CHAPTER 3

Passing Information with PHP

Now that you're comfortable with the basics of PHP, you're ready to go over how information is moved from page to page. You'll need this for nearly every aspect of the blog, from writing new posts to allowing users to comment.

In this chapter, you'll learn the following:

- What superglobal arrays are in PHP and how to use them
- How to send data using HTML forms using the POST method
- How to send data in the URL using the GET method
- How to store data in SESSIONS for use across multiple pages
- How to create and store COOKIES for returning users

Passing data is what separates dynamic web pages from static ones; by customizing an experience based on the user's choices, you're able to add an entirely new level of value to a web site.

Superglobal Arrays

PHP offers several types of *superglobal arrays* to developers, each with a different useful purpose. A superglobal array (refer back to Chapter 2 for more information on arrays) is a special variable that are always available in scripts, regardless of the current scope of the script (see the *Variable Scope* section later in this chapter). PHP includes several superglobals:

- `$GLOBALS`: Variables available in the global scope
- `$_SERVER`: Information about the server
- `$_GET`: Data passed using the HTTP GET method
- `$_POST`: Data passed using the HTTP POST method
- `$_REQUEST`: Data passed via an HTTP request
- `$_FILES`: Data passed by an HTML file input
- `$_SESSION`: Current session data specific to the user
- `$_COOKIE`: Data stored on the user's browser as a cookie

Variable Scope

In programming, *scope* refers to the context you declare a variable in. Most variables in PHP have a single scope: *global*. Using this scope means that a variable is available in the script that declares it, as well as in any script that is included after the variable is declared or in any script that includes the file in which the variable is declared.

For example, try opening test.php again and experimenting with variable scope:

```
<?php

    $foo = "some value";

    include_once 'extras.php'; // $foo is available in extras.php

    $bar = "another value"; // $bar is not available in extras.php

    echo "test.php: Foo is $foo, and bar is $bar. <br />";

?>
```

Now open up extras.php and insert the following code:

```
<?php

    echo "extras.php: Foo is $foo, and bar is $bar. <br />";

?>
```

When you load http://localhost/simple_blog/test.php in your browser, you get the following output:

```
extras.php: Foo is some value, and bar is .
test.php: Foo is some value, and bar is another value.
```

■ **Note** When error_reporting(E_ALL) is active, a notice will display, letting you know that $bar is an undefined variable. Error reporting is a great way to make sure all the bases are covered in your scripts. When developing an application, best practice dictates that you should use error_reporting(E_ALL), which means all errors and notices are displayed. You don't want our users to see nasty error codes in your production scripts, however, so you should use error_reporting(0) to turn off errors in your finished application.

Scope changes a bit when you start using functions, because variables declared within a function have *local scope*, meaning they're only available within the function that declares them. Additionally, variables declared in the global scope are only available if they are explicitly declared as `global` inside the function.

You're now ready to see how `error_reporting(E_ALL)` affects your scripts. In `test.php`, write the following code:

```
<?php

    error_reporting(E_ALL);

    $foo = "I'm outside the function!";

    function test()
    {
        return $foo;
    }

    echo test(); // A notice is issued that $foo is undefined

?>
```

`$foo` is undefined if you run your `test()` function, which issues a notice. You can clear this up by declaring `$foo` as global within your `test()` function; this means that `$foo` in local scope will now refer to the variable `$foo` in global scope:

```
<?php

    $foo = "I'm outside the function!";

    function test()
    {
        global $foo; // Declare $foo as a global variable
        return $foo;
    }

    echo test();

?>
```

Visiting `test.php` executes `test()`, you can see the result in this output:

```
I'm outside the function!
```

A variable declared within a function is not available outside that function unless it is specified as the function's return value. For instance, consider the following code:

```
<?php
    error_reporting(E_ALL);

    function test()
    {
        $foo =  "Declared inside the function. <br />";
        $bar = "Also declared inside the function. <br />";

        return $bar;
    }

    $baz = test();

    /*
     * Notices are issued that $foo and $bar are undefined
     */
    echo $foo, $bar, $baz;

?>
```

This code gives the following results:

```
Notice: Undefined variable: foo in /Applications/xampp/xamppfiles/htdocs/simple_blog/➥
test.php on line 19

Notice: Undefined variable: bar in /Applications/xampp/xamppfiles/htdocs/simple_blog/➥
test.php on line 19
Also declared inside the function.
```

You need to declare two variables within a function and return both; next, you use an array and the `list()` function to access the values easily.

```
<?php

    function test()
    {
        $foo = "Value One";
        $bar = "Value Two";

        return array($foo, $bar);
    }
```

```
    /*
     * The list() function allows us to assign a variable
     * to each array index as a comma-separated list
     */
    list($one, $two) = test();

    echo $one, "<br />", $two, "<br />";

?>
```

Running this code produces the desired output:

```
Value One
Value Two
```

Using `list()` is a way to declare multiple variables in one line; for example this line declares the variables `$one` and `$two`:

```
list($one, $two) = test();
```

The following handful of lines accomplishes the same thing:

```
    $array = test();

    $one = $array[0];
    $two = $array[1];
```

■ **Note** Only numerically indexed arrays will work when using `list()`. Using an array with text-based keys issues a notice that indicates you have undefined array indexes.

$GLOBALS

PHP provides another option for accessing variables in the global scope: the `$GLOBALS` superglobal array. All variables in the global scope are loaded into the `$GLOBALS` array, enabling you to access them using the variable name as the array key.

You can try out this array in `test.php`:

```
<?php

    $foo = "Some value.";

    function test()
```

```
    {
        echo $GLOBALS['foo'];
    }

    test();

?>
```

This code produces the following output:

```
Some value.
```

■ **Tip** It is generally a good practice to avoid using globals wherever possible. The preferred method of accessing global variables inside functions is to pass them as arguments. This makes your scripts more readable, which simplifies maintenance over the long term.

$_SERVER

The `$_SERVER` superglobal stores information about the server and the current script. It also has features that allow you to access the IP address of a site visitor, what site referred the visitor to this script, and many other useful pieces of information. I won't cover all of the capabilities of `$_SERVER` here, so be sure to check the PHP manual to learn more about this feature.

One of the most useful pieces of information available in the `$_SERVER` superglobal is the name of the host site, which is stored in `HTTP_HOST`. The host site's name is useful because it allows you to create a simple template that you can use across different projects without requiring that you change any of your code.

For instance, you can use the following code snippet to welcome a visitor to your site:

```
<?php

    echo "<h1> Welcome to $_SERVER[HTTP_HOST]! </h1>";

?>
```

Running this code in `test.php` produces this output:

```
Welcome to localhost!
```

Loading the exact same snippet on a live web site produces this output:

```
Welcome to yoursite.com!
```

As you continue forward with creating your simple blog, you'll discover a few uses for HTTP_HOST in your scripts.

Other useful pieces of information are available as well, shown below (see comments for a description of what each piece means).

Note the use of print_r() at the bottom of the script. This is a great way to debug code, especially arrays, because it outputs a "human-readable" display of a variable's contents. You can see this at work when you load your test script in test.php:

```
<?php

    /*
     * Note that <pre> tags and newline characters (\n) are used
     * for the sake of legibility
     */

    // Path to the current file (i.e. '/simple_blog/test.php')
    echo $_SERVER['PHP_SELF'], "\n\n";

    // Information about the user's browser
    echo $_SERVER['HTTP_USER_AGENT'], "\n\n";

    // Address of the page that referred the user (if any)
    echo $_SERVER['HTTP_REFERER'], "\n\n";

    // IP address from which the user is viewing the script
    echo $_SERVER['REMOTE_ADDR'], "\n\n";

    // Human-readable export of the contents of $_SERVER
    print_r($_SERVER);

?>
```

Loading `test.php` in your browser displays output something like the following:

/simple_blog/test.php

```
Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US; rv:1.9.0.10) Gecko/2009042315
Firefox/3.0.10



Notice:  Undefined index:  HTTP_REFERER in
/Applications/XAMPP/xamppfiles/htdocs/simple_blog/test.php on line 19



::1

Array
(
    [UNIQUE_ID] => ShwUrQoAAQYAAAE4T@0AAAAA
    [HTTP_HOST] => localhost
    [HTTP_USER_AGENT] => Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;➥
 rv:1.9.0.10) Gecko/2009042315 Firefox/3.0.10
    [HTTP_ACCEPT] => text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
    [HTTP_ACCEPT_LANGUAGE] => en-us,en;q=0.5
    [HTTP_ACCEPT_ENCODING] => gzip,deflate
    [HTTP_ACCEPT_CHARSET] => ISO-8859-1,utf-8;q=0.7,*;q=0.7
    [HTTP_KEEP_ALIVE] => 300
    [HTTP_CONNECTION] => keep-alive
    [HTTP_COOKIE] =>

    [HTTP_CACHE_CONTROL] => max-age=0
    [PATH] => /usr/bin:/bin:/usr/sbin:/sbin
    [SERVER_SIGNATURE] =>

    [SERVER_SOFTWARE] => Apache/2.2.11 (Unix) DAV/2 mod_ssl/2.2.11 OpenSSL/0.9.7l ➥
PHP/5.2.9 mod_perl/2.0.4 Perl/v5.10.0
    [SERVER_NAME] => localhost
    [SERVER_ADDR] => ::1
    [SERVER_PORT] => 80
    [REMOTE_ADDR] => ::1
    [DOCUMENT_ROOT] => /Applications/XAMPP/xamppfiles/htdocs
    [SERVER_ADMIN] => you@example.com
    [SCRIPT_FILENAME] => /Applications/XAMPP/xamppfiles/htdocs/simple_blog/test.php
    [REMOTE_PORT] => 49310
```

```
    [GATEWAY_INTERFACE] => CGI/1.1
    [SERVER_PROTOCOL] => HTTP/1.1
    [REQUEST_METHOD] => GET
    [QUERY_STRING] =>
    [REQUEST_URI] => /simple_blog/test.php
    [SCRIPT_NAME] => /simple_blog/test.php
    [PHP_SELF] => /simple_blog/test.php
    [REQUEST_TIME] => 1243354285
    [argv] => Array
        (
        )

    [argc] => 0
)
```

Be sure to check whether HTTP_REFERER, is set before you use it. Otherwise, HTTP_REFERER is undefined if the visitor was not referred by another site a notice will be issued that. This code provides an easy way to tell whether it is set:

```
if(isset($_SERVER['HTTP_REFERER']))
{
    echo $_SERVER['HTTP_REFERER'];
}
else
{
    echo "No referer set!";
}
```

$_GET

One of the two most common methods of passing data between pages is the GET method. GET data is passed through a *query string* in the URL, which looks something like this:

```
http://example.com?var1=somevalue&var2=othervalue
```

The query string begins with a question mark (?) and then names a variable (var1 in the preceding example). The next part of the query string is an equals sign (=), followed by a value. You can add more variables by appending an ampersand (&) to the end of the first value, then declaring another variable name (var2 in the preceding example), followed by an equals sign and a value. Note that you do not enclose the values in the query string in quotes; this can cause problems when you use values that have spaces and/or special characters.

URL Encoding

Fortunately, PHP includes the `urlencode()` function, which you can use to get values ready to be passed in a URL. You use this function in tandem with `urldecode()`; together, the two functions enable you to pass complex values through the URL. Add this code to `test.php` to learn how URL encoding works:

```
<?php

    error_reporting(E_ALL);

    $foo = "This is a complex value & it needs to be URL-encoded.";

    // Output the original string
    echo $foo, "<br /><br />";

    // URL encode the string
    $bar = urlencode($foo)

    // Output the URL-encoded string
    echo $bar, "<br /><br />";

    // Decode the string and output
    echo urldecode($bar);

?>
```

Loading `test.php` in a browser produces the following result:

```
This is a complex value & it needs to be URL-encoded.
This+is+a+complex+value+%26+it+needs+to+be+URL-encoded.
This is a complex value & it needs to be URL-encoded.
```

Accessing URL Variables

You use the `$_GET` superglobal array to access the variables in the query string. You can access the variables in this array with the variable name as the array key. For instance, you can use this code to access your variables from the URL example at the beginning of this section:

```
<?php

    echo "var1: ", $_GET['var1'], "<br />";
    echo "var2: ", $_GET['var2'], "<br />";

?>
```

To test this snippet of code, navigate to this URL: `http://localhost/simple_blog/test.php?var1=somevalue&var2=anothervalue`. Running this code produces the following output:

```
var1: somevalue
var2: anothervalue
```

The `$_GET` superglobal allows you to determine what information displays on a page, depending on the values you pass in the URL. For example, you can use this code if you want to allow a user to view either a brief description of your site *or* your contact information:

```
<ul id="menu">
    <li> <a href="test.php">Home</a> </li>
    <li> <a href="test.php?page=about">About Us</a> </li>
    <li> <a href="test.php?page=contact">Contact Us</a> </li>
</ul>

<?php

    /*
     * Very basic security measure to ensure that
     * the page variable has been passed, enabling you to
     * ward off very basic mischief using htmlentities()
     */
    if(isset($_GET['page'])) {
        $page = htmlentities($_GET['page']);
    } else {
        $page = NULL;
    }

    switch($page) {

        case 'about':
            echo "
            <h1> About Us </h1>
                <p> We are rockin' web developers! </p>";
            break;

        case 'contact':
            echo "
                <h1> Contact Us </h1>
                    <p> Email us at
```

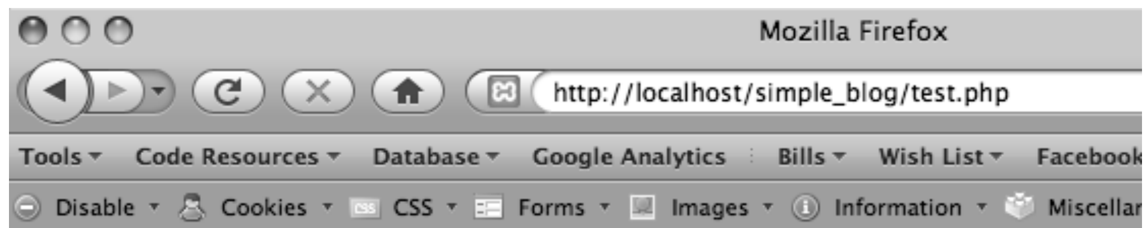
```
                    <a href=\"mailto:info@example.com\">
                    info@example.com
                        </a>
                </p>";
            break;

        /*
         * Create a default page in case no variable is passed
         */
        default:
            echo "
                <h1> Select a Page! </h1>
                <p>
                    Choose a page from above

                    to learn more about us!
                </p>";
                break;
    }

?>
```

Loading the page displays a screen similar to what you see in Figure 3-1.



- Home
- About Us
- Contact Us

Select a Page!

Choose a page from above to learn more about us!

Figure 3-1. *The default page if no* `$_GET` *variable is supplied*

After clicking the "About Us" link at the top, the page will reload (see Figure 3-2).

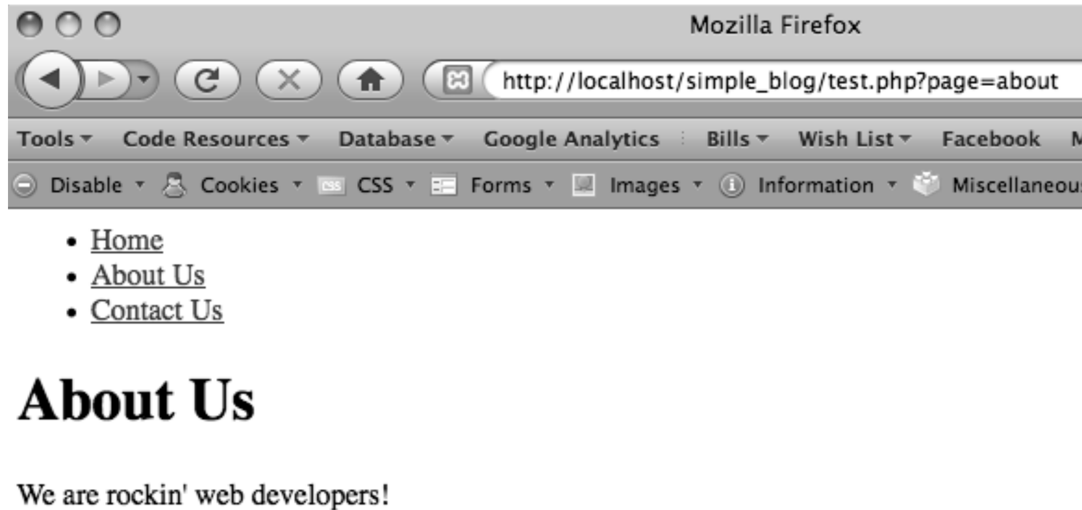


Figure 3-2. *The About Us page is loaded when the* `page=about` *query string is passed in the URL*

Click the Contact Us link to bring up a page similar to what you see in Figure 3-3.

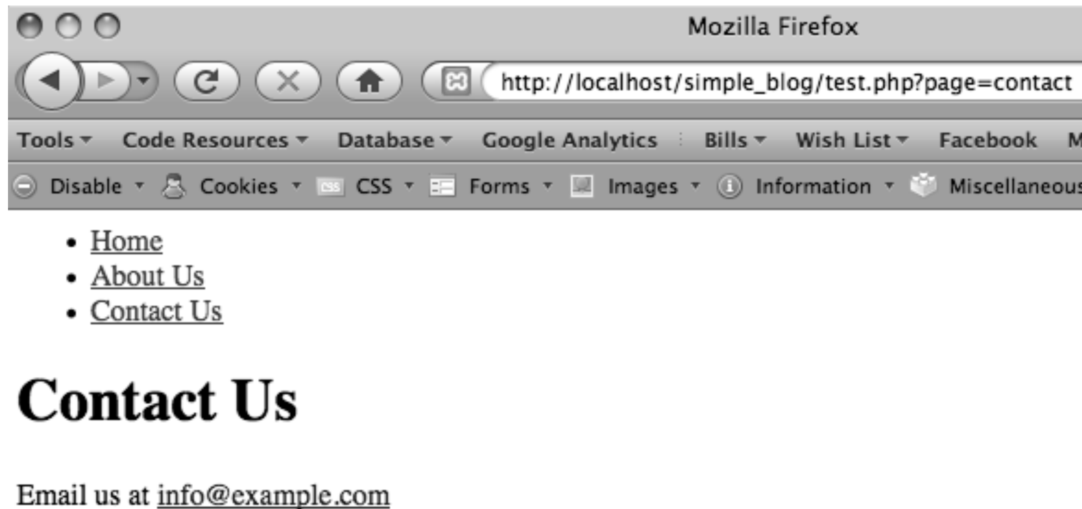


Figure 3-3. *The Contact Us page is loaded when* `page=contact`

■ **Caution** $_GET variables are useful, but they also allow malicious users to pass information directly into your script. Because of this, you should always sanitize the input. You should also avoid sending sensitive information via a GET request. Basic security precautions will be covered later in this book.

$_POST

PHP provides a second method for sending data: POST. Submitting most web forms (such as a contact form or site registration) requires using the POST method to submit the form information for processing.

Structurally, POST requests are very similar to GET requests, and you access their values in the same way: you use the $_POST superglobal and append the variable name as the array key.

In most cases, a POST value is passed by a web form that you build in HTML. For example, take a look at this registration form:

```
<form action="test.php" method="post">
    <input type="text" name="username" />
    <input type="text" name="email" />
    <input type="submit" value="Register!" />
</form>
```

The name attribute tells you the array key to use to access the information submitted with the form. For example, you use the $_POST['username'] variable in your script to access the “username” field.

Open test.php to build a simple form-handling script:

```
<?php

    /*
     * Checks if the form was submitted
     */
    if($_SERVER['REQUEST_METHOD'] == 'POST') {
        // Displays the submitted information
        echo "Thanks for registering! <br />",
        "Username: ", htmlentities($_POST['username']), "<br />",
        "Email: ", htmlentities($_POST['email']), "<br />";
    } else {
    // If the form was not submitted, displays the form

?>
```

```
<form action="test.php" method="post">
    <label for="username">Username:</label>
    <input type="text" name="username" />
    <label for="email">Email:</label>
    <input type="text" name="email" />
    <input type="submit" value="Register!" />
</form>

<?php } // End else statement ?>
```

When you load `http://localhost/simple_blog/test.php`, the script checks whether any data has been submitted via the `POST` method. If so, the script thanks the user for registering and the displays the selected username and supplied email address. If nothing is submitted via `POST`, the script displays the registration form (see Figure 3-4).

Figure 3-4. *Your registration form*

Set the action attribute set to `test.php`; doing so prompts the form to send the request to the script when the form is submitted. This enables you to both accept user input and process it within the same script.

Use the `$_SERVER` superglobal to check the `REQUEST_METHOD` value for information on whether a `POST` request has been made. If so, you perform basic escaping of the input (using `htmlentities()`) to filter the input.

Entering a name and email address causes your confirmation message to display (see Figure 3-5).

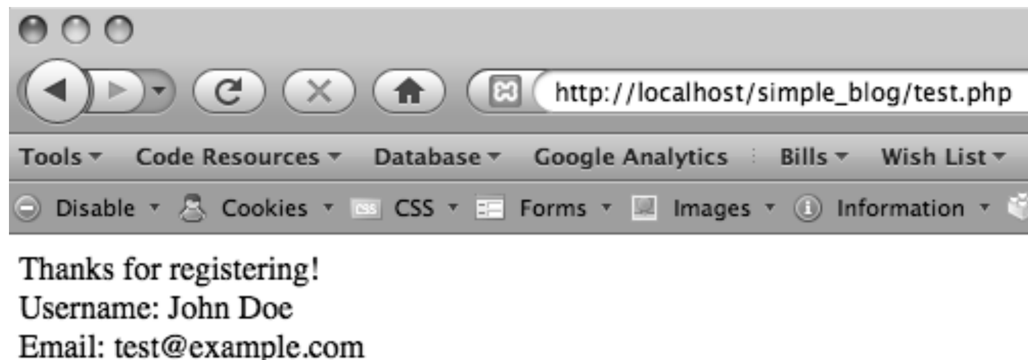


Figure 3-5. *This confirmation message is displayed after you use your registration form*

$_REQUEST

The `$_REQUEST` superglobal is an array that contains the contents of the `$_GET`, `$_POST`, and `$_COOKIE` superglobals. If you modify your form in `test.php` to submit to `test.php?submit=true` with a `username` and `email` text input, you can access all the submitted data using the following code:

```
<?php

    // Checks if the form was submitted
    if($_SERVER['REQUEST_METHOD'] == 'POST') {

        // Output the contents of $_REQUEST
        foreach($_REQUEST as $key => $val) {
            echo $key, " : ", $val, "<br />";
        }

    } else {
    // If the form was not submitted, displays the form HTML

?>

<form action="test.php?submit=true" method="post">
    <label for="username">Username:</label>
    <input type="text" name="username" />
    <label for="email">Email:</label>
    <input type="text" name="email" />
    <input type="submit" name="submit" value="Register!" />
</form>

<?php } // End else statement ?>
```

This script's output looks something like this:

```
submit : true
username : Test
email : test@example.com
```

The precedence of values is important in this case. Declaring an index in `$_COOKIE`, `$_POST`, and `$_GET` with the same name could cause confusion for new developers. `$_REQUEST` causes `$_COOKIE` to override both `$_POST` and `$_GET`. If `$_COOKIE` does not contain the variable name, `$_POST` will override `$_GET`. For example, if you set the `action` attribute to `test.php?username=agetvariable`, you won't see any change in the username value because `$_POST` will override `$_GET`, even if the `username` field is left blank.

$_FILES

Another feature of HTML forms is the ability to allow users to upload files. In an application such as the blog, you need to be able to accept images to include with your entries. In order to access an uploaded file, you need to use the `$_FILES` superglobal.

`$_FILES` works a little differently from `$_POST` and `$_GET` in that each file creates an array of related elements that provide information about the uploaded file. The provided information includes:

- `name`: The file name
- `type`: The file type (e.g. `image/jpeg`)
- `tmp_name`: The temporary location of the uploaded file
- `error`: An error code corresponding to error type (`0` if no errors found)
- `size`: The file size in bytes (e.g. `9347012`)

You store each file uploaded as a multidimensional array in the `$_FILES` superglobal, which you access using first the field name as the array key and then the name of the desired field value (*i.e.*, `$_FILES['upload1']['name']`).

You can experiment with `$_FILES` by creating a file upload field in `test.php`. Remember to add `enctype="multipart/form-data"` to the `<form>` tag to allow file uploads:

```
<?php

    // Checks if the form was submitted
    if($_SERVER['REQUEST_METHOD'] == 'POST') {

        // Checks if a file was uploaded without errors
        if(isset($_FILES['photo'])
        && is_uploaded_file($_FILES['photo']['tmp_name'])
```

```
        && $_FILES['photo']['error']==UPLOAD_ERR_OK) {

            // Outputs the contents of $_FILES
            foreach($_FILES['photo'] as $key => $value) {
                echo "$key : $value <br />";
            }
        } else {
            echo "No file uploaded!";
        }
    } else {
    // If the form was not submitted, displays the form HTML

?>

<form action="test.php" method="post"
        enctype="multipart/form-data">
    <label for="photo">User Photo:</label>
    <input type="file" name="photo" />
    <input type="submit" value="Upload a Photo" />
</form>

<?php } // End else statement ?>
```

Running this script in `test.php` displays the screen shown in Figure 3-6.

Figure 3-6. *Creating a file upload form*

Uploading a file creates output similar to the following:

```
name : sample_image.jpg
type : image/jpeg
tmp_name : /private/var/tmp/phpU6q76q
error : 0
size : 226452
```

Using an uploaded file is a little trickier—Chapter 8 explains how you do this in significantly more detail. However, you can add this script to `test.php` to see a very basic example of using an uploaded file:

```
<?php

    // Checks if the form was submitted
    if($_SERVER['REQUEST_METHOD'] == 'POST') {

        // Checks if a file was uploaded without errors
        if(isset($_FILES['photo'])
        && is_uploaded_file($_FILES['photo']['tmp_name'])

        && $_FILES['photo']['error']==UPLOAD_ERR_OK) {

            // Checks if the file is a JPG image
            if($_FILES['photo']['type']=='image/jpeg') {
                $tmp_img = $_FILES['photo']['tmp_name'];

                // Creates an image resource
                $image = imagecreatefromjpeg($tmp_img);

                // Tells the browser what type of file
                header('Content-Type: image/jpeg');

                // Outputs the file to the browser
                imagejpeg($image, '', 90);

                // Frees the memory used for the file
                imagedestroy($image);
            } else {
                echo "Uploaded file was not a JPG image.";
            }
        } else {
            echo "No photo uploaded!";
        }
    } else {
    // If the form was not submitted, displays the form HTML

?>
```

```
<form action="test.php" method="post"

        enctype="multipart/form-data">
    <label for="photo">User Photo:</label>
    <input type="file" name="photo" />
    <input type="submit" value="Upload a Photo" />
</form>

<?php } // End else statement ?>
```

Once a user selects a JPG file and clicks the “Upload a Photo” option, your script implements a series of actions:

1. It verifies that the form was submitted using the `POST` method
    a. If not, it displays the form and no further processing is performed
    b. If so, it proceeds to Step 2
2. It checks whether a file was uploaded and that the file has no errors
    a. If both conditions are not met, the script kicks out an error message
    b. If both conditions are met, it proceeds to Step 3
3. It verifies that the uploaded file is a JPG image
    a. If not, the script generates an error message
    b. If so, it proceeds to Step 4
4. It uses `imagecreatefromjpeg()` to create an image resource from the temporary file
5. It sends a `content-type` header to the browser, so the image is handled properly
6. It generates the image using `imagejpeg()`
7. It uses `imagedestroy()` to free the memory consumed by the image resource

After the file is uploaded, the script displays the image in the browser (see Figure 3-7).

Figure 3-7. *The image displayed by* `test.php` *after uploading and processing*

■ **Note** You can find a more in-depth explanation of PHP's image-handling functions in Chapter 8.

$_SESSION

In cases where you need to store a value for a user's entire visit, the `$_SESSION` superglobal provides a practical, easy solution. When a `$_SESSION` variable is declared, it remains in memory until it is explicitly unset, the session times out (the default time-out value is 180 minutes), or the browser is closed.

One common use of `$_SESSION` variables is to store a user's login status. You can build on your simple registration form to create a session for a user that has already registered.

Your script will accomplish the following tasks:

1. Display a registration form
2. Thank user for registering after she does so
3. Tell the user she has already registered on consequent page loads

You must initiate session data before you can use session variables; you do this by calling `session_start()` at the top of your script.

■ **Note** If you call `session_start()` more than once, PHP generates a notice and additional `session_start()` calls will be ignored. Also, you should call `session_start()` at the very beginning of any script you use it in, before you send any output to the browser. Failure to do so will also generate a notice and prevents the session from starting. Finally, the script that starts the session must be the first thing in the file, so make sure you place the opening `<?php` on line 1 of the file.

You can implement a `$_SESSION` variable by inserting the following code in `test.php`:

```
<?php

    // Initialize session data
    session_start();

    /*
     * If the user is already registered, display a
     * message letting them know.
     */
    if(isset($_SESSION['username'])) {
        echo "You're already registered as $_SESSION[username].";
    }

    // Checks if the form was submitted
    else if($_SERVER['REQUEST_METHOD'] == 'POST') {

        /*
         * If both the username and email fields were filled
         * out, save the username in a session variable and
         * output a thank you message to the browser. To
         * eliminate leading and trailing whitespace, we use the
         * trim() function.
         */
        if(!empty(trim($_POST['username']))
        && !empty(trim($_POST['email']))) {
```

```
            // Store escaped $_POST values in variables
            $uname = htmlentities($_POST['username']);
            $email = htmlentities($_POST['email']);

            $_SESSION['username'] = $uname;

            echo "Thanks for registering! <br />",
                "Username: $uname <br />",
                "Email: $email <br />";
        }

        /*
         * If the user did not fill out both fields, display
         * a message letting them know that both fields are
         * required for registration.
         */
        else {
            echo "Please fill out both fields! <br />";
        }
    }

    // If the form was not submitted, displays the form HTML
    else {

?>

<form action="test.php?username=overwritten" method="post">
    <label for="username">Username:</label>
    <input type="text" name="username" />
    <label for="email">Email:</label>
    <input type="text" name="email" />
    <input type="submit" value="Register!" />
</form>

<?php } // End else statement ?>
```

You should see your registration form after you load `test.php` in your browser, and your script will display the following message after you fill in the fields:

```
Thanks for registering!
Username: Test User
Email: test@example.com
```

Reloading the page should display this message:

```
You're already registered as Test User.
```

You use unset() to destroy a $_SESSION variable. Removing the username session variable and showing the registration form again is as simple as inserting the following code:

```
unset($_SESSION['username']);
```

Using session_destroy()

PHP provides a function called session_destroy(), the name of which can be somewhat misleading. When you call this function, a session will be destroyed the next time the page loads. However, the session variables remain available with the same script. Insert this code in test.php to see this behavior in action:

```
<?php

error_reporting(E_ALL);

session_start();

// Create a session variable
$_SESSION['test'] = "A value.";

// Destroy the session
session_destroy();

// Attempt to output the session variable (output: A value.)
echo $_SESSION['test'];

// Unset the variable specifically
unset($_SESSION['test']);

// Attempt to output the session variable (generates a notice)
echo $_SESSION['test'];

?>
```

Loading this script in a browser generates the following output:

```
A value.
Notice: Undefined index: test in /Applications/XAMPP/xamppfiles/htdocs/simple_blog/test.php on
line 20
```

This behavior can be confusing, so you should specifically `unset()` session variables that need you want destroyed before the end of the script to avoid unexpected behavior.

$_COOKIE

Cookies behave similarly to sessions, but they allow you to store information on a user's machine for a longer period of time. Information stored in cookies remains available even after the user closes her browser, assuming that the expiration date on the cookie is set far enough in the future.

You can use cookies to make a user's repeat visits more pleasant by retaining pertinent, non-sensitive data. This can include settings that customize the user's experience on your site, perform repetitive tasks automatically, or allow a user to stay logged in and avoid the extra step of logging in each time she visits.

In PHP, you use the `$_COOKIE` superglobal to access cookie values. However, setting a cookie requires that you use the `setcookie()` function. Open `test.php` again and set a cookie that contains your username and greet you whenever you load the page.

■ **Note** This snippet uses the `time()` function. The returned value from `time()` is the number of seconds since the Unix Epoch[1], which is commonly used in programming to determine dates and intervals.

```
<?php

    /*
     * If the user is returning to this page
     * after previously registering, use the
     * cookie to welcome them back.
     */
    if(isset($_COOKIE['username'])) {
        echo "Welcome back, ",
            htmlentities($_COOKIE['username']),
            "! <br />";
    }

    /*
     * If the user is new and submits the
     * registration form, set a cookie with
     * their username and display a thank
     * you message.
     */
```

1. The Unix Epoch was midnight Coordinated Universal Time (UTC) of January 1, 1970. It serves as a measuring stick for determining time when dealing with UNIX systems.

```
  else if($_SERVER['REQUEST_METHOD']=='POST'
            && !empty($_POST['username'])) {

        // Sanitize the input and store in a variable
        $uname = htmlentities($_POST['username']);

        // Set a cookie that expires in one week
        $expires = time()+7*24*60*60;
        setcookie('username', $uname, $expires, '/');

        // Output a thank you message
        echo "Thanks for registering, $uname! <br />";
    }

    /*
     * If the user has neither previously registered
     * or filled out the registration form, show the
     * registration form.
     */
    else {

?>

<form method="post">
    <label for="username">Username:</label>
    <input type="text" name="username" />
    <input type="submit" value="Register" />
</form>

<?php } // End else statement ?>
```

As in this book's earlier examples, a user will see the registration form the first time he navigates to `test.php`. After filling out and submitting the form, he will see the following message, assuming he entered the username, "Jason":

```
Thanks for registering, Jason!
```

Subsequent visits will load the `$_COOKIE` variable and display this message to the user:

```
Welcome back, Jason!
```

■**Note** Browsers allow users to refuse cookies, so it's a good idea not to use them for vital parts of your application. When testing a new application that uses cookies, you should turn cookies off in your browser and make sure that a user can still get the benefit of your application without them.

Summary

At the end of this chapter, you should feel comfortable using superglobal arrays to access data sent from one page to the next. You should also know the difference between sessions and cookies, as well as how to use superglobal arrays to access the values stored in them.

In the next chapter, I'll introduce you to MySQL and explain how to interact with database tables to store, retrieve, modify, and delete data, which form a core part of the blog's functionality.