Website Database Basics With PHP and MySQL

By Thomas Kehoe
http://www.FriendshipCenter.com/
January 11, 2000

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/PHP/DB_Basics/

Before the Beginning

Why put a database on a website?

The World Wide Web (WWW) does only one thing: provide information. If you have information about something, you can share it with the world by building a website. As your website grows you may run into two problems:

1. Your website has so much information that visitors can't quickly find what they want.
2. Visitors want to give you information.

Both of these problems can be solved by building a database on a website.

My Stuttering Science & Therapy Website has a page for persons who stutter to find penpals. This page became very popular. Nearly one thousand stutterers poured out their life stories, wanting to share their experiences with like souls. Men, women, young, old, students, attorneys, nurses, firefighters, from all over the world. Eventually the file took the better part of an hour to download on a 28.8K modem.

Maintaining the webpage took too much of my time. For every stutterer's request I approved, I rejected two or three requests from people who hadn't bothered to read that this webpage was for stutterers. Usually these were from teenagers. Some of these were sexually explicit.

Then there were the bad e-mail addresses. AOL users didn't understand that they had to attach "@aol.com" at the end of their e-mail address.

I needed a database. Users would fill out neat forms listing their age, location, profession, etc. Other users could search just for the people they wanted to meet, e.g., nursing students who stutter, or young women who stutter in Ohio. Within seconds users would find just who they were looking for.

The software could check if e-mail addresses were valid. Teenagers who didn't check the "stutterer" box could submit their penpal requests — and these requests wouldn't be read by users looking for stutterers.

Databases are everywhere on the WWW. Yahoo! and eBay are databases. When you track your Federal Express package, you search for it in a database. Amazon.com is a huge database of millions of books, CDs, and other merchandise.

Why PHP and MySQL?

There are many database applications. I'd used Filemaker Pro for almost 15 years, and 4th Dimension (4D) for six years. These applications run on Windows and the Macintosh. Filemaker Pro is easy to set up and use. You just type in the fields you want; click if they're text, numbers, dates, photos, etc.; resize boxes and change text colors to look nice on your monitor, and you're done. 4D is similar. but with more advanced "pro" features.

Filemaker Pro and 4D databases can be put on websites. I decided not to use these for three reasons:

1. My website runs on a UNIX server.
2. I kept running into things Filemaker Pro can't do.
3. Filemaker Pro and 4D can interact with other applications, but this can be difficult.

As far as I know, every major commercial website database uses a database called SQL. SQL is not a database application, but rather is a langauge for querying a database. It stands for Structured Query Language. The most popular "pro" SQL database application is Oracle. The big boys use this, and it costs a fortune.

In recent years several companies have developed "open source" SQL applications. The most popular is MySQL. It's more or less free, and more or less as powerful as Oracle, at least for small to medium-sized databases. MySQL runs under UNIX (there are also Windows versions).

To run MySQL on a website, you need a scripting language to make your HTML pages talk to the database. Perl used to be popular, but the consensus seems to be that PHP is better. Don't ask me to explain the differences — I used Perl once and it worked fine, but everyone seems to be using PHP.

The other main scripting langauge is Java. Java has the advantage of running client-side scripts, in other words, programs can be downloaded and run on the visitor's computer. This is a good idea if a program will be run many times, and the user has a slow modem connection. I don't know much about Java — again, it seems like everyone uses PHP with MySQL, and this works for me so I haven't learned Java.

Reference documentation

This is a tutorial. I'll tell you how to use the most popular features of PHP and MySQL. You'll also need reference documentation, to look up features I skipped or covered quickly.

The powerful UNIX operating system runs most web servers. UNIX is not like Windows or the Macintosh. MySQL runs only on UNIX (a Windows version is under development). I use O'Reilly's UNIX In A Nutshell reference book.

HTML is the language for the static (text, graphics) and structural parts of websites.

- I use O'Reilly's HTML: The Definitive Guide reference book.
- I've heard good things about John G. Gilson's HTML tutorial.

PHP is the language I use for the dynamic or interactive parts of websites.

- The main documentation for PHP is on-line. Comments from users are useful.
- The book Core PHP Programming is mostly a re-hash of the on-line documentation, but sometimes explains something better.
- The book PHP Programming: Browser-Based Applications is pretty good, although I haven't used it extensively.
- I haven't had time to read Professional PHP Programming, but the user comments on Amazon.com are good.
- The PHP e-mail list gets 100+ messages a day.

SQL is a language for interacting with databases. MySQL is a database that understands the SQL language.

- I've tried three reference books, and the one I like is Introduction To SQL, by Rick van der Lans.
- On-line documentation is on the MySQL website.
- O'Reilly's MySQL & mSQL covers what's unique about MySQL, but you still need a SQL reference book.
- The MySQL e-mail list gets 50+ messages a day.

There is also third-party documentation and tutorials on the WWW, including:

- Webmonkey has tutorials for HTML, Java, PHP, etc. They have a tutorial about building databases with PHP and MySQL.
- Philip and Alex's Guide to Web Publishing has stuff about HTML and databases.
- PHP Wizard has tutorials and examples.

Software applications you'll need

PHP and MySQL are more or less free, but getting the applications to run may be a challenge. If you have a computer running UNIX, you can download (PHP, MySQL) and install the applications. The documentation above explains how.

For me it's easier to pay for an account on a UNIX server, and let someone else do the administration. A web search for "MySQL website hosting" will turn up many host companies. I'm happy with phpwebhosting.com.

You'll need a World Wide Web browser, such as Netscape Navigator or Microsoft's Internet Explorer.

To access a remote UNIX server from a personal computer, you'll need a software application called a "terminal emulator". A terminal emulator opens a window into which you can type command lines to do stuff on the UNIX server. If you used computers before the Macintosh and Windows, this will be familiar to you. I don't know about terminal emulators for Windows. Macintosh terminal emulators are available from White Pine Software. You may be able to find an old shareware terminal emulator, but it may not connect via TCP/IP (in other words, it'll call a computer directly via a modem, but won't connect via the Internet).

If you're using a remote UNIX server, you'll also need a file transfer (FTP) application. You'll use this to move documents you created on your computer to the UNIX server. On the Macintosh, the most popular FTP applications is Fetch.

Another software application you'll need is an ASCII text editor. ASCII is "plain text" letters, numbers, and punctuation. It doesn't have bold or italic or different fonts or font sizes. Most word processors will "Save As…" a file to "text" or ASCII. I find this doesn't work well in Microsoft Word 98, and I hate how it goes into "browser mode" when it sees HTML code, so I don't recommend using Word 98. Instead, I use BBEdit, which runs on the Macintosh. Keith Edmunds wrote an excellent review of 19 text editing applications with an eye for how well they write PHP code.

I'm not aware of any "integrated design environments" (IDE) for PHP. An IDE such as Codewarrior supposedly makes it easier to program in C, C++, Java, and Pascal. However, I find these "power tools" are confusing for a beginner.

Running PHP

Let's see if PHP runs. We'll start with printing "Hello, world." on your browser.

In your text editor, create a document called `helloworld.php`. In the document, enter the following code:

```
<html>
<body>

<?php
print "Hello, world.";
?>

</body>
</html>
```

Save the document as ASCII text.

If you're running PHP on your computer, open the document with your browser. You should see "Hello, world." in the browser window.

If you're running PHP on a remote UNIX server:

1. Save the document as ASCII text.
2. Connect to the Internet.
3. Transfer the file (using Fetch or equivalent) `helloworld.php` to your server. You could use your terminal emulator and a UNIX text editor to create the document `helloworld.php` on the UNIX server, but I find it easier to use my computer's text editor and then transfer the file.
4. Use your browser to go to the file `helloworld.php` on your UNIX server. I.e., use Internet Explorer or Netscape Navigator to go to `http://www.yourwebsite.com/helloworld.php`.

You should see "Hello, world." in the browser window. Remember this three-step process for viewing your PHP creations: save as ASCII text, transfer to the UNIX server, view with your browser.

If it didn't work:

1. Use Fetch or your terminal emulator to make sure the file `helloworld.php` is on your UNIX server. There may have

been a problem transfering the file.
2. PHP may not be running. Contact your system administrator.

Running MySQL

Now we'll check if MySQL is running. If you're using a remote UNIX server, use your terminal emulator to go to your UNIX server.

At your prompt, type `mysql -u username -p`. The server should then ask for your password. Then you'll get a welcome message and the prompt changes to `mysql>`. The whole exchange should look like:

```
$ mysql -u username -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 31110 to server version: 3.22.25-log

Type 'help' for help.

mysql>
```

If your prompt changes to `mysql>`, then MySQL is running.

If it didn't work:

If you got a response such as

```
bash: mysql: command not found
```

then MySQL is not installed on your computer. Contact your system administrator.

If MySQL is running, enter your database by typing (replace "dbname" with the name of your MySQL database):

```
use dbname;
```

You should get the response:

```
Database changed
```

If it didn't work:

If you got a response such as

```
ERROR 1044: Access denied for user
```

the problem may be that you need to create a database. My system administrator did this when he set up my account, so I don't know how to do it. Look in the MySQL documentation.

Now create a table in your database. Replace "tablename" with the name of your table.

```
CREATE TABLE tablename (
first_name VARCHAR (25),
last_name VARCHAR (25)
);
```

Now check that your table is there by typing:

```
show tables;
```

You should get a list of tables:

```
+------------------+
| Tables in dbname |
+------------------+
| tablename        |
+------------------+
2 rows in set (0.00 sec)
```

If PHP and MySQL are running, then the next chapter will show you how to make HTML forms run PHP scripts that query a MySQL database. Now quit MySQL by typing

```
quit
```

You should get your UNIX prompt back. Now that everything is running, we'll work on getting PHP and MySQL to talk to each other and to HTML.

HTML talks to PHP talks to MySQL

1. Mixing HTML and PHP
2. PHP `prints` to HTML
3. PHP submits data to MySQL
4. PHP retrieves data from MySQL
5. Testing whether your query worked

Mixing HTML and PHP

Let's examine how `helloworld.php` worked. I assume you know the basics of HTML, so the first two lines and the last two lines should be familiar to you:

```
<html>
<body>

<?php
print "Hello, world.";
?>

</body>
</html>
```

`<html>` means that this is an HTML document, intended to be read by a browser such as Netscape or Internet Explorer.

`<body>` sets out the body of the HTML document, which is displayed to the user (as opposed to the header's invisible information).

`</body>` and `</html>` close the body and the HTML document.

Alternative means to the same end

`helloworld.php` is an HTML document with a embedded PHP script. We could alternatively write a PHP script with HTML commands:

```
<?php
```

```
print "<html>";
print "<body>";
print "Hello, world.";
print "</body>";
print "</html>";

?>
```

Either way produces the same result. With some PHP functions (e.g., cookies) you have to use the latter method (PHP scripts with embedded HTML).

PHP `prints` to HTML

The simplest way for PHP to talk to HTML is to throw some text onto the browser. We did that with the PHP script:

```
<?php
print "Hello, world.";
?>
```

Let's go over the `print` function more carefully.

PHP scripts always begin with `<?php`. You can also use `<?` but some applications (e.g., XML, FrontPage) prefer the full `<?php`. A PHP script is closed with `?>`.

This PHP script has one line:

```
print "Hello, world.";
```

PHP lines always end with a semi-colon (`;`).

`print` is the PHP function that sends text to the browser. Between the `print` and `;` we put a string (strings are text to be read literally). Strings are denoted by quotation marks. Whatever is between the quotation marks will be sent to the browser.

Alternative means to the same end

There are several other ways to send "Hello, world." to your browser. The following file is `print.php`.

```
<html>
<body>
<?php
print "This uses the print function.";
print "<p>";
echo "This uses the echo function.", "  ", "P.S. You can add a second string", " ", "if you separate strings wi
print "<p>";
printf ("This uses the printf function.");
print "<p>";
printf ("The printf function is mostly used to format numbers.");
print "<p>";
printf ("Remember the parentheses with printf.");
?>
</html>
</body>
```

`print.php` produces the following output:

```
This uses the print function.

This uses the echo function. P.S. You can add a second string if you separate strings with a comma.

This uses the printf function.

The printf function is mostly used to format numbers.

Remember the parentheses with printf.
```

`print` is the simplest function for throwing text onto the browser window.

`echo` is like `print` but you can add additional strings, separated by commas.

`printf` will format numbers as integers, scientific notation, etc. `printf` requires parentheses.

Parentheses are an issue with the three print functions:

- `echo` must not have parentheses.
- `printf` must have parentheses.
- `print` works with or without parentheses.

Printing strings and numbers is easy, but how do you print arrays? (Arrays are sets of things.) If you try

```
print $myarray;
```

the result will be `Array`, in other words, PHP informing you that *$myarray* is an array. This is useful when you're not sure whether a variable is an array, but doesn't help when you want to see the contents of the array.

You could use the `implode` function to convert the array into a string, then print the string. The first argument is the array, the second argument is a delimiter to separate the array elements:

```
$implodedarray = implode ($myarray, ", ");
print $implodedarray;
```

Another way to print an array uses `array_walk`. This function executes a function on each element in an array. You must create the function to be executed — you can't use a PHP function such as `print`:

```
function printelement ($element)
{
   print ("$element<p>");
}

array_walk($myarray, "printelement");
```

PHP submits data to MySQL

I assume you know something about HTML forms. The following **submitform.html** is very simple:

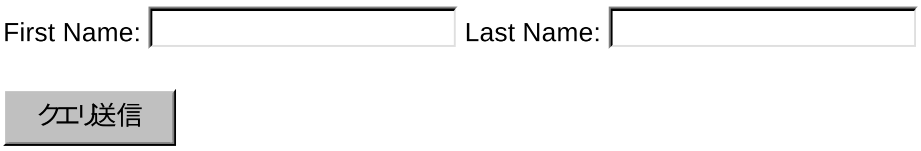
```
<html>
<body>
<form action=submitform.php method=GET>

First Name: <input type=text name=first_name size=25 maxlength=25>

Last Name: <input type=text name=last_name size=25 maxlength=25>
<p>
<input type=submit>

</form>
</body>
</html>
```

submitform.html produces the following form:



When you enter data in the fields and then click **Submit**, the form sends the data to **submitform.php**. Here is **submitform.php**:

```
<html>
<body>
<?php

mysql_connect (localhost, username, password);

mysql_select_db (dbname);

mysql_query ("INSERT INTO tablename (first_name, last_name)
                VALUES ('$first_name', '$last_name')
             ");

print ($first_name);

print (" ");

print ($last_name);

print ("<p>");

print ("Thanks for submitting your name.");

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

In line 3, you must replace "username" and "password" with your username and password. In line 5 you must replace "dbname" with the name of your MySQL database. In line 13 you must replace "tablename" with the name of one of your MySQL tables.

If you open **submitform.html**, enter a name, and press **Submit**, you should see the name you entered printed on a new webpage. Also note that in the browser URL address window, the URL includes the two fields, looking something like:

```
…/submitform.php?first_name=Fred&last_name=Flintstone
```

Because we used the `GET` form method, the fields are passed from `submitform.html` to the URL header of `submitform.php`. PHP then automatically creates variables for each input field name passed from the HTML form. PHP variables always start with a dollar sign, so the variables are `$first_name` and `$last_name`.

Check that your entered name was successfully entered into your database. Open MySQL, and at the `mysql>` prompt, enter

```
select * from tablename;
```

You should get a table showing the name that you entered:

```
+------------+------------+
| first_name | last_name  |
+------------+------------+
| Fred       | Flintstone |
+------------+------------+
1 rows in set (0.00 sec)
```

Let's examine how `submitform.php` works:

The first two lines are:

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

mysql_select_db (dbname);
```

These two functions open the MySQL database.

The next line does the work:

```
mysql_query ("INSERT INTO tablename (first_name, last_name)
                VALUES ('$first_name', '$last_name')
             ");
```

`mysql_query` enables PHP to throw SQL commands at the MySQL database. You can put any SQL commands after the `mysql_query` function. The SQL commands must be enclosed in parentheses and quotations.

Here's a weird thing: MySQL lines end with a semi-colon (`;`) and PHP lines end with a semi-colon, but MySQL lines in PHP lines don't end with a semi-colon. In other words, when you're at the MySQL command line you would type in:

```
INSERT INTO tablename (first_name, last_name)
  VALUES ('$first_name', '$last_name');
```

But when included in a PHP line, you leave off the semi-colon. The weird part is that `SELECT` and `INSERT` will work with or without the extra semi-colon, but `UPDATE` won't work. I haven't experimented with `DELETE`, which is the fourth SQL command.

After that, there are five **print** lines to display the entered name (separated by a space) and a thank-you closing line (preceded by a paragraph code).

PHP retrieves data from MySQL

Now we'll create another HTML form to search the database. We'll call it **searchform.html**:

```
<html>
<body>

<form action=searchform.php method=GET>

Search For:
<p>
First Name: <input type=text name=first_name size=25 maxlength=25>
<p>
Last Name: <input type=text name=last_name size=25 maxlength=25>
<p>
<input type=submit>

</form>
</body>
</html>
```

You will also have to create the following file **searchform.php**:

```
<html>
<body>

<?php

mysql_connect (localhost, username, password);

mysql_select_db (dbname);

if ($first_name == "")
{$first_name = '%';}

if ($last_name == "")
{$last_name = '%';}

$result = mysql_query ("SELECT * FROM tablename
                         WHERE first_name LIKE '$first_name%'
                         AND last_name LIKE '$last_name%'
                       ");

if ($row = mysql_fetch_array($result)) {

do {
  print $row["first_name"];
  print (" ");
  print $row["last_name"];
  print ("<p>");
} while($row = mysql_fetch_array($result));

} else {print "Sorry, no records were found!";}

?>

</body>
</html>
```

Remember to replace "username", "password", "dbname", and "tablename".

When you open **searchform.html** you should see a form. If you enter a name and click **Submit**, you should get a new webpage showing the full record or records matching your search terms.

Going over **searchform.php** carefully, it begins with the familiar lines to open the MySQL database. Then there are four lines:

```
if ($first_name == "")
{$first_name = '%';}

if ($last_name == "")
```

```
{$last_name = '%';}
```

These lines check if the form fields are empty. The `if` function is followed by parentheses, and what's in the parentheses is the statement to be tested. The statement `$first_name == ""` means "The variable $first_name is empty." Note that the double equals signs means "equals". A single equals sign means "assign the value from what's on the right to the on the left."

The next line is what is done when the `if` statement is evaluated as true. What's done is a PHP line, so it ends with a semi-colon. (Note that the `if` function doesn't end with a semi-colon.) The PHP line is put in curly brackets.

The `%` is SQL's character string wildcard. (Don't confuse it with SQL's `*` column name wildcard.) The first two lines together mean "If the 'first name' field is empty, search for any first name." The latter two lines similarly checks if the `$last_name` variable is empty, it searches for any last name. This enables you to find everyone named "Fred" or everyone named "Flintstone".

```
$result = mysql_query ("SELECT * FROM tablename
                         WHERE first_name LIKE '$first_name%'
                         AND last_name LIKE '$last_name%'"
                       ");
```

This line does most of the work. When `mysql_query` does a `QUERY`, the result is an integer identifier. The result identifier could be "2".

This query selects all the columns from the specified table, then searches for records in which the "first_name" column matches the "$first_name" variable from `searchform.html` and the "last_name" column matches the "$last_name" variable. Note the `%` wildcard: this enables the user to type only the first few letters of a name, e.g, "Flint" finds "Flintstone".

```
if ($row = mysql_fetch_array($result)) {

do {
  print $row["first_name"];
  print (" ");
  print $row["last_name"];
  print ("<p>");
} while($row = mysql_fetch_array($result));

} else {print "Sorry, no records were found!";}
```

The last part of the script throws the search results onto the new webpage. `mysql_fetch_array` grabs the first row of the query results. This function's argument is the result identifier (`$result`). Successive calls to `mysql_fetch_array` will return the next rows of the query.

The array variable `$row` is created and filled with the first row of the query results.

If a matching record was found, then the block in the outermost curly brackets is done:

```
do {
  print $row["first_name"];
  print (" ");
  print $row["last_name"];
  print ("<p>");
} while($row = mysql_fetch_array($result));
```

This is a `do…while` loop. Unlike a `while` loop, the block of code is done, then afterwards a decision is made whether to do the block of code again. A `while` loop decides first whether to do the block of code.

What is done is inside the nest set of curly brackets:

```
print $row["first_name"];
print (" ");
print $row["last_name"];
print ("<p>");
```

The integer `$row` specified which record to print. The first time through, the first "first_name" is printed, followed by a space, and then the first "last_name" is printed, followed by a paragraph mark.

Now we get to the `while` decision. The MySQL database is called again, using the `mysql_fetch_array` function. `mysql_fetch_array` calls the next row until no rows are left. The `do` block is then executed again.

When no rows are left, `mysql_fetch_array` returns `false`, the `do…while` loop is stopped, and the `if` function is exited.

Why didn't we just `implode` the results of `mysql_fetch_array($result)` and print the imploded string? This results in printing each element twice. `mysql_fetch_array($result)` allows you to specify elements by the field name (e.g., `"first_name"` or by the number of the field, with the first field being "0", the second field is "1" and so on. Thus we could have written this block of code as:

```
print $row[0];
print (" ");
print $row[1];
print ("<p>");
```

We could also shorten these four lines of code into one line:

```
echo $row[0], " ", $row[1], "<p>";
```

If no matching records were found, then `mysql_fetch_array` returns nothing and no value can be assigned to *$row*, and the `else` clause is executed:

```
else {print "Sorry, no records were found!";}
```

Testing whether your query worked

Did your `SELECT`, `DELETE`, or other query work? Good question, and not always easy to answer.

Testing an `INSERT` query is relatively simple:

```
$result = mysql_query ("INSERT INTO tablename (first_name, last_name)
                          VALUES ('$first_name', '$last_name')
                       ");

if(!$result)
{
   echo "<b>INSERT unsuccessful:</b> ", mysql_error();
   exit;
}
```

But that code doesn't work with a `SELECT` query. Instead, I use:

```
$selectresult = mysql_query ("SELECT * FROM tablename
                                WHERE first_name = '$first_name'
                                AND last_name = '$last_name'
                             ");

if (mysql_num_rows($selectresult) == 1)
{
  print "Your SELECT query was successful.";
}

elseif (mysql_num_rows($selectresult) == 0)
{
  print "Your SELECT query was not successful.";
  exit;
}
```

And that code doesn't work with a `DELETE` queries. Here's how to test those:

```
$deleteresult = mysql_query ("DELETE FROM tablename
                                WHERE first_name = '$first_name'
                                AND last_name = '$last_name'
                             ");

if (mysql_affected_rows($deleteresult) == 1)
{
  print "Your DELETE query was successful.";
}

elseif (mysql_affected_rows($deleteresult) != 1)
{
  print "Your DELETE query was not successful.";
  exit;
}
```

Verifying form data

Some visitors to your website will try to enter invalid data into your database. You'll want to write a verification script to clean up entered data.

Trimming white space

`trim` will remove white spaces from the beginning and end of the data:

```
trim ($first_name);
```

Required fields

Some of your fields will be required. In other words, visitors have to enter something in the field. The following script checks that a first name was entered:

```
if (ereg(".", $first_name) == 1)
{
    echo "First name: ", "$first_name";
    $verify = "OK";
}
else
{
    print ("<b>Error:</b> A first name is required.");
    $verify = "bad";
}
```

`ereg` means "evaluate regular expression". "Regular expressions" are the UNIX function for finding patterns in strings of letters and numbers. `ereg` is followed by parentheses, and you can put three arguments in the parentheses. The arguments are separated by commas. The first argument is the pattern to search for, usually surrounded by quotation marks. The second argument is where `ereg` is to search, usually a variable. The third, optional, argument is an array to put matches into. This argument is a variable.

`ereg` returns either a "0" (false) or a "1" (true).

The dot `.` or period is a regular expression wild card meaning "any character."

`(ereg(".", $first_name) == 1)` means "the variable '$first_name' contains anything". If this expression is true, then the first name is printed, and the variable `$verify` is set to "OK".

The `else` argument executes when `ereg` returns "0" (false).

There are three other versions of the `ereg` function.

1. `ereg_replace` uses three arguments: the first is the pattern to search for, the second is the pattern to replace the first pattern, and the third is where to search (a variable).
2. `eregi` is the same as `ereg`, except that it's not case-sensitive (i.e., it doesn't differentiate upper- and lower-case letters).
3. `eregi_replace` is not case sensitive

Alternative means to the same end

The line `if (ereg(".", $first_name) == 1)` can be simplified to `if ($first_name)`. I used the longer form to show how to use `ereg` in a simple example.

Checking e-mail addresses

The following `ereg` arguments test validity of e-mail addresses:

"\@"	Must include @
"^\@"	Can't begin with @
"\@.*\."	Must have characters between @ and .
"\....*"	At least two characters after the .
" "	No spaces permitted
"<" ">"	No angle brackets permitted

Checking that the username is unique

You may need to make sure no two visitors try to use the same name:

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

mysql_select_db (dbname);

$result = mysql_query ("SELECT * FROM tablename
                         WHERE USER_NAME = '$USER_NAME'
                       ");

if ($row = mysql_fetch_array($result))
{
    print ("<b>Error:</b> The user name <b>");
    print ("$USER_NAME");
    print ("</b> has been taken.  Go back and enter a new user name.");
    print ("<p>");
    $verify = "bad";
}
else
{
    print ("User name: ");
    print ("$USER_NAME");
}
```

This code connects to MySQL, then searches the database for the entered username. Note that the `=` equals sign is used for an exact search, when previously we used `LIKE` to do wildcard searches. If any result is found, the visitor is told to enter a new username.

Using cookies to identify and track visitors

By assigning a "cookie," your website can identify and track a visitor across webpages. A cookie is a set of data stored in the visitor's browser. We'll look at how to assign a unique customer ID number to each visitor and set this as a cookie. Your visitors can then repeatedly access your database, with their data always going to their own record.

Note that the on-line PHP manual describes cookies in the HTTP functions chapter, not in the chapter titled Cookies.

Viewing your browser's cookies

Take a look at the cookies stored by your browser. With Microsoft Internet Explorer, go to `Edit...Preferences...Receiving Files...Cookies`. Alternatively, you can search your hard drive for a file called `cookies.txt`. You should see a list of servers, e.g., `amazon.com`. Choose a cookie and click `View`. You'll see six properties set by the website that created the cookie, plus whether the cookie is enabled. For example:

Cookie Properties

Name:	ubid-main
Server:	amazon.com
Path:	/
Value:	012-3456789-0123456
Expires:	Tue, Jan 1, 2036 8:00 AM GMT
Secure:	No
Status:	Enabled

When your browser pulls a webpage from the Internet, it compares the domain name of the webpage to the domain names in its cookie list. If there's a domain name match, then the browser sends the matching cookie(s) to the server. The server then can create a custom webpage using data from that cookie.

For example, when I connect to Amazon.com, my browser sends four cookies to Amazon before Amazon sends me the first webpage. `ubid-main` is the cookie that identifies me. Amazon looks up the `Value:` number in its database, matches it to my name, and creates a custom webpage that begins:

Season's greetings, Thomas D Kehoe. (If you're not Thomas D Kehoe, click here.)

Setting cookies

Cookies have to be set before the server sends anything to the browser. E.g., Amazon had to identify me before it a custom webpage welcoming me. To accomplish this, cookies must be set before the `<HEAD>` HTML tag. Actually, the cookie must be set before the `<HTML>` tag, as the following example shows:

```
<?php

setcookie("CookieID", $USERID);

?>

<HTML>
<BODY>

</BODY>
</HTML>
```

If you get the error `Warning: Oops, php3_SetCookie called after header has been sent`, then you sent the <HTML> tag before setting the cookie. The error message is confusing because the above example doesn't send any header information using the `<HEAD>` tag.

The `setcookie` function takes up to six arguments, separated by commas:

1. The cookie name, a string, e.g., `"CookieID"`. Semi-colons, commas, and white spaces are not allowed. The cookie name is required (all other arguments are optional). If only the cookie name is included, and no other arguments, the cookie will be deleted.
2. The cookie value or contents, a string, e.g. `$USERID`. To skip, use an empty string (""). Slashes apparently are not allowed.
3. The time the cookie expires, an integer. If this is omitted (or filled with a zero) the cookie will expire when the session ends. The time can be an absolute time, in the format DD-Mon-YY HH:MM:SS, e.g., `"Wed, 24-Nov-99 08:26:00"`. Or, more usefully, the date can be in relative time. This is done with the UNIX `time()` or `mktime` functions. For example, `time()+3600` makes the cookie expire in one hour. Some older browsers don't handle cookies properly if the time argument is omitted.
4. The UNIX directory path. This is used to identify cookies beyond the domain name identification. The path "/" is the same as omitting this argument — except that some browsers don't handle cookies correctly if the path is not set, so use the slash instead of omitting this argument. Note that Netscape's cookie specification puts `domain` before `path`, but PHP puts `path` before `domain`.
5. The domain name of the server, for matching cookies. If omitted, the domain name is taken from the webpage the cookie is sent from. Note that you must put a period (`.`) before the domain name, e.g., ".friendshipcenter.com". Cookies are rejected unless they have at least two periods (for the domains com, edu, net, org, gov, mil, and int; all other domains require at least three periods).
6. `secure` is set by an integer. `1` means that the cookie can only be sent via a secure network. `0` (or omitting this argument) allows the cookie to go over unsecured networks.

There are many bugs in older browsers that screw up cookies. See the reader's notes in the on-line PHP manual for details and fixes.

Cookies are variables

When a PHP script receives a cookie from client browser, it's automatically converted into a variable. E.g., a cookie named `CookieID` becomes the variable *$CookieID*.

To see a cookie, print the variable:

```
print $CookieID;
```

Alternative means to the same end

Cookies are stored in the array `HTTP_COOKIE_VARS`. You can print a cookie's value with:

```
print $HTTP_COOKIE_VARS[CookieID];
```

Setting a cookie from a database lookup

Going back to our webpage `submitform.php`, which inserted the visitor's name into our database, let's add code to look up the USERID number our database automatically assigns to each submitted name, and then send a cookie to the visitor's browser with the value set as the USERID number.

But first, let's look at `AUTO_INCREMENT`. MySQL can be set to assign a number to each new record, starting with "1". The next inserted record gets "2", the next is "3", etc. You can add such a column, in this case called `USERID`, with this bit of SQL:

```
ALTER TABLE dbname
    ADD COLUMN USERID INT(11) NOT NULL PRIMARY KEY AUTO_INCREMENT;
```

The new field `USERID` is set as an 11-digit integer (allowing nearly 100 billion records); the field is not allowed to be empty (`NOT NULL`), the database is indexed by this field (`PRIMARY KEY`), and, lastly, `AUTO_INCREMENT` is set.

To set a cookie in the visitor's browser after he or she inserts his name into your database, with the value of the cookie taken from the USERID, you could do this:

```
<?php

mysql_connect (localhost, username, password);

mysql_select_db (dbname);

mysql_query ("INSERT INTO tablename (first_name, last_name)
              VALUES ('$first_name', '$last_name')
             ");

setcookie("CookieID", mysql_insert_id(), time()+94608000, "/");  /* expires in 3 years */

?>
```

The PHP function `mysql_insert_id()` returns the `AUTO_INCREMENT` number assigned in the last `INSERT` query. No arguments are required, although you can put in a variable which has been assigned the value of the `mysql_query`.

Try it out and then look at your browser's cookie list. You should see "CookieID" listed. Use your terminal emulator to view the contents of your MySQL table and see that the `USERID` of the last submission is the same as the value of the cookie listed in your browser.

Receiving a cookie

Let's write a PHP script for a webpage like Amazon.com. First, the PHP script checks if the client's browser has sent a cookie. If so, the visitor's name is displayed. If no cookie is found, a form is displayed for the visitor to submit their name, which is then added to the database and a cookie is set in the client's browser.

First, let's create a webpage that displays the visitor's cookie:

```
<?php

print $CookieID;

?>
```

Save this script as `cookiepage.php`. If you save this to your UNIX server, then open the webpage after running the last version of `submitform.php`, you should get the value of your cookie. You can check it against your browser's cookie list and your MySQL database.

Now let's make `cookiepage.php` welcome me by name:

```
<?php

mysql_connect (localhost, username, password);

mysql_select_db (dbname);

$selectresult = mysql_query ("SELECT * FROM tablename
                                WHERE USERID = '$CookieID'
                             ");

$row = mysql_fetch_array($selectresult);

echo "Welcome ", $row[first_name], "!";

?>
```

Weird SQL: What The Books Don't Tell You

SQL is supposed to be like English. You just tell your database what you want, and it finds it, right? Well, there's a few surprises…

1. The trailing semi-colon, or lack thereof
2. Datatypes
3. Wild cards
4. `NOT NULL` and empty records

The trailing semi-colon, or lack thereof

The first thing about MySQL you learn is that every line ends with a semi-colon (;). Well…there are (at least) two exceptions.

In the section PHP submits data to MySQL I pointed out that when a MySQL line is part of a PHP line, the semi-colon at the end of the MySQL line of omitted. For example:

```
mysql_query ("INSERT INTO tablename (first_name, last_name)
                VALUES ('$first_name', '$last_name')
             ");
```

This is done because PHP lines also end with a semi-colon, so an extra semi-colon might confuse the PHP parser. You leave off the semi-colon, and PHP automatically puts it back in for you.

As I wrote, the weird part is that SELECT and INSERT will work with or without the extra semi-colon, but UPDATE won't work. SELECT and INSERT are the first MySQL functions you use, so you're happily coding with both semi-colons, and then when you want to UPDATE a record everything stops working. I haven't experimented with DELETE, which is the fourth SQL command.

The other time you don't use a semi-colon is when you want to see all the fields (what SQL calls "columns") displayed vertically down your monitor, instead of horizontally across your monitor. With a terminal emulator (at least with my old terminal emulator) you have a choice of 80 or 132 columns (of characters), but you can't scroll over to see stuff farther to the right. So you end the SQL line with \G instead:

```
SELECT * FROM PENPALS
  WHERE USER_ID = 1\G
```

TEXT, DATE, and SET datatypes

MySQL fields must have a datatype specified. There are about 25 choices. Most are fairly straightforward. Some details to remember:

TEXT is not a datatype, despite what some books tell you. The datatype is called LONG VARCHAR or MEDIUMTEXT.

For VARCHAR weirdness, see the section on wildcards.

The DATE dataset formats dates as YYYY-MM-DD, e.g., 1999-12-08. This is logical because we write numbers with the biggest (e.g., millions) to the left, then smaller numbers (e.g., thousands, hundreds, tens, ones) progressively to the right. You can retrieve the current date, in the same format with the PHP function

```
date("Y-m-d")
```

It's simple to subtract a stored date (e.g., someone's birthdate) from the current date:

```
$age = ($current_date - $birthdate);
```

SET is a useful datatype. It's like ENUM except that it'll store multiple values. Also, it can only take up to 64 predefined (enumerated) values, when ENUM can handle up to 65,535 predefined values. But if you need more than 64 values, it's easy to divide your list into two or more columns. More about the SET datatype in the checkboxes chapter.

Wildcards

SQL sometimes uses the asterisk (*) as a wildcard, and sometimes use a percent sign (%). E.g., suppose you want to see all of the records in your database:

```
SELECT * FROM dbname
  WHERE USER_ID LIKE '%';
```

Yes, I know that SELECT * FROM dbname; will work just as well. My point is that there are two wildcards, meaning the same thing but used in different contexts, for no reason I can see.

Another non-obvious thing is that the % wildcard requires using LIKE. It won't work with =.

There's another wildcard, with a different meaning. The underscore (_) means "any single character."

NOT NULL and empty records

What happens when a user leaves a field blank? If you require a value in the field, you wrote a verification script requiring a value. But some fields are OK to leave empty. MySQL will do any of three things:

- Insert the value NULL. This is the default action.
- If you declared the column NOT NULL (when you created the column, or by modifying the column), MySQL will leave the record empty.
- In an ENUM datatype, if you declared the column NOT NULL, MySQL will insert the first value of the enumerated set. In other words, MySQL treats the ENUM datatype as if you declared the first value to be the DEFAULT value. To work around this weirdness, make the first value a pair of single quotes (''), which means " empty set".

The difference between NULL and an empty record is that the % wild card finds empty records, but doesn't find NULL records. I have yet to find a situation where the latter result is desirable. In my experience, all columns should be declared NOT NULL. Then the following SELECT query works:

```
if (!$CITY) {$CITY = "%";}

$selectresult = mysql_query ("SELECT * FROM dbname
                              WHERE FIRST_NAME = 'Bob'
                              AND LAST_NAME = 'Smith'
                              AND CITY LIKE '$CITY'
                           ");
```

The first line says that if the user doesn't specify a city, the % wild card is used for the search, to find any city, or empty CITY records.

If every record contains a city, then the query returns all Bob Smiths in your database. If some CITY records are empty, the query also returns all Bob Smiths in your database. But if some CITY records contain NULL, your query won't return the Bob Smiths with a NULL value in the CITY column.

Can we solve that problem with:

```
if (!$CITY) {$CITY = "%";}

$selectresult = mysql_query ("SELECT * FROM dbname
                              WHERE FIRST_NAME = 'Bob'
                              AND LAST_NAME = 'Smith'
                              AND (CITY LIKE '$CITY' OR CITY IS NULL)
                           ");
```

Note that to search for NULL you must use IS. = or LIKE will not find NULL values.

If the user enters "Altoona" for the city, the query returns every Bob Smith in

```
Altoona, and every Bob Smith with NULL in the CITY field. That isn't what the user
wanted. It'd better to declare every column to be NOT NULL and avoid this problem.

One last pitfall to watch out for. If you add (or modify) columns after some
records are already in your database, you may get a mixture of NULL and empty
records. This is certain to screw up your SELECTqueries.
```

Checkboxes and other HTML form processing
HTML forms are easy to design, as long as you allow one value per field. When you allow more than one value, the processing gets tricky.

1. Checkboxes
2. `SELECT multiple` scrolling lists
3. Searching with multiple values

Checkboxes

Checkboxes are the simplest way to allow users to enter more than one value into a field:

What pets do you have?

☐ Dog
☐ Cat
☐ Fish

You can check one, two, or all of the pets. The HTML code looks like this:

```
What pets do you have?
<FORM>
<INPUT TYPE=checkbox NAME=PET_ARRAY[] value=dog> Dog<br>
<INPUT TYPE=checkbox NAME=PET_ARRAY[] value=cat> Cat<br>
<INPUT TYPE=checkbox NAME=PET_ARRAY[] value=fish> Fish<br>
</FORM>
```

The MySQL field name is `PET`, but here we use `PET_ARRAY[]`. When the user clicks the `SUBMIT` button, the values are passed to the header looking like this:

```
http://www.mywebsite.com/myform.php3?PET_ARRAY%5B%5D=dog&PET_ARRAY%5B%5D=cat
```

`5B` is 91 in hexadecimal, and the HTML character entity for left square bracket — `[` — is &#091, so `%5B` means left square bracket. `5D` is 93 in hexadecimal, and the HTML character entity for right square bracket — `]` — is &#093, so `%5D` means right square bracket.

When we get to the PHP script that processes this form, we use this script to put both values into one field. The `PET` field is a `SET` datatype.

```
if ($PET_ARRAY)
{
          $PET = implode($PET_ARRAY, ",");

          $result = mysql_query ("UPDATE dbname
                                    SET PET = '$PET'
                                 ");
          if(!$result)
          {
             echo "<B>UPDATE unsuccessful:</b> ", mysql_error();
             exit;
          }
}
```

`if ($PET_ARRAY)` checks if the user checked any of the boxes. If the user doesn't have any pets, the field is left empty.

`$PET = implode($PET_ARRAY, ",");` converts the array into a string, with the elements separated by commas. The values passed to the header above would come out as

```
dog,cat
```

The query then puts the string into the `PET` of the database.

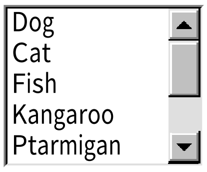
To search a `SET` datatype, remember to put `%` wildcards before and after the search value. This is necessary to find one of several values, and ignore the commas. E.g.,

```
SELECT * FROM dbname
  WHERE PET LIKE '%$PET%';
```

`SELECT multiple` scrolling lists

Another way to allow selection of more than one value is to use pull-down menus or scrolling lists. E.g.,

What pets do you have?



```
<FORM>
What pets do you have?
<FORM>
<SELECT NAME=PET_ARRAY[] size=5 multiple>
<option>Dog
<option>Cat
<option>Fish
<option>Kangaroo
<option>Ptarmigan
<option>3-Toed Sloth
<option>Lemur
<option>Narwhal
</select>
</FORM>
```

If you `shift-click`, you can select two or more adjacent values, e.g., Dog, Cat, and Fish. With Windows, you `ctrl-click` to select two or more non-adjacent values. On the Macintosh you hold down the butterfly key as you click your mouse.

99.999% of Internet users don't know this, and 99.99% aren't going to read instructions you provide, so I don't use `SELECT multiple` scrolling lists. Checkboxes are more obvious.

Searching with multiple values

This is another short section. I haven't figured out how to search for more than one value, e.g., allow users to query a `SET` of pets to find people with a dog and a fish. If I figure it out I'll explain it here.

Using include() to bring in outside files
Sometimes you have a block of text that is repeated on many webpages. For example, all of these chapters start with "PHP and MySQL Website Database Basics" and the copyright, etc. Later you need to change one word, and you have to go into 20 documents to change the same word.

A better way is to put the block of text into its own document, and use `include()` to bring the text into each webpage. Any change you make in the text document will appear in all your webpages.

1. `include()` with text
2. `include()` with applets
3. Using `<object>`

include() with text

Write the text object as a PHP file, e.g.,

```
<?php
print "This is my text block.";
?>
```

Save it as a PHP files, e.g., `textblock.php3`.

In each webpage, you put in this PHP function:

```
include ('textblock.php');
```

include() with applets

`include()` can do more than text. Any PHP file can be included. You should be able to call other types of files, but I haven't tried this.

Using `<object>`

HTML's code `<object>` is another way to include external files. It was created for applets, such as Java and Microsoft's ActiveX, but supposedly works with other types of files. I've never gotten it to work.