

Learning PHP, Part 1: Register for an account, upload files for approval, and view and download approved files

Skill Level: Intermediate

[Nicholas Chase](#)
Web Developer
Author, Consultant

[Tyler Anderson \(tyleranderson5@yahoo.com\)](mailto:tyleranderson5@yahoo.com)
Engineer
Stexar Corp.

14 Jun 2005

This tutorial is Part 1 of a three-part "[Learning PHP](#)" series teaching you how to use PHP through building a simple workflow application. This tutorial walks you through creating a basic PHP page using HTML forms and covers accessing databases.

Section 1. Before you start

About this tutorial

This tutorial walks you through building a simple workflow application with PHP. Users will register for an account, upload files for approval, and view and download files that have already been approved. Users designated as administrators can view uploaded files and approve them to make the files available to all users. [Part 2](#) and [Part 3](#) of this series explore HTTP password protection and other relevant issues.

This tutorial covers the following:

- Creating a basic page

- Variables, loops, and if-then statements
- Functions
- Connecting to a database
- Using include files
- Tools

Who should take this tutorial?

If you're a programmer who wants to learn how to use PHP to build Web-based applications, start here with Part 1 of a three-part series of tutorials. PHP is a script-based language that is easy to learn, but still enables you to build complex applications with robust functionality. This tutorial walks you through creating a basic PHP page using HTML forms and also covers accessing databases.

Prerequisites

This tutorial assumes you have no PHP experience. In fact, while it's useful for you to be familiar with the concepts of HTML, no other programming is necessary for this tutorial.

System requirements

You need to have a Web server, PHP, and a database installed and available. If you have a hosting account, you can use it as long as the server has PHP V5 installed and has access to a MySQL database. Otherwise, download and install the following packages:

Web server

Whether you're on Windows or Linux (or Mac OS X, for that matter), you have the option of using the Apache Web server. Feel free to choose either V1.3 or 2.0, but the instructions in this tutorial concentrate on V2.0. If you're on Windows, you also have the option of using Internet Information Services, which is part of Windows.

PHP V5

You will need a distribution of PHP. Both PHP V4 and V5 are in use at the time of this writing, but because of changes in V5, we'll concentrate on that version. (The version isn't terribly important in this tutorial, but it makes a difference for later parts of this series.)

MySQL

Part of this project involves saving data to a database, so you'll need one of those, as well. In this tutorial, we'll concentrate on MySQL because it's so commonly used with PHP.

Section 2. Basic PHP syntax

A basic PHP page

Let's take a look at the basics of creating a page with PHP. In the next section, you'll look at using an HTML form to submit information to PHP, but first you need to know how to do some of the basic tasks.

Start by creating the most basic PHP page:

```
<html>
<title>Workflow
Registration</title>
<body>
<p>You
entered:</p>
<p><?php
echo
"Some
Data";
?></p>
</body>
</html>
```

Overall, you have a simple HTML page with a single PHP section in bold. When the server encounters the `<?php` symbol, it knows to evaluate the commands that follow, rather than simply send them out to the browser. It keeps following instructions -- which will be discussed in a moment -- until the end of the section, as indicated by the `?>` symbol.

In this case, you have just one command, `echo`, which tells the server to output the indicated text. That means that if you save the page and call it with your browser, the browser receives:

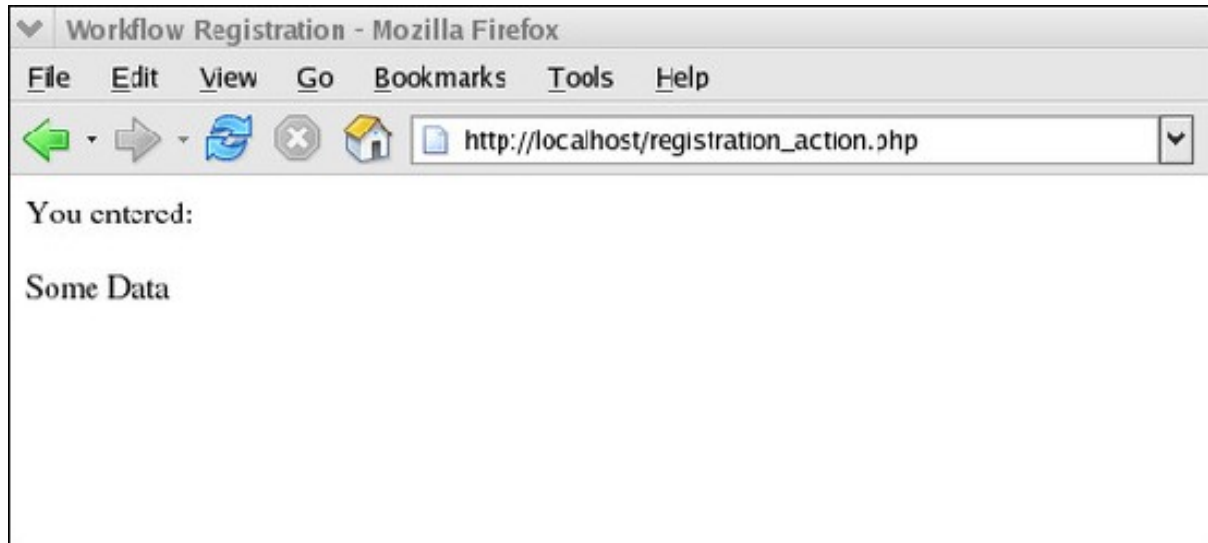
```
<html>
<title>Workflow Registration</title>
<body>
  <p>You entered:</p>
  <p>Some Data</p>
</body>
```

```
</html>
```

To see this in action, save the file as `registration_action.php` and move it to the document root of your server. For Apache, this will likely be `/var/www/html`. For Internet Information Services, it will be `C:\inetpub\wwwroot`.

Open your browser and point it to `http://localhost/registration_action.php`. You should see something similar to Figure 1.

Figure 1. Output from echo command



You've now written your first PHP page.

Variables

A variable is a placeholder for data. You can assign a value to it, and from then on, any time PHP encounters your variable, it will use the value instead. For example, change your page to read:

```
<html>
<title>Workflow Registration</title>
<body>
  <p>You entered:</p>

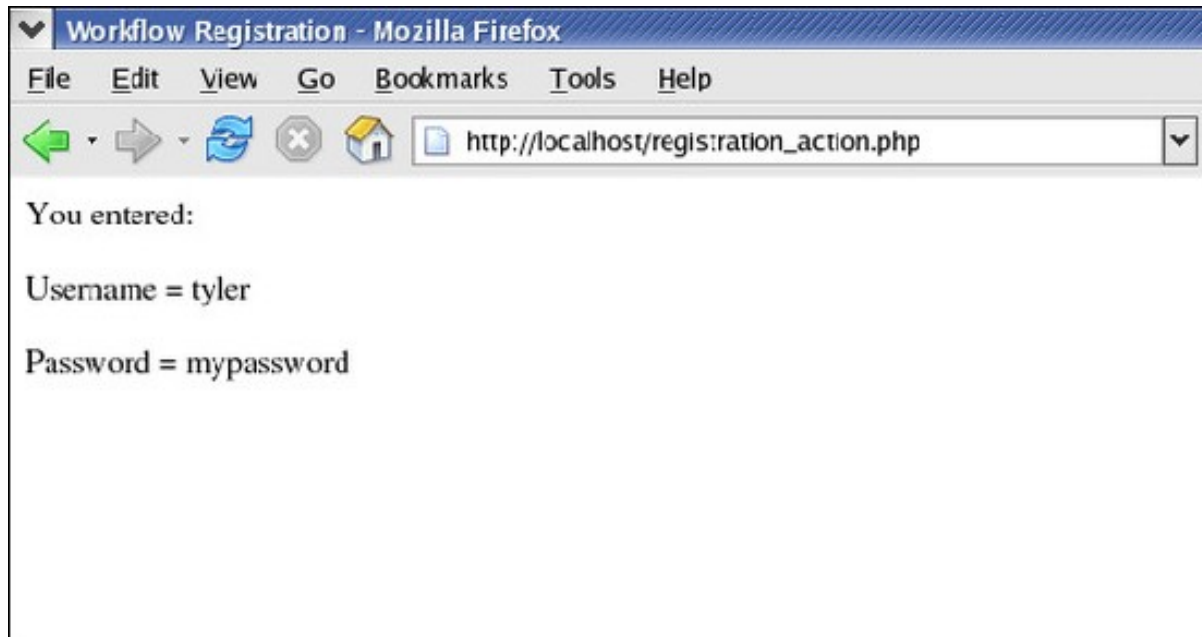
  <?php
    $username = "tyler";
    $password = "mypassword";

    echo "<p>Username = " . $username . "</p>";
    echo "<p>Password = " . $password . "</p>";
  ?>

</body>
</html>
```

Save the file (and upload it if necessary) and refresh your browser. You should see something similar to Figure 2.

Figure 2. Browser after refresh



First, notice that each line must end with a semicolon. Also, notice that you use a period to *concatenate* text, or put it together. You can put together any number of *strings*, or chunks of text, this way.

One more note about variables: In PHP, variable names are case-sensitive, so `$UserName` is a different variable from `$username`.

A consistent naming convention, such as deciding that all variables will be lowercase, can go a long way in preventing hard-to-catch errors.

Before moving on, let's look at a special kind of variable.

Constants

You can change the value of a variable as many times as you want, but sometimes you want to set up a variable with the expectation that the value will not change. These items are not called variables -- they're *constants*. For example, you might want to define a constant that represents the title of each page:

```
<?php
define("PAGE_TITLE", "Workflow Registration");
?>
```

```
<html>
<title><?php echo PAGE_TITLE ?></title>
<body>
  <p>You entered:</p>
  ...
```

(It may seem a little trivial now, but later you'll see how this definition can be used on multiple pages.)

Notice that you're defining the name of the constant and its value. If you try to change its value after it's been defined, you'll get an error.

Notice also that when you reference the constant, as in the `title` element, you don't use a dollar sign, just the name of the constant. You can name a constant anything you like, but it's customary to use all capital letters.

Easier output

Up to now, you've used the `echo` command to output information, but when you have just one piece of data to output, this command can be a little cumbersome.

Fortunately, PHP provides a simpler way. By using the output operator `<?= ?>` construct, you can specify information to output:

```
<?php
  define("PAGE_TITLE", "Workflow Registration");
?>
<html>
<title><?= PAGE_TITLE ?></title>
<body>
  <p>You entered:</p>
  ...
```

Notice that when you use the output operator, you don't follow the information with a semicolon.

Later, you'll learn about other basic PHP constructs, such as if-then statements, because you'll need them in building the application.

Section 3. PHP and forms

Creating and using forms in PHP

Developers created PHP as a Web programming language. In fact, while you can run PHP from the command line, it's rare for anyone to use the language outside of the Web application arena. The upshot is that one of your most common tasks as a PHP programmer will be to use Web forms.

You create Web forms using HTML, and when a user submits the form, the browser sends an *array* of information to the server.

In this section, you'll look at arrays, and you'll look at the ways in which you can work with form data. You'll also look at ways of controlling the flow of a PHP script, such as loops and if-then statements.

Creating a form in HTML

Start by creating the registration page for your application. Ultimately, users will enter their information, and you'll *validate* it, or check it for completeness, before saving it in a database. For now, just create the basic form. Create a new file called `registration.php` and add the following:

```
<html>
<head><title>Workflow System</title></head>
<body>
<h1>Register for an Account:</h1>
<form action="registration_action.php" method="GET">

Username: <input type="text" name="name" /><br />
Email: <input type="text" name="email" /><br />
Password: <input type="password" name="pword" /><br />
<input type="submit" value="GO" />
</form>

</body>
</html>
```

Here you have a simple form (contained within the `form` element) with two text inputs: a password input, and a submit button. If you save the file in the document root (with `registration_action.php`) and type in each of the fields, you should see something like Figure 3.

Figure 3. Register for an Account form



Workflow System - Mozilla Firefox

File Edit View Go Bookmarks Tools Help

← → ↻ × 🏠 http://localhost/registration.php

Register for an Account:

Username:

Email:

Password:

Notice that the password box does not display the actual content you're typing. But what happens when you click the **GO** button?

Submitting a form

When you created the form, you created the actual `form` element as:

```
<form action="registration_action.php" method="GET">
```

The element has two pieces of information. The first, `action`, tells the browser *where* to send the information. In this case, it's going to the page you created earlier, `registration_action.php`. The second, `method`, tells the browser *how* to send the data.

Let's see how it works. Fill in some data and click the **GO** button. You should see something similar to Figure 4.

Figure 4. Outputting the data



In this case, you didn't submit the information it's saying you did, but that's because you haven't yet adjusted that page to look at the data being submitted. But take a look at the URL.

```
http://localhost/registration_action.php?name=roadnick&email=
ibmquestions%40nicholaschase.com&pwd=supersecretpassword
```

Notice that for each form element that has a name, you have a name-value pair in the URL, separated by ampersands. The URL looks like this because you used the GET method. You'll also look at [Using POST](#), but first take a look at actually retrieving this data from within a PHP page.

Accessing form data

Now that you've submitted the form, you've got to get the data into the actual response page, `registration_action.php`. Make the following changes to that file:

```
...
<body>
  <p>You entered:</p>

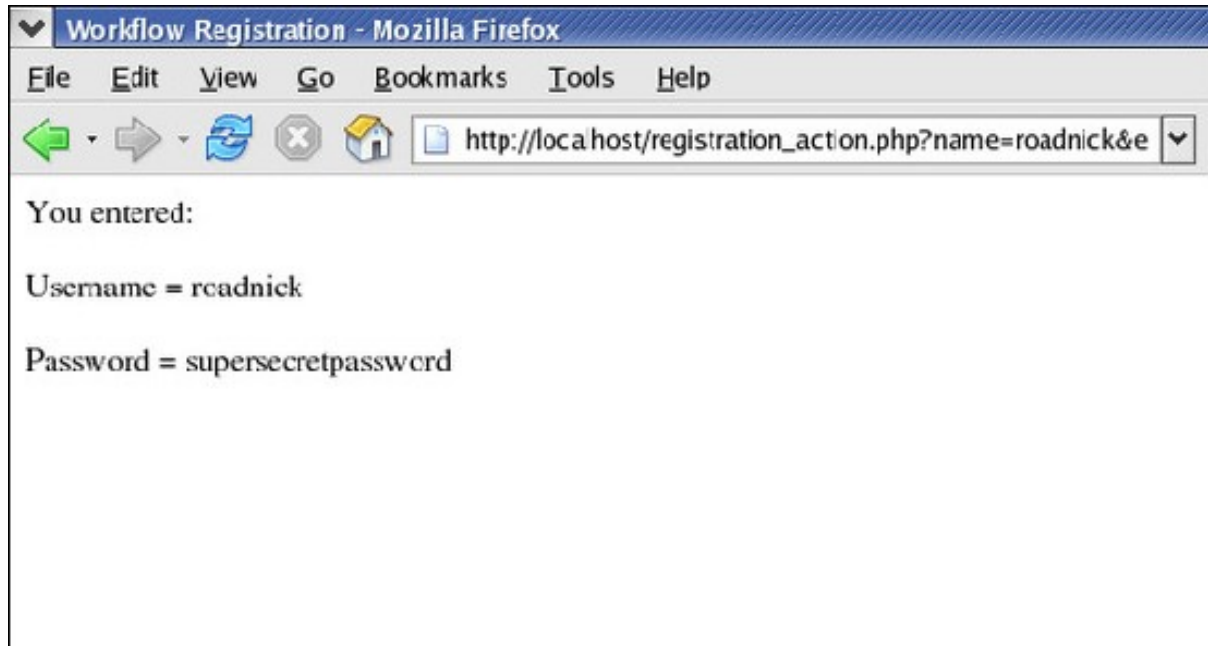
  <?php
    $username = $_GET['name'];
    $password = $_GET['pwd'];

    echo "<p>Username = " . $username . "</p>";
    echo "<p>Password = " . $password . "</p>";
  ?>

</body>
</html>
```

What you're doing is pulling the named value out of the `$_GET` array. There will be more about arrays in a moment, but for now notice that if you refresh the browser, your actual answers appear, as shown in Figure 5.

Figure 5. Correct information in the browser



You can pull any piece of submitted information by its name, but because this is an array, you also have other options.

Arrays

PHP enables you to create arrays, or lists of values, which allow you to move a group of values at one time fairly easily. For example, you can create an array of values and output them to the page:

```
$formnames = array("name", "email", "pword");  
echo "0=".$formnames[0]."<br />";  
echo "1=".$formnames[1]."<br />";  
echo "2=".$formnames[2]."<br />";
```

The `array()` function returns a value that, in this case, happens to be an array. (Functions will be dealt with later, but for now, understand that you call it and it returns a value you assign to a variable.)

This script produces the output:

```
0=name<br />
```

```
1=email<br />
2=password<br />
```

Notice that the first value has an *index* of 0, rather than 1. Notice also that you specified which value you wanted by adding the index in brackets after the name of the array variable. This action is similar to the way in which you access the form values, and that's no accident. The `$_GET` variable is a special kind of an array called an *associative* array, which means that instead of a numeric index, each value has a *key*.

When you submit the form, you're essentially creating an array like so:

```
$_GET = array("name" => "roadnick",
             "email" => "ibmquestions@nicholaschase.com",
             "password" => "supersecretpassword");
```

That's what enables you to extract individual values, such as `$_GET["name"]`. It doesn't have to be done individually, however.

Getting array information

Associative arrays can be extremely handy in dealing with data, but situations frequently arise in which you don't actually know what the structure of the array looks like. For example, you might be building a generic database routine that receives an associative array from a query.

Fortunately, PHP provides two functions that make your life a little easier:

```
<body>
  <p>You entered:</p>

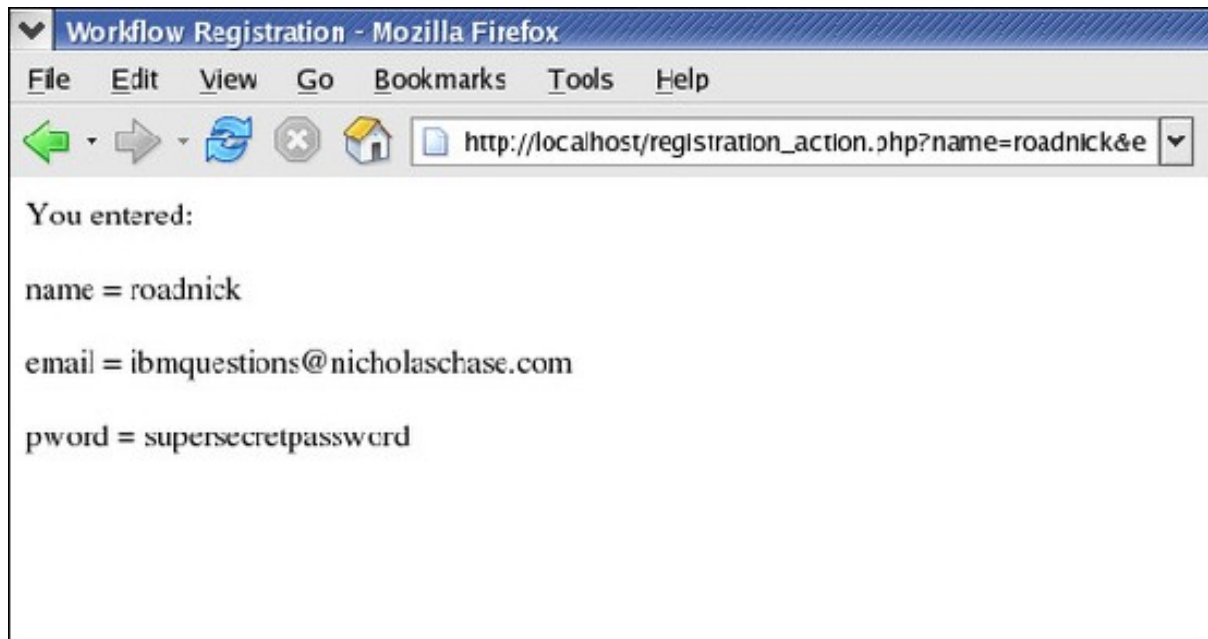
  <?php
    $form_names = array_keys($_GET);
    $form_values = array_values($_GET);

    echo "<p>" . $form_names[0] . " = " . $form_values[0] . "</p>";
    echo "<p>" . $form_names[1] . " = " . $form_values[1] . "</p>";
    echo "<p>" . $form_names[2] . " = " . $form_values[2] . "</p>";
  ?>

</body>
</html>
```

The `array_keys()` and `array_values()` functions each return regular numeric arrays of information, so you can use those arrays to pull the data out using the numeric indexes, as shown in Figure 6.

Figure 6. Arrays to pull out data using numeric indexes



Still, there's got to be a more convenient way. For example, what if you don't actually know how many values there are? PHP provides several ways of dealing with associative arrays, the most convenient being determined by what information you already have. Let's look at two other ways of accomplishing this same task next.

Using a for-next loop

One very common task in PHP is looping through a number of values. You can accomplish that easily using a for-next loop. A for-next loop runs through a number of values based on its definition. For example, the loop:

```
for ($i = 0; $i < 10; $i++) {  
    echo $i . " ";  
}
```

produces the output:

```
0 1 2 3 4 5 6 7 8 9
```

PHP initially assigns a value of 0 to `$i` because that is what's specified at the beginning of the loop. The loop continues as long as `$i` is less than 10, and each time the loop executes, PHP increments the value of `$i` by one.

What this means is that if you can find out how many values are in the `$_GET` array -- which you can do -- you can easily loop through all of the values provided by the form:

```
<body>
  <p>You entered:</p>

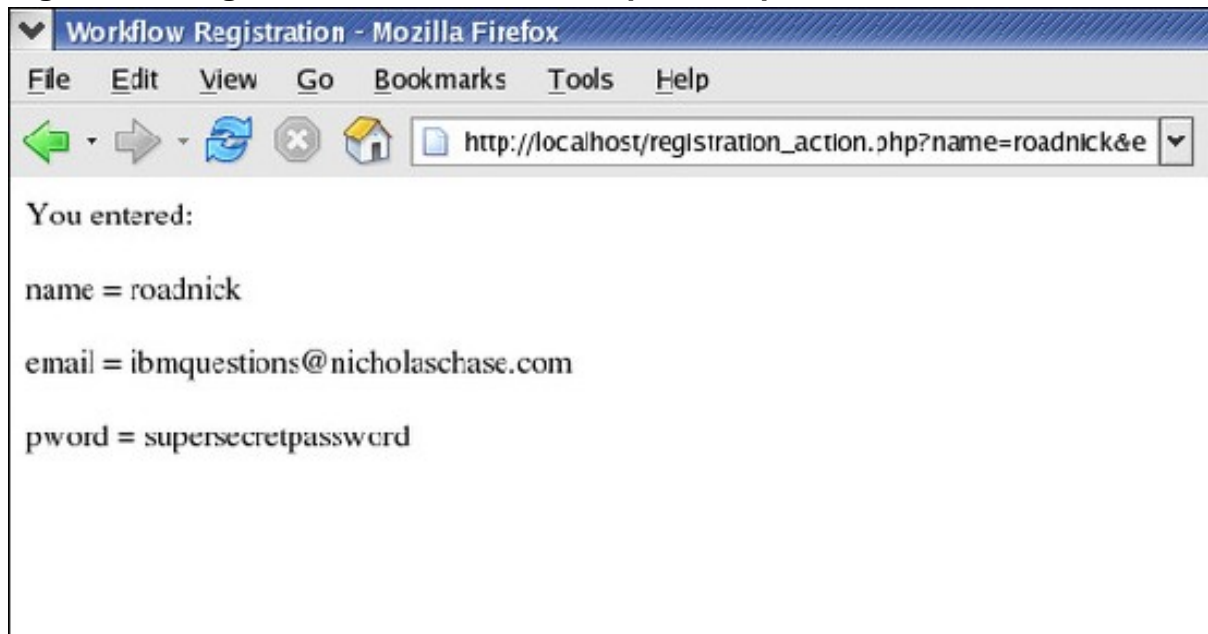
  <?php
    $form_names = array_keys($_GET);
    $form_values = array_values($_GET);

    for ($i = 0; $i < sizeof($_GET); $i++) {
      echo "<p>".$form_names[$i]." = " . $form_values[$i] . "</p>";
    }
  ?>

</body>
</html>
```

The `sizeof()` function gives you the number of values in the `$_GET` array. You can use that data to tell you when to stop the loop, as shown in Figure 7.

Figure 7. Using the `sizeof` function to stop the loop



With `$_GET` as an associative array, you actually have yet another option: the `foreach` loop.

Using a `foreach` loop

Associative arrays are so common in PHP that the language also provides an easy way to get at the data without having to go through the process of extracting the keys and values. Instead, you can use a `foreach` loop, which directly manipulates the array. Consider, for example, this code:

```
...
<?php
```

```
foreach ($_GET as $value) {  
    echo "<p>" . $value . "</p>";  
}  
?>
```

The first time PHP executes the loop, it takes the first value in the `$_GET` array and assigns that value to `$value`, which it then outputs. It then returns to the top of the loop and assigns the next value to `$value`, doing this for each value in `$_GET` (hence, the name). The end result is the output:

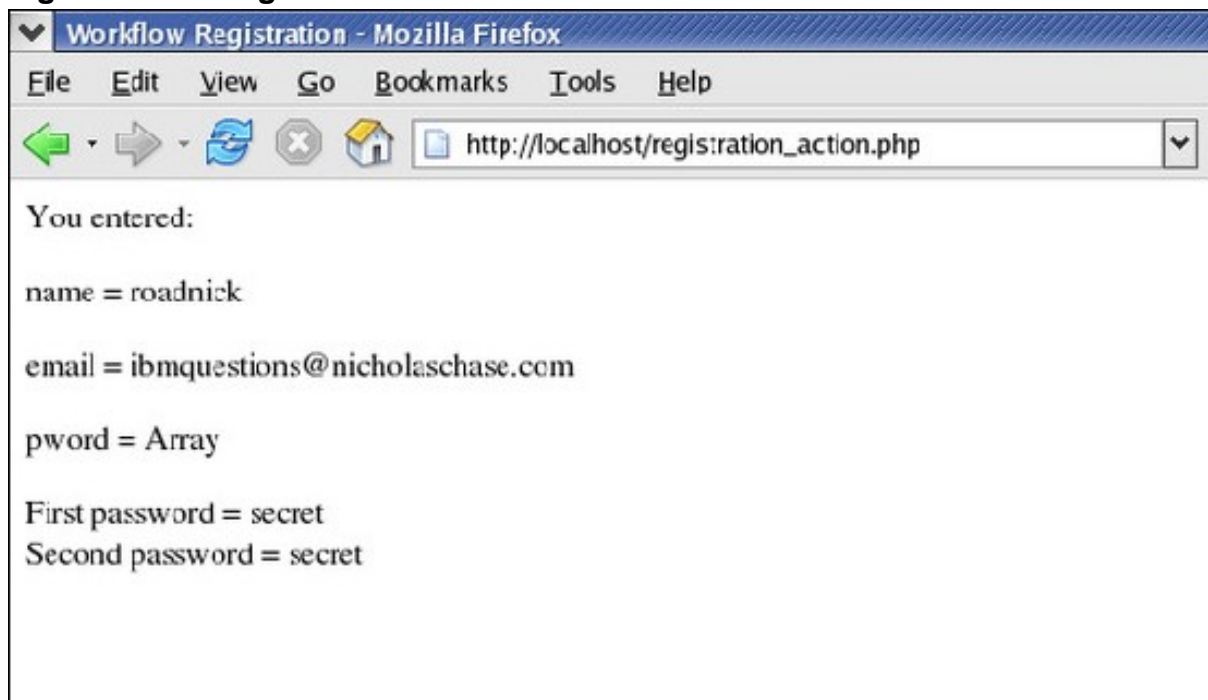
```
<p>roadnick</p>  
<p>ibmquestions@nicholaschase.com</p>  
<p>supersecretpassword</p>
```

Even more handy, however, is the ability to extract the value *and* the key:

```
...  
<?php  
    foreach ($_GET as $key=>$value) {  
        echo "<p>".$key." = " . $value . "</p>";  
    }  
?<  
...
```

This brings us back to our original result:

Figure 8. The original result



Multiple form values

While on the subject of form values, you need to deal with a situation that comes up occasionally: when you have multiple form values with a single name. For example, since the users can't see what they are typing for the password, you may want to make them type it twice to confirm that they haven't made a mistake:

```
...
Username: <input type="text" name="name" /><br />
Email: <input type="text" name="email" /><br />
Password: <input type="password" name="pword[]" /><br />
Password (again): <input type="password" name="pword[]" /><br />
<input type="submit" value="GO" />
...
```

Notice that the name of the `pword` field has changed slightly. Because you're going to retrieve multiple values, you need to treat the password itself as an array. Yes, that means you have an array value that is another array. So, if you submit the form now, it creates a URL of:

```
http://localhost/registration_action.php?name=roadnick&email=ibmquestions%40nicholaschase.com&pword[]=supersecretpassword&pword[]=supersecretpassword
```

Submitting the form is the same as creating arrays, such as:

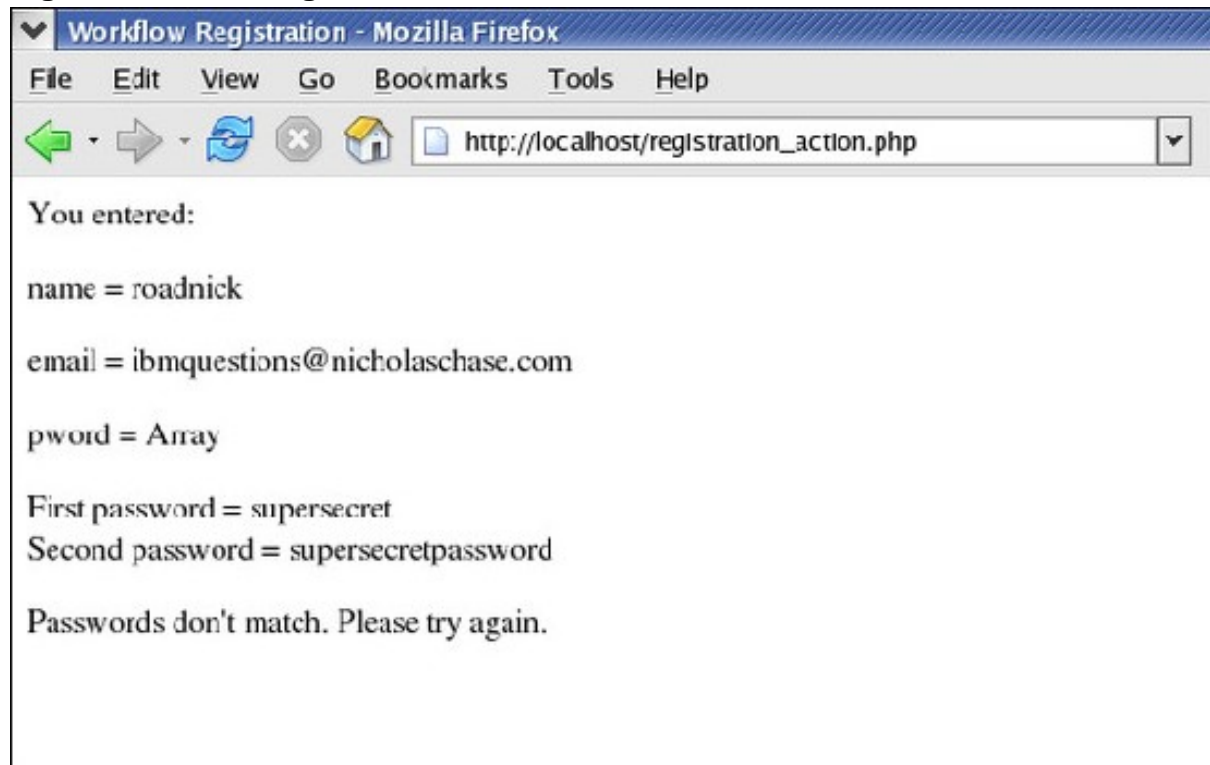
```
$passwords = array("supersecretpassword", "supersecretpassword");
$_POST = array("name"=>"roadnick",
               "email"=>"ibmquestions@nicholaschase.com",
               "pword"=>$passwords);
```

All this means that if you want to see the password values, you'll need to access them as a numeric array, as in:

```
...
foreach ($_GET as $key=>$value) {
    echo "<p>".$key." = " . $value . "</p>";
}

$passwords = $_GET["pword"];
echo "First password = ".$passwords[0];
echo "<br />";
echo "Second password = ".$passwords[1];
...
```

If you submit the form (or refresh the page), you can see the difference, as shown in Figure 9.

Figure 9. Submitting the form

Notice that the password field is now output as `Array`, but you can access its values directly.

GET vs. POST

So far, you've been using the `GET` method for submitting data, which, as you have seen, places the data right in the URL. Now, sometimes this is appropriate, and sometimes it's not. For example, you can use this technique to simulate submitting a form using a link, but if you have a large amount of data -- coming, say, from a `textarea` in which users can enter comments -- this technique isn't the best way to accomplish your goal. For one thing, Web servers typically limit the number of characters that they'll accept in a `GET` request.

For another thing, good technique and standards requirements say that you never use `GET` for an operation that has "side effects," or that actually *does* something. For example, right now you're just looking at data, so no side effects affect the operation. But, ultimately, you're going to add the data to the database, which is, by definition, a side effect.

Many Web programmers aren't aware of this particular restriction, which can lead to problems. Using `GET`, particularly as a URL, can lead to situations in which systems perform operations multiple times because a user has bookmarked the page, or

because a search engine is indexing the URL, not knowing it's actually updating a database or performing some other action.

So, in these instances, you'll have to use `POST` instead.

Using POST

Using the `POST` method instead of the `GET` method is actually pretty straightforward. First you need to change the `registration.php` page:

```
...
<h1>Register for an Account:</h1>
<form action="registration_action.php" method="POST">

Username: <input type="text" name="name" /><br />
...
```

Now when you submit the form, the URL is bare:

```
http://localhost/registration_action.php
```

To retrieve the data, you need to use the `$_POST` array rather than the `$_GET` array in `registration_action.php`:

```
...
<body>
  <p>You entered:</p>

  <?php
    foreach ($_POST as $key=>$value) {
      echo "<p>".$key." = " . $value . "</p>";
    }

    $passwords = $_POST["pword"];
    echo "First password = ".$passwords[0];
    echo "<br />";
    echo "Second password = ".$passwords[1];
  ?>

</body>
</html>
```

You can work with the `$_POST` array in exactly the same way you worked with the `$_GET` array.

Error-checking: The if-then statement

Before moving on, it doesn't make any sense to request that the user type the password twice if you don't make sure that both attempts match. To do that, you'll use an if-then statement:

```
...
$passwords = $_POST["pword"];
echo "First password = ".$passwords[0];
echo "<br />";
echo "Second password = ".$passwords[1];

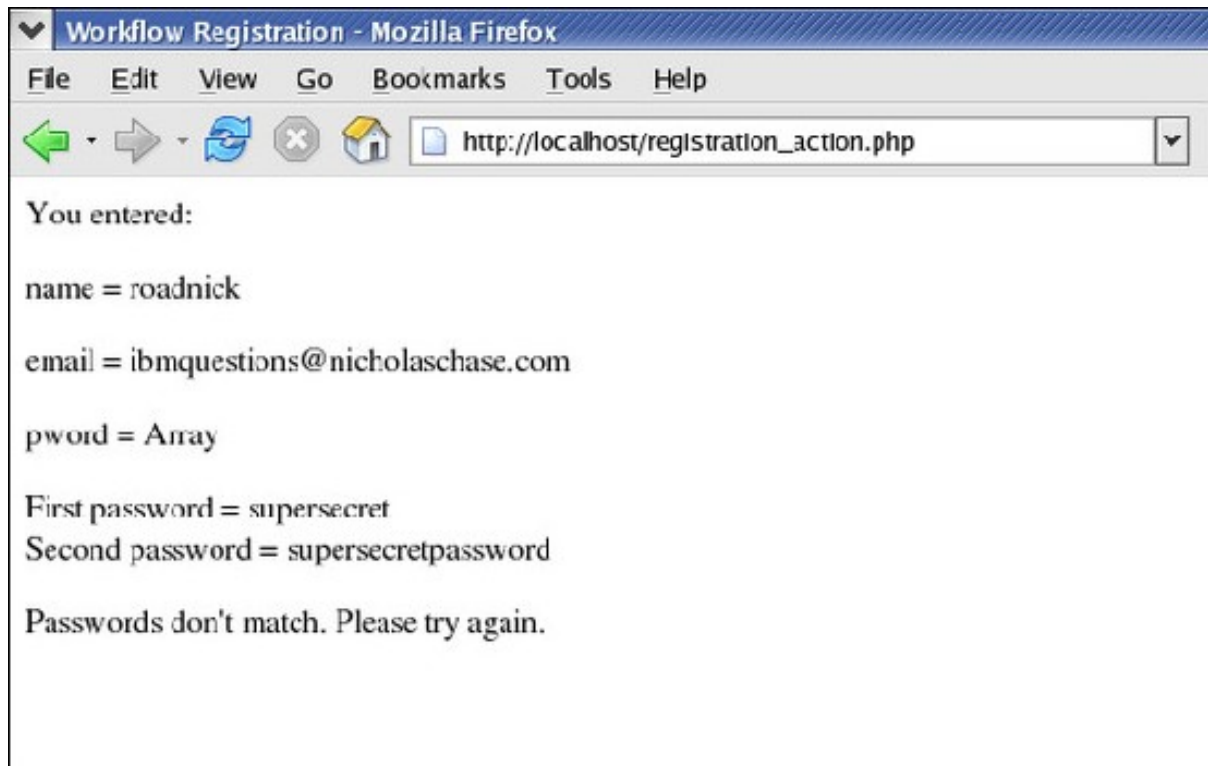
if ($passwords[0] == $passwords[1]) {
    echo "<p>Passwords match. Thank you.</p>";
} else {
    echo "<p>Passwords don't match. Please try again.</p>";
}
...
```

In an if-then statement, if the expression in the parentheses (in this example, `$passwords[0] == $passwords[1]`) is true, PHP executes the statements in the first set of brackets. If it's false, it doesn't. In this case, you've also included an alternate course of action to take if the statement is false.

Notice that rather than saying `$passwords[0] = $passwords[1]` with a single equals sign, you said `$passwords[0] == $passwords[1]` with a double equals sign. The double equals sign is the comparative operator. It actually detects whether the two are equal. The single equals sign is the assignment operator. With a single equals sign, when you executed the statement, PHP would assign the value of `$passwords[1]` to `$passwords[0]`, which is clearly not what you wanted.

In this case, the page gives the user a warning if the passwords don't match, as shown in Figure 10.

Figure 10. Warning issued if passwords don't match



Two more handy operators are the *and* operator (`&&`) and the *or* operator (`| |`). For example, you could say:

```
if (($today == "Monday") && ($status == "Not a holiday")) {  
    echo "GO TO WORK!!!";  
}
```

In this case, the expression is true only if today is Monday *and* it's not a holiday. The *or* operator returns true if any of the components are true.

Section 4. Functions

Creating a function

When you're building an application of any significant size, it's common to run across actions, calculations, or other sections of code you use over and over.

In those cases, it's helpful to take the code and use it to create a *function*. For example, you can take the password validation and put it into a separate function,

like so:

```
...
<body>
<p>You
entered:</p>

<?php

function
checkPasswords($firstPass,
$secondPass){

    if
($firstPass
==
$secondPass)
{
echo
"<p>Passwords
match.
Thank
you.</p>";
}
else {
echo
"<p>Passwords
don't
match.
\
Please
try
again.</p>";
}

}

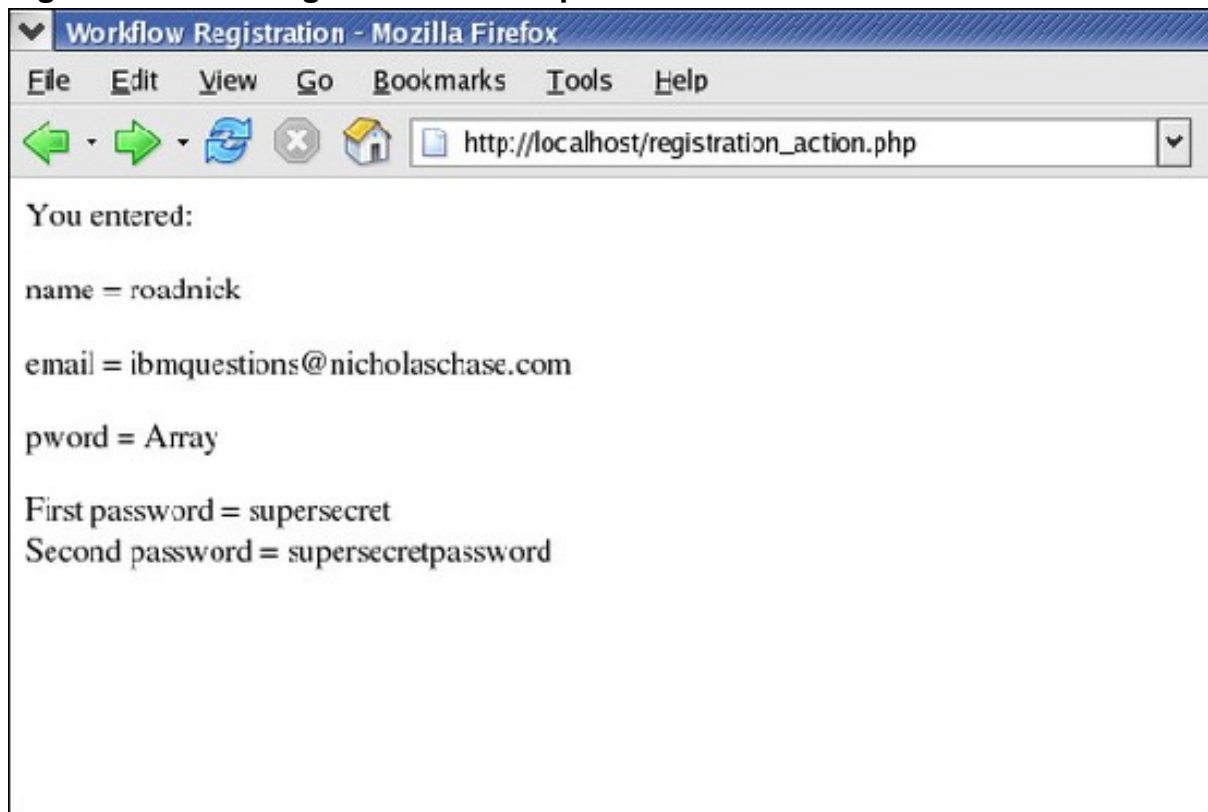
foreach
($_POST
as
$key=>$value)
{
echo
"<p>". $key. "
= " .
$value
.
"</p>";
}

$passwords
=
$_POST["pword"];
echo
"First
password
=
" . $passwords[0];
echo
"<br
/>";
echo
"Second
password
=
" . $passwords[1];
```

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

When the server processes this page, it gets to the `function` keyword and knows that it shouldn't execute that section of code until specifically requested. Therefore, the `foreach` loop is still the first thing executed on this page, as you can see in Figure 11.

Figure 11. Executing the foreach loop



So, how do you actually use the function?

Calling a function

To call a function, you use its name and follow it with a pair of parentheses. If you expect any arguments, as in this case, they go in the parentheses, like so:

```
...
<body>
  <p>You entered:</p>

<?php
function checkPasswords($firstPass, $secondPass){
```

```

    if ($firstPass == $secondPass) {
        echo "<p>Passwords match. Thank you.</p>";
    } else {
        echo "<p>Passwords don't match. Please try again.</p>";
    }
}

foreach ($_POST as $key=>$value) {
    echo "<p>".$key." = " . $value . "</p>";
}

$passwords = $_POST["pword"];
echo "First password = ".$passwords[0];
echo "<br />";
echo "Second password = ".$passwords[1];

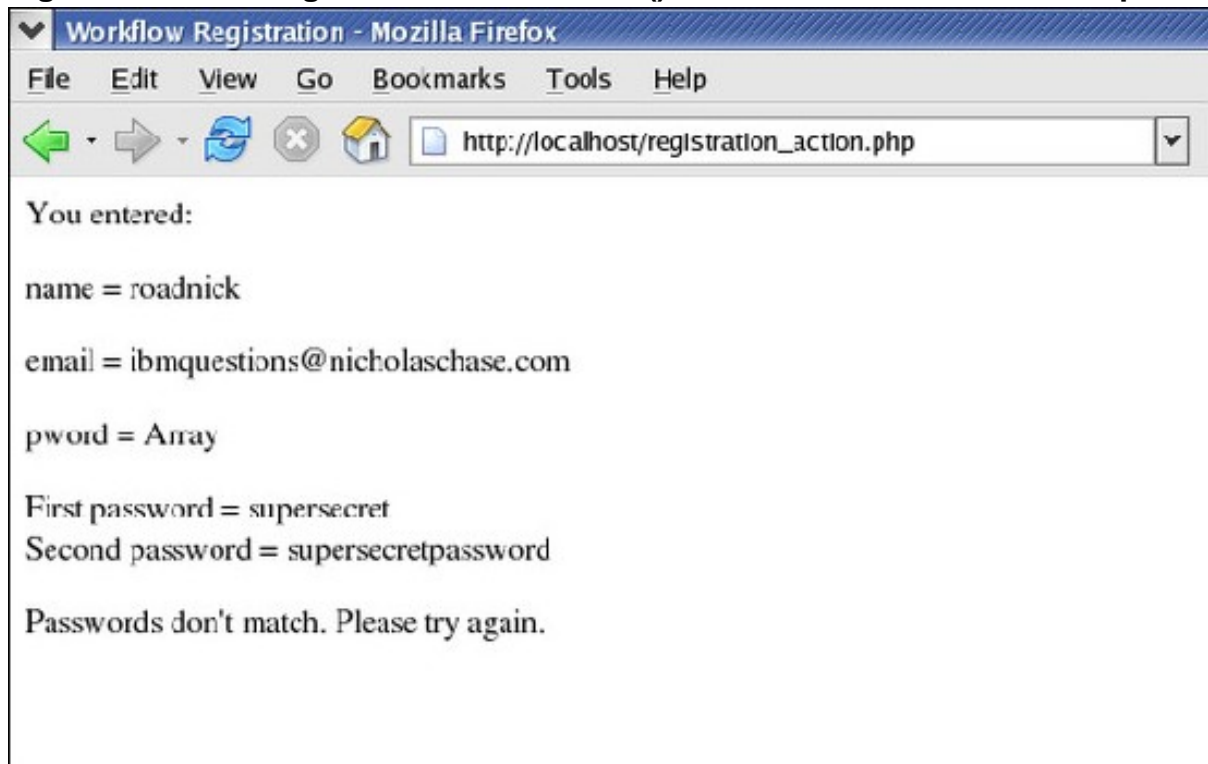
checkPasswords($passwords[0], $passwords[1]);

?>
</body>
</html>

```

When PHP executes this page, it starts with the `foreach` loop, outputs the password, then executes the `checkPasswords()` function, passing as arguments the two password attempts. (You could also have passed the array and pulled out the individual values from within the function.)

Figure 12. Executing the `checkPasswords()` function after the `foreach` loop



If you've programmed in certain other languages, you may consider this more of a

subroutine because the objective is to execute a chunk of code, rather than to return a value. You can use functions either way, as you'll see next.

Returning a value

In addition to using a function to execute a chunk of code, it's often helpful to use a function to perform some sort of action and return a value. For example, you can create a validation function that performs a number of actions, then returns a value indicating whether there's a problem:

```
...
<body>
  <p>You entered:</p>

  <?php
  function validate($allSubmitted){

    $message = "";

    $passwords = $allSubmitted["pword"];
    $firstPass = $passwords[0];
    $secondPass = $passwords[1];
    $username = $allSubmitted["name"];

    if ($firstPass != $secondPass) {
      $message = $message."Passwords don't match<br />";
    }
    if (strlen($username) < 5 || strlen($username) > 50){
      $message = $message."Username must be \
        between 5 and 50 characters<br />";
    }

    if ($message == ""){
      $message = "OK";
    }

    return $message;

  }
  ...
```

The function takes as an argument the `$_POST` array, and pulls out the information it needs to look at. You start out with an empty `$message` string, and if the passwords don't match, or if the length of the username (as returned by the `strlen()`, or *string length*, function) is wrong, you add text to the `$message` string. If the passwords match and the username length is correct, you wind up with an empty string, which you assign a value of "OK" so you can check for it in the body of the page, which is next.

Validating the data

You've created a function that returns a value based on whether the user's input is appropriate, so now you can test for that value:

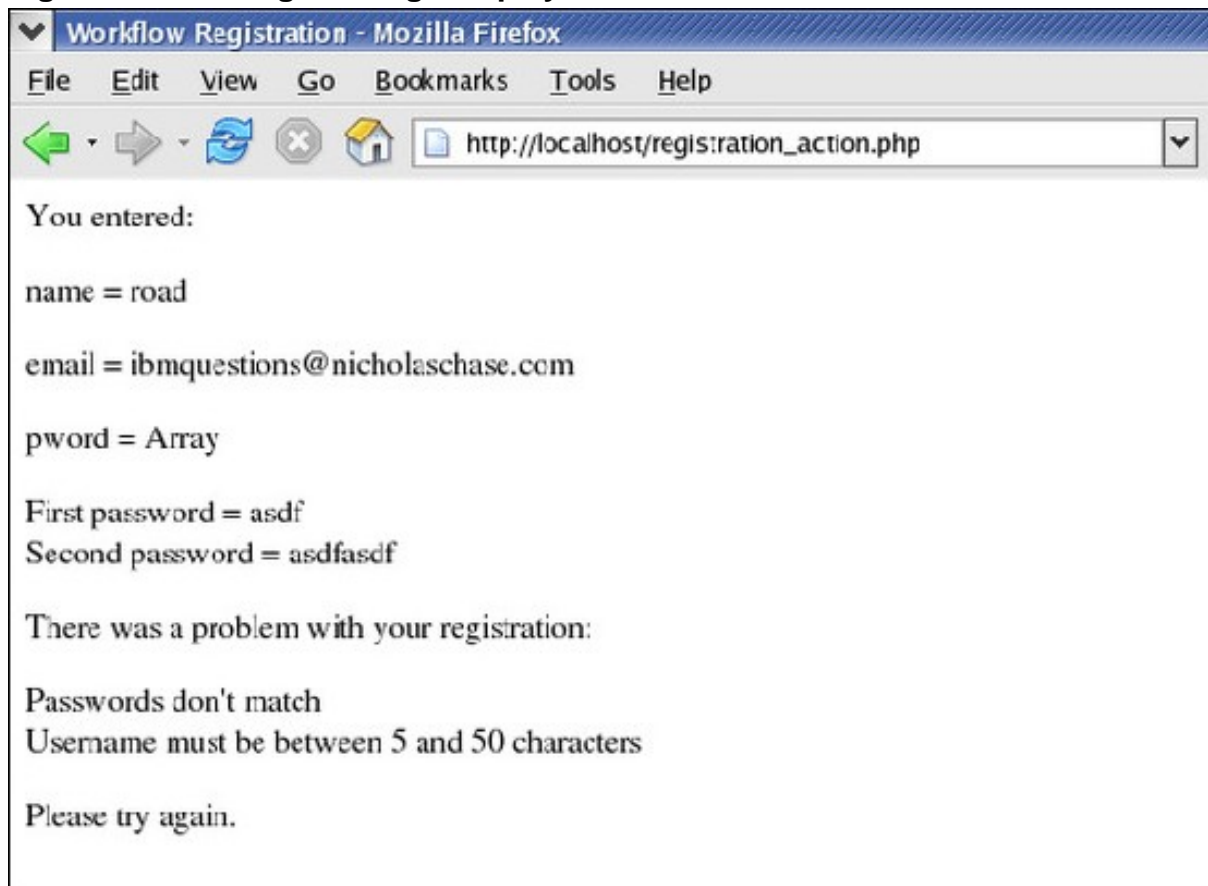
```
...
echo "<br />";
echo "Second password = ".$passwords[1];

if (validate($_POST) == "OK") {
    echo "<p>Thank you for registering!</p>";
} else {
    echo "<p>There was a problem with your registration:</p>";
    echo validate($_POST);
    echo "<p>Please try again.</p>";
}

?>
...
```

In the if-then statement, the value returned by the `validate()` function is checked. If it equals "OK", a simple thank-you message is provided; otherwise, the message itself is displayed, as shown in Figure 13.

Figure 13. Warning message displayed



Note first that this technique is much more convenient for testing for a specific result. Can you imagine the chaos if you'd tried to put all those conditions in the if-then statement? Also, note that the function is being called twice here, which isn't efficient. In a production application, you'd assign the return value to a variable, then

check against that rather than unnecessarily repeating operations.

Now that you know the data is all right, you can enter it into the database.

Section 5. Connecting to and using MySQL

Setting up

Before going any further, you need to do a little preparation in MySQL. You need to create a database, add a table, and create a new user who has access to it.

From the MySQL console, type the following:

```
create
database
workflow;

use
workflow;

create
table
users
(id int
auto_increment
primary
key,
username
varchar(50),
email
varchar(255),
password
varchar(50));

show
tables;
```

The final output should look something like this:

```
+-----+
| Tables_in_workflow |
+-----+
|      users         |
+-----+
1 row in set (0.00 sec)
```

Finally, add the new user, `wfuser`, with a password of `wfpass`:

```
GRANT ALL PRIVILEGES ON *.* TO 'wfuser'@'localhost'
```

```
IDENTIFIED BY 'wfpass' WITH GRANT OPTION;
```

Now you can move on to actually using the database.

Connecting to MySQL

It's virtually impossible to create a Web application of any significant size without having to interact with a database of some sort. In your sample application, you'll be using a MySQL database to store username and password information. In this section, you'll add functionality to the registration action page so that it checks that the submitted username is unique and inserts the data into the table if it is. You'll also look at displaying information that's already in a database. Finally, you'll create the application's login page.

You'll start by connecting to the database. PHP has a number of functions that exist solely for dealing with MySQL databases, and you'll use them in this section.

The first step is to create a function that connects to the workflow database you created in [Setting up](#):

```
...
    return $message;
}

function db_connect($user='wfuser',
                   $password='wfpass', $db='workflow'){

    mysql_connect('localhost', $user, $password)
        or die('I cannot connect to db: ' . mysql_error());

}

foreach ($_POST as $key=>$value) {
    echo "<p>".$key." = " . $value . "</p>";
}

...
if (validate($_POST) == "OK") {
    echo "<p>Thank you for registering!</p>";

    db_connect();

} else {
    echo "<p>There was a problem with your registration:</p>";
}
...
```

Here, you're creating a function, `db_connect()`, which attempts to open a connection between PHP and the MySQL database. Notice that in the definition of the function, you included values for the arguments. These are *default* values, which means that if you don't provide a username, password, and database name, PHP will use these values. (In fact, you'll do just that in a moment.)

The function attempts to connect to the database on the local machine, `localhost`. Note that in this case, "local machine" means local to the PHP server, so you're talking about the Web server and *not* the client.

If PHP can't open a connection, processing stops (or *dies*), and PHP simply displays a message explaining what's happened.

Assuming all goes well, this connection will stay open until you close it, or until the page is finished processing. Any other database commands you issue will be directed at that connection.

Finally, the function is called to make all this happen.

Selecting a database

One MySQL database server can house multiple databases, so once you've opened a connection to the server, you'll need to specify which database you want:

```
...
function db_connect($user='wfuser',
                   $password='wfpass', $db='workflow'){

    mysql_connect('localhost', $user, $password)
        or die('I cannot connect to db: ' . mysql_error());
    mysql_select_db($db);
}
...
```

At this point, you have a function you can reuse to connect to any MySQL database on the local server. In fact, one nice thing about this function is that it is, in a sense, database-independent; you could easily change not only the name but the type of database you're accessing, and this function is the only place you'd have to make alterations.

Now you're ready to insert the user data.

Inserting the record

Now it's time to add data to the users table you created previously. To add data, you're going to create a SQL statement that inserts the data into that table, then you're going to execute that statement.

The statement has the form:

```
insert into users (username, email, password) values
```

```
('roadnick', 'ibmquestions@nicholaschase.com', 'supersecretpassword')
```

Now, if you were paying particular attention when you created the table, you might be wondering what happened to the `id` column. You specified that first column as `AUTO_INCREMENT`, so if you leave it out, as you're doing here, MySQL will automatically fill it in with the next available integer. So, all you have to do here is substitute the user-submitted data for your placeholders and execute the statement:

```
...
    if (validate($_POST) == "OK") {
        echo "<p>Thank you for registering!</p>";

        db_connect();

        $sql = "insert into users (username, email, password) values
                ('".$_POST["name"]."', '".$_POST["email"]."', \
                '".$_PASSWORDS[0]."')";
        $result = mysql_query($sql);

        if ($result){
            echo "It's entered!";
        } else {
            echo "There's been a problem: ".mysql_error();
        }
    } else {
        echo "<p>There was a problem with your registration:</p>";
    }
...

```

Notice that when you call the `mysql_query()` function, it returns a value that is being stored in the `$result` variable. That value will be true if the operation went smoothly and false if any problems arose. You can then use that value as the expression in an if-then statement to take action depending on the results.

If any problems came up, MySQL will set a value for the `mysql_error()` function to return, which you can then output to the page.

Now that you've added information to the database, it's time to look at getting it back out.

Selecting records

At this point, you can add data to the database, but how do you know that the username is unique? At the moment, you don't, but you can remedy that by checking the users table before you do the actual insert:

```
...
    if (validate($_POST) == "OK") {
        echo "<p>Thank you for registering!</p>";

        db_connect();

```

```

$sql = "select * from users where username='".$_POST["name"]."";
$result = mysql_query($sql);
if (!$result) {

    $sql = "insert into users (username, email, password) values
    ('".$_POST["name"]."', '".$_POST["email"]."', '".$_passwords[0]."'");
    $result = mysql_query($sql);

    if ($result){
        echo "It's entered!";
    } else {
        echo "There's been a problem: ".mysql_error();
    }
} else {

    echo "There is already a user with that name: <br />";
    $sqlAll = "select * from users";
    $resultsAll = mysql_query($sqlAll);

}

} else {
    echo "<p>There was a problem with your registration:</p>";
...

```

Start by creating a SQL statement that selects any records that have a username matching the one you're thinking of inserting. You can then execute that statement against the database, just as you did with the insert statement. If the statement returns any results, `mysql_query()` returns a value that evaluates as true, and if not, it returns false.

Now, what you want is no results for this username, so you *want* the value of `$result` to be false. But when you use an if-then statement, you're looking for a *true* statement, not a false one. So, you're using the *negation* operator, the exclamation point, to say, in essence, "If the *opposite* of this value is true, then do this." And the "this" in this case is insert the data into the database.

But what if `$result` is true to start with? Then the opposite of `$result` will be false, and you'll execute the else statement. Ultimately, you'll list the existing usernames and e-mail addresses, so start by creating and executing that SQL statement.

You'll retrieve the results next.

Retrieving the results

Of course, in the real world, you would *never* show all of the existing usernames if someone entered one that already existed, but you're going to do it here so you can see how this kind of thing is done.

Previously, you created a SQL statement that selects all of the records in the users table and represents those results in the `$resultsAll` variable. Now you're going

to retrieve the data from that variable:

```
...
    } else {
        echo "There is already a user with that name: <br />";
        $sqlAll = "select * from users";
        $resultsAll = mysql_query($sqlAll);
        $row = mysql_fetch_array($resultsAll);

        echo $row["username"]." -- ".$row["email"]."<br />";
    }
...

```

The first step in retrieving the data is to extract a single row from `$resultsAll`, which is actually a *resource* representing the entire data set. The `mysql_fetch_array()` function, as the name implies, returns an associative array that includes the data for a single row. The keys are the same as the column names, so you can output the data for that row very easily by requesting the appropriate values from the `$row` array.

But this example is just a single row. How do you access *all* the data?

Seeing all the results: The while loop

If, during the first retrieval, there is at least one row to retrieve, `$row` will represent true in an if-then statement -- or while loop, which you're using here. PHP gets to the `while ($row)` statement and says, "OK, the value of this expression is true, so go ahead and execute the statements in this block." It outputs the data for that row, then attempts to fetch another row. It then goes back to the top of the loop.

```
...
    } else {
        echo "There is already a user with that name: <br />";
        $sqlAll = "select * from users";
        $resultsAll = mysql_query($sqlAll);
        $row = mysql_fetch_array($resultsAll);
        while ($row) {
            echo $row["username"]." -- ".$row["email"]."<br />";

            $row = mysql_fetch_array($result);
        }
    }
...

```

If that attempt to retrieve another row is successful, `$row` will once again evaluate as true, and the loop executes again. This goes on until no more rows remain, and `mysql_fetch_array()` returns false. At that point, PHP knows to skip the loop

and move on with the rest of the script.

One handy note: If you leave out the last step, in which you try to retrieve another row, `$row` will always be true, and your server will keep running the loop until it runs out of memory or times out. So, when you create a loop like this, the very first statement you should add to it is the one that increments whatever it is you're looking at.

Close the database connection

Before moving on, you need to make sure that the database connection you opened gets closed again:

```
...
    if (validate($_POST) == "OK") {
        echo "<p>Thank you for registering!</p>";

        db_connect();

        $sql = "select * from users where username='".$_POST["name"]."'";
        $result = mysql_query($sql);
        if (!$result) {
...
            }

            mysql_close();
        } else {
...
    }
```

Now it's time to clean things up a bit.

Section 6. Cleaning up: including files

Why include files?

So far, each script you've written has been self-contained, with all of the code in a single PHP file. In this section, you'll look at organizing your code into multiple files. You'll take sections of code that you use on multiple pages and place them into a separate file, which you'll then include in the original pages.

PHP provides two ways to include files. One is for including support files, such as interface elements, and the other is for crucial files, such as functions called within the page.

Including the definitions

Start by creating the files you'll eventually include. Whenever you create a Web site, one of the first things you need to do is create a header and footer file that contains the major interface elements. That way, you can build as many pages as you want without worrying about what the page looks like until the coding work is done. At that point, you can create the interface just once, in the include files, and the entire site will be instantly updated.

So, to start, create a file called `top.txt` and add the following:

```
<html>
<head>
<title>Workflow System</title>
</head>
<body>
<table>
<tr><td colspan="2"><h2>The Workflow System</h2></td></tr>

<tr>
  <td width="30%">
    <h3>Navigation</h3>

    <p><a href="register.php">Register</a></p>

  </td>
  <td>
```

In a separate file called `bottom.txt`, add the following:

```
  </td>
</tr>
</table>
</body>
</html>
```

Save both files in the same directory as `registration.php`.

Including the files

Now go ahead and add these files to the registration page. Edit `registration.php` to look like this:

```
        <?php
        include("top.txt");
        ?>

        <h1>Register for an Account:</h1>
```

```
<form action="registration_action.php" method="POST">
Username: <input type="text" name="name" /><br />
Email: <input type="text" name="email" /><br />
Password: <input type="password" name="pword[]" /><br />
Password (again): <input type="password" name="pword[]" /><br />
<input type="submit" value="GO" />

</form>

<?php
    include("bottom.txt");
?>
```

Notice that you've removed the HTML that normally surrounds the content of the page and replaced it with a command to include the files you just created. Now it's time to see what that action does.

The results

If you now point your browser back to the registration page, you're going to see a much different look, as shown in Figure 14.

Figure 14. New look of registration page



If you do a "view source" on the page, you can see that all three files have now been

merged in the output:

```
<html>
<head>
<title>Workflow System</title>
</head>
<body>
<table>
<tr><td colspan="2"><h2>The Workflow System</h2></td></tr>

<tr>
  <td width="30%">
    <h3>Navigation</h3>

    <p><a href="register.php">Register</a></p>

  </td>
  <td>

<h1>Register for an Account:</h1>
<form action="registration_action.php" method="POST">

Username: <input type="text" name="name" /><br />
Email: <input type="text" name="email" /><br />
Password: <input type="password" name="pword[]" /><br />
Password (again): <input type="password" name="pword[]" /><br />
<input type="submit" value="GO" />

</form>

  </td>
</tr>
</table>
</body>
</html>
```

If you go ahead and make the same changes to `registration_action.php` and submit the form, you'll see that the changes take place immediately.

Now, this page isn't a work of art, and that's OK. Later, you can get a designer to make it look nice, and you'll have to make the changes only once -- to the included files -- rather than to every page on the site.

Requiring files

If PHP can't find interface files, it's a problem, but it isn't necessarily a catastrophe, especially if all you're worried about is the functionality of the application. As a result, if PHP can't find a file specified by the `include()` function, it displays a warning and continues processing the page.

In some cases, however, not being able to find an include file *is* a catastrophe. For example, you can pull the `validate()` and `db_connect()` scripts out into a separate file and include them in the `registration_action.php` file. If PHP can't find that file, that's a problem because you're calling those functions within the page. So,

to avoid that, you can use the `require()` function, instead of `include()`:

```
<?php
    include("top.txt");
    require("scripts.txt");
?>

    <p>You entered:</p>

<?php
    foreach ($_POST as $key=>$value) {
        echo "<p>".$key." = " . $value . "</p>";
    }

    $passwords = $_POST["pword"];
    echo "First password = ".$passwords[0];
    echo "<br />";
    echo "Second password = ".$passwords[1];

    if (validate($_POST) == "OK") {
        echo "<p>Thank you for registering!</p>";
    }
...

```

If PHP can't find a page that's required, it sends a fatal error and stops processing.

Avoiding duplicates

There's nothing to stop you from including a file in a file that is itself included in another file. In fact, with all of these include files floating around, it can get confusing, and you may inadvertently include the same file more than once. This duplication can result in interface elements appearing multiple times, or errors due to the redefinition of functions or constants. To avoid that, PHP provides special versions of the `include()` and `require()` functions. For example, you can be sure that the `registration_action.php` file will load the files only once:

```
<?php
    include_once("top.txt");
    require_once("scripts.txt");
?>

    <p>You entered:</p>

<?php
    foreach ($_POST as $key=>$value) {
        echo "<p>".$key." = " . $value . "</p>";
    }
...

```

When PHP encounters the `include_once()` or `require_once()` function, it checks to see if the file has already been included in the page before including it

again.

Section 7. Summary

In this tutorial, you began the process of building a Web-based application using PHP. You looked at the basic syntax of a PHP script, using it to build a page that accepts input from an HTML form. In processing the form, you reviewed basic structures, such as variables, if-then statements, and loops. You also examined numeric and associative arrays, and how to access their data. You then moved on to looking at and moving data into and out of a MySQL database by creating a SQL statement and executing it, then working with the arrays that represent each row of data. Finally, you looked at using include files.

The purpose of this series of tutorials is to teach you how to use PHP through building a workflow application. Here in Part 1, you began the process by enabling users to sign up for a new account, which you then stored in a database. Subsequent parts of this series explore HTTP password protection and other important issues that will help you on your path to becoming a PHP developer.

Resources

Learn

- [PHP documentation](#) is available in many languages.
- [MySQL documentation](#) is also available.
- Read "[Connecting PHP applications to IBM DB2 Universal Database.](#)"
- Check out "[Connecting PHP Applications to Apache Derby.](#)"
- Don't miss "[Develop IBM Cloudscape and DB2 Universal Database applications with PHP.](#)"
- For information about PHP with MySQL, read "[Creating dynamic Web sites with PHP and MySQL.](#)"
- "[Using HTML forms with PHP](#)" covers some additional issues not discussed in this tutorial.
- Visit IBM developerWorks' [PHP project resources](#) to learn more about PHP.
- Stay current with [developerWorks technical events and webcasts](#).
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.
- To listen to interesting interviews and discussions for software developers, be sure to check out [developerWorks podcasts](#).

Get products and technologies

- Download [PHP](#).
- Download [MySQL](#).
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- Get involved in the developerWorks community by participating in [developerWorks blogs](#).

About the authors

Nicholas Chase

Nicholas Chase has been involved in Web-site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. He has been a high school physics teacher, a low-level-radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the chief technology officer of an interactive communications company. He is the author of several books, including *XML Primer Plus* (Sams 2002).

Tyler Anderson

Tyler Anderson has graduated with a degree in computer science in 2004 and a Master of Science degree in computer engineering in December, 2005, both from Brigham Young University. Tyler is currently a freelance writer and developer for Backstop Media.