developerWorks®

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

Nicholas Chase

October 23, 2012
(First published June 14, 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.
23 Oct 2012 - *Nicholas Chase updated content throughout this tutorial to reflect current PHP technology.*

View more content in this series

Before you start

In this first part of the series you will learn about basic PHP syntax, forms and functions and how to connect to and use MySQL or any other database with your PHP application.

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 approved files. 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, XML, JSON, 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

© Copyright IBM Corporation 2005, 2012

Trademarks

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. It also explains how to access a database.

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. Feel free to download the source code used in 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:

XAMPP
: Whether you're on Windows®, Linux®, or even Mac, the easiest way to get all of the necessary pieces of software for this tutorial is to install XAMPP, which includes a web server, PHP, and the MySQL database engine. If you choose to go this route, simply install, then run the control panel to start up the Apache and MySQL processes. You also have the option of installing the various pieces separately, but keep in mind you will then have to configure them to work together—a step which is already taken care of by XAMPP.

Web server
: If you choose not to use XAMPP, you have several options for a web server. If you're using PHP 5.4 (as of this writing, XAMPP is only using PHP 5.3.8) you can use the built-in web server for testing. For production, however, I will going to assume that you're using the Apache Web server, version 2.x.

PHP 5.x
: If you're not using XAMPP, you will need to download PHP 5.x separately. The standard distribution includes everything you're going to need for this tutorial. Feel free to download the binaries; you will not need the source for this tutorial (or ever, unless you want to hack on PHP itself). This tutorial was written and tested on PHP 5.3.8.

MySQL
: Part of this project involves saving data to a database, so you'll need one of those, as well. Again, if you've installed XAMPP, you can skip this step, but if you choose to, you can install a database separately. In this tutorial, I'll concentrate on MySQL because it's so commonly used with PHP, so if you choose to go this route, you can download and install the Community Server.

Basic PHP syntax

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.

A basic PHP page

Start by opening your text editor and creating the most basic PHP page (see Listing 1).

Listing 1. 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 you'll see 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 (which you'll do in a moment), the browser receives the page shown in Listing 2.

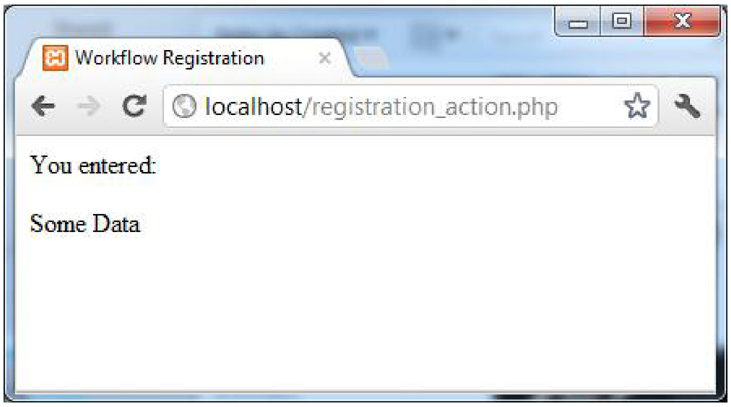
Listing 2. The browser receives the PHP page

```
<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 XAMPP, this directory is <XAMPP_HOME>/htdocs. (For Apache, this will likely be /var/www/html.)

To call this page, open your browser and point it to http://localhost/registration_action.php. You should see something similar to Figure 1.

Figure 1. Output from the echo command



You entered:

Some Data

Congratulations, you've written your first PHP page. Everything else you'll do in the language builds on this.

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 Listing 3.

Listing 3. Using variables in your PHP page

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

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

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

   </body>
</html>
```

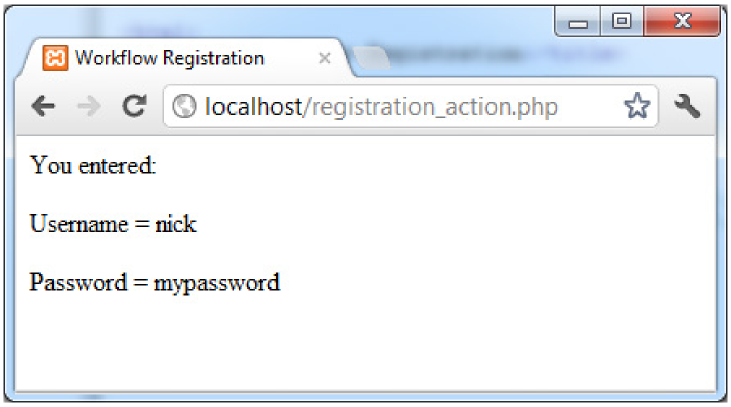
First, notice that each line must end with a semicolon. Also, notice that you use a period (`.`) to *concatenate* text, or put it together. You can put together any number of *strings*, or chunks of text, this way.

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.

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



You entered:

Username = nick

Password = mypassword

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 as shown in Listing 4.

Listing 4. Defining a constant

```
<?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.

Later, when you learn about objects, you'll see a slightly more compact way of specifying a constant.

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 as shown in Listing 5.

Listing 5. Using the output operator

```
<?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.

PHP and forms

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

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.

Creating a form in HTML

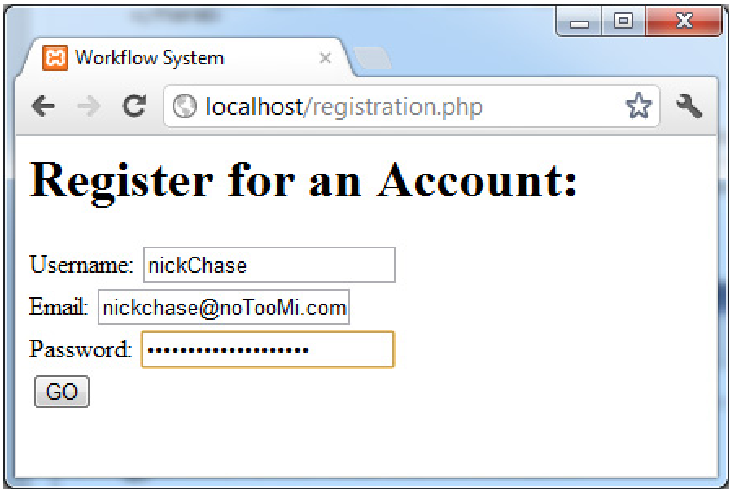
Start by creating the registration page for your application. Ultimately, users will enter their information, and you'll *validate* it, or check it for completeness, before saving it in a database. For now, just create the basic form. Create a new file called registration.php and add the following in Listing 6.

Listing 6. Creating a registration page

```
<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. Save the file in the document root (with registration_action.php). To open it, point your browser at http://localhost/registration.php and type in each of the fields. You should see something like Figure 3.

Figure 3. Register for an Account form



Register for an Account:

Username: nickChase

Email: nickchase@noTooMi.com

Password: ••••••••••••••••••••

Notice that the password box does not display the actual content you're typing; this type of form element is meant to obscure sensitive information from someone who might be looking over your shoulder. But what happens when you click the **GO** button?

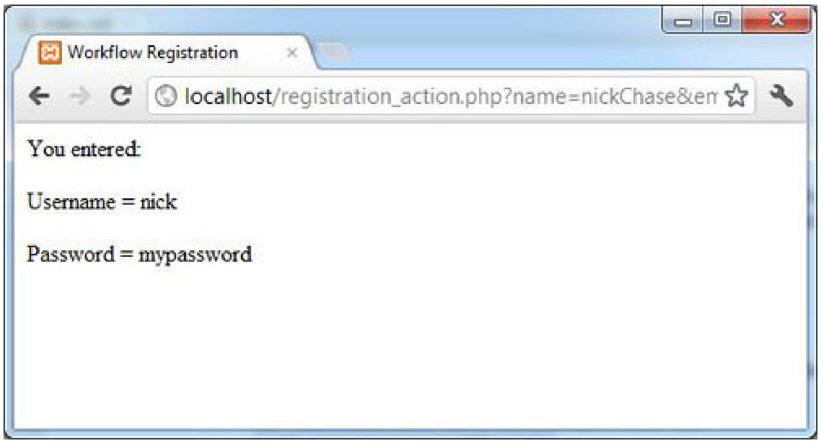
Submitting a form

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 (if you haven't already) and click the **GO** button. You should see a result similar to Figure 4.

Figure 4. Outputting the data



You entered:

Username = nick

Password = mypassword

In this case, notice that the information seems to be incorrect—it's not what you actually submitted—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=nickChase&email=nickchase%40noTooMi.com&pwor d=asupersecretpassword`.

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, which tells the browser to send the data this way. You'll also look at Using POST, which involves sending the data differently, but first take a look at actually retrieving this data from within a PHP page.

Accessing form data

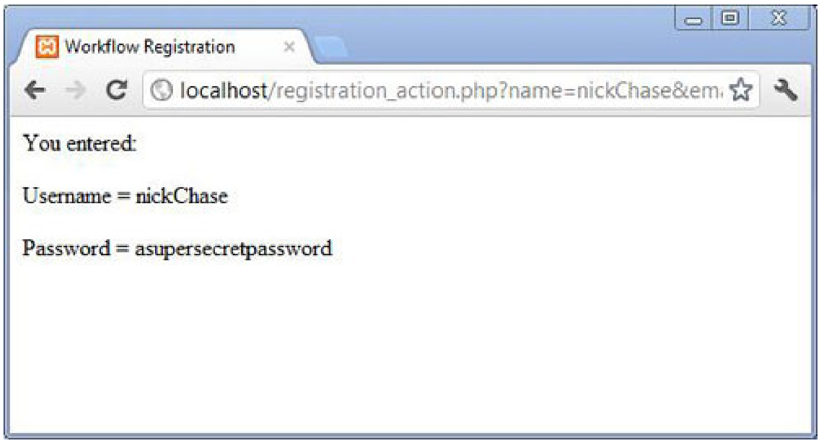
Now that you've submitted the form, you've got to get the data into the actual response page, registration_action.php. Make the changes shown in Listing 7 to that file.

Listing 7. Getting the data to the response page

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

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

         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 entered:

Username = nickChase

Password = asupersecretpassword

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 easily reference a group of values all at one time. For example, you can create an array of values and output them to the page (see Listing 8).

Listing 8. Creating an array of values

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

The `array()` function returns a value that is an array of the values passed to it. (Functions will be dealt with later, but for now, understand that you call it and it returns a value you assign to a variable.)

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 script produces the output shown in Listing 9.

Listing 9. Array output

```
0=name<br />
1=email<br />
2=pword<br />
```

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 as shown in Listing 10.

Listing 10. Submitting the form creates an array

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

That's what enables you to extract individual values, such as`$_GET["name"]`. You don't have to do this individually, however.

Getting array information by the numbers

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.

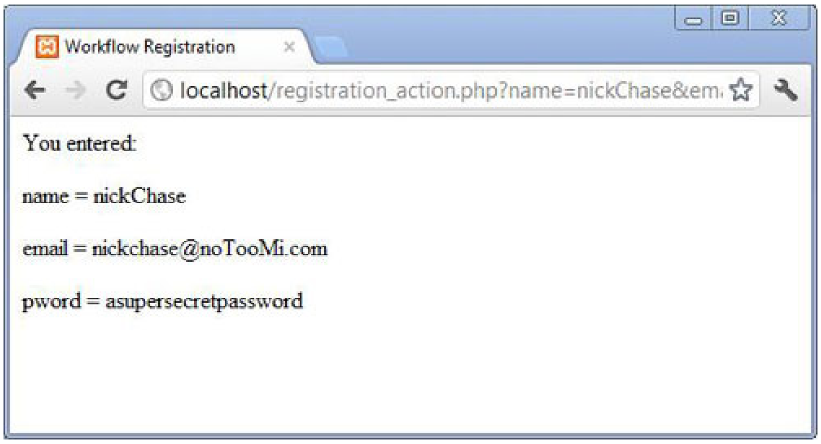
Fortunately, PHP provides two functions that make your life a little easier (see Listing 11).

Listing 11. The `array_keys()` and the `array_values()` functions

```
...
<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>
...
```

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



You entered:

name = nickChase

email = nickchase@noTooMi.com

pword = asupersecretpassword

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. Let's look at an example of this the loop in Listing 12.

Listing 12. for-next loop

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

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. This the output:

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

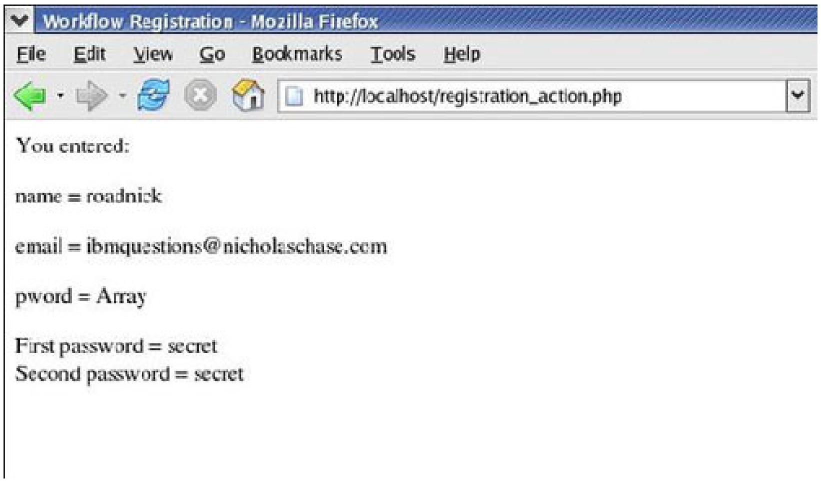
What this means is that if you can find out how many values are in the `$_GET` array—which you can do—you can easily loop through all of the values provided by the form as shown in Listing 13.

Listing 13. Looping through all 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>
...
```

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


Workflow Registration - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/registration_action.php

You entered:

name = roadnick

email = ibmquestions@nicholaschase.com

pword = Array

First password = secret
Second password = secret

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, the code in Listing 14.

Listing 14. Using a `foreach` loop

```
...
   $form_values = array_values($_GET);

   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 shown in Listing 15.

Listing 15. Output from the `foreach` loop

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

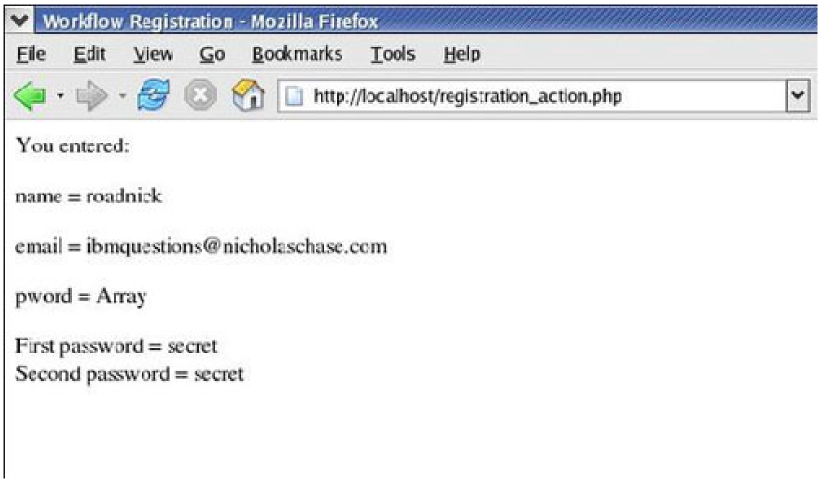
Even more handy, however, is the ability to extract the value *and* the key as shown in Listing 16.

Listing 16. Extracting the value and the key

```
...
   $form_values = array_values($_GET);

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

This brings you back to the original result (see Figure 8).

Figure 8. The original result


Workflow Registration - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost/registration_action.php

You entered:

name = roadnick

email = ibmquestions@nicholaschase.com

pword = Array

First password = secret
Second password = secret

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 by adding the code in Listing 17 to registration.php:

Listing 17. Multiple form values with a single name

```
...
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: `http://localhost/registration_action.php?name=nickChase&email=nickchase%40noTooMi.com &pword%5B%5D=asupersecretpassword&pword%5B%5D=asupersecretpassword`.

(The value `%5B%5D` is the url-encoded version of `[]`.)

Submitting the form is the same as creating arrays (see Listing 18).

Listing 18. Submitting the form creates an array

```
$passwords = array("asupersecretpassword", "asupersecretpassword");
$_GET = array("name"=>"nickChase",
                "email"=>"nickChase@noTooMi.com",
                "pword"=>$passwords);
```

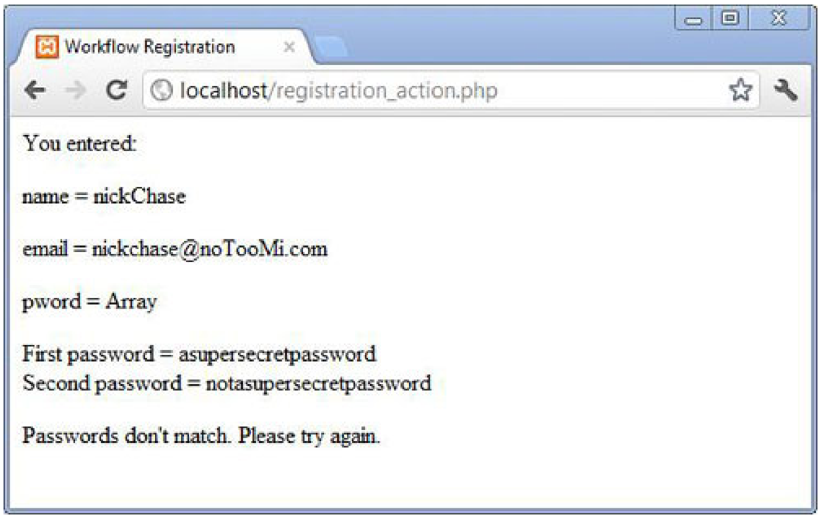
All this means that if you want to see the password values, you'll need to access them as a numeric array, as shown in Listing 19.

Listing 19. Accessing the password values as a numeric array

```
...
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



```
You entered:

name = nickChase

email = nickchase@noTooMi.com

pword = Array

First password = asupersecretpassword
Second password = notasupersecretpassword

Passwords don't match. Please try again.
```

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

Using `GET` versus using `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 has indexed the URL, unaware 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 as shown in Listing 20.

Listing 20. Using `POST` method instead of `GET` on the registration 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 have two choices. The first is to use the `$_POST` array rather than the `$_GET` array in registration_action.php, as shown in Listing 21.

Listing 21. Using the `$_POST` array to retrieve the data

```
...
<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>
...
```

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

The good thing about this method is that you control precisely what it is your script is looking at. The bad thing is that if you want to debug by putting something into the URL, you have to change your script. One common way to get around this problem is to use a third array, `$_REQUEST`, which includes data passed in both places (see Listing 22).

Listing 22. Using a `$_REQUEST` array for debugging

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

      $passwords = $_REQUEST["pword"];
      echo "First password = ".$passwords[0];
...
```

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 as shown in Listing 23.

Listing 23. Using an if-then statement

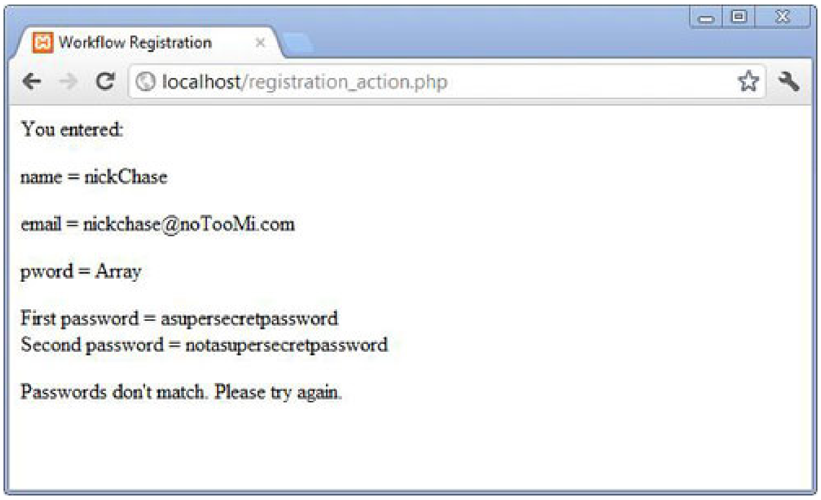
```
...
$passwords = $_POST["pword"];
echo "First password = ".$passwords[0];
echo "<br />";
echo "Second password = ".$passwords[1];

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] == $passwords[1]` with a double equals sign. The double equals sign is the comparative operator. It actually detects whether the two are equal. The single equals sign is the assignment operator. With a single equals sign, when you executed the statement, PHP would simply assign the value of `$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



```
You entered:

name = nickChase

email = nickchase@noTooMi.com

pword = Array

First password = asupersecretpassword
Second password = notasupersecretpassword

Passwords don't match. Please try again.
```

Two more handy operators are the *and* operator (`&&`) and the *or* operator (`||`). Listing 24 shows them in action.

Listing 24. The *and* and the *or* operators

```
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.

Functions

It's time to dive in a little deeper and see how implementing functions in PHP can help you save time.

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.

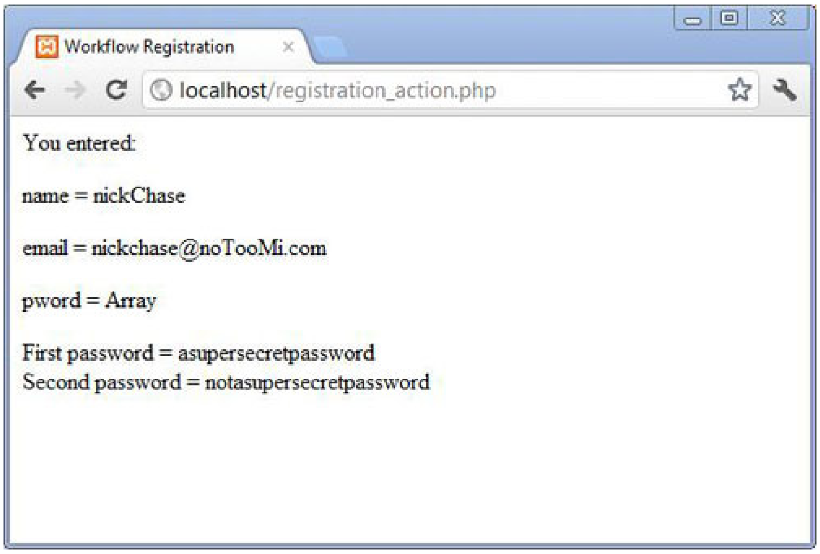
In those cases, it's helpful to take the code and use it to create a *function*. For example, you can take the password validation and put it into a separate function, as shown in Listing 25.

Listing 25. Creating a function

```
...
<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>

...
```

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



You entered:

name = nickChase

email = nickchase@noTooMi.com

pword = Array

First password = asupersecretpassword
Second password = notasupersecretpassword

So, how do you actually use the function?

Calling a function

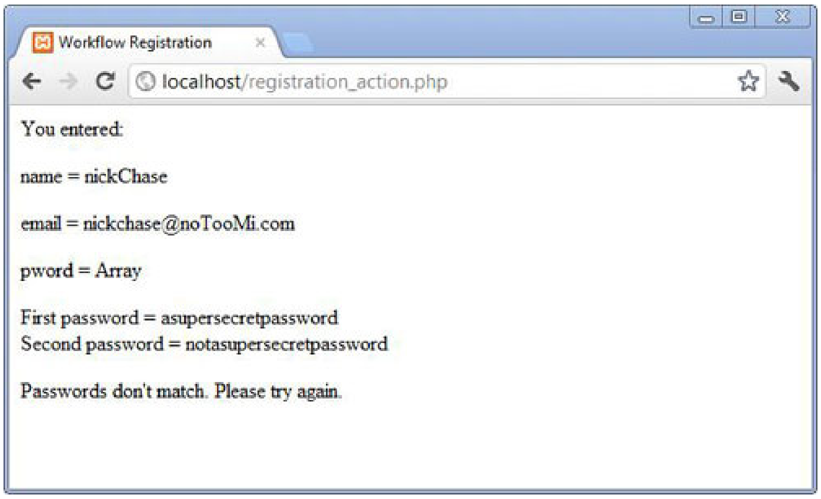
To call a function, you use its name and follow it with a pair of parentheses. If you expect any arguments, as in this case, they go in the parentheses, as shown in Listing 26.

Listing 26. Calling a function

```
...
<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>

...
```

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 (see Figure 12). (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



You entered:

name = nickChase

email = nickchase@noTooMi.com

pword = Array

First password = asupersecretpassword
Second password = notasupersecretpassword

Passwords don't match. Please try again.

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 (see Listing 27).

Listing 27. Returning a value

```
...
<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;

      }


      function checkPasswords($firstPass, $secondPass){

...
```

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

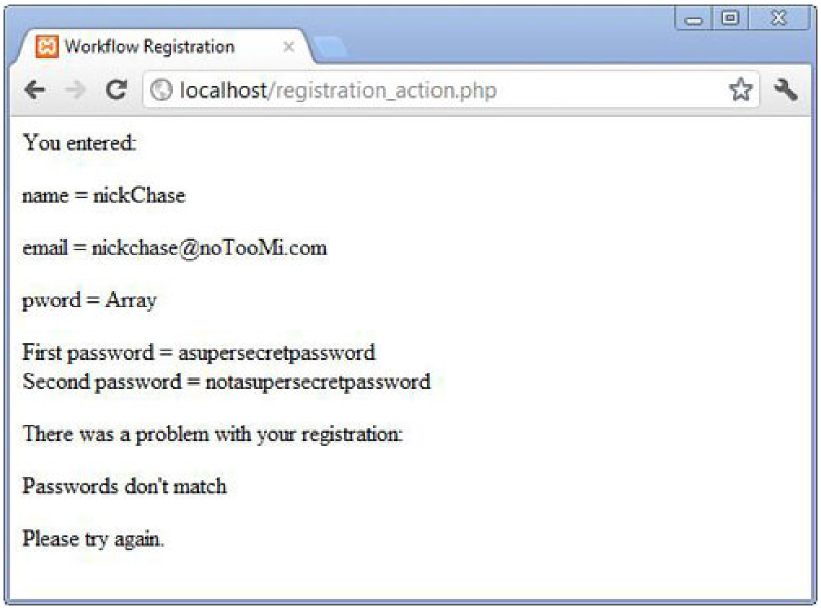
You've created a function that returns a value based on whether the user's input is appropriate, so now you can test for that value (see Listing 28).

Listing 28. Validating the data

```
...
   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



```
You entered:

name = nickChase

email = nickchase@noTooMi.com

pword = Array

First password = asupersecretpassword
Second password = notasupersecretpassword

There was a problem with your registration:

Passwords don't match

Please try again.
```

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 have a way to know that the data is all right, you can enter it into the database.

Connecting to and using a database

Many of your PHP applications will need to use a database, which is what I cover in this section.

Choosing a database—or not: Using PHP Data Objects (PDO)

Because it's freely available and configuration is included in many hosting plans and distributions of PHP, MySQL has generally been the database of choice for the vast majority of PHP applications. Which is fine, except that the resulting code has been specific to MySQL.

Unfortunately that means that if, further down the line, you needed to change databases—say to Sqlite, IBM® DB2® or Oracle—all of your database-related code had to be rewritten. Fortunately, with the advent of PHP Data Objects, or PDO, and its inclusion with the core install of PHP as of version 5.3, it's possible to write PHP database-related code that isn't tied to a specific database.

The way it works is that you write your database access code using PDO, and then include the database-specific PDO driver. To change your database, you simply need to include the appropriate driver and change the database "connect string."

In this case, you are going to use PDO to connect to a MySQL database. If you're using XAMPP, you don't need to do any additional configuration; the MySQL driver is enabled by default. If you're not using XAMPP, you will need to make sure that the following lines are uncommented in php.ini: `extension=php_pdo.dll` and `extension=php_pdo_mysql.dll`.

These dlls should already be present in the location specified in the php.ini file as the `extension_dir`.

Now you'll need to take care of the database itself.

Setting up

Before going any further, you need to create a database, add a table, and create a new user who has access to it.

To get to the MySQL console if you have installed XAMPP, type the commands in Listing 29.

Listing 29. Getting to the MySQL console

```
cd <XAMPP_HOME>
mysql/bin/mysql -h localhost -u root
```

(If you've installed MySQL separately and created a password for the `root` user, you will need to add the `-p` parameter and enter the password when requested.)

Once you're in the console, type the code in Listing 30.

Listing 30. Creating the database

```
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 Listing 31.

Listing 31. Final output

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

Finally, add the new user, `wfuser`, with a password of `wfpass` (see Listing 32).

Listing 32. Add the new user

```
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 a database

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 to see if 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.

Start by connecting to the workflow database you created in Setting up (see Listing 33).

Listing 33. Connecting to the workflow database

```
...
if (validate($_POST) == "OK") {

   $dbh = new PDO('mysql:host=localhost;dbname=workflow', 'wfuser', 'wfpass');

   echo "<p>Thank you for registering!</p>";

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

Here, you're creating a new object, `$dbh`. You'll learn more about objects in later parts of this series, but for now, you can think of an object as a collection of functions for you to use.

The object uses the database's "connect string" (`mysql:host=localhost;dbname=workflow`) along with the username and password to connect to the MySQL database. (Different databases have different types of connect strings.)

Assuming all goes well, this connection will stay open until you destroy the object (which you'll see shortly), or until the page is finished processing.

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 shown in Listing 34.

Listing 34. SQL statement that inserts data into the table

```
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.

But actually inserting the data requires more than just building the insert statement. It used to be common practice to do something like Listing 35.

Listing 35. Old way of inserting data

```
$sql = "insert into users (username, email, password) values \
                  ('".$_POST["name"]."', '".$_POST["email"]."', '".$passwords[0]."')
```

However, this kind of construction is vulnerable to a hacking technique called "SQL injection," which can lead to data being stolen or deleted. Fortunately, one of the advantages of using PDO rather than directly accessing the database is the ability to easily use "prepared statements," which eliminates this risk. Here's how it works:

You need to create, or prepare the statement (see Listing 36).

Listing 36. Creating the statement

```
...
if (validate($_POST) == "OK") {

   $dbh = new PDO('mysql:host=localhost;dbname=workflow', 'wfuser', 'wfpass');

   $stmt = $dbh->prepare("insert into users (username, email, password) \
                                         values (:name, :email, :pword)");

   echo "<p>Thank you for registering!</p>";

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

Notice the `->` construction. That's how you reference an object's functions or properties. In this case, you're telling the database to prepare an insert statement with three parameters.

Next you need to "bind" variables to the parameters (see Listing 37).

Listing 37. Binding variables to the parameters

```
...
   $stmt = $dbh->prepare("insert into users (username, email, password) \
                                         values (?, ?, ?)");

   $stmt->bindParam(1, $name);
   $stmt->bindParam(2,  $email);
   $stmt->bindParam(3, $pword);

   echo "<p>Thank you for registering!</p>";

} else {
...
```

This step tells the database to take whatever value is in the variable and use it for that particular parameter. You then set the values and execute the statement, as shown in Listing 38.

Listing 38. Setting the values and executing the statement

```
...
   $stmt = $dbh->prepare("insert into users (username, email, password) \
                                     values (?, ?, ?)");

   $stmt->bindParam(1, $name);
   $stmt->bindParam(2,  $email);
   $stmt->bindParam(3, $pword);

   $name = $_POST["name"];
   $email = $_POST["email"];
   $pword = $passwords[0];

   $stmt->execute();

   echo "<p>Thank you for registering!</p>";

} else {
...
```

The end result is that the database executes the query with the proper values, but without the risk of a SQL Injection attack. (Proper steps, such as escaping specific characters, are taken on the back end.)

Another advantage of doing things this way is that you can re-use the prepared statement. Simply assign new values to the `$name`, `$email` and `$pword` variables, and re-execute the statement.

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 (see Listing 39).

Listing 39. Ensuring that the username is unique

```
...
if (validate($_POST) == "OK") {
    $dbh = new PDO('mysql:host=localhost;dbname=workflow', 'wfuser', 'wfpass');

    $checkUserStmt = $dbh->prepare("select * from users where username = ?");
    $checkUserStmt->bindParam(1, $name);
    $name = $_POST["name"];
    $checkUserStmt->execute();

    if ($checkUserStmt->rowCount() == 0) {

        $stmt = $dbh->prepare("insert into users (username, email, password)\
                                        values (?, ?, ?)");
...
        echo "<p>Thank you for registering!</p>";

    } else {

        echo "There is already a user with that name: <br />";

    }

} else {
```

```
...
```

In this case, you're creating a prepared statement just as you were the first time, but the `execute()` function may seem a little less intuitive. After all, you're not really *doing* anything, right? Wrong. What you're doing is populating the statement with the data that satisfies the query—any database rows that have the username the user has entered.

To check and see whether the database found any records, you can check the `rowCount()` function, which returns the number of rows found by the query. If there are any, it means the username isn't unique.

In actual fact, the statement contains all of the information on any rows that satisfy the query. You'll retrieve a set of 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 saw how to use a prepared statement. If you're not using any untrusted data (such as the username or other information submitted by the user, then you have the option to simply execute the query directly (see Listing 40).

Listing 40. Executing the query directly

```
...
    } else {

        echo "<p>There is already a user with that name: </p>";

        $users = $dbh->query("SELECT * FROM users");
        $row = $users->fetch();
        echo $row["username"] . " -- " . $row["email"] . "<br />";

    }
...
```

The first step is to populate the `$users` statement by executing the query, which selects all rows in the `users` table.

Next, you'll use the `fetch()` function to extract a single row. This function 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

To see all of the data, you want the page to keep fetching rows as long as there are rows to fetch. In order to do that, you can set up a `while` loop (see Listing 41).

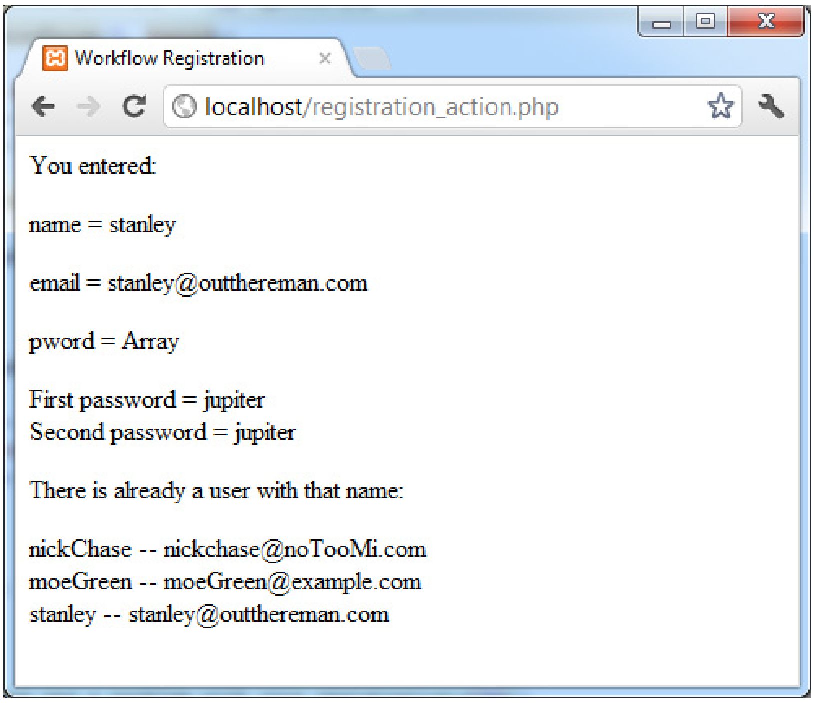
Listing 41. Setting up a `while` loop

```
                  ...
                        $users = $dbh->query("SELECT * FROM users");
        while ($row = $users->fetch()) {
            echo $row["username"] . " -- " . $row["email"] . "<br />";
        }
                  ...
```

PHP starts by attempting to `fetch` a row. If it succeeds, the result is `true` and it goes on to display the information for that row. If it fails—in other words, if there are no more rows to `fetch`—the expression returns `false` and the `while` loop ends.

The result is a listing of all of the accounts that currently exist, as in Figure 14.

Figure 14. Displaying multiple rows from a table



You entered:

name = stanley

email = stanley@outthereman.com

pword = Array

First password = jupiter
Second password = jupiter

There is already a user with that name:

nickChase -- nickchase@noTooMi.com
moeGreen -- moeGreen@example.com
stanley -- stanley@outthereman.com

Close the database connection

Before moving on, you need to make sure that the database connection you opened gets closed again. You do that by destroying the PDO object that connects to the database (see Listing 42).

Listing 42. Destroying the PDO object

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

     $dbh = null;

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

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

Cleaning up: including 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.

Why include files?

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 website, 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.

To start, create a file called top.txt and add the code in Listing 43.

Listing 43. Creating a file called top.txt

```
  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>Workflow Manager</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
  </head>
  <body>
    <div id="wrapper"><div id="bg"><div id="header"></div>
      <div id="page"><div id="container"><div id="banner"></div>
        <div id="nav1">
          <ul style='float: left'>
            <li><a href="#" shape="rect">Home</a></li>
            <li><a href="#" shape="rect">Upload</a></li>
            <li><a href="#" shape="rect">Files</a></li>
          </ul>
        </div>
        <div id="content">
          <div id="center">
```

In a separate file called bottom.txt, add the following shown in Listing 44.

Listing 44. bottom.txt

```
          </div>
        </div>
      </div>
    </div></div></div>
  </body>
</html>
```

Save both files in the same directory as registration.php. Copy the images directory and the style.css file from the PHPPart1SampleFiles.zip in Downloadable resources to that directory as well, as they're referenced by the HTML code in top.txt.

Including the files

Now go ahead and add these files to the registration page. Edit registration.php to look like Listing 45.

Listing 45. Adding top.txt and bottom.txt to the registration page

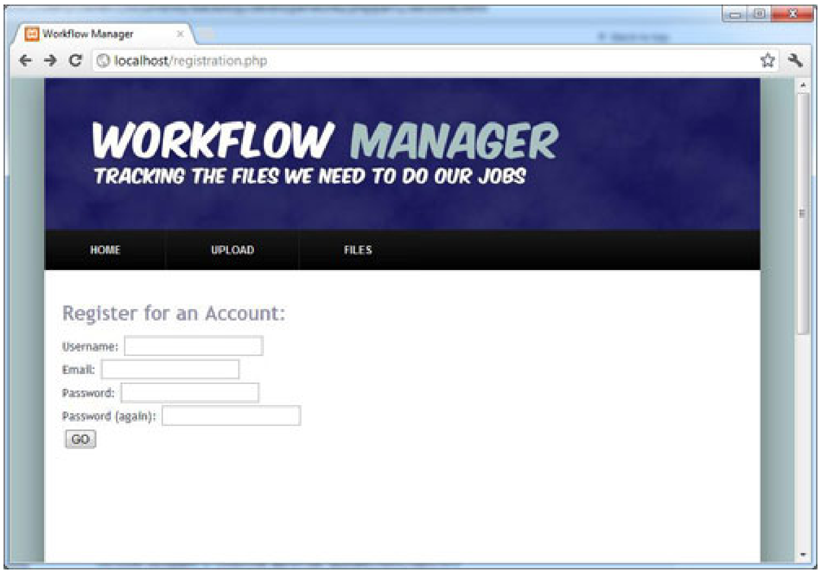
```
<?php

   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 will see a much different look, as shown in Figure 15.

Figure 15. 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 (see Listing 46).

Listing 46. Files have been merged in the output

```
  <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en">
  <head>
    <title>Workflow Manager</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
  </head>
  <body>
    <div id="wrapper"><div id="bg"><div id="header"></div>
      <div id="page"><div id="container"><div id="banner"></div>
        <div id="nav1">
          <ul style='float: left'>
            <li><a href="#" shape="rect">Home</a></li>
            <li><a href="#" shape="rect">Upload</a></li>
            <li><a href="#" shape="rect">Files</a></li>
          </ul>
        </div>
        <div id="content">
          <div id="center">
<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>

          </div>
        </div>
      </div>
    </div></div></div>
  </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()` script out into a separate file and include that file in the registration_action.php file. If PHP can't find it, that's a problem because you're calling that functions within the page. So, to avoid that, you can use the `require()` function, instead of `include()` (see Listing 47).

Listing 47. Using the `require()` function

```
<?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 (see Listing 48).

Listing 48. Ensuring 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
...
```

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.

Conclusion

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.

Downloadable resources

Description	Name	Size
Part 1 source code	PHPPart1SampleFiles.zip	62KB

Related topics

- PHP documentation is available in many languages.
- MySQL documentation is also available.
- Learn more about using "PHP Data Objects" to create database-independent code.
- Read "Connecting PHP applications to IBM DB2 Universal Database" (Dan Scott, developerWorks, July 2001).
- Check out "Connecting PHP Applications to Apache Derby" (Moira Casey, developerWorks, September 2004).
- Don't miss "Develop IBM Cloudscape and DB2 Universal Database applications with PHP: (Dan Scott, developerWorks, February 2005).
- For information about PHP with MySQL, read "Creating dynamic Web sites with PHP and MySQL" (Md. Ashraful Anam, developerWorks, May 2001).
- "Using HTML forms with PHP" covers some additional issues not discussed in this tutorial (Nicholas Chase, developerWorks, August 2002).
- Download XAMPP.
- Download PHP.
- Download MySQL.
- Visit IBM developerWorks' PHP project resources to learn more about PHP.
- Follow developerWorks on Twitter.

© Copyright IBM Corporation 2005, 2012
(www.ibm.com/legal/copytrade.shtml)
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)