CHAPTER 13

Cookies and Session Tracking

The ability to track users and customize user information based on personal preferences has become both one of the hottest and most debated features to be offered on the Web. While the advantages of being able to offer users services based on exactly what they desire are obvious, many questions have been raised regarding privacy in terms of the ramifications of being able to "follow" a user as that user navigates from page to page, and even from site to site.

Barring privacy concerns, the process of tracking user information through cookies or other technologies can be immensely beneficial to both the user and the site offering these services. It is to the user's benefit that these services provide the opportunity to customize content, weeding out any information that may be uninteresting or useless. This capability is also highly beneficial to the site administrators, as tracking user preferences and habits opens up a whole new realm of possibilities for user interaction, including targeted marketing and a vastly superior analysis of the popularity of their onsite content. On the commerce-dominated Web, these capabilities are by now practically the de facto standard.

This idea of tracking a user while navigating through your site can be defined as session tracking. Given the vast amount of knowledge that could be gained from introducing session tracking into your site architecture, it could be said that the advantages of session tracking and providing customized content far outweigh the disadvantages. With that said, this could hardly be considered a complete PHP textbook without devoting a chapter to PHP's session-tracking capabilities. In this chapter, I introduce several concepts closely intertwined with session tracking, namely, session cookies and their uses, unique session identification numbers, before concluding the chapter with a synopsis of PHP's predefined session-tracking configuration and predefined functions.

What Is a Cookie?

A *cookie* is nothing more than a small parcel of information that is sent by a Web server and stored on a client browser. This can be advantageous to the developer because useful data regarding the user session can be stored and then later retrieved, resulting in the creation of a state of persistence between the client and

server. Cookies are commonly used by many Internet sites as a means to enhance both user experience and site efficiency, providing a way to track user navigation, actions, and preferences. The ability to store this information is a key feature for sites offering such services as online shopping, site personalization, and targeted advertising.

Due to the usercentric purpose of cookie usage, the key piece of information stored is likely to be a unique user identification number (UIN). This ID is subsequently stored in a database and is used as the key for retrieving any information stored in the database that is mapped to this UIN. Of course, it is not mandatory that the cookie is used to store a UIN; you could store anything you like in the cookie, provided that its total size does not surpass four kilobytes (4096 bytes).

Cookie Components

Interestingly, other pieces of information are also stored in the cookie, enabling the developer to tailor its usage in terms of domain, time frame, path, and security. Here are descriptions of the various cookie components:

- **name**—The cookie name is a mandatory parameter because the name is the parameter from which the cookie is referenced. The cookie name can be essentially thought of in terms of a variable name.

- **value—**A cookie value is simply a piece of data mapped to the cookie name. This could be a user identification number, background color, date, anything.

- **expiration date—**This date defines the lifetime of the cookie. Once this timestamp equals the current date and time, the cookie will expire and be rendered unusable. According to cookie specifications, inclusion of the expiration date is optional. However, PHP's cookie-setting functionality requires that this expiration date is set. According to the cookie specifications, if an expiration date is not included, the cookie will expire at the end of the user session (that is, when the user exits the site).

- **domain—**This is the domain that both created and can read the cookie. If a domain has multiple servers and would like all servers to be able to access the same cookie, then the domain could be set in the form of .phprecipes.com. In this case all potential third-level domains falling under the PHPrecipes site, such as wap.phprecipes.com or news.phprecipes.com, would have access to the cookie. For security reasons, a cookie cannot be set for any domain other than the one mapped to the server attempting to

set the cookie. This parameter is optional. If it is not included, it will default to the domain name from which the cookie is emanating.

- **path**—The path setting specifies the URL path from which the cookie is valid. Any attempt to retrieve a cookie from outside of this path will fail. Setting path is optional. If it is not set, then the path will be set to the path of the document from which the cookie is created.

- **security**—This determines whether or not the cookie can be retrieved in a nonsecure setting. Because the cookie will be primarily used in a nonsecure setting, this optional parameter will default to FALSE.

Although all cookies must abide by the same set of syntax rules when they are set, the cookie storage format is browser dependent. For example, Netscape Communicator stores a cookie in a format similar to the following:

```
.phprecipes.com    FALSE    /    FALSE    971728956    bgcolor    blue
```

In Internet Explorer, the same cookie would be stored as:

```
bgcolor
blue
localhost/php4/php.exe/book/13/
0
2154887040
29374385
522625408
29374377
*
```

To correctly view a cookie stored by Internet Explorer, just open it up using a text editor. Keep in mind that certain text editors do not properly process the newline character found at the end of each line, causing them to appear as squares in the cookie document.

NOTE *Internet Explorer stores its cookie information in a folder aptly entitled "Cookies," while Netscape Communicator stores it in a single file entitled "cookies." Just perform a search on your drive to find these files.*

Cookies and PHP

OK, enough background information. By now, I'm sure you're eager to learn how you can begin using PHP to store and retrieve your own cookies. You'll be happy to know that it is surprisingly easy, done with a simple call to the predefined function `setcookie()`.

The function `setcookie()` stores a cookie on a user's machine. Its syntax is:

```
int setcookie (string name [, string val [, int date [, string path [, string
domain [, int secure]]]]])
```

If you took a moment to read the introduction to cookies, you are already familiar with the parameters in the `setcookie()` syntax. If you've skipped ahead and are not familiar with the mechanics of persistent cookies, I suggest that you return to the beginning of this section and read through the introduction, as all of the `setcookie()` parameters are introduced there.

Before proceeding, I ask that you read the following sentence not once, not twice, but three times. A cookie must be set *before* any other page-relevant information is sent to the browser. Write this 500 times on a blackboard, get a tattoo stating this rule, teach your parrot to say it: I don't care, just get it straight. In other words, you cannot just set a cookie where you wish in a Web page. It must be sent before any browser-relevant information is sent; otherwise *it will not work*.

Another important restriction to keep in mind is that you cannot set a cookie and then expect to use that cookie in the same page. Either the user must refresh the page (don't count on it), or you will have to wait until the next page request before that cookie variable can be used.

This example illustrates how `setcookie()` is used to set a cookie containing a user identification number:

```
$userid = "4139b31b7bab052";
$cookie_set = setcookie ("uid", $value, time()+3600, "/", ".phprecipes.com", 0);
```

After analyzing this code, you'll notice these results of setting the cookie:

- After reloading or navigating to any subsequent page, the variable `$userid` becomes available, producing the user id 4139b31b7bab052.
- This cookie will expire (thus be rendered unusable) exactly one hour (3600 seconds) after it has been sent.
- The cookie is available for retrieval in all directories on the server.

- This cookie is only accessible via the phprecipes.com domain.
- This cookie is accessible via a nonsecured protocol.

The next example, shown in Listing 13-1, illustrates how a cookie can be used to store page-formatting preferences, in this case the background color. Notice how the cookie will only be set *if* the form action has been executed.

Listing 13-1: Storing a user's favorite background color

```
<?
// If the variable $bgcolor exists…
if (isset($bgcolor)) :
     setcookie("bgcolor", $bgcolor, time()+3600);
?>

<html>
<body bgcolor="<?=$bgcolor;?>">

<?
// else, $bgcolor is not set, therefore show the form
else :
?>
<body bgcolor="white">
<form action="<? print $PHP_SELF; ?>" method="post">
     What's your favorite background color?
     <select name="bgcolor">
          <option value="red">red
          <option value="blue">blue
          <option value="green">green
          <option value="black">black
     </select>
          <input type="submit" value="Set background color">
</form>

<?
endif;
?>
</body>
</html>
```

On loading of this page to the browser, the script will verify whether the cookie entitled "bgcolor" has been set. If it has, then the background color of the page will be set to the value specified by the variable `$bgcolor`. Otherwise, an

HTML form will appear, prompting the user to specify a favorite background color. Once the color is specified, subsequent reloading of the page or traversal to any page using the cookie value `$bgcolor` will be recognized.

Interestingly, you can also use array notation to specify cookie names. You could specify cookie names as uid[1], uid[2], uid[3], and so on, and then later access these values just as you would a normal array. Check out Listing 13-2 for an example of how this works.

Listing 13-2: Assigning cookie names according to array index value

```
<?
setcookie("phprecipes[uid]", "4139b31b7bab052", time()+3600);
setcookie("phprecipes[color]", "black", time()+3600);
setcookie("phprecipes[preference]", "english", time()+3600);

if (isset ($phprecipes)) {
   while (list ($name, $value) = each ($phprecipes)) {
      echo "$name = $value<br>\n";
   }
}
?>
```

Executing this script results in the following output, in addition to three cookies being set on the user's computer:

```
uid = 4139b31b7bab052
color = black
preference = english
```

NOTE *Although the use of array-based cookies may seem like a great idea for storing all kinds of information, keep in mind that certain browsers (such as Netscape Communicator) limit the number of cookies to 20 per domain.*

Perhaps the most common use of cookies is for storage of a user identification number that will be later used for retrieving user-specific information. This process is illustrated in the next listing, where a UIN is stored in a MySQL database. The stored information is subsequently retrieved and used to set various pieces of information regarding the formatting of the page.

To set the stage for the next listing, assume that a table entitled user_info resides on a database named user. The user_info table contains three pieces of

information: a user ID, first name, and email address. This table was created using the following syntax:

```
mysql>create table user_info (
->user_id char(18),
->fname char(15),
->email char(35));
```

Listing 13-3 actually picks up about halfway through what would be a complete "registration" script, starting where the user information (user ID, first name, and email address) has already been inserted into the database. To eliminate the need for the user to later log in, the user ID (set to 15 in Listing 13-3 for the sake of illustration) is stored on the user's computer by way of a cookie.

Listing 13-3: Retrieving user information from a database

```
<?
if (! isset($userid)) :
     $id = "15";
     setcookie ("userid", $id, time()+3600);
     print "A cookie containing your userID has been set on your machine. Please
refresh the page to retrieve your user information";
else:
     @mysql_connect("localhost", "web", "4tf9zzzf")
             or die("Could not connect to MySQL server!");
     @mysql_select_db("user") or die("Could not select user database!");
     // declare query
     $query = "SELECT * FROM user_info WHERE user_id = '$userid'";
     // execute query
     $result = mysql_query($query);

     $row = mysql_fetch_array($result);
     print "Hi ".$row["fname"].",<br>";
     print "Your email address is ".$row["email"];

     mysql_close();
endif;
?>
```

Listing 13-3 highlights just how useful cookies can be for identifying users. The above scenario could be applied to any number of situations, ranging from eliminating the need to log in to effectively tracking user preferences.

The listing in the next section, "Unique Identification Numbers," illustrates the complete process of user registration and subsequent storage of the unique user ID.

> **NOTE** *The MySQL functions used in Listing 13-3 are introduced in Chapter 11, "Databases."*

Unique Identification Numbers

By now you are probably curious just how easy it is to create a unique UIN. Put your college calculus books away; there is no need for funky 17th-century algorithms. PHP provides an easy way to create a unique UIN through its predefined function `uniqid()`.

The function `uniqid()` generates a 13-character unique identification number based on the current time. Its syntax is:

```
int uniqid (string prefix [, boolean lcg])
```

The input parameter `prefix` can be used to begin the UIN with a particular string value. Since `prefix` is a required parameter, you must designate at least an empty value. If set to TRUE, the optional input parameter `lcg` will cause `uniqid()` to produce a 23-character UIN. To quickly create a unique ID, just call `uniqid()` using an empty value as the sole input parameter:

```
$uniq_id = uniqid("");
// Some 13 character value such as ' 39b3209ce8ef2' will be generated.
```

Another way to create a unique ID is to prepend the derived value with a string, specified in the input parameter `prefix`, as shown here:

```
$uniq_id = uniqid("php", TRUE);
// Some 16 character value such as 'php39b3209ce8ef2' will be generated.
```

Given the fact that `uniqid()` creates its UIN based on the current time of the system, there is a remote possibility that it could be guessed. Therefore, you may want to ensure that its value is truly random by first randomly choosing a prefix using another of PHP's predefined functions, `rand()`. The following example demonstrates this usage:

```
srand ((double) microtime() * 1000000);
$uniq_id = uniqid(rand());
```

The function `srand()` acts to initiate the random number generator. If you want to ensure that `rand()` consistently produces a random number, you must execute `srand()` first. Placing `rand()` as an input parameter to `uniqid()` will result in `rand()` first being executed, returning a prefix value to `uniqid()`, which will then execute, producing a UIN that would be rather difficult to guess.

Armed with the knowledge of how to create unique user IDs, you can now create a practical user registration scheme. On first request of the script in Listing 13-4, the user is greeted with a short form requesting a name and email address. This information will be then inserted along with a generated unique ID into the table user_info, first described along with Listing 13-3. A cookie containing this unique ID is then stored on the user's computer. Any subsequent visit to the page will prompt the script to query the database based on the unique user ID stored in the cookie, displaying the user information to the screen.

Listing 13-4: A complete user registration process

```
<?
// build form
$form = "
<form action=\"Listing13-4.php\" method=\"post\">
<input type=\"hidden\" name=\"seenform\" value=\"y\">
Your first name?:<br>
<input type=\"text\" name=\"fname\" size=\"20\" maxlength=\"20\" value=\"\"><br>
Your email?:<br>
<input type=\"text\" name=\"email\" size=\"20\" maxlength=\"35\" value=\"\"><br>
<input type=\"submit\" value=\"Register!\">
</form>
";
// If the form has not been displayed and the user does not have a cookie.
if ((! isset ($seenform)) && (! isset ($userid))) :

     print $form;

// If the form has been displayed but the user information
// has not yet been processed
elseif (isset ($seenform) && (! isset ($userid))) :

     srand ((double) microtime() * 1000000);
     $uniq_id = uniqid(rand());
     // connect to the MySQL server and select the users database
     @mysql_pconnect("localhost", "web", "4tf9zzzf")
              or die("Could not connect to MySQL server!");
     @mysql_select_db("user") or die("Could not select user database!");
```

```
     // declare and execute query
     $query = "INSERT INTO user_info VALUES('$uniq_id', '$fname', '$email')";
     $result = mysql_query($query) or die("Could not insert user information!");

     // set cookie "userid" to expire in one month.
     setcookie ("userid", $uniq_id, time()+2592000);

     print "Congratulations $fname! You are now registered! Your user information
     will be displayed on each subsequent visit to this page.";
// else if the cookie exists, use the userID to extract
// information from the users database
elseif (isset($userid)) :
     // connect to the MySQL server and select the users database
     @mysql_pconnect("localhost", "web", "4tf9zzzf")
              or die("Could not connect to MySQL server!");
     @mysql_select_db("user") or die("Could not select user database!");

     // declare and execute query
     $query = "SELECT * FROM user_info WHERE user_id = '$userid'";
     $result = mysql_query($query) or die("Could not extract user information!");

     $row = mysql_fetch_array($result);
     print "Hi ".$row["fname"].",<br>";
     print "Your email address is ".$row["email"];

endif;

?>
```

The judicious use of several *if* conditionals makes it possible to use one script to take care of each step of the registration and subsequent user recognition process. There are three scenarios involved in this script:

- The user has not seen the form and does not have a valid cookie. This is the step where the user is presented with the form.

- The user has filled in the form and does not yet have a valid cookie. This is the step where the user information is entered into the database, and the cookie is set, due to expire in one month.

- The user returns to the script. If the cookie is still valid (has not expired), the cookie is read in and the relevant information is extracted from the database.

The general process shown in Listing 13-4 could of course be applied to any database. This illustrates, on a very basic level, how many of the larger sites are able to apply user-specified preferences to their site, resulting in a "tailor-made" look for each user.

This ends the introduction to PHP and cookies. If you are interested in learning more about the cookie mechanism, check out the online resources that I've cited in the sidebar "Relevant Links."

Relevant Links

For more information regarding cookies and their usage, take a moment to read through a few of the resources that I've gleaned from the Web:

- http://www.cookiecentral.com
- http://home.netscape.com/newsref/std/cookie_spec.html
- http://builder.com/Programming/Cookies/ss01.html
- http://www.w3.org/Protocols/rfc2109/rfc2109

As you have learned, cookies can be very useful for "remembering" user-specific information that can be retrieved in subsequent visits to your site. However, cookies can not be solely relied on since users can set their browsers to refuse to accept cookies. Thankfully, PHP offers an alternative methodology for storing persistent information; This method is called *session tracking* and is the subject of the next section.

Session Handling

A *session* is best defined as the period of time beginning when a user enters your site and ending when the user exits. Throughout this session, you may wish to assign various variables that will accompany the user while navigating around your site, without having to manually code a bunch of hidden forms or appended URL variables. This otherwise tedious process becomes fairly easy with *session handling*.

Consider the following scenario. Using session handling, a user entering your site would be assigned a unique session id (SID). This SID is then sent to the user's browser in a cookie entitled PHPSESSID. If cookie support is disabled or not supported, this SID can be automatically appended to all local URLs throughout the user session. At the same time, a file with the same name as the SID is stored on

the server. As the user navigates throughout the site, you may wish to record certain variables as session variables. These variables are stored in that user's file. Any subsequent call to any of those variables deemed to be of the "session" type will cause the server to grab that user's session file and search it for the session variable in question. And voilà! The session variable is displayed. In a nutshell, this is the essence of session handling. Of course, you can also direct this user information to be stored in databases or other files, whatever you wish.

Sounds interesting? You bet it does. Armed with this information, you will surely have a better understanding of the various configuration issues at hand, which I will now discuss. There are three particularly important configuration flags. The first, entitled `–enable-trans-id`, must be included in the configuration process if you wish to take advantage of its features (described below). The other two, entitled `track_vars` and `register_globals`, can be enabled and disabled as necessary in the php.ini file. The ramifications of activating these three flags are discussed next.

–enable-trans-sid

When PHP is compiled with this flag, all relative URLs will automatically be rewritten with the session ID attached. This appendage of the session ID is written in the form `session-name=session-id`, where `session-name` is defined in the php.ini file (explained later in this section). If you decide not to do so, you can use the constant SID.

track_vars

Enabling `track_vars` allows `$HTTP_*_VARS[]` arrays, where * is one of the EGPCS (Environment, Get, Post, Cookie, Server) values. This must be enabled in order for the SID to propagate from one page to another. As of PHP 4.03, this setting is always enabled.

register_globals

Enabling this option will result in all EGCPS variables being globally accessible. You want this disabled if you don't want your global array filling with perhaps unnecessary data. If this is disabled and `track_vars` is enabled, all GPC variables can be accessed through the *$HTTP_*_VARS[]* arrays. As an example, if `register_globals` is disabled, you would have to refer to the predefined variable `$PHP_SELF` as `$HTTP_SERVER_VARS["PHP_SELF"]`.

There are also a number of preferential configuration issues that you should take care of. These directives are described in Table 13-1, shown in their default form as seen in the php.ini file. They are introduced in the order that they actually appear in the file.

Table 13-1. Session-handling directives in the php.ini file

DIRECTIVE		DESCRIPTION
`session.save_handler`	`= files`	Specifies how the session information will be stored on the server. There are three ways to do so: in a file (`files`), shared memory (`mm`), or through user-defined functions (`User`). The user-defined functions allow you to easily store the information in any format you wish, for example, in a database.
`session.save_path`	`= /tmp`	Designates the directory in which the PHP session files will be stored. On the Linux platform, the default setting (`'/tmp'`) is probably just fine. On the Windows platform, you will need to change this to some Windows path; otherwise errors will occur.
`session.use_cookies`	`= 1`	When enabled, cookies are used to store the session ID on the user's computer.
`session.name`	`= PHPSESSID`	If `session.use_cookies` is enabled, then `session.name` will be used as the cookie name. The characters comprising the name can only be alphanumeric.
`session.auto_start`	`= 0`	When enabled, `session.auto_start` will automatically initiate a session when a client makes an initial request.
`session.cookie_lifetime`	`= 0`	If `session.use_cookies` is enabled, then `session.cookie_lifetime` will determine the lifetime of the sent cookies. If it is set to 0, then any sent cookies will expire on the termination of the user session.
`session.cookie_path`	`= /`	If `session.use_cookies` is enabled, then `session.cookie_path` determines the parent path directory for which sent cookies are valid.
`session.cookie_domain`	`=`	If `session.use_cookies` is enabled, then `session.cookie_domain` determines the domain for which sent cookies are valid.
`session.serialize_handler`	`= php`	This specifies the name of the handler that will be used to serialize data. There are currently two possible values for this: `php` and `WDDX`.

Table 13-1. (Continued)

DIRECTIVE		DESCRIPTION
session.gc_probability	= 1	This specifies the percentual probability that PHP's *g*arbage *c*ollection routine will be activated.
session.gc_maxlifetime	= 1440	Specifies the time (in seconds) before session data is considered invalid and will be destroyed. This timer begins counting down after the last access to the session.
session.referer_check	=	When set to a string, each request to a session-enabled page will begin with a verification that the specified string is in the global variable $HTTP_REFERER. If it is not found, any accompanying session ID will be ignored.
session.entropy_file	=	Points to an external file that supplies additional random information used during the creation of the session ID. There are typically two devices on UNIX systems made for this purpose, /dev/random and /dev/urandom. The /dev/random device collects random data from inside the kernel, while the /dev/urandom device relies on the MD5 hashing algorithm to produce a random string. In short, /dev/random is faster, but /dev/urandom produces a more "random" string.
session.entropy_length	= 0	Assuming session.entropy_file is set, session.entropy_length specifies the number of bytes to be read from the file specified by session.entropy_length.
session.cache_limiter	= nocache	Determines the cache control method for session pages. There are three possible values for this setting: nocache, public, and private.
session.cache_expire	= 180	Determines the TTL (time to live) in minutes for cached session pages.

NOTE *The configuration directive* `session.save_handler` *is so useful that I felt an entire section should be devoted to it. It is located at the conclusion of this chapter under "Specifying User Callbacks as Storage Modules".*

Now that you have presumably made any necessary configuration adjustments to your server, I will turn attention toward the mechanics of how you can implement session handling on your site. It is actually a rather simple process, made possible through the use of several predefined functions. The first concept that you need to know is that a session is initiated with the function `session_start()`. Of course, you could eliminate the need to use this function if you had enabled `session.auto_start` in the php.ini file as discussed earlier in this section. However, for the remainder of this section, I will assume that you have not done this so to ensure consistency in my examples. The syntax of `session_start()` is simple, as it requires no input parameters and returns only a boolean informing the developer as to its success.

session_start()

The function `session_start()` is twofold in purpose. Once called, it checks to see if the user has already started a session, and if the user has not, it starts one. Its syntax is:

```
boolean session_start()
```

If it starts a session, it performs three functions, assigning the user a SID, sending a cookie (if `session.use_cookies` is enabled in the php.ini file), and creating the session file on the server. Its second purpose is that it informs the PHP engine that other session variables may be used in the script from which it (`session_start()`) is executed.

A session is started simply by calling `session_start()` like this:

```
session_start();
```

Just as a session can be created, it can be destroyed. This is accomplished via the function `session_destroy()`.

TIP *The* `session_start()` *function returns TRUE no matter what the actual outcome is. Therefore, it does no good to use it in if conditionals or in conjunction with* `die()` *statements.*

session_destroy()

The function `session_destroy()` will destroy all persistent data corresponding to the current user session. Its syntax is:

```
boolean session_destroy()
```

Keep in mind that this will *not* destroy any cookies on the user's browser. However, if you are not interested in using the cookie beyond the end of the session, just set `session.cookie_lifetime` to 0 (its default value) in the php.ini file. An example of the function's usage is:

```
<?
session_start();
// do some session stuff
session_destroy();
?>
```

Now that you know how to create and destroy sessions, you are ready to begin working with the various session variables. Perhaps the most important one is the SID. This is easily obtainable through using the `session_id()` function.

session_id()

The function `session_id()` returns the user's SID originally created by `session_start()`. This is its syntax:

```
string session_id([string sid])
```

If you supply a session ID as the optional input parameter `sid`, the user's session ID will be changed. Keep in mind, however, that this will *not* resend the cookie. Executing this example:

```
<?
session_start ();
print "Your session identification number is ".session_id();
session_destroy();
?>
```

results in output similar to the following being displayed to the browser:

```
Your session identification number is 967d992a949114ee9832f1c11cafc640
```

So how can you begin creating your own session variables? The function `session_register()` takes care of this job handily.

session_register()

The function `session_register()` registers one or more variable names with the user's current session. Its syntax is:

```
boolean session_register (mixed varname1 [, mixed varname2 …])
```

Keep in mind that you are *not* registering variables, but rather the names of the variables. `Session_register()` will also call `session_start()` internally, implicitly beginning a new session if one does not already exist.

Before exemplifying the usage of `session_register()`, I would like to introduce another session-oriented function that can verify whether or not a particular variable has been registered. The function is entitled `session_is_registered()`.

session_is_registered()

It is often useful to determine whether or not a variable has already been registered. This task can be accomplished with `session_is_registered()`. Its syntax is:

```
boolean session_is_registered (string varname)
```

To illustrate the usage of `session_register()` and `session_is_registered()`, I'll refer to what seems to be everyone's favorite basic session example: a hit counter. This is illustrated in Listing 13-5.

Listing 13-5: A user-specific hit counter

```
<?
session_start();
if (! session_is_registered('hits')) :
     session_register('hits');
endif;
$hits++;
print "You've seen this page $hits times";
?>
```

Just as you can create session variables, you can destroy them. This is accomplished with `session_unregister()`.

session_unregister()

A session variable can be destroyed with a call to `session_unregister()`. Its syntax is:

```
boolean session_unregister (string varname)
```

The input parameter `varname` is the name of the session variable that you would like to destroy.

```
<?
session_start();
session_register('username');
// ...use the variable $username as needed, then destroy it.
session_unregister('username');
session_destroy();
?>
```

As is the case with `session_register()`, remember that you do not specify the input parameter `varname` as an actual variable (that is, with a preceding dollar sign [$]). Instead, you just use the *name* of the variable.

session_encode()

The function `session_encode()` offers a particularly convenient method for formatting session variables for storage, for example in a database. Its syntax is:

```
boolean session_encode()
```

Executing this function will result in all session data being formatted into a single string. This string can then be inserted into a database for storage purposes.

Consider Listing 13-6 for an example of how `session_encode()` is used. Assume that a "registered" user has a cookie containing that user's unique ID stored on a computer. When the user requests the page containing Listing 13-6, the user ID is retrieved from the cookie. This value is then assigned to be the session ID. Certain session variables are created and assigned values, and then all of this information is encoded using `session_encode()` and inserted into a MySQL database.

Listing 13-6: Using session_encode() to store data in a MySQL database

```
<?
// Initiate session and create a few session variables
session_register('bgcolor');
session_register('fontcolor');

// assume that the variable $usr_id (containing a unique user ID)
// is stored in a cookie on the user's machine.

// use session_id() to set the session ID to be the user's
// unique user ID stored in the cookie and In the database
$id = session_id($usr_id);

// these variables could be set by the user via an HTML form
$bgcolor = "white";
$fontcolor = "blue";

// encode all session data into a single string
$usr_data = session_encode();

// connect to the MySQL server and select users database
@mysql_pconnect("localhost", "web", "4tf9zzzf") or die("Could not connect to MySQL
server!");
@mysql_select_db("users") or die("Could not select user database!");

// update the user's page preferences
$query = "UPDATE user_info set page_data='$usr_data' WHERE user_id= '$id'";
$result = mysql_query($query) or die("Could not update user information!");
?>
```

As you can see, the capability to quickly convert all of the session variables into a single string eliminates the need to keep track of several column names when storing and retrieving data and eliminates several lines of code that would otherwise be needed to store and retrieve this data.

session_decode()

Any session data previously encoded with `session_encode()` can be decoded with `session_decode()`. Its syntax is:

```
string session_decode(string session_data)
```

The input parameter session_data is the encoded string of session variables, presumably returned from a file or database retrieval. The string is decoded, and all session variables in the string are regenerated back to their original variable format.

Listing 13-7 illustrates how previously encoded session variables are regenerated by using session_decode(). Assume that a MySQL table entitled "user_info" is built from just two columns: user_id and page_data. The user's UID, stored in a cookie on the user's computer, is used to retrieve encoded session data stored in the page_data column. The page_data column stores an encoded string of variables, one of which is the user's preferential background color, stored in the variable $bgcolor.

Listing 13-7: Decoding session data stored in a MySQL database

```
<?
// assume that the variable $usr_id (containing a unique user ID)
// is stored in a cookie on the user's machine.

$id = session_id($usr_id);

// connect to the MySQL server and select user's database
@mysql_pconnect("localhost", "web", "4tf9zzzf") or die("Could not connect to MySQL
server!");
@mysql_select_db("users") or die("Could not select company database!");

// select data from the MySQL table
$query = "SELECT page_data FROM user_info WHERE user_id= '$id'";
$result = mysql_query($query);
$user_data = mysql_result($result, 0, "page_data");

// decode the data
session_decode($user_data);

// output one of the regenerated session variables
print "BGCOLOR: $bgcolor";

?>
```

As you can see from the previous two listings, session_encode() and session_decode() are enormously useful and efficient for storing and retrieving session data.

Specifying User Callbacks as Storage Modules

While storing session information in files works pretty well, you may be interested in storing data using other mediums, probably a database. Or perhaps you are interested in reusing the same scripts on different sites, but with different databases. Another common dilemma is the need to share session data across various servers, something that is rather difficult when using PHP's default routines of storing session data in a file. You'll be happy to know that realizing all of these extensions to PHP's session handling is really an easy task, given PHP's capability to allow users to specify their own storage routines via a predefined function called `session_set_save_handler()`.

The function `session_set_save_handler()` defines the user-level session storage and retrieval functions. Its syntax is:

```
void session_set_save_handler (string open, string close, string read, string
write, string destroy, string gc)
```

The six input parameters correspond to the six functions that are transparently called by PHP's session-handling functions. The function `session_set_save_handler()` allows you to redefine these functions without affecting the scripts that call PHP's predefined session functions. Although you can change the names of the functions to be whatever you wish, each must take as input a specified set of parameters. Before proceeding to an example, take a look at Table 13-2 to understand the roles of these six functions and their input parameters.

NOTE *In order to make use of* `session_set_save_handler()`*, you must set* `session.save_handler` *to* `user` *in the php.ini file.*

Table 13-2: Six input parameters for the function session_set_savehandler()

PARAMETER	DESCRIPTION
`sess_close()`	Called when a script implementing the session functions finishes. This is *not* the same as sess_destroy(), which is used to actually destroy the session variables. There aren't any input parameters for `sess_close()`.
`sess_destroy($session_id)`	Deletes all session data. The input parameter `$session_id` specifies which session is to be destroyed.
`sess_gc($maxlifetime)`	Deletes any sessions that have expired. The expiration time is denoted by the input parameter `$maxlifetime, specified in seconds`. This parameter is read from the php.ini file and corresponds to `session.gc_lifetime`.
`sess_open($sess_path, $sess_name)`	Called when a new session is initialized, either by `session_start()` or `session_register()`. The two input parameters `$sess_path` and `$sess_name` are read from the php.ini file and correspond to the `session.save_path` and `session.name` parameters, respectively.
`sess_read($key)`	Used to retrieve the value corresponding to a session variable, denoted by the input parameter `$key`.
`sess_write($key, $value)`	Used to write the session data. Any data saved by `sess_write()` can later be retrieved by `sess_read()`. The input parameter `$key` corresponds to a session variable name, and `$value` corresponds to the value assigned to `$key`.

Now that you know more about the functions that you need to define, I'll provide an example of a MySQL-based implementation of the session-handling functions. This example is given in Listing 13-8.

Listing 13-8: MySQL implementation of the session-handling functions

```
<?
// MySQL implementation of session-handling functions

// mysql server host, username, and password values
$host = "localhost";
$user = "web";
$pswd = "4tf9zzzf";

// database and table names
$db = "users";
$session_table = "user_session_data";
$SESS_TBLNAME = "user_session_data";

// retrieve sess.gc_lifetime value from php.ini file
$sess_life = get_cfg_var("sess.gc_lifetime");

// Function: mysql_sess_open()
// mysql_sess_open() connects to the MySQL server
// and selects the database.

function mysql_sess_open($save_path, $session_name) {
     GLOBAL $host, $user, $pswd, $db;

     @mysql_pconnect($host, $user, $pswd)
              or die("Can't connect to MySQL server!");
     @mysql_select_db($SESS_$db) or die("Can't select session database!");
}

// Function: mysql_sess_close()
// mysql_sess_close() is not needed in the MySQL implementation.
// *However*, it still must be defined.

function mysql_sess_close() {
     return true;
}

// Function: mysql_sess_read()
// mysql_sess_read() reads the information from the MySQL database.

function mysql_sess_read($key) {
     GLOBAL $session_table;
```

```
     $query = "SELECT value FROM $session_table WHERE sess_key = '$key' AND
                    sess_expiration >". time();
     $result = mysql_query($query);

     // If session value Is found, return it
     if (list($value) = mysql_fetch_row($result)) :
          return $value;
     endif;

     return false;
}

// Function: mysql_sess_write()
// mysql_sess_write() writes the information to the MySQL database.

function mysql_sess_write($key, $val) {
     GLOBAL $sess_life, $session_table;

     // set expiration time
     $expiration = time() + $sess_life;

     $query = "INSERT INTO $session_table VALUES('$key', '$expiration',
              '$value')";
     $result = mysql_query($query);

     // if the insert query failed because of the primary key already exists,
     // perform an update instead.

     if (! $result) :
   $query = "UPDATE $session_table SET sess_expiration = '$expiration',
                                sess_value='$value' WHERE sess_key = '$key'";
              $result = mysql_query($result);
     endif;
}

// Function: mysql_sess_destroy()
// mysql_sess_destroy() deletes all table rows having the session key = $sess_id

function mysql_sess_destroy($sess_id) {
     GLOBAL $session_table;

     $query = "DELETE FROM $session_table WHERE sess_key = '$sess_id'";
     $result = mysql_result($query);
```

```
     return $result;
}

// Function: mysql_sess_gc()
// mysql_sess_gc() deletes all table rows
// having an expiration < current time - session.gc_lifetime

function mysql_sess_gc($max_lifetime) {
     GLOBAL $session_table;

     $query = "DELETE FROM $session_table WHERE sess_expiration < " . time();
     $result = mysql_query($query);

     return mysql_affected_rows();
}

session_set_save_handler("mysql_sess_open", "mysql_sess_close", "mysql_sess_read",
"mysql_sess_write", "mysql_sess_destroy", "mysql_sess_gc");
?>
```

Once you have defined these six functions, you are then free to execute each through its abstract name (`sess_close()`, `sess_destroy()`, `sess_gc()`, `sess_open()`, `sess_read()`, or `sess_write()`). The convenience in this lies in the fact that you could then build as many implementations as necessary and then redefine `session_set_save_handler()` whenever necessary.

Project: Create a Visitor Log

It's often useful to record information about your site's visitors. As you already know, this is a common practice among Web advertising agencies, portals, and any of a number of other sites interested in learning more about their visitors. While these systems can get enormously complicated, there are still a number of benefits that can be obtained from the creation of a relatively simple logging system. I'll show you how to build just such a simple system using PHP, MySQL, and cookies.

> **CAUTION** *This project incorporates the Chapter 8 browser detection project. If you skipped over either Chapter 8 or the project, I would strongly recommend at least reviewing the project code before proceeding.*

As I've already said, our system will be relatively simple, monitoring only visits to the site index page. When the visitor arrives, the PHP script checks to see whether or not a valid cookie resides on the visitor's computer. If one does, this signifies that the user has previously visited in a specified time frame (preset by the site administrator in an initialization file), and the script will not count this visit. If there is no cookie (or there is a previously set cookie that has expired), then either the user has never visited or the preset time frame between visits has been surpassed, and the information is recorded to the MySQL table. Furthermore, a new cookie is sent to the visitor's computer.

How can this script be constructed using PHP? First, you need to create the MySQL table that holds the information:

```
mysql>create table visitors (
         ->browser char(85) NOT NULL,
         ->ip char(30) NOT NULL,
         ->host char(85) NOT NULL,
         ->timeOfVisit datetime NOT NULL
         ->);
```

What is the purpose of each column? The column browser contains information directly relating to the user's browser. This information is supplied by the PHP variable `$HTTP_USER_AGENT`. The column ip contains the user's IP address. The column host contains ISP information from where the IP address emanates. Finally, the column timeOfVisit specifies the date and time that the visitor arrived at the site.

NOTE *A powerful visitor-logging application is available for free download from the PHP resource site phpinfo.net (http://www.phpinfo.net). You can also check out a live implementation onsite. However, you'll need to dust off that French textbook before going there!*

Next, create the application initialization file, init.inc, as shown in Listing 13-9. It holds both the global variables and core functions. Notice that the functionality of the Chapter 8 project script sniffer.php is used in the `viewStats()` function. This script will be included along with the init.inc file when necessary. Take a moment to review this script and its comments.

Listing 13-9: Creating the application initialization file (init.inc)

```
<?
// file: init.inc
// purpose: initialization file for Visitor Logging project

// Database connection variables
$host = "localhost";
$user = "web";
$pswd = "4tf9zzzf";

// database name
$database = "myTracker";

// polls table name
$visitors_table = "visitors";

// Connect to the MySQL Server
@mysql_pconnect($host, $user, $pswd) or die("Couldn't connect to MySQL server!");

// Select the database
@mysql_select_db($database) or die("Couldn't select $database database!");

// Number of recent visitors to display in table
$maxNumVisitors = "5";

// Cookie Name. You can set this to whatever you wish.
// However, the current setting will work just fine.
$cookieName = "visitorLog";

// Value stored in the cookie.
$cookieValue="1";

/*
Timeframe between acknowledgement of subsequent visit by same user
If $timeLimit is set to 0, every user visit to that page will be recorded
regardless of the frequency. All other integer settings will be regarded as number
of SECONDS that must pass between visits in order to be recorded.
*/
$timeLimit = 3600;
```

```
// How would you like the table displayed?
$header_color = "#cbda74";
$table_color = "#000080";
$row_color = "#c0c0c0";
$font_color = "#000000";
$font_face = "Arial, Times New Roman, Verdana";
$font_size = "-1";

// function: recordUser
// purpose: Record user Information in the MySQL table $visitors_table
function recordUser() {

     GLOBAL $visitors_table, $HTTP_USER_AGENT, $REMOTE_ADDR, $REMOTE_HOST;

/*
If the visitor is operating on the internal site server, set the $REMOTE_HOST to
'localhost'. Alternatively, you may want to eliminate the recording of all
internal visitors, since it's likely to be yourself or another development team
member.
/*
     if ($REMOTE_HOST == "") :
          $REMOTE_HOST = "localhost";
     endif;

     // format a valid MySQL datetime format
     $timestamp = date("Y-m-d H:i:s");

     // Insert the user data into the MySQL table
     $query = "INSERT INTO $visitors_table VALUES
              ('$HTTP_USER_AGENT', '$REMOTE_ADDR',
              '$REMOTE_HOST', '$timestamp')";

     $result = @mysql_query($query);

} // recordUser

// function: viewStats
// purpose: Extract and format information  in the MySQL table $visitors_table
function viewStats() {

     // Include some global variables
     GLOBAL $visitors_table, $maxNumVisitors, $table_color, $header_color;
```

```
    GLOBAL $row_color, $font_color, $font_face, $font_size;

    // Select the most recent $maxNumVisitors from the MySQL table.
    $query = "SELECT browser, ip, host, timeOfVisit FROM $visitors_table
                  ORDER BY timeOfVisit desc LIMIT 0, $maxNumVisitors";

    $result = @mysql_query($query);

    // format and print the retrieved data
    print "<table cellpadding=\"2\" cellspacing=\"1\" width = \"800\" border =
    \"0\" bgcolor=\"$table_color\">";
    print "<tr bgcolor= \"$header_color\">
           <th>Browser</th><th>IP</th><th>Host</th><th>TimeofVisit</th>
           </tr>";

    while($row = mysql_fetch_array($result)) :

         // These functions are in 'sniffer.inc'
         list ($browse_type, $browse_version) = browser_info ($row["browser"]);
         $op_sys = opsys_info ($row["browser"]);

         print "<tr bgcolor=\"$row_color\">";
         print "<td><font color=\"$font_color\" face=\"$font_face\"
         size=\"$font_size\">";
         print "$browse_type $browse_version - $op_sys</font></td>";
         print "<td><font color=\"$font_color\" face=\"$font_face\"
         size=\"$font_size\">".$row["ip"]."</font></td>";
         print "<td><font color=\"$font_color\" face=\"$font_face\"
         size=\"$font_size\">".$row["host"]."</font></td>";
         print "<td><font color=\"$font_color\" face=\"$font_face\"
         size=\"$font_size\">";
         print $row["TimeofVisit"]."</font></td>";
         print "</tr>";

    endwhile;
    print "</table>";
} // viewStats
?>
```

Next, insert the script you see in Listing 13-10; it will be used to check for a valid cookie and call the `recordUser()` function when necessary. I'll include this code along with a very simple index file entitled "index.php."

Listing 13-10: Checking for a valid cookie (index.php)

```
<?
include("init.inc");
// If no valid cookie is found
if (! isset($$cookieName)) :
     // Set a new cookie
     setcookie($cookieName, $cookieValue, time()+$timeLimit);
     // Record the visitor information
     recordUser();
endif;
?>

<html>
<head>
<title>Welcome to My Site!</title>
</head>
<body bgcolor="#c0c0c0" text="#000000" link="#808040" vlink="#808040"
alink="#808040">
Welcome to my site. <a href = "visitors.php">Check out who else has recently
visited</a>.
</body>
</html>
```

How is the information that is stored in the MySQL database viewed in the browser? This is accomplished simply by placing the function `viewStats()`in a separate file(visitors.php), as shown here:

```
<html>
<?
// Include browser detection functionality
include("sniffer.inc");
// Include the initialization file
include("init.inc");
?>
<head>
<title>Most recent <?=$maxNumVisitors;?> visitors</title>
</head>
<body bgcolor="#ffffff" text="#000000" link="#808040" vlink="#808040"
alink="#808040">
<?
viewStats();
?>
</body>
</html>
```

Alternatively, you could place the entire HTML code in the `viewStats()` function and then just include sniffer.inc, init.inc, and a call to `viewStats()` in a separate file. It depends on how much you would like to consolidate the formatting of the page. Using the current table format settings in init.inc, a sample output produced by `viewStats()` is shown in Figure 13-1.

Browser	IP	Host	TimeofVisit
Netscape 4.75 - Linux	134.43.112.41	host12.cob.ohio-state.edu	2000-10-16 12:16:00
Opera 4.02 - Windows	212.23.456.123	nat-7.cat.com	2000-10-16 11:55:00
IE 5.0 - Windows	62.4.156.931	paris11-nas4-46-208.dial.proxad.net	2000-10-16 11:24:00
IE 5.0 - Windows	213.43.48.52	ppp-127.dialup.osu.edu	2000-10-16 11:21:00
IE 5.0 - Windows	245.12.234.512	ca.ol23.att.net	2000-10-16 11:20:00

Figure 13-1. Sample output produced by `viewStats()`

There are many modifications that you could make to this script to expand its practicality. One commonly used way to track visitors is to assign an identification number to each page that you would like to log and then track users as they navigate from page to page. This could be accomplished using the above project by simply expanding your MySQL table to include a column that stores a page identification number. Then modify the `recordUser()` function to have an input parameter from which this ID number could be passed in for recording. You could then vary each cookie to hold that page ID and check for that specific cookie as the visitor requests each logged page.

What's Next?

This chapter introduced one of the most exciting features of the PHP language: session handling. In particular, the following topics were covered:

- Cookie basics
- Cookies and PHP
- Unique identification numbers
- User registration scenarios
- Introduction to sessions
- The php.ini session parameters

- PHP's predefined session functions
- The `session_set_save_handler()` function
- A visitor-logging application

Sessions offer an enormous administrative advantage to developers interested in creating truly user-oriented Web sites. I strongly urge you to experiment with PHP's session-handling functionality, as I think you will find it particularly useful.

This chapter concludes Part II of this book. Part III, "Advanced PHP," begins with a survey of PHP and XML integration. Stay tuned, as things are about to get really interesting.