A Pain-Free Introduction to Building Interactive Web Sites

Learning PHP 5

O'REILLY®

David Sklar

CHAPTER 8

Remembering Users with Cookies and Sessions

A web server is a lot like a clerk at a busy deli full of pushy customers. The customers at the deli shout requests: "I want a half pound of corned beef!" and "Give me a pound of pastrami, sliced thin!" The clerk scurries around slicing and wrapping to satisfy the requests. Web clients electronically shout requests ("Give me */catalog/yak.php*!" or "Here's a form submission for you!"), and the server, with the PHP interpreter's help, electronically scurries around constructing responses to satisfy the requests.

The clerk has an advantage that the web server doesn't, though: a memory. She naturally ties together all the requests that come from a particular customer. The PHP interpreter and the web server can't do that without some extra steps. That's where *cookies* come in.

A cookie identifies a particular web client to the web server and to the PHP interpreter. Each time a web client makes a request, it sends the cookie along with the request. The interpreter reads the cookie and figures out that a particular request is coming from the same web client that made previous requests, which were accompanied by the same cookie.

If deli customers were faced with a memory-deprived clerk, they'd have to adopt the same strategy. Their requests for service would look like this:

```
"I'm customer 56 and I want a half-pound of corned beef."
"I'm customer 29 and I want three knishes."
"I'm customer 56 and I want two pounds of pastrami."
"I'm customer 77 and I'm returning this rye bread -- it's stale."
"I'm customer 29 and I want a salami."
```

The "I'm customer so-and-so" part of the requests is the cookie. It gives the clerk what she needs to be able to link a particular customer's requests together.

A cookie has a name (such as "customer") and a value (such as "77" or "ronald"). "Working with Cookies," next, shows you how to work with individual cookies in your programs: setting them, reading them, and deleting them.

One cookie is best at keeping track of one piece of information. Often, you need to keep track of more about a user (such as the contents of their shopping cart). Using multiple cookies for this is cumbersome. PHP's *session* capabilities solve this problem.

A session uses a cookie to distinguish users from each other and makes it easy to keep a temporary pile of data for each user on the server. This data persists across requests. On one request, you can add a variable to a user's session (such as putting something into the shopping cart). On a subsequent request, you can retrieve what's in the session (such as on the order checkout page when you need to list everything in the cart). Later in this chapter, "Activating Sessions" describes how to get started with sessions, and "Storing and Retrieving Information" provides the details on working with sessions.

Working with Cookies

To set a cookie, use the `setcookie()` function. This tells a web client to remember a cookie name and value and send them back to the server on subsequent requests. Example 8-1 sets a cookie named `userid` to value `ralph`.

Example 8-1. Setting a cookie

```
setcookie('userid','ralph');
```

To read a previously set cookie from your PHP program, use the `$_COOKIE` auto-global array. Example 8-2 prints the value of the `userid` cookie.

Example 8-2. Printing a cookie value

```
print 'Hello, ' . $_COOKIE['userid'];
```

The value for a cookie that you provide to `setcookie()` can be a string or a number. It can't be an array or more complicated data structure.

When you call `setcookie()`, the response that the PHP interpreter generates to send back to the web client includes a special header that tells the web client about the new cookie. On subsequent requests, the web client sends that cookie name and value back to the server. This two-step conversation is illustrated in Figure 8-1.

Usually, you must call `setcookie()` before the page generates any output. This means that `setcookie()` must come before any `print` statements. It also means that there can't be any text before the PHP `<?php` start tag in the page that comes before the `setcookie()` function. Later in this chapter, "Why setcookie() and session_start() Want to Be at the Top of the Page" explains why this requirement exists, and how, in some cases, you can get around it.

Example 8-3 shows the correct way to put a `setcookie()` call at the top of your page.

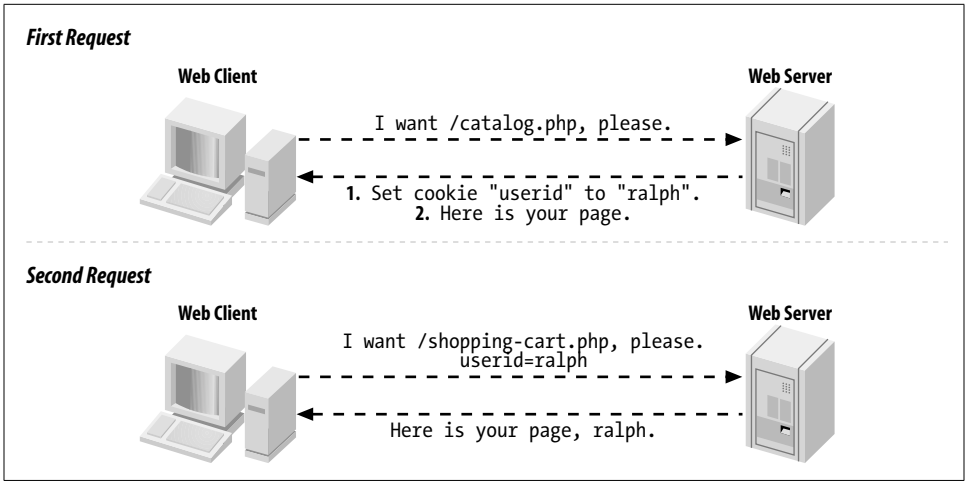

Figure 8-1. Client and server communication when setting a cookie

Example 8-3. Starting a page with setcookie()

```
<?php
setcookie('userid','ralph');
?>
<html><head><title>Page with cookies</title><head>
<body>
This page sets a cookie properly, because the PHP block
with setcookie( ) in it comes before all of the HTML.
</body></html>
```

Cookies show up in `$_COOKIE` only when the web client sends them along with the request. This means that a name and value do not appear in `$_COOKIE` immediately after you call `setcookie( )`. Only after that cookie-setting response is digested by the web client does the client know about the cookie. And only after the client sends the cookie back on a subsequent request does it appear in `$_COOKIE`.

The default lifetime for a cookie is the lifetime of the web client. When you quit Internet Explorer or Mozilla, the cookie is deleted. To make a cookie live longer (or shorter), use the third argument to `setcookie( )`. This is an optional cookie expiration time. Example 8-4 shows some cookies with different expiration times.

Example 8-4. Setting cookie expiration

```
// The cookie expires one hour from now
setcookie('short-userid','ralph',time( ) + 60*60);

// The cookie expires one day from now
setcookie('longer-userid','ralph',time( ) + 60*60*24);

// The cookie expires at noon on October 1, 2006
setcookie('much-longer-userid','ralph',mktime(12,0,0,10,1,2006));
```

The cookie expiration time needs to be given to `setcookie()` expressed as the number of seconds elapsed since midnight on January 1, 1970. (As crazily arbitrary as that sounds, there are some good reasons for expressing time values that way, which are explained in Chapter 9.)

Two functions make coming up with appropriate expiration times easier: `time()` and `mktime()`. The `time()` function returns the current number of elapsed seconds since January 1, 1970. So if you want the cookie expiration time to be a certain number of seconds from now, add that value to what `time()` returns. There are 60 seconds in a minute and 60 minutes in an hour, so 60*60 is the number of seconds in an hour. That makes `time()` + 60*60 equal to the "elapsed seconds" value for an hour from now. Similarly, 60*60*24 is the number of seconds in a day, so `time()` + 60*60*24 is the "elapsed seconds" value for a day from now.

The `mktime()` function computes an appropriate "elapsed seconds" value for a given date and time. The arguments to `mktime()` are hour, minute, second, month, day, and year. So, `mktime(12,0,0,10,1,2006)` returns the correct value for noon (hour: 12, minute: 0, second: 0), on October 1, 2006 (month: 10, day: 1, year: 2006).

Setting a cookie with a specific expiration time makes the cookie last even if the web client exits and restarts.

Aside from expiration time, there are two other cookie parameters that are helpful to adjust: the path and the domain. Each of these affect with what requests the web client sends back the cookie.

Normally, cookies are only sent back with requests for pages in the same directory (or below) as the page that set the cookie. A cookie set by *http://www.example.com/buy.php* is sent back with all requests to `www.example.com`, because *buy.php* is in the top-level directory of the web server. A cookie set by *http://www.example.com/catalog/list.php* is sent back with other requests in the `catalog` directory, such as *http://www.example.com/catalog/search.php*. It is also sent back with requests for pages in subdirectories of `catalog`, such as *http://www.example.com/catalog/detailed/search.php*. But it is not sent back with requests for pages above or outside the `catalog` directory such as *http://www.example.com/sell.php* or *http://www.example.com/users/profile.php*.

The part of the URL after the hostname (such as */buy.php*, */catalog/list.php*, or */users/profile.php*) is called the *path*. To tell the web client to match against a different path when determining whether to send a cookie to the server, provide that path as the fourth argument to `setcookie()`. The most flexible path to provide is `/`, which means "send this cookie back with all requests to the server." Example 8-5 sets a cookie with the path set to `/`.

Example 8-5. Setting the cookie path

```
setcookie('short-userid','ralph',0,'/');
```

In Example 8-5, the expiration time argument to `setcookie( )` is 0. This tells `setcookie( )` to use the default expiration time (when the web client exits) for the cookie. When you specify a path to `setcookie( )`, you have to fill in something for the expiration time argument. It can be a specific time value (such as `time( ) + 60*60`), or it can be 0 to use the default expiration time.

Setting the path to something other than `/` is a good idea if you are on a shared server and all of your pages are under a specific directory. For example, if your web space is under `http://students.example.edu/~alice/`, then you should set the cookie path to `/~alice/`, as shown in Example 8-6.

Example 8-6. Setting the cookie path to a specific directory

```
setcookie('short-userid','ralph',0,'/~alice/');
```

With a cookie path of `/~alice/`, the `short-userid` cookie is sent with a request to *http://students.example.edu/~alice/search.php*, but not with requests to other students' web pages such as *http://students.example.edu/~bob/sneaky.php* or *http://students.example.edu/~charlie/search.php*.

The last argument that affects which requests the web client decides to send a particular cookie with is the domain. The default behavior is to send cookies only with requests to the same host that set the cookie. If *http://www.example.com/login.php* set a cookie, then that cookie is sent back with other requests to `www.example.com`—not with requests to `shop.example.com`, `www.yahoo.com`, or `www.example.org`.

You can alter this behavior slightly. A fifth argument to `setcookie( )` tells the web client to send the cookie with requests that have a hostname whose end matches the argument. The most common use of this feature is to set the cookie domain to something like `.example.com`. (The period at the beginning is important.) This tells the web client that the cookie should accompany future requests to the servers `www.example.com`, `shop.example.com`, `testing.development.example.com`, and any other server name that ends in `.example.com`. Example 8-7 shows how to set a cookie like this.

Example 8-7. Setting the cookie domain

```
setcookie('short-userid','ralph',0,'/','.example.com');
```

The cookie in Example 8-7 expires when the web client exits and is sent with requests in any directory (because the path is `/`) on any server that ends with `.example.com`.

The path that you provide to `setcookie( )` must match the end of the name of your server. If your PHP programs are hosted on the server `students.example.edu`, you can't supply `.yahoo.com` as a cookie path and have the cookie you set sent back to all servers in the `yahoo.com` domain. You can, however, specify `.example.edu` as a cookie domain to have your cookie sent with all requests to any server in the `example.edu` domain.

To delete a cookie, call `setcookie()` with the name of the cookie you want to delete and the empty string as the cookie value, as shown in Example 8-8.

Example 8-8. Deleting a cookie

```
setcookie('short-userid','');
```

If you've set a cookie with nondefault values for an expiration time, path, or domain, you must provide those same values again when you delete the cookie for the cookie to be deleted properly.

Most of the time, any cookies you set are fine with the default values for expiration time, path, or domain. But understanding how these values can be changed helps you understand how PHP's sessions behavior can be customized.

Activating Sessions

Sessions use a cookie called `PHPSESSID`. When you start a session on a page, the PHP interpreter checks for the presence of this cookie and sets it if it doesn't exist. The value of the `PHPSESSID` cookie is a random alphanumeric string. Each web client gets a different session ID. The session ID in the `PHPSESSID` cookie identifies that web client uniquely to the server. That lets the interpreter maintain separate piles of data for each web client.

The conversation between the web client and the server when starting up a session is illustrated in Figure 8-2.

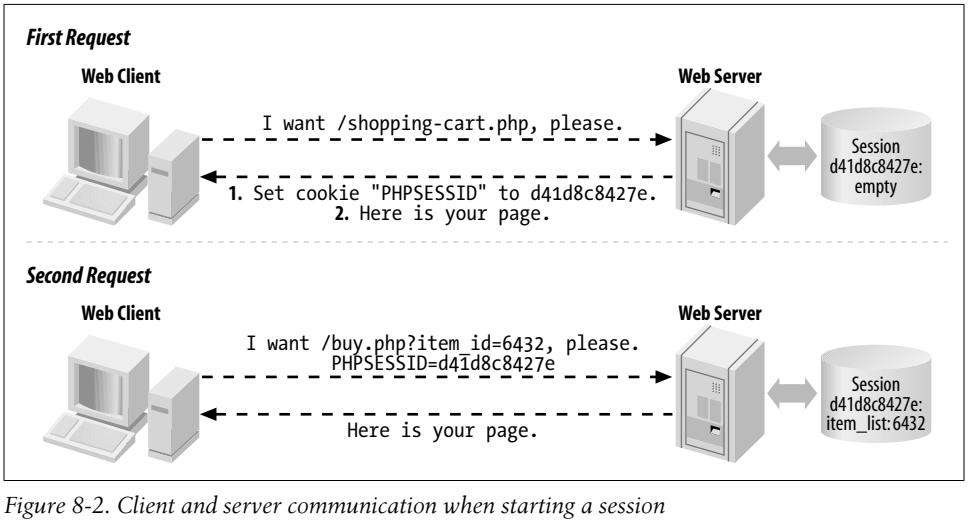

Figure 8-2. Client and server communication when starting a session

To use a session in a page, call `session_start()` at the beginning of your script. Like `setcookie()`, this function must be called before any output is sent. If you want to

use sessions in all your pages, set the configuration directive `session.auto_start` to `On`. (Appendix A explains how to change configuration settings.) Once you do that, there's no need to call `session_start( )` in each page.

Storing and Retrieving Information

Session data is stored in the `$_SESSION` auto-global array. Read and change elements of that array to manipulate the session data. Example 8-9 shows a page counter that uses the `$_SESSION` array to keep track of how many times a user has looked at the page.

Example 8-9. Counting page accesses with a session

```
session_start( );

$_SESSION['count'] = $_SESSION['count'] + 1;

print "You've looked at this page " . $_SESSION['count'] . ' times.';
```

The first time a user accesses the page in Example 8-9, no `PHPSESSID` cookie is sent by the user's web client to the server. The `session_start( )` function creates a new session for the user and sends a `PHPSESSID` cookie with the new session ID in it. When the session is created, the `$_SESSION` array starts out empty. So, `$_SESSION['count'] = $_SESSION['count'] + 1` sets `$_SESSION['count']` to 1. The `print` statement outputs:

```
You've looked at this page 1 times.
```

At the end of the request, the information in `$_SESSION` is saved into a file on the web server associated with the appropriate session ID.

The next time the user accesses the page, the web client sends the `PHPSESSID` cookie. The `session_start( )` function sees the session ID in the cookie and loads the file that contains the saved session information associated with that session ID. In this case, that saved information just says that `$_SESSION['count']` is 1. Next, `$_SESSION['count']` is incremented to 2 and `You've looked at this page 2 times.` is printed. Again, at the end of the request, the contents of `$_SESSION` (now with `$_SESSION['count']` equal to 2) are saved to a file.

The PHP interpreter keeps track of the contents of `$_SESSION` separately for each session ID. When your program is running, `$_SESSION` contains the saved data for one session only—the active session corresponding to the ID that was sent in the `PHPSESSID` cookie. Each user's `PHPSESSID` cookie has a different value.

As long as you call `session_start( )` at the top of a page (or if `session.auto_start` is on), you have access to a user's session data in your page. The `$_SESSION` array is a way of sharing information between pages.

Example 8-10 is a complete program that displays a form in which a user picks a dish and a quantity. That dish and quantity are added to the session variable `order`.

Example 8-10. Saving form data in a session

```
<?php
require 'formhelpers.php';

session_start( );

$main_dishes = array('cuke' => 'Braised Sea Cucumber',
                     'stomach' => "Sauteed Pig's Stomach",
                     'tripe' => 'Sauteed Tripe with Wine Sauce',
                     'taro' => 'Stewed Pork with Taro',
                     'giblets' => 'Baked Giblets with Salt',
                     'abalone' => 'Abalone with Marrow and Duck Feet');

if ($_POST['_submit_check']) {
    if ($form_errors = validate_form( )) {
        show_form($form_errors);
    } else {
        process_form( );
    }
} else {
    show_form( );
}

function show_form($errors = '') {
    print '<form method="POST" action="'.$_SERVER['PHP_SELF'].'">';

    if ($errors) {
        print '<ul><li>';
        print implode('</li><li>',$errors);
        print '</li></ul>';
    }
    // Since we're not supplying any defaults of our own, it's OK
    // to pass $_POST as the defaults array to input_select and
    // input_text so that any user-entered values are preserved
    print 'Dish: ';
    input_select('dish', $_POST, $GLOBALS['main_dishes']);
    print '<br/>';

    print 'Quantity: ';
    input_text('quantity', $_POST);
    print '<br/>';

    input_submit('submit','Order');

    print '<input type="hidden" name="_submit_check" value="1"/>';
    print '</form>';
}
```

Example 8-10. Saving form data in a session (continued)

```
function validate_form( ) {
    $errors = array( );

    // The dish selected in the menu must be valid
    if (! array_key_exists($_POST['dish'], $GLOBALS['main_dishes'])) {
        $errors[ ] = 'Please select a valid dish.';
    }

    if ((! is_numeric($_POST['quantity'])) || (intval($_POST['quantity']) <= 0)) {
        $errors[ ] = 'Please enter a quantity.';
    }

    return $errors;
}

function process_form( ) {
    $_SESSION['order'][ ] = array('dish'     => $_POST['dish'],
                                  'quantity' => $_POST['quantity']);

    print 'Thank you for your order.';
} ?>
```

The form-handling code in Example 8-10 is mostly familiar. As in Examples 7-30 and 7-56, the form element printing helper functions are loaded from the *formhelpers.php* file. The `show_form( )`, `validate_form( )`, and `process_form( )` functions display, validate, and process the form data.

Where Example 8-10 takes advantage of sessions, however, is in `process_form( )`. Each time the form is submitted with valid data, an element is added to the `$_SESSION['order']` array. Session data isn't restricted to strings and numbers such as cookies. You can treat `$_SESSION` like any other array. The syntax `$_SESSION['order'][ ]` says "treat `$_SESSION['order']` as an array and add a new element onto its end." In this case, what's being added on to the end of `$_SESSION['order']` is a two-element array containing information about the dish and quantity that were submitted in the form.

The program in Example 8-11 prints a list of dishes that have been ordered by accessing the information that's been stored in the session by Example 8-10.

Example 8-11. Printing session data

```
<?php
session_start( );

$main_dishes = array('cuke' => 'Braised Sea Cucumber',
                     'stomach' => "Sauteed Pig's Stomach",
                     'tripe' => 'Sauteed Tripe with Wine Sauce',
                     'taro' => 'Stewed Pork with Taro',
                     'giblets' => 'Baked Giblets with Salt',
                     'abalone' => 'Abalone with Marrow and Duck Feet');
```

Example 8-11. Printing session data (continued)

```
if (count($_SESSION['order']) > 0) {
    print '<ul>';
    foreach ($_SESSION['order'] as $order) {
        $dish_name = $main_dishes[ $order['dish'] ];
        print "<li> $order[quantity] of $dish_name </li>";
    }
    print "</ul>";
} else {
    print "You haven't ordered anything.";
}
?>
```

Example 8-11 has access to the data stored in the session by Example 8-10. It treats `$_SESSION['order']` as an array: if there are elements in the array (because `count()` returns a positive number), then it iterates through the array with `foreach()` and prints out a list element for each dish that has been ordered.

Configuring Sessions

Sessions work great with no additional tweaking. Turn them on with the `session_start()` function or the `session.auto_start` configuration directive, and the `$_SESSION` array is there for your enjoyment. However, if you're more particular about how you want sessions to function, there are a few helpful settings that can be changed.

Session data sticks around as long as the session is accessed at least once every 24 minutes. This is fine for most applications. Sessions aren't meant to be a permanent data store for user information—that's what the database is for. Sessions are for keeping track of recent user activity to make their browsing experience smoother.

Some situations may need a shorter session length, however. If you're developing a financial application, you may want to allow only 5 or 10 minutes of idle time to reduce the chance that an unattended computer can be used by an unauthorized person. If your application doesn't work with very critical data and you have easily distracted users, you may want to set the session length to longer than 24 minutes.

The `session.gc_maxlifetime` configuration directive controls how much idle time is allowed between requests to keep a session active. It's default value is 1,440—there are 1,440 seconds in 24 minutes. You can change `session.gc_maxlifetime` in your server configuration or by calling the `ini_set()` function from your program. If you use `ini_set()`, you must call it before `session_start()`. Example 8-12 shows how to use `ini_set()` to change the allowable session idle time to 10 minutes.

Example 8-12. Changing allowable session idle time

```
<?php
ini_set('session.gc_maxlifetime',600'); // 600 seconds == ten minutes
session_start( );
?>
```

Expired sessions don't actually get wiped out instantly after 24 minutes elapses. Here's how it really works: at the beginning of any request that uses sessions (because the page calls `session_start( )` or `session.auto_start` is on), there is a 1% chance that the PHP interpreter scans through all of the sessions on the server and deletes any that are expired. "A 1% chance" sounds awfully unpredictable for a computer program. It is. But that randomness makes things more efficient. On a busy site, searching for expired sessions to destroy at the beginning of every request would consume too much server power.

You're not stuck with that 1% chance if you'd like expired sessions to be removed more promptly. The `session.gc_probability` configuration directive is the percent chance that the "erase old sessions" routine runs at the start of a request. To have that happen on every request, set it to 100. Like with `session.gc_maxlifetime`, if you use `ini_set( )` to change the value of `session.gc_probability`, you need to do it before `session_start( )`. Example 8-13 shows how to change `session.gc_probability` with `ini_set( )`.

Example 8-13. Changing the expired session cleanup probability

```
<?php
ini_set('session.gc_probability',100); // 100% : clean up on every request
session_start( );
?>
```

If you are activating sessions with the `session.auto_start` configuration directive and you want to change the value of `session.gc_maxlifetime` or `session.gc_probability`, you can't use `ini_set( )` to change those values—you have to do it in your server configuration.

Login and User Identification

A session establishes an anonymous relationship with a particular user. Requiring a user to log in to your web site lets them tell you who they are. The login process typically requires a user to provide you with two pieces of information: one that identifies them (a username or an email address) and one that proves that they are who they say they are (a secret password).

Once a user is logged in, they can access private data, submit message board posts with their name attached, or do anything else that the general public isn't allowed to do.

Adding user login on top of sessions has five parts:

- Displaying a form asking for username and password
- Checking the form submission
- Adding the username to the session (if the submitted password is correct)
- Looking for the username in the session to do user-specific tasks
- Removing the username from the session when the user logs out

The first three steps are handled in the context of regular form processing. The `validate_form( )` function gets the responsibility of checking to make sure that the supplied username and password are acceptable. The `process_form( )` function adds the username to the session. Example 8-14 displays a login form and adds the username to the session if the login is successful.

Example 8-14. Displaying a login form

```
<?php
require 'formhelpers.php';

// This is identical to the input_text( ) function in formhelpers.php but
// prints a password box (in which asterisks obscure what's entered)
// instead of a plain text field
function input_password($field_name, $values) {
    print '<input type="password" name="' . $field_name .'" value="';
    print htmlentities($values[$field_name]) . '">';
}

session_start( );

if ($_POST['_submit_check']) {
    if ($form_errors = validate_form( )) {
        show_form($form_errors);
    } else {
        process_form( );
    }
} else {
    show_form( );
}

function show_form($errors = '') {
    print '<form method="POST" action="'.$_SERVER['PHP_SELF'].'">';

    if ($errors) {
        print '<ul><li>';
        print implode('</li><li>',$errors);
        print '</li></ul>';
    }
    print 'Username: ';
    input_text('username', $_POST);
    print '<br/>';
```

Example 8-14. Displaying a login form (continued)

```
    print 'Password: ';
    input_password('password', $_POST);
    print '<br/>';

    input_submit('submit','Log In');

    print '<input type="hidden" name="_submit_check" value="1"/>';
    print '</form>';
}

function validate_form( ) {
    $errors = array( );

    // Some sample usernames and passwords
    $users = array('alice'   => 'dog123',
                   'bob'     => 'my^pwd',
                   'charlie' => '**fun**');

    // Make sure user name is valid
    if (! array_key_exists($_POST['username'], $users)) {
        $errors[ ] = 'Please enter a valid username and password.';
    }

    // See if password is correct
    $saved_password = $users[ $_POST['username'] ];
    if ($saved_password != $_POST['password']) {
        $errors[ ] = 'Please enter a valid username and password.';
    }

    return $errors;
}


function process_form( ) {
    // Add the username to the session
    $_SESSION['username'] = $_POST['username'];

    print "Welcome, $_SESSION[username]";
}
?>
```

Figure 8-3 shows the form that Example 8-14 displays, Figure 8-4 shows what happens when an incorrect password is entered, and Figure 8-5 what happens when a correct password is entered.

In Example 8-14, `validate_form( )` checks two things: whether a valid username is entered and whether the correct password was supplied for that username. Note that the same error message is added to the `$errors` array in either case. If you use different error messages for a missing username (such as "User name not found") and bad passwords (such as "Password doesn't match"), you provide helpful information for someone trying to guess a valid username and password. Once this

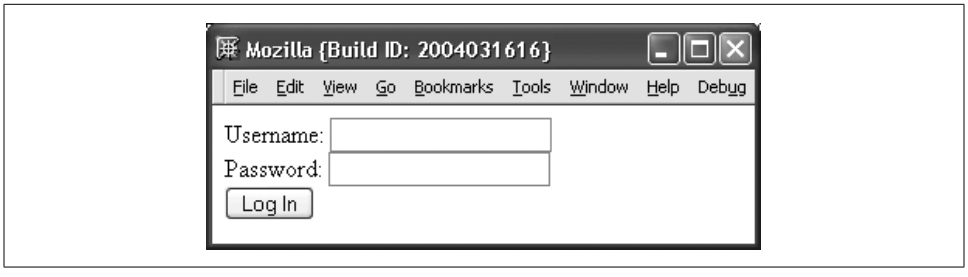

Figure 8-3. Login form

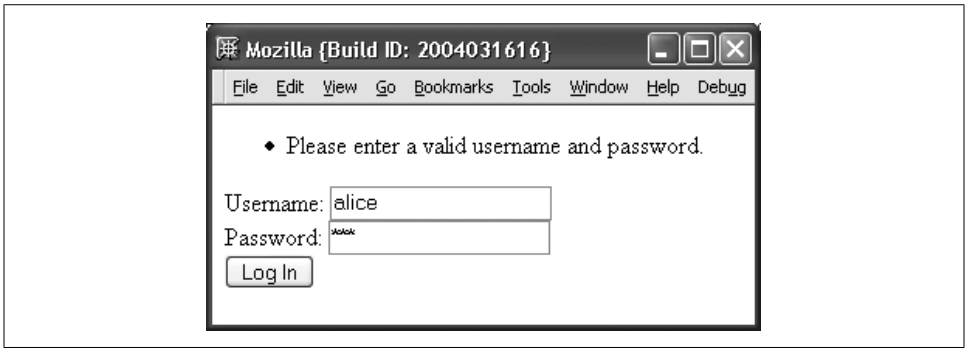

Figure 8-4. Unsuccessful login

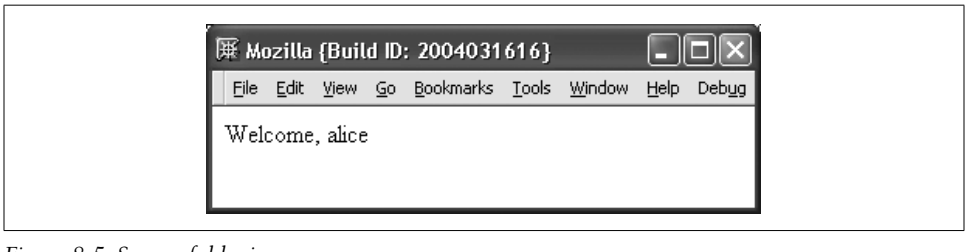

Figure 8-5. Successful login

attacker stumbles on a valid username, she sees the "Password doesn't match" error message instead of the "User name not found" message. She then knows that she's working with a real username and has to guess the password only. When the error messages are the same in both cases, all the attacker knows is that something about the username/password combination she tried is not correct.

If the username is valid and the right password is submitted, `validate_form()` returns no errors. When this happens, `process_form()` is called. The `process_form()` function adds the submitted username (`$_POST['username']`) to the session and prints out a welcome message for the user. This makes the username available in the session for other pages to use. Example 8-15 demonstrates how to check for a username in the session in another page.

Example 8-15. Doing something special for a logged-in user

```
<?php
session_start( );

if (array_key_exists('username', $_SESSION)) {
    print "Hello, $_SESSION[username].";
} else {
    print 'Howdy, stranger.';
}
?>
```

The only way a `username` element can be added to the `$_SESSION` array is by your program. So if it's there, you know that a user has logged in successfully.

The `validate_form( )` function in Example 8-14 uses a sample array of usernames and passwords called `$users`. Storing passwords without encrypting them is a bad idea. If the list of unencrypted passwords is compromised, then an attacker can log in as any user. Storing encrypted passwords prevents an attacker from getting the actual passwords even if she gets the list of encrypted passwords, because there's no way to go from the encrypted password back to the unencrypted password you'd have to enter to log in. Operating systems that require you to log in with a password use this same technique.

A better `validate_form( )` function is shown in Example 8-16. The `$users` array in that function contains passwords that have been encrypted with PHP's `crypt( )` function. Because the passwords are stored as encrypted strings, they can't be compared directly with the unencrypted password that the user enters. Instead, the submitted password in `$_POST['password']` is also encrypted with `crypt( )`, and the result is compared with the stored encrypted password. If they match, then the user has submitted the correct password.

Example 8-16. Using encrypted passwords

```
function validate_form( ) {
    $errors = array( );

    // Sample users with encrypted passwords
    $users = array('alice'   => '$1$LdBOG7jx$zVu.6YDfT2M3PcIq3xUdDO',
                   'bob'     => '$1$YY/mMevB$6KEH9LLrjZnuemGml9GRE/',
                   'charlie' => '$1$q.hxaUR9$Pu/NxLQeyMgF7lmCJ3FBo/');

     // Make sure user name is valid
    if (! array_key_exists($_POST['username'], $users)) {
        $errors[ ] = 'Please enter a valid username and password.';
    }

    // See if password is correct
    $saved_password = $users[ $_POST['username'] ];
    if ($saved_password != crypt($_POST['password'], $saved_password)) {
        $errors[ ] = 'Please enter a valid username and password.';
```

Example 8-16. Using encrypted passwords (continued)

```
    }

    return $errors;
}
```

The `crypt( )` function needs to have the stored encrypted password passed to it as a second argument to make sure that the `$_POST['password']` is encrypted properly. (If you're interested in more details about how `crypt( )` works, read about in the PHP online manual at *http://www.php.net/crypt* and in Recipe 14.5 of *PHP Cookbook*, by David Sklar and Adam Trachtenberg [O'Reilly].)

Putting an array of users and passwords inside `validate_form( )` makes these examples self contained. However, more typically, your usernames and passwords are stored in a database table. Example 8-17 is a version of `validate_form( )` that retrieves the username and encrypted password from a database. It assumes that a database connection has already been set up outside the function and is available in the global variable `$db`.

Example 8-17. Retrieving a username and password from a database

```
function validate_form( ) {
    global $db;

    $errors = array( );

    $encrypted_password = $db->getOne('SELECT password FROM users WHERE username = ?',
                                      array($_POST['username']));

    if ($encrypted_password != crypt($_POST['password'], $encrypted_password)) {
        $errors[ ] = 'Please enter a valid username and password.';
    }

    return $errors;
}
```

The query that `getOne( )` sends to the database returns the encrypted password for the user identified in `$_POST['username']`. If the username supplied in `$_POST['username']` doesn't match any rows in the database, then `$encrypted_password` is empty. Either way, `$encrypted_password` is compared to the results of encrypting the supplied password (`$_POST['password']`); if they don't match, then an error is added to the `$errors` array.

Just like any other array, use `unset( )` to remove a key and value from `$_SESSION`. This is how to log out a user. Example 8-18 shows a logout page.

Example 8-18. Logging out

```
<?php
session_start( );
unset($_SESSION['username']);

print 'Bye-bye.';
?>
```

When the `$_SESSION` array is saved at the end of the request that calls `unset( )`, the `username` element isn't included in the saved data. The next time that session's data is loaded into `$_SESSION`, there is no `username` element, and the user is once again anonymous.

Why setcookie( ) and session_start( ) Want to Be at the Top of the Page

When a web server sends a response to a web client, most of that response is the HTML document that the browser renders into a web page on your screen: the soup of tags and text that Internet Explorer or Mozilla formats into tables or changes the color or size of. But before that HTML is a section of the response that contains *headers*. These don't get displayed on your screen but are commands or information from the server for the web client. The headers say things such as "this page was generated at such-and-such a time," "please don't cache this page," or (and the one that's relevant here) "please remember that the cookie named `userid` has the value `ralph`."

All of the headers in the response from the web server to the web client have to be at the beginning of the response, before the *response body*, which is the HTML that controls what the browser actually displays. Once some of the body is sent—even one line—no more headers can be sent.

Functions such as `setcookie( )` and `session_start( )` add headers to the response. In order for the added headers to be sent properly, they must be added before any output starts. That's why they must be called before any `print` statements or any HTML appearing outside `<?php ?>` PHP tags.

If any output has been sent before `setcookie( )` or `session_start( )` is called, the PHP interpreter prints an error message that looks like this:

```
Warning: Cannot modify header information - headers already sent by
(output started at /www/htdocs/catalog.php:2) in /www/htdocs/catalog.php on line 4
```

This means that line 4 of *catalog.php* called a function that sends a header, but something was already printed by line 2 of *catalog.php*.

If you see the "headers already sent" error message, scrutinize your code for errant output. Make sure there are no print statements before you call `setcookie( )` or

`session_start()`. Check that there is nothing before the first `<?php` PHP start tag in the page. Also, check that there is nothing outside the `<?php ?>` tags in any included or required files—even blank lines.

An alternative to hunting down mischievous blank lines in your files is to use *output buffering*. This tells the PHP interpreter to wait to send *any* output until it's finished processing the whole request. Then, it sends any headers that have been set, followed by all the regular output. To enable output buffering, set the `output_buffering` configuration directive to `On` in your server configuration. Web clients have to wait a few additional milliseconds to get the page content from your server, but you save megaseconds fixing your code to have all output happen after calls to `setcookie()` or `session_start()`.

With output buffering turned on, you can mix `print` statements, cookie and session functions, HTML outside of `<?php` and `?>` tags, and regular PHP code without getting the "headers already sent" error. The program in Example 8-19 works only when output buffering is turned on. Without it, the HTML printed before the `<?php` start tag triggers the sending of headers, which prevents `setcookie()` from working properly.

Example 8-19. A program that needs output buffering to work

```
<html>
<head>Choose Your Site Version</head>
<body>
<?php
setcookie('seen_intro', 1);
?>
<a href="/basic.php">Basic</a>
 or
<a href="/advanced.php">Advanced</a>
</body>
</html>
```

Chapter Summary

Chapter 8 covers:

- Understanding why cookies are necessary to identify a particular web browser to a web server.
- Setting a cookie in a PHP program.
- Reading a cookie value in a PHP program.
- Modifying cookie parameters such as expiration time, path, and domain.
- Deleting a cookie in a PHP program.
- Turning on sessions from a PHP program or in the PHP interpreter configuration.
- Storing information in a session.

- Reading information from a session.
- Saving form data in a session.
- Removing information from a session.
- Configuring session expiration and cleanup.
- Displaying, validating, and processing a validation form.
- Using encrypted passwords.
- Understanding why `setcookie()` and `session_start()` must be called before anything is printed.

Exercises

1. Make a web page that uses a cookie to keep track of how many times a user has viewed the page. The first time a particular user looks at the page, it should print something like "Number of views: 1." The second time the user looks at the page, it should print "Number of views: 2," and so on.
2. Modify the web page from the first exercise so that it prints out a special message on the 5th, 10th, and 15th time the user looks at the page. Also modify it so that on the 20th time the user looks at the page, it deletes the cookie and the page count starts over.
3. Write a PHP program that displays a form for a user to pick their favorite color from a list of colors. Make another page whose background color is set to the color that the user picks in the form. Store the color value in `$_SESSION` so that both pages can access it.
4. Write a PHP program that displays an order form. The order form should list six products. Next to each product name there should be a text box into which a user can type in how many of that product they want to order. When the form is submitted, the submitted form data should be saved into the session. Make another page that displays the contents of the saved order, a link back to the order form page, and a Check Out button. If the link back to the order form page is clicked, the order form page should be displayed with the saved order quantities from the session in the text boxes. When the Check Out button is clicked, the order should be cleared from the session.