webmonkey/programming/

PHP/MySQL Tutorial

by Graeme Merrall

Welcome to the third and final lesson for this tutorial. If you've gone through Lesson 1 and Lesson 2, you already know the essentials for installing and writing useful scripts with MySQL and PHP. We're going to look at some useful PHP functions that should make your life a lot easier. First, let's look at include files.

We all know the basics of includes, right? Contents of an external file are referenced and imported into the main file. It's pretty easy: You call a file and it's included. When we do this in PHP there are two functions we need to talk about: `include()` and `require()`. The difference between these two functions is subtle but important, so let's take a closer look. The `require()` function works in a XSSI-like way; files are included as part of the original document as soon as that file is parsed, regardless of its location in the script. So if you decide to place a `require()` function inside a conditional loop, the external file will be included even if that part of the conditional loop is false.

The `include()` function imports the referenced file each time it is encountered. If it's not encountered, PHP won't bother with it. This means that you can use `include` in loops and conditional statements, and they'll work exactly as planned.

Finally, if you use `require()` and the file you're including does not exist, your script will halt and produce an error. If you use `include()`, your script will generate a warning, but carry on. You can test this yourself by trying the following script. Run the script, then replace `include()` with `require()` and compare the results.

```
<html>

<body>


<?php

include("emptyfile.inc");

echo "Hello World";

?>


</body>

</html>
```

I like to use the suffix .inc with my include files so I can separate them from normal PHP scripts. If you do this, make sure that you set your Web server configuration file to parse .inc files as PHP files. Otherwise, hackers might be able to guess the name of your include files and display them through the browser as text files. This could be bad if you've got sensitive information - such as database passwords - contained in the includes.

So what are you going to do with include files? Simple! Place information common to all pages inside them. Things like HTML headers, footers, database connection code, and user-defined functions are all good candidates. Paste this text into a file called header.inc.

```
<?php

$db = mysql_connect("localhost", "root");

mysql_select_db("mydb",$db);

?>

<html>

<head>

<title>

<?php echo $title ?>

</title>

</head>

<body>

<center><h2><?php echo $title ?></h2></center>
```

Then create another file called footer.txt that contains some appropriate closing text and tags.

Now let's create a third file containing the actual PHP script. Try the following code, making sure that your MySQL server is running.

```
<?php

$title = "Hello World";

include("header.inc");

$result = mysql_query("SELECT * FROM employees",$db);

echo "<table border=1>\n";

echo "<tr><td>Name</td><td>Position</tr>\n";

while ($myrow = mysql_fetch_row($result)) {

  printf("<tr><td>%s %s</td><td>%s</tr>\n", $myrow[1], $myrow[2], $myrow[3]);

}

echo "</table>\n";

include("footer.inc");

?>
```

See what happens? The include files are tossed into the main file and then the whole thing is executed by PHP. Notice how the variable `$title` was defined before header.inc is referenced.

Its value is made available to the code in header.inc; hence, the title of the page is changed. You can now use header.inc across all your PHP pages, and all you'll have to do is change the value of `$title` from page to page.

Using a combination of includes, HTML, conditional statements, and loops, you can create complex variations from page to page with an absolute minimum of code. Includes become especially useful when used with functions, as we'll see down the road.

On to the exciting world of data validation.

Simple Validation

Imagine for a moment that we've got our database nicely laid out and we're now requesting information from users that will be inserted into the database. Further, let's imagine that you have a field in your database waiting for some numeric input, such as a price. Finally, imagine your application falling over in a screaming heap because some smart aleck put text in that field. MySQL doesn't want to see text in that portion of your SQL statement - and it complains bitterly.

What to do? Time to validate.

Validation simply means that we'll examine a piece of data, usually from an HTML form, and check to make sure that it fits a certain model. This can range from ensuring that a element is not blank to validating that an element meets certain criteria (for example, that a numeric value is stipulated or that an email address contains an @ for an email address).

Validation can be done on the server side or on the client side. PHP is used for server-side validation, while JavaScript or another client-based scripting language can provide client-side validation. This article is about PHP, so we're going to concentrate on the server end of things. But if you're looking for some ready-made, client-side validation scripts, check out the Webmonkey code library.

Let's ignore our database for the moment and concentrate on PHP validation. If you wish, you can add additional fields to our employee database quite simply by using the MySQL ALTER statement - that is, if you want to commit to the values that we'll validate.

There are several useful PHP functions we can use to validate our data, and they range from simple to highly complex. A simple function we could use might be `strlen()`, which tells us the length of the variable.

A more complex function would be `ereg()`, which uses full regular expression handling for complex queries. I won't delve into the complexities of regex here, as entire books have been written on the subject, but I will provide some examples on the next page.

Let's start with a simple example. We'll check to see whether a variable does or does not exist.

```
<html>

<body>

<?php

if ($submit) {
     if (!$first || !$last) {

           $error = "Sorry! You didn't fill in all the fields!";

           } else {

                      // process form

                      echo "Thank You!";

           }


}


if (!$submit || $error) {

     echo $error;

     ?>

     <P>

     <form method="post" action="<?php echo $PHP_SELF ?>">

     FIELD 1: <input type="text" name="first" value="<?php echo $first ?>">

     <br>

     FIELD 2: <input type="text" name="last" value="<?php echo $last ?>">

     <br>

     <input type="Submit" name="submit" value="Enter Information">

     </form>

     <?php

} // end if

?>


</body>

</html>
```

The keys to this script are the nested conditional statements. The first checks to see whether the Submit button has been pressed. If it has, it goes on to check that both the variables `$first` and `$last` exist. The || symbol means "or" and the ! symbol means "not." We could also rewrite the statement to say, "If $first does not exist or $last does not exist, then set $error to the following."

Next, let's extend things a little by checking to see whether a string is a certain length. This would be ideal for passwords, since you don't want some lazy user entering a password of only one or two letters. You'd rather it be, say, six or more characters.

The function for this is, as you already know, `strlen()`. It simply returns a number equal to the number of characters in the variable being tested. Here, I modified the script above to check the length of `$first` and `$last`.

```
<html>

<body>

<?php

if ($submit) {

     if (strlen($first) < 6 || strlen($last) < 6) {
          $error = "Sorry! You didn't fill in all the fields!";

          } else {

                    // process form

                    echo "Thank You!";

          }


}


if (!$submit || $error) {

     echo $error;

     ?>

     <P>

     <form method="post" action="<?php echo $PHP_SELF ?>">

     FIELD 1: <input type="text" name="first" value="<?php echo $first ?>">

     <br>

     FIELD 2: <input type="text" name="last" value="<?php echo $last ?>">

     <br>

     <input type="Submit" name="submit" value="Enter Information">

     </form>

     <?php

} // end if

?>


</body>

</html>
```

Run this script and try entering six or fewer letters to see what happens. It's simple yet quite effective.

Not-So-Simple Validation

Let's talk a bit about using regular expressions with the `ereg()` and `eregi()` functions. As I said earlier, these can be either quite complex or very simple, depending on what you need.

Using regular expressions, you can examine a string and intelligently search for patterns and variations to see whether they match the criteria you set. The most common of these involves checking whether an email address is valid (although, of course, there's no fail-safe way of doing this).

Rather than delve into the mysteries of regular expressions, I'll provide some examples. You can use the same form we created on the previous page - just paste in the lines below to see how they work.

First, let's make sure that text only has been entered into a form element. This regular expression tests true if the user has entered one or more lowercase characters, from a to z. No numbers are allowed:

```
        if (!ereg("[a-Z]", $first) || !ereg("[a-Z]", $last)) {
```

Now, let's extend this expression to check whether the string is four to six characters in length. Using `[[:alpha:]]` is an easy way to check for valid alphabetic characters. The numbers in the braces check for the number of occurrences. And note that the ^ and $ indicate the beginning and end of the string.

```
        if (!ereg("^[[:alpha:]]{4,6}$", $first) || !ereg("^[[:alpha:]]{4,6}$", $last)) {
```

Finally, let's build a regular expression that will check an email address' validity. There's been plenty of discussion about the effectiveness of checking for email addresses in this way. Nothing's completely foolproof, but what I have below works pretty well.

I took this gem from the PHP mailing list. It's a great resource - use it. And yes, this is as scary as it looks.

```
            if (!ereg('^[-!#$%&\'*+\\./0-9=?A-Z^_`a-z{|}~]+'.

'@'.

'[-!#$%&\'*+\\/0-9=?A-Z^_`a-z{|}~]+\.'.

'[-!#$%&\'*+\\./0-9=?A-Z^_`a-z{|}~]+$', $last)) {
```

Don't spend too much time looking at this. Just move on to the next page.

Functions

Enjoy that last regex expression? Fun, wasn't it? Wouldn't it be even more fun to enter that chunk on a dozen different pages that need to process email addresses?! Think about the joy of finding a typo in that mess - and doing it a dozen times no less. But of course, there's a better way.

Remember when we talked about include files earlier in this lesson? They'll allow us to create a piece of code like the email checker and include it multiple times across several pages. This way, when we want to change the code, we need edit only one file, not many.

But if we want to get this done, we'll have to use functions.

We've already used functions plenty of times. Every time we query the database or check the length of a string we're using functions. These functions are built into PHP. If you're a keen coder, you can extend PHP with your own customized functions. But that's a bit advanced for this tutorial. Instead we'll create functions that will reside within our PHP script.

A function is simply a block of code that we pass one or more values to. The function then processes the information and returns a value. The function can be as simple or complex as we like, but as long as we can pass a value in and get one out, we don't really care how complex it is. That's the beauty of functions.

Functions in PHP behave similarly to functions in C. When we define the functions, we must specify what values the function can expect to receive. It's tricky to get a handle on at first, but it prevents weird things from happening down the road. This is done because the variables inside a function are known as private variables. That is, they exist only inside the function. You may, for instance, have a variable in your script called `$myname`. If you created a function and expected to use the same `$myname` variable (with the same value), it wouldn't work. Alternatively, you could have the variable `$myname` in your script and also create another variable called `$myname` in your function, and the two would co-exist quite happily with separate values. I do not recommend doing this, however! When you come back and edit it six months later, you'll be breaking things left and right. There are exceptions to this rule as with all things, but that's outside the scope of this article.

So let's create a function. We'll start simply. We need to give the function a name and tell it what variables to expect. We also need to define the function before we call it.

```
<html>

<body>

<?php


function  addnum($first, $second) {

          $newnum = $first + $second;

          return $newnum;

}

echo addnum(4,5);

?>


</body>

</html>
```

That's it! First, we created our function. Notice how we defined two new variables, called `$first` and `$second`. When we call the function, each variable is assigned a value based on the order in which it appears in the list - 4 goes to `$first`, 5 to `$second`. Then we simply added the two numbers together and returned the result. "Return" here simply means to send the result back. At the end of the script we print the number 9.

Let's create something that's more useful to our database application. How about something that gracefully handles errors? Try this:

```
<html>

<body>

<?php


function  do_error($error) {

          echo  "Hmm, looks like there was a problem here...<br>";

          echo "The reported error was $error.\n<br>";

          echo "Best you get hold of the site admin and let her know.";

          die;

}


if (!$db = @mysql_connect("localhost","user", "password")) {

          $db_error = "Could not connect to MySQL Server";

          do_error($db_error);
}

?>


</body>

</html>
```

Before running this, try shutting down MySQL or using a bogus username or password. You'll get a nice, useful error message. Observant readers will notice the @ symbol in front of `mysql_connect()`. This suppresses error messages so that you get the information only from the function. You'll also see we were able to pass a variable into the function, which was defined elsewhere.

Remember that I said functions use their own private variables? That was a little white lie. In fact, you can make variables outside of a function accessible to the function. You might create a function to query a database and display a set of results over several pages. You don't want to have to pass the database connection identifier into the function every time. So in this situation, you can make connection code available as a global variable. For example:

```
<html>

<body>

<?php


function  db_query($sql) {

          global $db;

          $result = mysql_query($sql,$db);

          return $result;

}


$sql = "SELECT * FROM mytable";

$result = db_query($sql);

?>


</body>

</html>
```

This is a basic function, but the point is that you don't need to send `$db` through when you call the function - you can make it available using the word global. You can define other variables as global in this statement, just separate the variable names by a comma.

Finally, you can look like a real pro by using optional function variables. Here, the key is to define the variable to some default in the function, then when you call the function without specifying a value for the variable, the default will be adopted. But if you do specify a value, it will take precedence.

Confused? For example, when you connect to a database, you nearly always connect to the same server and you'll likely use the same username and password. But sometimes you'll need to connect to a different database. Let's take a look.

```
<html>

<body>

<?php


function  db_connect($host = "localhost", $user="username", $pass="graeme") {

           $db = mysql_connect($host, $username, $password);

           return $db;

}


$old_db = db_connect();


$new_host = "site.com";

$new_db = db_connect($new_host);

?>


</body>

</html>
```

Isn't that cool? The variables used inside the function were defined when the function was defined. The first time the function is called, the defaults are used. The second time, we connect to a new host, but with the same username and password. Great stuff!

Think about where you could use other functions in your code. You could use them for data checking, performing routine tasks, and so on. I use them a lot when processing text for display on a Web page. I can check, parse, and modify the text to add new lines and escape HTML characters in one fell swoop.

Now all that's left to do is to impart some words of wisdom.

Closing Advice

When it comes to databasing, there's a lot to learn. If you haven't done it already, find a good book about database design and learn to put together a solid database - on any platform. It's an invaluable skill and it will save you plenty of time and headache in the long run. Also, learn about MySQL. It's a complex but interesting database with a wealth of useful documentation. Learn about table structure, data types, and SQL. You can actually achieve some pretty impressive stuff if you know enough SQL.

Finally, there's PHP. The PHP Web site has nearly everything you need, from a comprehensive manual to mailing-list archives to code repositories. An excellent way to learn about PHP is to study the examples used in the manual and to check out the code archives. Many of the posted scripts consist of functions or classes that you can use for free in your own scripts without having to reinvent the wheel. Additionally, the mailing list is an excellent spot to check out if you get stuck. The developers themselves read the list and there are plenty of knowledgeable people there who can help you along the way.

Good luck and good coding!

Graeme Merrall works in New Zealand for KC Multimedia, an Internet/intranet design company. While sometimes being forced to use ASP, he enjoys spending his time coding in PHP, keeping the grass trimmed, and scaring his work mates with song lyrics repeated out of context.