba"
```

In the above, `\\1` gets replaced with `ba` in the replace string, which corresponds to `(.*)` (zero or more non-new line characters) in the regular expression. `\\2` gets replaced with `nana`, which corresponds to `(nana)` in the regular expression.

The same principle may be used to create our hyperlinks. Let's begin with a simple form of link, where the text of the link is the same as the URL. We want to support the following syntax:

```
Visit [L]http://www.php.net/[EL].
```

The corresponding HTML code, which we want to output, is as follows:

```
Visit <A HREF="http://www.php.net/">http://www.php.net/</A>.
```

First, we need a regular expression that will match links of this form. The regular expression is as follows:

```
\[L][-_./a-zA-Z0-9!&%#?,'=:~]+\[EL]
```

Again, we have put backslashes in front of the opening square brackets in `[L]` and `[EL]` to indicate that they are to be taken literally. We then use square brackets to list all the characters we wish to accept as part of the URL. We place a `+` after the square brackets to indicate that the URL will be composed of one or more characters taken from this list.

To output our link, we're going to need to capture the URL and output it both as the `HREF` attribute of the `A` tag, and as the text of the link. To capture the URL, we surround the corresponding portion of our regular expression with parentheses:

```
\[L]([-_./a-zA-Z0-9!&%#?,'=:~]+)\[EL]
```

So we do the link conversion with the following code:

```
$joketext = ereg_replace(
"\[L]([-_./a-zA-Z0-9!&%#?,'=:~]+)\[EL]",
"<A HREF=\"\\1\">\\1</A>", $joketext);
```

Note that we had to place backslashes in front of the double quotes in the HTML code for the link to keep PHP from confusing them with the quotes surrounding the replace string. Anyway, `\\1` gets replaced by the URL for the link, and the output is as expected!

We would also like to support hyperlinks that have link text that differs from their URL. Let's say the form of our link is as follows:

```
Check out [L=http://www.php.net/]PHP[EL].
```

Here's our regular expression:

```
\[L=([-_./a-zA-Z0-9!&%#?,'=:~]+)]([-_./a-zA-Z0-9 !&%#?,'=:~]+)\[EL]
```

Quite a mess, isn't it? Squint at it for a little while, and you'll see it does exactly what we need it to do, capturing both the URL (`\\1`) and the text (`\\2`) for the link. The PHP code to perform the substitution is as follows:

```
$joketext = ereg_replace(
  "\[L=([-_./a-zA-Z0-9!&%#?,'=:~]+)]".
  "([-_./a-zA-Z0-9 !&%#?,'=:~]+)\[EL]",
  "<A HREF=\"\\1\">\\2</A>", $joketext);
```

Splitting Text into Pages

While no joke is likely to be so long that it will require more than one page, many content-driven sites (like SitePoint.com!) provide lengthy content that is often best presented broken up into pages. Yet another regular expression function in PHP makes this exceedingly easy to do.

`split` is a function that takes a regular expression and a string of text and uses matches for the regular expression to break the text apart into an array. Consider the following example:

```
$regexp="[ \n\r\t]+"; // One or more whitespace characters
$text="This is a test.";
$textarray=split($regexp,$text);
echo("$textarray[0]<BR>"); // Outputs "This<BR>"
echo("$textarray[1]<BR>"); // Outputs "is<BR>"
echo("$textarray[2]<BR>"); // Outputs "a<BR>"
echo("$textarray[3]<BR>"); // Outputs "test.<BR>"
```

If instead of searching for a whitespace character we search for a `[PAGEBREAK]` tag, and instead of displaying all of the resulting portions of the text we display only the page we are interested in (indicated by a `$page` variable passed with the page request, for example), we can successfully divide our content into pages.

```
// If no page specified, default to the
// first page ($page = 0)
if (!isset($page)) $page = 0;
// Split the text into an array of pages
$textarray=split("\[PAGEBREAK]",$text);
```

```
// Select the page we want
$pagetext=$textarray[$page];
```

Of course, we'll want to provide some way of moving between pages. Let's put a link to the previous page at the top of the current page, and a link to the next page at the bottom.

If this is the first page we don't need a link to the previous page. We know we're on the first page if `$page` equals zero. Likewise, we don't need a link to the next page on the last page. To detect the last page, we need a new PHP function called `count`, which takes an array and returns the number of elements in the array. Passed our array of pages, `count` will tell us how many pages there are. If there are 10 pages, then `$textarray[9]` will contain the last page. Thus, we know we're on the last page if `$page` equals `count($textarray)` minus one.

The code for our page-turning links looks like this:

```
if ($page != 0) {
  $prevpage = $page - 1;
  echo("<P><A HREF=\"$PHP_SELF?id=$id&page=$prevpage\">".
       "Previous Page</A></P>");
}
// Output page content here...
if ($page < count($textarray) - 1) {
  $nextpage = $page + 1;
  echo("<P><A HREF=\"$PHP_SELF?id=$id&page=$nextpage\">".
       "Next Page</A></P>");
}
```

Putting it all Together

The completed code for outputting our joke text (with all special character and custom tag conversion in place) is as follows:

```
<!-- joke.php -->
...
// Get the joke text from the database
$joke = mysql_query("SELECT JokeText FROM Jokes ".
                    "WHERE ID=$id");
$joke = mysql_fetch_array($joke);
$joketext = $joke["JokeText"];
// Filter out HTML code
$joketext = htmlspecialchars($joketext);
// If no page specified, default to the
// first page ($page = 0)
if (!isset($page)) $page = 0;
// Split the text into an array of pages
$textarray=split("\[PAGEBREAK]",$joketext);
// Select the page we want
$joketext=$textarray[$page];
// Bold and italics
$joketext = eregi_replace("\[b]","<B>",$joketext);
$joketext = eregi_replace("\[eb]","</B>",$joketext);
$joketext = eregi_replace("\[i]","<I>",$joketext);
$joketext = eregi_replace("\[ei]","</I>",$joketext);
// Paragraphs and line breaks
$joketext = ereg_replace("\r","",$joketext);
$joketext = ereg_replace("\n\n","<P>",$joketext);
$joketext = ereg_replace("\n","<BR>",$joketext);
// Hyperlinks
$joketext = ereg_replace(
  "\[L]([-_./a-zA-Z0-9!&%#?,'=:~]+)\[EL]",
  "<A HREF=\"\\1\">\\1</A>", $joketext);
$joketext = ereg_replace(
  "\[L=([-_./a-zA-Z0-9!&%#?,'=:~]+)]".
  "([-_./a-zA-Z0-9 !&%#?,'=:~]+)\[EL]",
  "<A HREF=\"\\1\">\\2</A>", $joketext);
if ($page != 0) {
  $prevpage = $page - 1;
  echo("<P><A HREF=\"$PHP_SELF?id=$id&page=$prevpage\">".
       "Previous Page</A></P>");
}
echo( "<P>$joketext" );
if ($page < count($textarray) - 1) {
  $nextpage = $page + 1;
  echo("<P><A HREF=\"$PHP_SELF?id=$id&page=$nextpage\">".
       "Next Page</A></P>");
}
...
```

Don't forget to provide documentation so that users of your joke submission form know what tags are available and what they do.

Automatic Content Submission

It seems a shame to have spent so much time and effort on a content management system so easy that anyone could use it if the only people allowed using it are site administrators. Furthermore, while it is extremely convenient for an administrator not to have to edit HTML to make updates to the site's content, he or she must still transcribe submitted documents into the "Add New Joke" form and convert any text formatting into the custom formatting language we developed above, a tedious and mind-numbing task to say the least.

What if we put the "Add New Joke" form in the hands of casual site visitors? If you recall, we actually did this in Part Four by providing a form for users to submit their own jokes. At the time, this was simply a device for demonstrating how `INSERT` statements could be made from within PHP scripts. We discarded it almost immediately because of the inherent security risks involved. After all, who wants to open the content of his or her site for just anyone to tamper with?

But accepting joke submissions doesn't have to mean that those submissions appear on the site immediately. What if we added a new column to the `Jokes` table called `Visible` that could take one of two values: `'Y'` and `'N'`. Newly submitted jokes could be set to `Visible='N'` automatically, and could be excluded from appearing on the site by simply adding `WHERE Visible='Y'` to any query of the `Jokes` table for which the results are intended for public viewing. Jokes with `Visible='N'` would just sit in the database awaiting review by a content manager, who could edit each joke before making it visible, or just delete it out of hand.

Creating a table that may contain one of two values, one of which is the default, involves a new MySQL column type called `ENUM`:

```
mysql> ALTER TABLE Jokes ADD COLUMN
    -> Visible ENUM('N','Y') NOT NULL;
```

The first value listed in the parentheses (`'N'` in this case) is the default value, which is assigned to new entries if no value is specified in the `INSERT` statement.

With new jokes hidden from the public eye, the only remaining security detail becomes author identification. We want to be able to identify which author in the database submitted a particular joke, but it is inappropriate to rely on the old drop-down list of authors in the "Add New Joke" form, since any author could pose as any other. Obviously some sort of username/password authentication scheme is in order.

Storing a password in the `Authors` table is as simple as adding another column. You can then require an author to correctly enter his or her email address and password when submitting a joke to the database. You'd want to require the same login procedure before allowing an author to modify his or her details (name, email address, etc.). You might even like to give each author a "control center" of sorts, where he or she could view the status of the jokes he or she has submitted to the site.

Wrap-Up

While it would be interesting to delve into the details of the content-submission system described above, you should already have all the skills necessary to build it yourself. Want to let users rate the jokes on the site? How about letting joke authors make changes to their jokes, but requiring an administrator to approve the changes before they go live on the site? The power and complexity of the system is limited only by your imagination.

At this point in the series, you should now be equipped with all the basic skills and concepts you need to build your very own database-driven Web site. In the rest of this series, we'll be covering more advanced topics that will help make your site work better. Oh, and of course we'll be seeing more exciting features of PHP and MySQL.

In Part Eight, we'll take a step away from our Joke database and have a close-up look at MySQL server maintenance and administration. We'll learn how to make backups of our database (a critical task for any Web-based company!), administer MySQL users and their passwords, and of course we'll see how to log into a MySQL server if you've forgotten your password. :)

Part 8: MySQL Administration

At the core of any well-designed, content-driven site is a relational database. In this series, we've used the MySQL Relational Database Management System (RDBMS) to create our database. MySQL is a popular choice among Web developers not only because it is free for non-commercial use on all platforms under the GPL, but also because it is fairly simple to get a MySQL server up and running. As we demonstrated in Part One of this series, armed with proper instructions a new user can get a MySQL server up and running in less than 30 minutes (less than 10 if you practice a little!).

If all you want to do is have a MySQL server around so you can play with a few examples and experiment a little, then the initial installation process we went thought in Part One is likely to be all you'll need. If, on the other hand, you want to set up a database backend to a real, live Web site -- perhaps a site upon which your company depends -- then there are a few more things you'll need to learn how to do before relying on a MySQL server day-in and day-out.

Backups of data important to you or your business should be part of any Internet-based enterprise. Unfortunately, since setting up backups isn't the most interesting part of an administrator's duties, such procedures are usually arranged once out of necessity and can be deemed "good enough" for all applications. If your answer to "Should we be backing up our databases?" up to now has been "It's okay; they'll be backed up along with everything else." then you should really stick around. We'll be seeing why a generic file backup solution is inadequate for many MySQL installations, and we'll be demonstrating the "right way" to back up and restore a MySQL database.

In Part One, we set up the MySQL server so that you could connect as 'root' with a password of your choosing. This 'root' MySQL user (which, incidentally, has nothing to do with the Unix 'root' user) had read/write access to all databases and tables. In many organizations, it can be necessary to create other users with access to only particular databases and tables, and restrict that access in some way (e.g. read-only access to a particular table). In this part, we'll be learning how this can be done using two new MySQL commands: `GRANT` and `REVOKE`.

In some situations, such as power outages, MySQL databases can become damaged. Such damage need not always send you scrambling for your backups, however. We'll finish off our look at MySQL database administration by learning how to use the MySQL database check and repair utility to fix simple database corruptions.

Why Standard Backups aren't Enough

Like Web servers, most MySQL servers are called upon to be online 24 hours a day, 7 days a week. This makes backups of MySQL database files problematic. Because the MySQL server uses memory caches and buffers for improving the efficiency of writing updates to the database files stored on disk, these files may be in an inconsistent state at any given time. Since standard backup procedures involve just making copies of system and data files, backups of MySQL data files cannot be relied upon, since they cannot guarantee that the files that are copied are in a fit state to be used as replacements in the event of a crash.

Furthermore, since many databases will be receiving new information at all hours of the day, standard backups can only provide "snapshots" of database data. Any information added to or changed in the database after the time of the last backup will be lost in the event that the MySQL data files are destroyed or become unusable. In many situations, such as when a MySQL server is responsible for tracking customer orders on an eCommerce site, this is an unacceptable loss.

Facilities exist in MySQL for keeping up-to-date backups that are not adversely affected by server activity at the time backups are generated. Unfortunately, they require you to set up a backup scheme specifically for your MySQL data, completely apart from whatever backup measures you have established for the rest of your data. As with any good backup system, however, you'll appreciate it when the time comes to use it.

In this part of the series, the instructions we'll be providing will be designed for use on a computer running Linux, or some other Unix-based operating system. If you are running your MySQL server under Windows, the methods and advice provided here will all apply equally well, but you'll have to come up with some of the specific commands yourself. If you have any trouble, don't hesitate to post your questions in the SitePoint.com Forums.

Database Backups using `mysqldump`

In addition to `mysqld`, the MySQL server, and `mysql`, the MySQL client, a MySQL installation comes with many useful utility programs. We have seen `mysqladmin`, responsible for controlling and getting information about a running MySQL server, for example.

`mysqldump` is another such program. When run, it connects to a MySQL server (in much the same way as the `mysql` program or the PHP language does) and downloads the complete contents of the database you specify. It then outputs these as a series of SQL `CREATE TABLE` and `INSERT` commands that, if run in an empty MySQL database, would create a MySQL database with exactly the same contents as the original.

By redirecting the output of `mysqldump` to a file, you can store a "snapshot" of the database as a backup. The following command connects to the MySQL server running on `myhost` as user `root` with password `mypass` and saves a backup of the database called `dbname` into the file `dbname_backup.sql`:

```
% mysqldump -h myhost -u root -pmypass dbname > dbname_backup.sql
```

To restore this database after a server crash, you would use the following commands:

```
% mysqladmin -h myhost -u root -pmypass create dbname
% mysql -h myhost -u root -pmypass dbname < dbname_backup.sql
```

The first command uses the `mysqladmin` program to create the database. The second connects to the MySQL server using the usual `mysql` program, and feeds in our backup file as the commands to be executed.

In this way, we can use `mysqldump` to create backups of our databases. Since `mysqldump` performs its backups by connecting through the MySQL server, rather than by directly accessing the database files in the MySQL data directory, the backup produced is guaranteed to be a valid copy of the database, and not a snapshot of the database files, which may be in a state of flux as long as the MySQL server is online.

What remains to be seen is how to bridge the gap between these snapshots to maintain a backup of a database that is always up to date. To do this, you must instruct the server to keep an update log.

Incremental Backups using Update Logs

As we mentioned above, many situations in which a MySQL database may be used would make the loss of data -- any data -- unacceptable. In cases like these, we need some way of bridging the gap between backups made using mysqldump as described above. The solution is to instruct the MySQL server to keep an update log. An update log is a record of all SQL queries received by the database that modified the contents of the database in some way. This includes `INSERT`, `UPDATE`, and `CREATE TABLE` statements (among others), but does not include `SELECT` statements.

The general idea of keeping an update log is that you can restore the contents of the database at the very moment some disaster occurred by first applying a backup (made using `mysqldump`) then applying the contents of the update logs that were generated since that backup was made.

You can also edit update logs to undo mistakes that may have been made. For example, if a co-worker comes to you after having issued a `DROP TABLE` command without thinking, you can edit your update log to remove that command before restoring your database using your last backup and then applying the log. In this way, you can even keep changes to other tables that were made after the accident. As a precaution, you should probably also revoke your co-worker's `DROP` privileges (see the next section to find out how). `:)`

Telling the MySQL server to keep update logs is as simple as adding an option to the server command line:

```
% safe-mysqld --log-update=update
```

The above command starts the MySQL server and tells it to create files named update.001, update.002, and so on in the server's data directory (/usr/local/mysql/var if you set up the server according to the instructions in Part One). A new such file will be created each time the server flushes its log files (in practice, this is whenever the server is restarted). If you want to store your update logs someplace else (usually a good idea -- if the disk containing your data directory dies, you don't want it to take your backups along with it!), you can specify the full path to the update files.

If you're running your MySQL server full time, however, you probably have your system set up to launch the MySQL server at startup. Adding command-line options to the server can be difficult in this case. A simpler way to have update logs created is to add the option to the MySQL configuration file.

If you're thinking "WHAT MySQL configuration file??" don't worry. Until now, we have had no need for a server configuration file. To create one, log into Linux as the MySQL user we created in Part One (mysqlusr if you followed the instructions provided there). Using your favorite text editor, create a file called my.cnf in your MySQL data directory (/usr/local/mysql/var unless you chose someplace else to install MySQL). In the file, type the following:

```
[mysqld]
log-update=/usr/backups/mysql/update
```

Of course, feel free to specify whatever location you want for the server to write the update logs. Save the file and restart your MySQL server. From now on, the server will behave by default as if you'd specified the --log-update option on the command line.

Obviously, update logs can take up a lot of space on an active server. For this reason, and because MySQL will not automatically delete old log files as it creates new ones, it's up to you to manage your update log files. The following Unix shell script, for example, deletes all update files that were last modified more than a week ago, then tells MySQL to flush its log files.

```
#! /bin/sh
find /usr/backups/mysql/ -name "update.[0-9]*" \
  -type f -mtime +6 | xargs rm -f
/usr/local/mysql/bin/mysqladmin -u root \
  -ppassword flush-logs
```

This last step (flushing the log files) creates a new update log in case the current one has just been deleted, which will happen if the server has been online and has not received any queries that changed database contents for over a week.

If you're an experienced user, setting a script up using "cron" to periodically (say, once a week) perform a database backup and delete old update logs should be fairly easy for you. If you need a little help with this, speak to your local Unix guru, or post a message to the SitePoint.com Forums (we'll be glad to help!). The book 'MySQL' by Paul DuBois also has a fairly detailed guide for setting up such a system in its chapter on MySQL administration.

Assuming you have a backup and a copy of the update logs since it was made, restoring your database is now fairly simple. After creating the empty database and applying the backup as described in the previous section, apply the update logs using the --one-database command-line option for mysql. This instructs the server to run only those queries appearing in the update log that pertain to the database you want to restore (dbname in this example):

```
% mysql -u root -ppassword --one-database dbname < update.100
% mysql -u root -ppassword --one-database dbname < update.102
...
```

MySQL Access Control

Early on in this series, I mentioned that the database called `mysql`, which appears on every MySQL server, is used to keep track of users, their passwords, and what they are allowed to do. Until now, however, we have always logged into the server as the `root` user, which has access to all databases and tables.

If your MySQL server will only be accessed through PHP, and you're careful about who is given the password to the root MySQL account, then the `root` account may be sufficient. In cases where a MySQL server is shared among many users, however (for example, if a Web host wishes to use a single MySQL server to provide a database to each of its users), it is usually a good idea to set up user accounts with more restricted access.

The MySQL access control system is fully documented in Chapter 6 of the MySQL Reference Manual. In essence, user access is governed by the contents of five tables in the `mysql` database: `user`, `db`, `host`, `tables_priv`, and `columns_priv`. If you plan on editing these tables directly using `INSERT`, `UPDATE`, and `DELETE` statements, I would suggest reading the section of the MySQL manual on the subject beforehand. Since version 3.22.11, however, MySQL provides a simpler method of managing user access. Using `GRANT` and `REVOKE`, non-standard commands provided by MySQL, you can create users and set their privileges without having to worry about the details of how these are represented in the tables mentioned above.

Using `GRANT`

The `GRANT` command, used for creating new users, assigning user passwords, and adding user privileges, looks like this:

```
mysql> GRANT <privileges> ON <what>
    -> TO <user> [IDENTIFIED BY "<password>"]
    -> [WITH GRANT OPTION];
```

As you can see, there are a lot of blanks to be filled in with this command. Let's describe each of them in turn, and then look at some examples to give you an idea of how they work together.

`<privileges>` is a comma-separated list of the privileges you wish to grant. The privileges you can specify can be sorted into three groups:

- **Database/Table/Column privileges:**
    - `ALTER`: Modify existing tables (e.g. add/remove columns) and indexes.
    - `CREATE`: Create new databases and tables.
    - `DELETE`: Delete table entries.
    - `DROP`: Delete tables and/or databases.
    - `INDEX`: Create and/or delete indexes.
    - `INSERT`: Add new table entries.
    - `SELECT`: View/search table entries.
    - `UPDATE`: Modify existing table entries.

- **Global administrative privileges:**
  - `FILE`: Read and write files on the MySQL server.
  - `PROCESS`: View and/or kill server threads belonging to other users.
  - `RELOAD`: Reload the access control tables, flush the logs, etc.
  - `SHUTDOWN`: Shut down the MySQL server.
- **Special privileges:**
  - `ALL`: Allowed to do anything (like root).
  - `USAGE`: Only allowed to log in -- nothing else.

Some of these privileges apply to features of MySQL that we have not yet seen, but many should be familiar to you.

`<what>` defines what areas of the database sever the privileges apply to. `*.*` means the privileges apply to all databases and tables. `dbName.*` means the privileges apply to all tables in the database called `dbName`. `dbName.tblName` means the privileges apply only to the table called `tblName` in the database called `dbName`. You can even specify privileges for individual table columns by listing the columns between parentheses following the privileges to be granted (we'll see an example of this in a moment).

`<user>` specifies the user to which these privileges should apply. In MySQL, a user is specified both by the username given at login, and the hostname/IP of the machine from which the user connects. Both values may contain the `%` wildcard character (e.g. `kevin@%` will allow the username `kevin` to log in from any host and have the privileges you specify).

`<password>` specifies the password required for the user to connect to the MySQL server. As indicated by the square brackets above, the `IDENTIFIED BY "<password>"` portion of the `GRANT` command is optional. Any password specified will replace the existing password for that user. If no password is specified for a new user, a password will not be required to connect.

The optional `WITH GRANT OPTION` portion of the command specifies that the user be allowed to use `GRANT/REVOKE` to give any privileges granted to him or her to another user. Be careful with this -- the repercussions are not always obvious! For example, two users with this option enabled can get together and share their privileges with each other.

Using `GRANT` (cont'd.)

Let's consider a couple of examples. To create a user named `dbmanager` that can connect from `server.host.net` with password `managedb` and has full access to the database named `db` only (including the ability to grant access to that database to other users), use the following `GRANT` command:

```
mysql> GRANT ALL ON db.*
    -> TO dbmanager@server.host.net
    -> IDENTIFIED BY "managedb"
    -> WITH GRANT OPTION;
```

To subsequently change that user's password to `funkychicken`, use the following:

```
mysql> GRANT USAGE ON *.*
    -> TO dbmanager@server.host.net
    -> IDENTIFIED BY "funkychicken";
```

Notice that we aren't granting any additional privileges (the `USAGE` privilege doesn't let a user do anything but log in), but the user's existing privileges remain unchanged.

Now let's create a new user named `jessica`, who will be connecting from various machines in the `host.net` domain. Say she's responsible for keeping the names and email addresses of users in the database up to date, but may need to refer to other database information at times. As a result, she will have read-only (i.e. `SELECT`) access to the `db` database, but will be able to `UPDATE` the `name` and `email` columns of the `Users` table. Here are the commands:

```
mysql> GRANT SELECT ON db.*
    -> TO jessica@%.host.net
    -> IDENTIFIED BY "jessrules";
mysql> GRANT UPDATE (name,email) ON db.Users
    -> TO jessica@%.host.net;
```

Notice in the first command how we use the `%` (wildcard) character in the hostname to indicate where Jessica can connect from. Notice also that we have not given her the ability to pass her privileges onto other users, as we didn't put `WITH GRANT OPTION` on the end of the command. The second command demonstrates how to grant privileges for specific table columns, by listing the column(s) separated by commas in parentheses following the privilege(s) being granted.

Using `REVOKE`

The `REVOKE` command, as you would expect, is used to strip previously granted privileges from a user. The syntax for the command is as follows:

```
mysql> REVOKE <privileges> [(<columns>)]
    -> ON <what> FROM <user>;
```

All the fields in this command work just as they do in `GRANT` above. To revoke a co-worker of Jessica's `DROP` privileges (for instance, if he or she has demonstrated a habit of occasionally deleting tables and databases by mistake), you would use the following command:

```
mysql> REVOKE DROP ON *.* FROM idiot@%.host.net;
```

Revoking a user's login privileges is about the only thing that can't be done using `REVOKE`. `REVOKE ALL ON *.*` would definitely prevent a user from doing anything of consequence besides logging in, but to remove a user completely requires that you delete the corresponding entry in the user table:

```
mysql> DELETE FROM user
    -> WHERE User="idiot" AND Host="%.host.net";
```

Access Control Tips

Due to the way the access control system in MySQL works, there are a couple of idiosyncrasies that you should be aware of before launching into creating your users.

When creating users that can only log into the MySQL server from the computer on which that server is running (i.e. you require them to telnet to the server and run the MySQL client from there, or communicate using server-side scripts like PHP), you may ask yourself what the `<user>` part of the `GRANT` command should be. Say the server is running on `www.host.net`. Should you set up the user as `username@www.host.net`, or `username@localhost`?

The answer is that you can't rely on either one handling all connections. In theory, if the user specifies the hostname when connecting (either with the `mysql` client or with PHP's `mysql_connect` function), that hostname will have to match the entry in the access control system. But since you probably don't want to force your users to specify the hostname a particular way (in fact, users of the `mysql` client probably won't want to specify the hostname at all), it's best to use the following work-around.

For users that need to be able to connect from the same machine on which the MySQL server is running, create two user entries in the MySQL access system: one with the actual hostname of the machine (e.g. `username@www.host.net`), the other with `localhost` (e.g. `username@localhost`). Of course, you will have to grant/revoke all privileges to both of these user entries individually, but this is the only work-around that you can really rely upon.

Another common problem faced by MySQL administrators is that user entries with wildcards in their hostnames (e.g. `jessica@%.host.net` above) fail to work. When this happens, it is usually due to the way MySQL prioritizes the entries in the access control system. Specifically, it orders entries so that more specific hostnames appear first (e.g. `www.host.net` is completely specific, `%.host.net` is less specific, and `%` is totally unspecific).

In a fresh installation, the MySQL access control system contains two anonymous user entries (which allow connections from the local host using any username -- the two entries are to support connections from `localhost` and the server's actual hostname, as described above), and two `root` user entries. The problem described above happens when the anonymous user entries take precedence over our new entry because their hostname is more specific.

Let's look at the abridged contents of the user table on `www.host.net`, our fictitious MySQL server, after adding Jessica's entry. The rows here are sorted in the order that the MySQL server considers them when validating a connection:

```
+--------------+---------+-------------------+
| Host         | User    | Password          |
+--------------+---------+-------------------+
| localhost    | root    | (encrypted value) |
| www.host.net | root    | (encrypted value) |
| localhost    |         |                   |
| www.host.net |         |                   |
| %.host.net   | jessica | (encrypted value) |
+--------------+---------+-------------------+
```

As you can see, since Jessica's entry has the least specific hostname, it comes last in the list. When Jessica attempts to connect from `www.host.net`, the MySQL server matches her connection attempt to one of the anonymous user entries (a blank `User` value matches anyone). Since these anonymous entries don't require a password, and presumably Jessica enters her password, MySQL rejects the connection attempt. Even if Jessica connected without a password, she would be given the (very limited) privileges that are assigned to the anonymous users, as opposed to the privileges assigned to her entry in the access control system.

The solution to this problem is to either make your first order of business as a MySQL administrator to delete those anonymous user entries (`DELETE FROM user WHERE User=""`), or to give all users that need to connect from `localhost` two more entries (i.e. for `localhost` and the actual hostname of the server):

```
+--------------+---------+-------------------+
| Host         | User    | Password          |
+--------------+---------+-------------------+
| localhost    | root    | (encrypted value) |
| www.host.net | root    | (encrypted value) |
| localhost    | jessica | (encrypted value) |
| www.host.net | jessica | (encrypted value) |
| localhost    |         |                   |
| www.host.net |         |                   |
| %.host.net   | jessica | (encrypted value) |
+--------------+---------+-------------------+
```

Since maintaining three user entries (and three sets of privileges) for each user is excessive, we recommend removing the anonymous users unless you have a particular need for them:

```
+--------------+---------+-------------------+
| Host         | User    | Password          |
+--------------+---------+-------------------+
| localhost    | root    | (encrypted value) |
| www.host.net | root    | (encrypted value) |
| %.host.net   | jessica | (encrypted value) |
+--------------+---------+-------------------+
```

Locked Out?

Like locking your keys in the car, forgetting your password after spending an hour installing and tweaking a new MySQL server can be embarrassing to say the least. Fortunately, if you have root access to the computer on which the MySQL server is running, or if you can log in as the user you set up to run the MySQL server (`mysqlusr` if you followed the instructions in Part One), all is not lost. The following procedure will let you regain control of the server.

First, you must shut down the MySQL server. Since you would normally do this using `mysqladmin`, which requires your forgotten password, you'll have to do this by killing the server process. Using the `ps` command or by looking in the server's PID file (in the MySQL data directory), determine the process ID of the MySQL server, then terminate it using the following command:

```
% kill <pid>
```

where `<pid>` is the process ID of the MySQL server. This should be enough to stop the server. Do not use `kill -9` unless absolutely necessary, as this may damage your table files. If you are forced to do so, the next section provides instructions on how to subsequently check and repair those files.

With the server down, you can now restart it by running `safe-mysqld` (`mysqld` or `mysqld-nt` under Windows) with the `--skip-grant-tables` command line option. This instructs the MySQL server to allow unrestricted access to anyone. Obviously, you want to run the server in this mode as little as possible to avoid the inherent security risks.

Once connected, change your root password to something you'll remember:

```
mysql> USE mysql;
mysql> UPDATE user SET Password=PASSWORD("newpassword")
    -> WHERE User="root";
```

Finally, disconnect and instruct the MySQL server to reload the grant tables to begin requiring passwords:

```
% mysqladmin flush-privileges
```

That does it -- and nobody ever has to know what you did. As for locking your keys in your car, you're on your own there. ;)

Checking and Repairing MySQL Data Files

In power outages, situations where you need to `kill -9` the MySQL server process, or when Jessica's friend `idiot@%.host.net` kicks the plug out of the wall, there is a risk that the MySQL data files may be damaged. This can occur if the server is in the middle of making changes to the files at the time of the disturbance, as the files may be left in a corrupt or inconsistent state. Since this type of damage can be subtle, it can go undetected for days, weeks, even months. As a result, when you do discover the problem, all your backups may contain the same corruption.

Chapter 15 of the MySQL Reference Manual describes the `myisamchk` utility that comes with MySQL, and how to use it to check and repair your MySQL data files. While that chapter is recommended reading for anyone who wants to set up a heavy-duty preventative maintenance schedule for their MySQL server, we will cover all the essentials here.

Before we go any further, it is important for you to realize that the `myisamchk` program expects to have sole access to the MySQL data files it is checking and modifying. If the MySQL server is working with the files at the same time, and makes a modification to a file that `myisamchk` is in the middle of checking, `myisamchk` might incorrectly detect an error and try to fix it -- which in turn could trip up the MySQL server! Thus, to avoid making things worse instead of better, it's usually a good idea to shut down the MySQL server while working on the data files. Alternatively, shut down the server just long enough to make a copy of the files, and then do the work on the copies. When you're done, shut down the server again briefly to replace the files with the new ones (and perhaps apply any update logs that were made in the interim).

The MySQL data directory isn't too difficult to understand. It contains a subdirectory for each database, and each of these subdirectories contains the data files for the tables in the corresponding database. Each table is represented by three files, which have the same name as the table but with three different extensions. The `tblName.frm` file is the table definition, which keeps track of what columns the table contains, and their type. The `tblName.MYD` file contains all the table data. The `tblName.MYI` file contains any indexes for the table (for example, it might contain the lookup table that helps the table's primary key column speed up queries based on it).

To check a table for errors, just run `myisamchk` (in the MySQL `bin` directory) and provide the location of these files and the name of the table, or the name of the table index file:

```
% myisamchk /usr/local/mysql/var/dbName/tblName
% myisamchk /usr/local/mysql/var/dbName/tblName.MYI
```

Either of the above will perform a check of the specified table. To check all tables in the database, use a wildcard:

```
% myisamchk /usr/local/mysql/var/dbName/*.MYI
```

And to check all databases in all tables, use two:

```
% myisamchk /usr/local/mysql/var/*/*.MYI
```

Without any options, `myisamchk` performs a normal check of the table files. If you suspect problems with a table and a normal check fails to turn up anything, you can perform a much more thorough (but also much slower!) check using the `--extend-check` option:

```
% myisamchk --extend-check /path/to/tblName
```

Checking for errors is non-destructive, which means that you don't have to worry about making an existing problem worse by performing a check on your data files. Repair operations, on the other hand, while usually safe, make changes to your data files that cannot be undone. For this reason, it is strongly recommended that you make a copy of any damaged table files before attempting to repair them. As usual, make sure your MySQL server is shut down before making copies of the data files.

There are three types of repair that you can try to fix a problem with a damaged table. These should be tried in order, with fresh copies of the data files each time (i.e. don't try the second recovery method on a set of files resulting from a failed attempt of the first recovery method). If at any point you get an error message indicating that a temporary file can't be created, delete the file the message refers to and try again -- the offending file is a remnant of a previous repair attempt.

The three repair methods can be executed as follows:

```
% myisamchk --recover --quick /path/to/tblName
% myisamchk --recover /path/to/tblName
% myisamchk --safe-recover /path/to/tblName
```

The first is the quickest, and fixes the most common problems; the last is the slowest, and fixes a few problems that the other methods do not.

Checking and Repairing MySQL Data Files (cont'd.)

If the above methods fail to resurrect a damaged table, there are a couple more tricks you can try before giving up:

- If you suspect that the table index file (`*.MYI`) is damaged beyond repair, or even missing entirely, it can be regenerated from scratch and used with your existing data (`*.MYD`) and table form (`*.frm`) files. Begin by making a copy of your table data (`tblName.MYD`) file. Restart your MySQL server and connect to it, then delete the contents of the table using the following command:

  ```
  mysql> DELETE FROM tblName;
  ```

  In addition to deleting the contents of the table, this creates a brand new index file for the table. Log out and shut down the server again, then copy your saved data file (`tblName.MYD`) over the new (empty) data file. Finally, perform a standard repair (the second method above) using `myisamchk` to regenerate the index data based on the contents of the data and table form files.

- If your table form file (`tblName.frm`) is missing or damaged beyond repair, but you know the table well enough to reproduce the `CREATE TABLE` statement that defines it, you can generate a new `.frm` file and use it with your existing data file and index file (if the index file is no good, use the above method to generate a new one afterwards). Begin by making a copy of your data and index files, and then delete the originals (removing any record of the table from the data directory).

  Start up the MySQL server and create a new table using the exact same `CREATE TABLE` files over top of the new (empty files). The new `.frm` file should work with them, but perform a standard table repair (the second method above) for good measure.

Wrap-Up

Okay, I admit this part of the series hasn't been the usual non-stop, action-packed code fest that you may have become accustomed to. But in concentrating on the topics that we did -- backing up and restoring MySQL data, administering the MySQL access control system, and table checking and repair -- we have armed ourselves with the tools we'll need to be able to set up a MySQL database server that will stand the test of time (not to mention the constant traffic that your site will endure during that time).

In Part 9, the penultimate chapter of this series, we'll get back to the fun stuff and learn some advanced SQL techniques for making a relational database server do things that you may never have thought possible.

Part 9: Advanced SQL

As we worked through our example of an Internet Joke Database website, we had opportunities to explore most aspects of Structured Query Language (SQL). From the basic form of a `CREATE TABLE` query, to the two syntaxes of `INSERT` queries, you probably know many of these commands by heart by now.

This week, in an effort to tie up loose ends, we'll be looking at a few more SQL tricks that we haven't seen before, either because they were too advanced, or simply because "it didn't come up". As is typical, most of these will expand on our knowledge of what is already the most complex and potentially confusing SQL command available to us: the `SELECT` query.

Sorting `SELECT` Query Results

Long lists of information are always easier to use when they are provided in some kind of order. Finding a single author in a listing of our `Authors` table, for example, could become an exercise in frustration if we had more than a few dozen registered authors in our database. While at first it might appear that they are sorted in order of database insertion (with the oldest records first and the newest records last), you'll quickly notice that deleting records from the database leaves invisible gaps in this order, which get filled in by newer entries as they are inserted.

What this amounts to is no reliable built-in sorting of results from `SELECT` queries. Fortunately, there is another optional part of the `SELECT` query that lets us specify a column by which to sort our table of results. Let's say we wanted to print out a listing of the entries in our `Authors` table for future reference. If you'll recall, this table has three columns: `ID`, `Name`, and `eMail`. Since `ID` isn't really interesting in and of itself (it just provides a means to associate entries in this table with entries in the `Jokes` table), we will usually just list the remaining two columns when working with this table. Here's a short listing of a table of authors:

```
mysql> SELECT Name, eMail FROM Authors;
+-----------------+---------------------+
| Name            | eMail               |
+-----------------+---------------------+
| Joan Smith      | jsmith@somewhere.net |
| Ted E. Bear     | ted@e-bear.net      |
| Kevin Yank      | kevin@sitepoint.com  |
| Amy Mathieson   | amym@hotmail.com    |
+-----------------+---------------------+
```

As you can see, the entries are sorted in no particular order. This is fine for a short list like this, but if we were looking for a particular author's email address (that of Amy Mathieson, for example) in a very long list of authors (say a few hundred or so), having the authors' names appear in alphabetical order would make finding the entry quite a bit easier. Here's how:

```
mysql> SELECT Name, eMail FROM Authors ORDER BY Name;
+-----------------+---------------------+
| Name            | eMail               |
+-----------------+---------------------+
| Amy Mathieson   | amym@hotmail.com    |
| Joan Smith      | jsmith@somewhere.net |
| Kevin Yank      | kevin@sitepoint.com  |
| Ted E. Bear     | ted@e-bear.net      |
+-----------------+---------------------+
```

The entries now appear sorted alphabetically by their names. Just as we can add a `WHERE` clause to a `SELECT` statement to narrow down the list of results, we can also add an `ORDER BY` clause to specify a column by which a set of results should be sorted.

By adding the keyword `DESC` following the name of the sort column, you can have the entries sorted in descending order:

```
mysql> SELECT Name, eMail FROM Authors ORDER BY Name DESC;
+-----------------+---------------------+
| Name            | eMail               |
+-----------------+---------------------+
| Ted E. Bear     | ted@e-bear.net      |
| Kevin Yank      | kevin@sitepoint.com  |
| Joan Smith      | jsmith@somewhere.net |
| Amy Mathieson   | amym@hotmail.com    |
+-----------------+---------------------+
```

You can actually use a comma-separated list of several column names in the `ORDER BY` clause to have MySQL sort the entries by the first column, then sort any sets of tied entries by the second, and so on. Any of the columns listed in the `ORDER BY` clause may use the `DESC` keyword to reverse the sort order.

Setting `LIMIT`s

Often you'll be working with a large database table, but are only really interested in a few entries of it. Let's say you wanted to track the popularity of different jokes on your site. You could add a column named `TimesViewed` to your `Jokes` table. By starting it with a value of zero for new jokes and adding one to the value of the requested joke every time the joke page is viewed, you can keep count of the number of times each joke in your database has been read.

The PHP code for the query that adds one to the `TimesViewed` column of a joke with a given ID is as follows:

```
$sql = "UPDATE Jokes SET TimesViewed=TimesViewed+1 ".
             "WHERE ID=$id";
if (!mysql_query($sql)) {
  echo("<P>Error adding to times viewed ".
       "for this joke!</P>\n");
}
```

A common use of this "joke view counter" would be to present a "Top 10 Jokes" list on the front page of the site, for example. Using `ORDER BY TimesViewed DESC` to list the jokes from highest `TimesViewed` to lowest, we would just have to pick the 10 first values from the top of the list. But if we have thousands of jokes in our database, retrieving the complete list would be quite wasteful in terms of the processing time and server system resources required (such as memory and CPU load) to use only ten of them.

Using a `LIMIT` clause, we can specify that we only want a certain number of results to be returned. In our example, we need only the first ten:

```
$sql = "SELECT * FROM Jokes ORDER BY TimesViewed DESC LIMIT 10";
```

Although much less interesting, we could get rid of the word `DESC` and get the 10 least popular jokes in the database.

Often, you want to let users view a long list of entries (say, the results of a search), but only wish to display a few at a time. Think of the last time you went looking through pages of search engine results to find the web site you were looking for. You can use a `LIMIT` clause to do this sort of thing by specifying both the result to begin the list with and the maximum number of results to display. The following query, for example, will list the 21st to 25th most popular jokes in the database:

```
$sql = "SELECT * FROM Jokes ORDER BY TimesViewed DESC LIMIT 20, 5";
```

Remember, the first entry in the list of results is entry number 0. Thus, the 21st entry in the list is entry number 20.

LOCKing TABLES

Notice how, in the `UPDATE` query given above (and repeated here for convenience), we use the existing value of `TimesViewed` and add one to it to set the new value.

```
$sql = "UPDATE Jokes SET TimesViewed=TimesViewed+1 WHERE ID=$id";
```

If you hadn't known that you were allowed to do this, you might have done a separate `SELECT` to get the current value, add one to it, and then do an `UPDATE` using that newly calculated value. Besides the fact that this requires two queries instead of one, and thus will take about twice as long, there is a danger to using this method. What if, while that new value was being calculated, someone else viewed the same joke? The PHP script would be run a second time for that new request. When it performed the `SELECT` to get the "current" value of `TimesViewed`, it would get the same value as the first script did, because the value has not yet been updated. Both scripts would then add one to the same value, and write the new value into the table. See what happened? Two users viewed the joke, but the `TimesViewed` counter only got incremented by one!

In some situations, this kind of fetch-calculate-update procedure cannot be avoided, and the possibility of interference between simultaneous requests (of the nature described above) must be dealt with. Other situations where this may be necessary include cases where you need to update several tables in response to a single action (e.g. updating inventory and shipping tables in response to a sale on an eCommerce Web site). Many high-end database servers (e.g. Oracle, MS SQL Server, etc.) support a feature called "transactions", which lets you define complex operations like those above so they may be performed in a single, uninterrupted step. Support for transactions in MySQL is still in the works, but there is an alternative!

By "locking" the table or tables with which you are working in a multiple-query operation, you can obtain exclusive access for the duration of that operation to prevent potentially damaging interference from concurrent operations occurring mid-stream. The syntax for locking a table is fairly simple:

```
LOCK TABLES tblName { READ | WRITE }
```

As shown, when locking a table, you must specify whether you want a "read lock" or a "write lock". The former prevents other processes from making changes to the table, but allows others to read the table. The latter stops all other access to the table.

When you're done with a table you have locked, you must release the lock to give other processes access to the table again:

```
UNLOCK TABLES
```

A `LOCK TABLES` query implicitly releases whatever locks you may already have; so to safely perform a multi-table operation, you must lock all the tables you'll be using with a single query. Here's what the PHP code might look like for the eCommerce application we mentioned above:

```
mysql_query("LOCK TABLES inventory WRITE, shipping WRITE");

// Perform the operation...

mysql_query("UNLOCK TABLES");
```

Column and Table Name Aliases

In some situations, it can be convenient to be able to refer to MySQL columns and tables using different names. Let's take the example of a database used by an airline's online booking system (this example actually came up in the SitePoint.com forums). To represent the flights offered by the airline, the database contains two tables: `Flights` and `Cities`. Each entry in the `Flights` table represents an actual flight between two cities -- the origin and destination of the flight. Obviously, `Origin` and `Destination` are columns in the `Flights` table, with other columns for things like the date and time of the flight, the type of aircraft, the flight number, and the various fares.

The `Cities` table contains a list of all the cities to which the airline flies. Thus, both the `Origin` and `Destination` columns in the `Flights` table will just contain ID's referring to entries in the `Cities` table. Now, consider the following queries.

To get a list of flights with their origins:

```
mysql> SELECT Flights.Number, Cities.Name
    -> FROM Flights, Cities
    -> WHERE Flights.Origin = Cities.ID;
```

```
+--------+-----------+
| Number | Name      |
+--------+-----------+
| CP110  | Montreal  |
| CP226  | Sydney    |
| QF2026 | Melbourne |
  ...      ...
```

To get a list of flights with their destinations:

```
mysql> SELECT Flights.Number, Cities.Name
    -> FROM Flights, Cities
    -> WHERE Flights.Destination = Cities.ID;
+--------+----------+
| Number | Name     |
+--------+----------+
| CP110  | Sydney   |
| CP226  | Montreal |
| QF2026 | Sydney   |
  ...      ...
```

Now what if we wanted to list both the origin and destination of each flight with a single query? Pretty reasonable, right? Here's a query you might think to try:

```
mysql> SELECT Flights.Number, Cities.Name, Cities.Name
    -> FROM Flights, Cities
    -> WHERE Flights.Origin = Cities.ID
    -> AND Flights.Destination = Cities.ID;
Empty set (0.01 sec)
```

Why doesn't this work? Have another look at the query, this time focusing on what it actually says, rather than what you expect it to do. You're telling MySQL to join the `Flights` and `Cities` tables and list the flight number, city name, and city name (yes, twice!) of all entries obtained by matching up the `Origin` with the City `ID` and the `Destination` with the City `ID`. In other words, the `Origin`, `Destination`, and City `ID` must all be equal! This results in a list of all flights where the origin and the destination are the same! Unless your airline is offering scenic flights, there aren't likely to be any entries matching this description (thus the "Empty set" result above).

What we need is a way to be able to return two different entries from the `Cities` table (one for the origin and one for the destination) for each result. If we had two copies of the table, one called `Origins` and one called `Destinations`, this would be much easier to do, but why maintain two tables containing the exact same list of cities? The solution is to give the `Cities` table two different temporary names (aliases) for the purposes of this query.

By following the name of a table with `AS Alias` in the `FROM` portion of the `SELECT` query, we can give it a temporary name with which to refer to it elsewhere in the query. Here's that first query again (to display flight numbers and origins only), but this time we have given the `Cities` table an alias: `Origins`.

```
mysql> SELECT Flights.Number, Origins.Name
    -> FROM Flights, Cities AS Origins
    -> WHERE Flights.Origin = Origins.ID;
```

This doesn't actually change the way the query works -- in fact, it doesn't change the results at all -- but for long table names, it can save some typing. Consider, for example, if we had given aliases of `F` and `O` to `Flights` and `Cities`, respectively. The query would be much shorter as a result.

Let's now return to our problem query. By referring to the `Cities` table twice, using two different aliases, we can use a three-table join (where two of the tables are actually one and the same) to get the effect we want:

```
mysql> SELECT Flights.Number, Origins.Name,
    ->        Destinations.Name
    -> FROM Flights, Cities AS Origins,
    ->      Cities AS Destinations
    -> WHERE Flights.Origin = Origins.ID
    -> AND Flights.Destination = Destinations.ID;
+--------+-----------+----------+
| Number | Name      | Name     |
+--------+-----------+----------+
| CP110  | Montreal  | Sydney   |
| CP226  | Sydney    | Montreal |
| QF2026 | Melbourne | Sydney   |
  ...      ...         ...
```

You can also define aliases for column names. We could use this, for example, to differentiate the two "Name" columns in our result table above:

```
mysql> SELECT F.Number, O.Name AS Origin,
    ->        D.Name AS Destination
    -> FROM Flights AS F, Cities AS O, Cities AS D
    -> WHERE F.Origin = O.ID AND F.Destination = D.ID;
+--------+-----------+-------------+
| Number | Origin    | Destination |
+--------+-----------+-------------+
| CP110  | Montreal  | Sydney      |
| CP226  | Sydney    | Montreal    |
| QF2026 | Melbourne | Sydney      |
  ...      ...         ...
```

`GROUP`ing `SELECT` Results

Way back in Part Two, we saw the following query, which tells us how many jokes are stored in our `Jokes` table:

```
mysql> SELECT COUNT(*) FROM Jokes;
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

The MySQL function `COUNT` used in this query belongs to a special class of functions called "summary functions" or "group-by functions", depending on where you look. A complete list of these functions is provided in Section 7.4.13 of the MySQL Manual. Unlike other functions, which affect each entry in the result of the `SELECT` query individually, summary functions group together all the results and return a single result. In the above example, for instance, `COUNT` returns the total number of result rows.

Let's say you wanted to display a list of authors with the number of jokes they have to their names. Your first instinct (if you've been paying attention) might be to retrieve a list of all the authors' names and ID's, then use `COUNT` to count the number of results when you `SELECT` the jokes with each author's ID. The PHP code (without error handling, for simplicity) would look something like this:

```
// Get a list of all the authors
$authors = mysql_query( "SELECT Name, ID FROM Authors" );
// Process each author
```

```
while ($author = mysql_fetch_array($authors)) {
  $name = $author["Name"];
  $id = $author["ID"];
  // Get count of jokes attributed to this author
  $result = mysql_query(
    "SELECT COUNT(*) AS NumJokes ".
    "FROM Jokes WHERE AID=$id" );
  $row = mysql_fetch_array($result);
  $numjokes = $row["NumJokes"];
  // Display the author & number of jokes
  echo("<P>$name ($numjokes jokes)</P>");
}
```

Note the use of `AS` in the second query above to give a friendlier name (`NumJokes`) to the result of `COUNT(*)`.

This technique will work, but will require n+1 separate queries (where n is the number of authors in the database). Having the number of queries rely on a number of entries in the database is always something we want to avoid (a large number of authors would make this script unreasonably slow and resource-intensive!). Fortunately, another advanced feature of `SELECT` comes to the rescue!

By adding a `GROUP BY` clause to a `SELECT` query, you can tell MySQL to group the results of the query into sets that have the same value in the column(s) you specify. Summary functions like `COUNT` then operate on those groups -- not on the result set as a whole. The following single query, for example, lists the number of jokes attributed to each author in the database:

```
mysql> SELECT Authors.Name, COUNT(*) AS NumJokes
    -> FROM Jokes, Authors
    -> WHERE AID = Authors.ID
    -> GROUP BY AID;
+-----------------+----------+
| Name            | NumJokes |
+-----------------+----------+
| Kevin Yank      |        3 |
| Joan Smith      |        1 |
| Ted E. Bear     |        5 |
+-----------------+----------+
```

By grouping the results by author ID (`AID`), we get a breakdown of results for each author. Note that we could have specified `GROUP BY Authors.ID` and gotten the same result (since, as stipulated in the `WHERE` clause, these columns must be equal). `GROUP BY Authors.Name` would also work in most cases, but since you can't guarantee that two different authors won't have the same name (in which case their results would be lumped together), it's best to stick to the ID columns, which are guaranteed to be unique for each author.

LEFT JOINS

We can see from the results above that Kevin Yank has three jokes to his name, Joan Smith has one, and Ted E. Bear has five. What these results do not show is that there is a fourth author, Amy Mathieson, who doesn't have any jokes to her name. Since there are no entries in the `Jokes` table with `AID`'s matching her author `ID`, there will be no results satisfying the `WHERE` clause in the query above for her, and she will therefore be excluded from the table of results.

About the only practical way to overcome this with the tools we have seen so far would be to add another column to the `Authors` table and just store the number of jokes attributed to each author in that column. Keeping that column up to date would be a real pain, since we'd have to remember to update it every time a joke was added to, removed from, or changed in (if, for example, the value of `AID` was changed) the `Jokes` table. To keep things synchronized, we'd have to use `LOCK TABLES` whenever we were making such changes, as well. Quite a mess, to say the least!

MySQL provides another method for joining tables (fetching information from multiple tables at once), called a "left join", that is designed for just this type of situation. To understand how left joins differ from standard joins, we must first recall how standard joins work.

MySQL performs a standard join of two tables by listing all possible combinations of the rows of those tables. In a simple case, a standard join of two tables with two rows apiece will contain four rows: row 1 of table 1 with row 1 of table 2, row 1 of table 1 with row 2 of table 2, row 2 of table 1 with row 1 of table 2, and row 2 of table 1 with row 2 of table 2. With all of these result rows calculated, MySQL then looks to the `WHERE` clause for guidance on which rows should actually be kept (e.g. those where the `AID` column from table 1 matches the `ID` column from table 2).

The reason the above does not suit our purposes is that we'd like to also include rows in table 1 (i.e. `Authors`) that don't have any matching rows in table 2 (i.e. `Jokes`). A left join does exactly what we need, forcing a row to appear in the results for each row in the first (left-hand) table, even if no matching entries are found in the second (right-hand) table. Such "forced rows" are given `NULL` values for all of the columns in the "right-hand" table.

To perform a left join between two tables in MySQL, you separate the two table names in the `FROM` clause with `LEFT JOIN` instead of a comma. You then follow the second table's name with `ON <condition>`, where `<condition>` specifies the criteria for matching rows in the two tables (i.e. what you would normally put in the `WHERE` clause). Here's our revised query for listing authors and the number of jokes to their credit:

```
mysql> SELECT Authors.Name, COUNT(*) AS NumJokes
    -> FROM Authors LEFT JOIN Jokes
    -> ON AID = Authors.ID
    -> GROUP BY AID;
+---------------+----------+
| Name          | NumJokes |
+---------------+----------+
| Amy Mathieson |        1 |
| Kevin Yank    |        3 |
| Joan Smith    |        1 |
| Ted E. Bear   |        5 |
+---------------+----------+
```

Wait just a minute... Suddenly Amy Mathieson has one joke? That can't be right! In fact, it isn't -- the query is still wrong. `COUNT(*)` counts the number of rows returned for each author. If we look at the ungrouped results of the `LEFT JOIN`, we can see what's happening:

```
mysql> SELECT Authors.Name, Jokes.ID AS JokeID
    -> FROM Authors LEFT JOIN Jokes
    -> ON AID = Authors.ID;
+---------------+--------+
| Name          | JokeID |
+---------------+--------+
| Kevin Yank    |      1 |
| Kevin Yank    |      2 |
| Kevin Yank    |      4 |
| Joan Smith    |      3 |
| Ted E. Bear   |      5 |
| Ted E. Bear   |      6 |
| Ted E. Bear   |      7 |
| Ted E. Bear   |      8 |
| Ted E. Bear   |      9 |
| Amy Mathieson |   NULL |
+---------------+--------+
```

See? Amy Mathieson does have a row -- the "forced row" due to her not having any matching rows in the "right-hand" table of the `LEFT JOIN` (`Jokes`). The fact that the Joke `ID` value is `NULL` doesn't affect `COUNT(*)` -- it still counts it as a row. If instead of `*`, you specify an actual column name (say `Jokes.ID`) for the `COUNT` function to look at, it will ignore `NULL` values in that column, giving us the count we are looking for:

```
mysql> SELECT Authors.Name, COUNT(Jokes.ID) AS NumJokes
    -> FROM Authors LEFT JOIN Jokes
    -> ON AID = Authors.ID
    -> GROUP BY AID;
+---------------+----------+
| Name          | NumJokes |
+---------------+----------+
| Amy Mathieson |        0 |
| Kevin Yank    |        3 |
| Joan Smith    |        1 |
| Ted E. Bear   |        5 |
+---------------+----------+
```

Limiting Results with `HAVING`

How about if we wanted a list of only those authors that had no jokes to their name? Once again, let's look at the query that most users would try first:

```
mysql> SELECT Authors.Name, COUNT(Jokes.ID) AS NumJokes
    -> FROM Authors LEFT JOIN Jokes
    -> ON AID = Authors.ID
    -> WHERE NumJokes = 0
    -> GROUP BY AID;
ERROR 1054: Unknown column 'NumJokes' in 'where clause'
```

By now you're probably not surprised that it didn't work as expected. `:)` The reason `WHERE NumJokes = 0` didn't do the job is because conditions in the `WHERE` clause affect the entries that are selected before grouping due to the `GROUP BY` clause takes place. So if you wanted to exclude jokes containing the word "chicken" from the count, you could use the `WHERE` clause; however, since the `NumJokes` column doesn't even exist before `GROUP BY` does its thing, we'll need to use a different method to set conditions on its value.

Conditions that affect the results after grouping takes place must appear in a special `HAVING` clause. Here's the corrected query:

```
mysql> SELECT Authors.Name, COUNT(Jokes.ID) AS NumJokes
    -> FROM Authors LEFT JOIN Jokes
    -> ON AID = Authors.ID
    -> GROUP BY AID
    -> HAVING NumJokes = 0;
+---------------+----------+
| Name          | NumJokes |
+---------------+----------+
| Amy Mathieson |        0 |
+---------------+----------+
```

Some conditions work both in the `HAVING` and the `WHERE` clause. For example, if we wanted to exclude a particular author by name, we could use `Authors.Name != "AuthorName"` in either the `WHERE` or the `HAVING` clause to do it, because whether you filter out the author before or after grouping occurs, you'll get the same results. In such cases, it is always best to use the `WHERE` clause, because MySQL is better at internally optimizing such queries so they happen faster.

Wrap-up

This week we rounded out knowledge of Structured Query Language (SQL), as supported by MySQL. We focused mainly on features of `SELECT` that allow us to view information stored in a database with an unprecedented level of flexibility and power. With judicious use of the advanced features of `SELECT`, you can have MySQL do what it does best and lighten the load on PHP in the process.

There are still a few isolated query types we have not seen (mainly to do with indexes), and MySQL offers a whole library of built-in functions to do things like calculate dates and format text strings. To become truly proficient with MySQL, you should also have a firm grasp on the various column types offered by MySQL. The `TIMESTAMP` type, for example, can be a real time saver (no pun intended). All of these are fully documented in the MySQL Manual, as well as Paul DuBois' fantastic book "MySQL" (see my review), to which I refer you for further reading.

In Part Ten, the long-awaited conclusion to this series, we will look at some useful features of PHP we have not had the opportunity to cover. From tightening security to sending email, from handling file uploads to lightening the load on your server, I guarantee it will be a conclusion not to be missed.

Part 10: Advanced PHP

PHP's strength lies in its huge library of built-in functions, which allow even a novice user to perform very complicated tasks without having to install new libraries or worry about low-level details, as is often the case with other popular server-side languages like Perl. Because of the focus of this series, we've constrained ourselves to exploring only those functions that were directly related to interacting with a MySQL database (in fact, we didn't even see all of those). In this final installment, we'll broaden our horizons a little and see some of the other useful features PHP has to offer someone building a database driven Web site.

We'll begin by learning about PHP's `include` function, which allows us to use a single piece of PHP code in multiple pages, making the use of common code fragments much more practical. We'll also see how to add an extra level of security to our site using this feature.

PHP, while generally quick and efficient, nevertheless adds to the load time and the workload of the machine the server is running on. On high-traffic sites (SitePoint.com, for example!), this load can grow to unacceptable levels. But this doesn't mean we have to abandon the database-driven nature of our site. We'll see how to use PHP behind the scenes to create semi-dynamic pages that don't stress the server as much.

A common question asked on SitePoint.com and in other sites' forums is how to use an `<INPUT TYPE=FILE>` tag to accept file uploads from site visitors. We'll learn how to do this with PHP, and see how to make this fit in with the database-driven nature of our site.

Finally, an extremely powerful feature of PHP is the ability to easily send email messages with dynamically generated content. Whether you want to use PHP to let visitors send email versions of your site's content to their friends, or just provide a way for users to retrieve their forgotten passwords, PHP's `email` function will serve nicely!

Server-Side Includes with PHP

If you've been working on the Internet for a while, you've probably come across the term Server-Side Includes (SSI's); if not, you can read Matt Mickiewicz' mini-tutorial on the subject.

In essence, SSI's allow you to insert the content of one file stored on your Web server into the middle of another. The most common use for this capability is to encapsulate common design elements of a Web site in small HTML files that can then be included into Web pages on the fly. Any changes to these small files immediately affect all files that include them. And just like a PHP script, the Web browser doesn't need to know about any of it since the Web server does all the work before sending the requested page to the browser.

PHP has a function that provides similar capabilities. But in addition to being able to include regular HTML and other static elements in included files, you can also include common script elements. Let's look at an example:

```
<!-- include-me.inc -->
<?php
  echo( "<P>Soylent Green is made from people!\n" );
?>
```

The above file, `include-me.inc`, contains some simple PHP code. Notice that the name of the file ends in `.inc`, not `.php`. The idea here is to name the file something other than what your Web server expects for a PHP script. This will ensure that the file can only be executed when included in one of your `.php` files, and also helps you tell apart your PHP Web pages from your PHP include files.

You'll also need the following file:

```
<!-- testinclude.php -->
<HTML>
<HEAD>
<TITLE> Test of PHP Includes </TITLE>
</HEAD>
<BODY>
<?php
  include("include-me.inc");
?>
</BODY>
</HTML>
```

This file looks more like the PHP scripts you're used to, in that it is named with a `.php` extension (or `.php3` if your server requires that). Notice the call to the `include` function. We specify the name of the file we want to include (`include-me.inc`), and PHP will attempt to grab the named file and stick it into the file to replace the call to `include`. Upload both of the above files to your Web server (or copy them to your Web server's document folder if you're running the server on your computer) and load `testinclude.php` in your browser. You'll see a Web page containing the message from our include file, as expected.

If this example doesn't work, you may need to configure the `include_path` option in your `php.ini` file. Open the file in your favorite text editor and look for a line beginning with `include_path`, about halfway through the file. This setting works just like the system PATH environment variable that you may be familiar with, and contains a list of directories where PHP should look for files that you ask it to include. Set it so it contains "`.`" (the current directory).

Depending on whether your server is running under Windows or UNIX, you may need to surround your setting with quotes:

Under UNIX:

```
include_path=.:/another/directory
```

Under Windows:

```
include_path=".;c:\another\directory"
```

Increasing Security with Includes

PHP scripts will sometimes contain sensitive information like usernames, passwords, and other things you don't want the world to have access to. By now you're probably used to the `mysql_connect` function, which requires you to put your MySQL username and password in a PHP script that needs access to a database. While you can simply set up MySQL so that the username and password used by PHP cannot be used by potential hackers (by setting the Host field in the user table as described in Part 8), you would probably still rest easier knowing that your username and password are protected by an extra level of security.

"But wait a minute," you might be saying. "Since the PHP is processed by the server, nobody gets to see my password anyway, right?" Right. But consider what would happen if PHP stopped working on your server. Whether due to an accidental software misconfiguration made by a well-meaning associate or due to some other factor, if PHP stopped working on your server, the PHP pages would be served up as plain text files, with all your PHP code (including your password) there for the world to see!

To guard against this kind of security breach, you should put any security-sensitive code into an include file and put that file in a directory that is not part of your Web server's directory structure. By adding that directory to your PHP `include_path` setting (in `php.ini`), you can refer to the files directly with the PHP include function, but have them tucked away safely somewhere where your Web server can't display them as Web pages.

For example, if your Web server expects all Web pages to exist in `/home/httpd/` and its subdirectories, you could create a directory called `/home/phplib/` to house all of your include files. Add that directory to your `include_path`, and you're done! The following example shows how you can put your database connection code into an include file:

```
<!-- dbConnect.inc (in /home/phplib/) -->
<?php
  $cnx = mysql_connect("localhost",
                       "root", "rootpassword");
?>
```

And a file that uses this include:

```
<!-- dbSample.php (in /home/httpd/) -->
<?php
  // Connect to MySQL
  include("dbConnect.inc");
  mysql_select_db("myDatabase",$cnx);
  ...
```

As you can see, if PHP stops working on your server, all that will be exposed is a call to the include function. The username and password are safely stored in `dbConnect.inc`, which cannot be accessed directly from the Web.

Semi-Dynamic Pages

As the owner of a successful (or soon-to-be so) Web site, you most likely see site traffic as something you'd like to encourage. Unfortunately, high site traffic is just the kind of thing that a Web server administrator dreads -- especially when that site is primarily composed of dynamically generated, database-driven pages. Such pages take a great deal more horsepower from the computer running the Web server software than plain, old HTML files do, because every page request is like a miniature program running on that computer.

While some pages of a database-driven site must always display up-to-the-second data culled from the database, others don't necessarily. Consider the front page of a Web site like SitePoint.com. Typically, it presents a sort of "digest" of what's new and fresh on the site. But how often does that information actually change? Once a day? Once a week? And how important is it that visitors to your site see those changes the instant they occur? Would your site really suffer if changes took effect after a bit of a delay?

By converting high-traffic dynamic pages into "semi-dynamic" equivalents, which are static pages that get dynamically re-generated at regular intervals to "freshen" their content, you can go a long way towards reducing the toll that the database-driven components of your site take on your Web server's performance.

Say you have `index.php`, your front page, providing a summary of new content on your site. Through examination of server logs, you'll probably find that this is one of the most requested pages on your site. By asking yourself some of the questions above, you realize that this page really doesn't have to be dynamically generated for every request. As long as it is updated every time new content is added to your site, it'll be as dynamic as it needs to be. Using a PHP script, you can generate a static "snapshot" of the dynamic page's output and put it online in place of the dynamic version as `index.html`.

This little trick will require a bit of reading, writing, and juggling of files. PHP is perfectly capable of this, but we have not yet seen the functions we'll need:

- **`fopen`**
  Opens a file for reading and/or writing. This file can be stored on the server's hard disk, or can be loaded from a URL just like a browser would.
- **`fclose`**
  Tells PHP you're done reading/writing a particular file, releasing it for other programs or scripts to use.
- **`fread`**
  Reads data from a file into a PHP variable. Allows you to specify how much information (i.e. how many characters or bytes) to read.
- **`fwrite`**
  Writes data from a PHP variable into a file.
- **`copy`**
  Performs a run-of-the-mill file copy operation.
- **`unlink`**
  Deletes a file from the hard disk.

Do you see where this is going? If not, don't worry -- you will in a moment.

Create a file called `generateindex.php`. It will be the responsibility of this file to load `index.php` (the dynamic version of your front page) as a Web browser would, then write the static version of the file as an updated version of `index.html`. If anything goes wrong in this process, you want to avoid destroying the "good" copy of `index.html`, so we'll make this script write the new static version into a temporary file (`tempindex.html`) and then copy it over `index.html` if all is well.

Here's the code for `generateindex.php`, with ample comments so you can see what's going on:

```
<!-- generateindex.php -->
<?php
  // Sets the files we'll be using
  $srcurl         = "http://localhost/index.php";
  $tempfilename   = "tempindex.html";
  $targetfilename = "index.html";
?>
<HTML>
<HEAD>
<TITLE>
Generating <?php echo("$targetfilename"); ?>
</TITLE>
</HEAD>
<BODY>
<P>Generating <?php echo("$targetfilename"); ?>...</P>
<?php
  // Begin by deleting the temporary file, in case
  // it was left lying around. This might spit out an
  // error message if it were to fail, so we use
  // @ to suppress it.
  @unlink($tempfilename);
  // Load the dynamic page by requesting it with a
  // URL. The PHP will be processed by the Web server
  // before we receive it (since we're basically
  // masquerading as a Web browser), so what we'll get
  // is a static HTML page. The 'r' indicates that we
  // only intend to read from this "file".
  $dynpage = fopen($srcurl, 'r');
  // Check for errors
  if (!$dynpage) {
    echo("<P>Unable to load $srcurl. Static page ".
         "update aborted!</P>");
    exit();
  }
  // Read the contents of the URL into a PHP variable.
  // Specify that we're willing to read up to 1MB of
  // data (just in case something goes wrong).
  $htmldata = fread($dynpage, 1024*1024);
  // Close the connection to the source "file", now
  // that we're done with it.
  fclose($dynpage);
  // Open the temporary file (creating it in the
  // process) in preparation to write to it (note
  // the 'w').
  $tempfile = fopen($tempfilename, 'w');
  // Check for errors
  if (!$tempfile) {
    echo("<P>Unable to open temporary file ".
         "($tempfilename) for writing. Static page ".
         "update aborted!</P>");
    exit();
  }
  // Write the data for the static page into the
  // temporary file
  fwrite($tempfile, $htmldata);
  // Close the temporary file, now that we're done
  // writing to it.
  fclose($tempfile);
  // If we got this far, then the temporary file
  // was successfully written, and we can now copy
  // it on top of the static page.
  $ok = copy($tempfilename, $targetfilename);
  // Finally, delete the temporary file.
  unlink($tempfilename);
?>
<P>Static page successfully updated!</P>
</BODY>
</HTML>
```

The above code only looks daunting because of the large comments I've included. Remove them, and you'll see it's actually a fairly simple script.

Now, whenever `generateindex.php` is run (say, by requesting it with a browser), a fresh copy of `index.html` will be generated from `index.php`. By moving `index.php` and `generateindex.php` into a restricted-access directory, you can make sure that only site administrators have the ability to update the front page of your site in this way. Expand this script to generate all semi-dynamic pages on your site and add an "update front page" link to your content management system!

If you'd rather have your front page updated automatically, you'll need to set up your server to run `generateindex.php` at regular intervals (say, every hour). Under recent versions of Windows 9x, you can use the Task Scheduler (System Agent in older versions of Windows equipped with MS Plus Pack) to automatically run php.exe (a stand-alone version of PHP included with the Windows PHP distribution) every hour. Just create a batch file called `generateindex.bat` containing the following line of text.

```
C:\PHP\php.exe C:\WWW\generateindex.php
```

Adjust the paths and filenames as necessary, and then set up Task Scheduler to run `generateindex.bat` every hour (you'll need to set up 24 tasks to be run daily at the appropriate times). Done!

Under Linux (or other UNIX-based platforms) you can do a similar thing using `cron` -- a program installed on just about every UNIX system out there that lets you define tasks to be run at regular intervals. Ask your friendly neighborhood Linux know-it-all, check your favorite Linux Web site, or post a message on the SitePoint.com Forums if you need any help getting started with `cron`.

The task you'll set up `cron` to run will be very similar to the Windows task discussed above. The stand-alone version of PHP you'll need, however, doesn't come with the PHP Apache loadable module we compiled way back in Part 1. You'll need to compile it separately from the same package we used to compile the Apache module. Instructions for this are provided with the package and on the PHP Web site, but feel free to post in the SitePoint.com Forums if you need help!

For experienced cron users in a hurry, here's what the line in your `crontab` file should look like:

```
0 0-23 * * * php /path/to/generateindex.php > /dev/null
```

Handling File Uploads

All of our examples of database-driven Web sites in this series so far have dealt with sites based around textual data. Jokes, articles, authors... all of these things can be fully represented with strings of text. But what if you were running, say, an online digital photo gallery where people could upload pictures taken with digital cameras? For this to work, we need to be able to let visitors to our site upload their photos, and we need to be able to keep track of them.

Let's start with the basics: writing an HTML form that allows users to upload files. HTML makes this quite easy with its `<INPUT TYPE=FILE>` tag. By default, however, only the name of the file selected by the user is sent. To have the file itself submitted with the form data, we need to add `ENCTYPE="multipart/form-data"` to the `<FORM>` tag:

```
<FORM ACTION="fileupload.php" METHOD=POST
 ENCTYPE="multipart/form-data">
<P>Select file to upload:
<INPUT TYPE=FILE NAME="uploadedfile"></P>
<P><INPUT TYPE=SUBMIT NAME="submit" VALUE="Submit"></P>
</FORM>
```

As we can see, a PHP script (`fileupload.php`) will handle the data submitted with the form above. As you'd expect, a PHP variable named `$uploadedfile` (from the `NAME` attribute of the `<INPUT>` tag) will be automatically created. Instead of storing the contents of the uploaded file, however, `$uploadedfile` contains the name of the file stored on the Web server's hard disk, in the directory set by the `TEMP` environment variable (e.g. `C:\Windows\TEMP\` on most Windows 9x systems). This file is only kept for as long as the PHP script responsible for handling the form submission is running, so if you want to use it for anything (e.g. storing it for display on the site) you need to make a copy of it someplace else using the `copy` function described in the previous section.

In addition to `$uploadedfile`, three other variables are automatically created as well. `$uploadedfile_name` contains the name of the file before it was submitted (submitted files are stored as `phpx`, where `x` is a number, in the `TEMP` directory), `$uploadedfile_size` contains the size (in bytes) of the file, and `$uploadedfile_type` contains the MIME type (e.g. text/plain, image/gif, etc.) of the file. Remember, "`uploadedfile`" is just the `NAME` of the `INPUT` tag that submitted the file, so the actual names of these variables will depend on that attribute.

You can use these variables to decide whether to accept or reject an uploaded file. For example, in our photo gallery we will only really be interested in JPEG and possibly GIF files. These files have MIME types of `image/pjpeg` and `image/gif` respectively, so the code for validating uploaded files might look something like this:

```
if ("image/pjpeg" == $uploadedfile_type
    or "image/gif" == $uploadedfile_type) {
  // Handle the file...
} else {
  echo("<P>Please submit a JPEG or GIF image file.\n");
}
```

While you can use a similar technique to disallow files that are too large (by checking the `$uploadedfile_size` variable), this is not usually a good idea. Before this value can be checked, the file is already uploaded and saved in the `TEMP` directory. If you're trying to reject files due to limited disk space and/or bandwidth, the fact that large files can still be uploaded (even though they get deleted almost immediately) may be a problem for you.

Instead, you can tell PHP in advance the maximum file size you wish to accept. There are two ways to do this. The first is by adjusting the `upload_max_filesize` setting in your `php.ini` file. The default value is 2MB, so if you want to accept uploads larger than that you'll immediately need to change that value.

The second method is by including a hidden `INPUT` field in your form with the name `MAX_FILE_SIZE` and the maximum file size you want to accept with this form. For security reasons, this value cannot exceed the `upload_max_filesize` setting in your `php.ini`, but it does provide a way of accepting different maximum sizes on different pages. The following form, for example, will only allow uploads of up to 1 kilobyte (1024 bytes):

```
<FORM ACTION="fileupload.php" METHOD=POST
 ENCTYPE="multipart/form-data">
<P>Select file to upload:
<INPUT TYPE=FILE NAME="uploadedfile"></P>
<P><INPUT TYPE=SUBMIT NAME="submit" VALUE="Submit"></P>
<INPUT TYPE=HIDDEN NAME=MAX_FILE_SIZE VALUE=1024>
</FORM>
```

Assigning Unique File Names

As we said above, to keep an uploaded file we need to copy it to another directory for safekeeping. And while we have access to the name of each uploaded file with its `$uploadedfile_name` variable, we have no guarantee that two files with the same name will not be uploaded. In such a case, storing the file with its original name may result in newer uploads overwriting older ones.

For this reason, you will usually want to adopt a scheme for assigning a unique filename to all uploaded files. Using the system time (which we can access using the PHP `time` function), we can easily get a name based on the number of seconds since 1/1/1970. But what if two files happen to be uploaded within one second of each other? To help guard against this, we'll also use the client's IP address (automatically stored in `$REMOTE_HOST` by PHP) in the filename. Since we're unlikely to receive two files from the same IP address within one second of each other, this is an acceptable solution for our purposes.

```
// Pick a file extension
if ( "image/pjpeg" == $uploadedfile_type )
  $extension = ".jpg";
else
  $extension = ".gif";
// The complete path/filename
$filename = "C:\\Uploads\\" . time() .
  $REMOTE_HOST . $extension;
// Copy the file
if (copy($uploadedfile, $filename)) {
  echo("<P>File stored successfully as $filename.");
} else {
  echo("<P>Could not save file as $filename!");
}
```

Notice that we must use double-backslashes (\\) in our path under Windows since backslashes are used to signify special characters in PHP text strings. Under UNIX, we can just use single slashes (/) as usual.

Recording Uploaded Files in the Database

So we've created a system whereby visitors can upload JPEG and GIF images and have them saved on our server, but wasn't this series supposed to be about database-driven Web sites? If we used the system as it stands now, someone would have to collect the submitted images out of the folder where they get saved and then add them to the Web site by hand! Thinking back to Part Seven, when we developed a system that site visitors could use to submit jokes and have them stored in the database ready for quick approval by an administrator, we know there must be a better way!

MySQL has several column types that allow you to store binary data. In database parlance, these column types let us store BLOB's (Binary Large OBjects). Storing potentially large files in a relational database, however, is not usually a good idea. While there is a convenience factor in having all the data in one place, large files lead to large databases and large databases lead to reduced performance and much larger backup files.

The best alternative is usually to just store the filenames in the database. As long as you remember to delete files when you delete their corresponding entries in the database, everything should work just the way you need it to.

Since we've seen all the SQL code involved in this time and again, I'll leave the details up to you. As usual, the SitePoint.com Forum community is here to offer help if you need it!

Email in PHP

Email is a powerful force on the Internet. Whether you want to provide a weekly "what's new" newsletter to your users or a way for them to retrieve a lost or forgotten password, email is the way to go. PHP makes working with email exceedingly easy by letting you send messages using a single call to the `mail` function.

Before you can send email using the mail function, you have to first set up PHP's email-related options. Here are the relevant lines of an out-of-the-box `php.ini` file under Windows:

```
[mail function]
SMTP            = localhost         ;for win32 only
sendmail_from   = me@localhost.com        ;for win32 only
;sendmail_path  =                   ;for unix only ...
```

Depending on whether you are using the Windows or UNIX version, PHP will send mail through an SMTP server or the local sendmail system, respectively. Setting up either of these is beyond the scope of this article, and there's plenty of information out there to help you with either. If you're running on Windows, however, chances are your ISP has already provided an SMTP server for you to use. It's the same server you set up your email program to use when sending messages. Set the SMTP setting to the hostname/IP address of that server.

`sendmail_from` should be set to the default email address from which you would like emails sent by PHP to be from. If you're administering this server, then you should probably put your email address here.

Finally, `sendmail_path` under UNIX should be uncommented (i.e. remove the semicolon from the start of the line) and set to the path and filename of the `sendmail` program on your system. Under Linux, this will usually be `/usr/sbin/sendmail`.

With these setting set and your Web server restarted, PHP should be decked out with full email capabilities. Sending an email in PHP now couldn't be easier:

```
mail("to-address@somewhere.com", "Message Subject", "This is the body of
the message.");
```

Sending to multiple recipients can be accomplished by simply separating each address with commas:

```
mail("to1@mail.net, to2@mail.net, ...", "Message Subject", "Message
body");
```

Additional headers, to specify From: or Reply-To: addresses for example, is also very easy. Just add them as a fourth parameter, separated by carriage return-newline pairs:

```
mail("to@mail.net", "Message Subject", "Message body", "From:
webmaster@host.com\r\nReply-to:admin@host.com");
```

In combination with a database, a mailing list becomes very easy to manage! Just pull the list of addresses out of the database and use the `mail` function to fire off the messages. Personalizing the messages is also very easy. Consider the following example:

```
// Retrieve $email and $password from the database based
// on the $username provided in a form.
mail($email, "Your Password",
"Hi there!
You just filled out a form on our Web site
indicating that you had lost your password.
As requested, we are sending it to you by
email.
username: $username
password: $password
Please record this information in a safe
place so you have it on hand for your next
visit to pingpongballs.com!
-The Webmaster.
");
```

If you are running under UNIX and (for whatever reason) you do not have a local sendmail system available for sending email, all is not lost. PHP comes equipped with full TCP/IP networking capabilities, which allow it to connect to an SMTP server for sending messages if it needs to. Likewise, if you need to attach files to outgoing messages, PHP is capable of this as well.

Unfortunately, the built-in `mail` function does not support either of these features, and if you need them you'll have to write your own emailing function from scratch. The authors of WROX Press' "Professional PHP Programming" have done this for you, and full code is provided in Chapter 17 of that book. Although this is an excellent book that I highly recommend (see my review), the source code is also available for free download from WROX's Web site, so there is no need to buy the book just to get this functionality.

Despite these two little stumbling blocks, PHP's built-in `mail` function provides incredible convenience and ease when sending email messages from your Web page.

Wrap-up and Thanks

And so this ten-part series draws to an end. In the past three months we've explored the MySQL RDBMS and the PHP scripting language, two products that together provide everything needed to launch a database-driven Web site with all the capabilities and administrative (not to mention financial!) benefits that stem from it.

As I wrote this series, a lot of readers wrote -- both in the SitePoint.com Forums and in private emails -- to encourage me, provide suggestions, ask lots of questions, and thank me for taking what has always seemed like a rather daunting subject and making it into something approachable and (dare I say it?) fun. While your words are all greatly appreciated (this series would not have been what it is without the level of support I have had from its readers!), I'd like to take this opportunity to thank the people who worked to make MySQL and PHP. These two products, free for personal use, together continue the long tradition of letting just about anyone project as professional and polished presence on the Internet as any big company.

A project like this is never undertaken entirely alone. I'd like to thank my colleagues Mark, Matt, and Jason at SitePoint.com for their support in this endeavor (would you believe it was their idea?). Thanks also must go to my best friend, Amy Mathieson, who didn't mind me coming all the way to Australia to visit her and spend most of my time working on this, to her sister Lisa Mathieson, who let me use her computer during my stay, and to my good friends Tony and Helen Terbizan, who provided me with an Internet connection, as well as more than one gourmet dinner!

If you enjoyed this series, I'd encourage you to post a message on the SitePoint.com Forums or drop an email message in my inbox. With this major project complete, SitePoint.com will doubtless be looking for another. Let us know what you'd like to learn next!

Thanks for reading, and see you soon!

-Kevin Yank.
Newcastle, July 2000.

URL: http://www.webmasterbase.com/article.php?aid=228&pid=0