CHAPTER 18

Session Handlers

Over the course of the past few years, standard Web development practices have evolved considerably. Perhaps most notably, the practice of tracking user-specific preferences and data, once treated as one of those "gee whiz" tricks that excited only the most ambitious developers, has progressed from novelty to necessity. These days, foregoing the use of HTTP sessions is more the exception than the norm for most enterprise applications. Therefore, no matter whether you are completely new to the realm of Web development or simply haven't yet gotten around to considering this key feature, this chapter is for you.

This chapter introduces session handling, one of the most interesting features of PHP. Around since the release of version 4.0, session handling remains one of the coolest and most talked-about features of the language, yet it is surprisingly easy to use, as you're about to learn. This chapter introduces the spectrum of topics surrounding session handling, including its very definition, PHP configuration requirements, and implementation concepts. In addition, the feature's default session-management features are demonstrated in some detail. Furthermore, you'll learn how to create and define your own customized management plug-in, using a PostgreSQL database as the back end.

What Is Session Handling?

The Hypertext Transfer Protocol (HTTP) defines the rules used to transfer text, graphics, video, and all other data via the World Wide Web. It is a *stateless* protocol, meaning that each request is processed without any knowledge of any prior or future requests. Although such a simplistic implementation is a significant contributor to HTTP's ubiquity, this particular shortcoming has long remained a dagger in the heart of developers who wish to create complex Web-based applications that must be able to adjust to user-specific behavior and preferences. To remedy this problem, the practice of storing bits of information on the client's machine, in what are commonly called *cookies,* quickly gained acceptance, offering some relief to this conundrum. However, limitations on cookie size and the number of cookies allowed, and various inconveniences surrounding their implementation, prompted developers to devise another solution: *session handling*.

Session handling is essentially a clever workaround to this problem of statelessness. This is accomplished by assigning each site visitor a unique identifying attribute, known as the session ID (SID), and then correlating that SID with any number of other pieces of data, be it number of monthly visits, favorite background color, or middle name—you name it. In relational database terms, you can think of the SID as the primary key that ties all the other user attributes

together. But how is the SID continually correlated with the user, given the stateless behavior of HTTP? It can be done in two different ways, both of which are introduced in the following sections. The choice of which to implement is entirely up to you.

Cookies

One ingenious means for managing user information actually builds upon the original method of using a cookie. When a user visits a Web site, the server stores information about the user, such as their preferences, in a cookie and sends it to the browser, which saves it. As the user executes a request for another page, the server retrieves the user information and uses it, for example, to personalize the page. However, rather than storing the user preferences in the cookie, the SID is stored in the cookie. As the client navigates throughout the site, the SID is retrieved when necessary, and the various items of data correlated with that SID are furnished for use within the page. In addition, because the cookie can remain on the client even after a session ends, it can be read in during a subsequent session, meaning that persistence is maintained even across long periods of time and inactivity. However, keep in mind that because cookie acceptance is a matter ultimately controlled by the client, you must be prepared for the possibility that the user has disabled cookie support within the browser or has purged the cookies from their machine.

URL Rewriting

The second method used for SID propagation simply involves appending the SID to every local URL found within the requested page. This results in automatic SID propagation whenever the user clicks one of those local links. This method, known as *URL rewriting,* removes the possibility that your site's session-handling feature could be negated if the client disables cookies. However, this method has its drawbacks. First, URL rewriting does not allow for persistence between sessions, because the process of automatically appending a SID to the URL does not continue once the user leaves your site. Second, nothing stops a user from copying that URL into an e-mail and sending it to another user; as long as the session has not expired, the session will continue on the recipient's workstation. Consider the potential havoc that could occur if both users were to simultaneously navigate using the same session, or if the link recipient was not meant to see the data unveiled by that session. For these reasons, the cookie-based methodology is recommended. However, it is ultimately up to you to weigh the various factors and decide for yourself.

The Session-Handling Process

Because PHP can be configured to autonomously control the entire session-handling process with little programmer interaction, you may consider the gory details somewhat irrelevant. However, there are so many potential variations to the default procedure that taking a few moments to better understand this process would be well worth your time.

The very first task executed by a session-enabled page is to determine whether a valid session already exists or a new one should be initiated. If a valid session doesn't exist, one is generated and correlated with that user, using one of the SID propagation methods described earlier. An existing session is located by finding the SID either within the requested URL or within a cookie. Therefore, if the session name is sessionid and it's appended to the URL, you could retrieve the value with the following variable:

```
$_GET['sessionid']
```

If it's stored within a cookie, you can retrieve it like this:

```
$_COOKIE['sessionid']
```

With each page request, this SID is retrieved. Once retrieved, you can either begin correlating information with that SID or retrieve previously correlated SID data. For example, suppose that the user is browsing various news articles on the site. Article identifiers could be mapped to the user's SID, allowing you to compile a list of articles that the user has read, and display that list as the user continues to navigate. In the coming sections, you'll learn how to store and retrieve this session information.

■ **Tip** You can also retrieve cookie information via the `$_REQUEST` superglobal. For instance, `$_REQUEST['sessionid']` will retrieve the SID, just as `$_GET['sessionid']` or `$_COOKIE['sessionid']` would in the respective scenarios. However, for purposes of clarity, consider using the superglobal that best matches the variable's place of origin.

This process continues until the user ends the session, either by closing the browser or by navigating to an external site. If you use cookies, and the cookie's expiration date has been set to some date in the future, if the user were to return to the site before that expiration date, the session could be continued as if the user never left. If you use URL rewriting, the session is definitively ended, and a new one must begin the next time the user visits the site.

In the coming sections, you'll learn about the configuration directives and functions responsible for carrying out this process.

Configuration Directives

Twenty-five session configuration directives are responsible for determining the behavior of PHP's session-handling functionality. Because many of these directives play such an important role in determining this behavior, you should take some time to become familiar with the directives and their possible settings. The most relevant are introduced in this section.

session.save_handler (files, mm, sqlite, user)

Scope: `PHP_INI_ALL`; Default value: `files`

The `session.save_handler` directive determines how the session information will be stored. This data can be stored in four ways: within flat files (`files`), within shared memory (`mm`), using the SQLite database (`sqlite`), or through user-defined functions (`user`). Although the default setting, `files`, will suffice for many sites, keep in mind that the number of session-storage files could potentially run into the thousands, and even the hundreds of thousands over a given period of time. The shared memory option is the fastest of the group, but also the most volatile because the data is stored in RAM. The `sqlite` option takes advantage of the new SQLite extension to manage session information transparently using this lightweight database (see Chapter 22

for more about SQLite). The fourth option, although the most complicated to configure, is also the most flexible and powerful, because custom handlers can be created to store the information in any media the developer desires. Later in this chapter you'll learn how to use this option to store session data within a PostgreSQL database.

session.save_path (string)

Scope: `PHP_INI_ALL`; Default value: `/tmp`

If `session.save_handler` is set to the `files` storage option, then the `session.save_path` directive must point to the storage directory. Keep in mind that this should not be set to a directory located within the server document root, because the information could easily be compromised via the browser. In addition, this directory must be writable by the server daemon.

For reasons of efficiency, you can define `session.save_path` using the syntax *N*`;/path`, where *N* is an integer representing the number of subdirectories *N*-levels deep in which session data can be stored. This is useful if `session.save_handler` is set to `files` and your Web site processes a large number of sessions, because it makes storage more efficient since the session files will be fragmented into various directories rather than stored in a single, monolithic directory. If you decide to take advantage of this feature, note that PHP will not automatically create these directories for you. If you're using a Unix-based operating system, be sure to execute the `mod_files.sh` script located in the `ext/session` directory. If you're using Windows, this shell script isn't supported, although writing a compatible script using VBScript should be fairly trivial.

session.use_cookies (0|1)

Scope: `PHP_INI_ALL`; Default value: `1`

If you'd like to maintain a user's session over multiple visits to the site, you should use a cookie so that the handlers can recall the SID and continue with the saved session. If user data is to be used only over the course of a single site visit, then URL rewriting will suffice. Setting this directive to `1` results in the use of cookies for SID propagation; setting it to `0` causes URL rewriting to be used.

Keep in mind that when `session.use_cookies` is enabled, there is no need to explicitly call a cookie-setting function (via PHP's `set_cookie()`, for example), because this will be automatically handled by the session library. If you choose cookies as the method for tracking the user's SID, then there are several other directives that you must consider, each of which is introduced in the following entries.

session.use_only_cookies (0|1)

Scope: `PHP_INI_ALL`; Default value: `0`

This directive ensures that only cookies will be used to maintain the SID, ignoring any attempts to initiate an attack by passing a SID via the URL. Setting this directive to `1` causes PHP to use only cookies, and setting it to `0` opens up the possibility for both cookies and URL rewriting to be considered.

session.name (string)

Scope: PHP_INI_ALL; Default value: PHPSESSID

The directive session.name determines the cookie name. The default value can be changed to a name more suitable to your application, or can be modified as needed through the session_name() function, introduced later in this chapter.

session.auto_start (0|1)

Scope: PHP_INI_ALL; Default value: 0

A session can be initiated explicitly through a call to the function session_start(), or automatically by setting this directive to 1. If you plan to use sessions throughout the site, consider enabling this directive. Otherwise, call the session_start() function as necessary.

One drawback to enabling this directive is that it prohibits you from storing objects within sessions, because the class definition would need to be loaded prior to starting the session in order for the objects to be re-created. Because session.auto_start would preclude that from happening, you need to leave this disabled if you want to manage objects within sessions.

session.cookie_lifetime (integer)

Scope: PHP_INI_ALL; Default value: 0

The session.cookie_lifetime directive determines the session cookie's period of validity. This number is specified in seconds, so if the cookie should live 1 hour, then this directive should be set to 3600. If this directive is set to 0, then the cookie will live until the browser is restarted.

session.cookie_path (string)

Scope: PHP_INI_ALL; Default value: /

The directive session.cookie_path determines the path in which the cookie is considered valid. The cookie is also valid for all child directories falling under this path. For example, if it is set to /, then the cookie will be valid for the entire Web site. Setting it to /books causes the cookie to be valid only when called from within the http://www.example.com/books/ path.

session.cookie_domain (string)

Scope: PHP_INI_ALL; Default value: empty

The directive session.cookie_domain determines the domain for which the cookie is valid. This directive is a necessity because it prevents other domains from reading your cookies. The following example illustrates its use:

```
session.cookie_domain = www.example.com
```

If you'd like a session to be made available for site subdomains, say customers.example.com, intranet.example.com, and www2.example.com, set this directive like this:

```
session.cookie_domain = .example.com
```

session.serialize_handler (string)

Scope: PHP_INI_ALL; Default value: php

This directive defines the callback handler used to serialize and unserialize data. By default, this is handled by an internal handler called php. PHP also supports a second serialization handler, Web Development Data Exchange (WDDX), available by compiling PHP with WDDX support. Staying with the default handler will work just fine for the vast majority of cases.

session.gc_probability (integer)

Scope: PHP_INI_ALL; Default value: 1

This directive defines the numerator component of the probability ratio used to calculate how frequently the garbage collection routine is invoked. The denominator component is assigned to the directive session.gc_divisor, introduced next.

session.gc_divisor (integer)

Scope: PHP_INI_ALL; Default value: 100

This directive defines the denominator component of the probability ratio used to calculate how frequently the garbage collection routine is invoked. The numerator component is assigned to the directive session.gc_probability, introduced previously.

session.referer_check (string)

Scope: PHP_INI_ALL; Default value: empty

Using URL rewriting as the means for propagating session IDs opens up the possibility that a particular session state could be viewed by numerous individuals simply by copying and disseminating a URL. This directive lessens this possibility by specifying a substring that each referrer is validated against. If the referrer does not contain this substring, the SID will be invalidated.

session.entropy_file (string)

Scope: PHP_INI_ALL; Default value: empty

Those involved in the field of computer science are well aware that what is seemingly random is often anything but. For those skeptical of PHP's built-in SID-generation procedure, this directive can be used to point to an additional entropy source that will be incorporated into the generation process. On Unix systems, this source is often /dev/random or /dev/urandom. On Windows systems, installing Cygwin (http://www.cygwin.com/) will offer functionality similar to random or urandom.

session.entropy_length (integer)

Scope: PHP_INI_ALL; Default value: 0

This directive determines the number of bytes read from the file specified by `session.entropy_file`. If `session.entropy_file` is empty, this directive is ignored, and the standard SID-generation scheme is used.

session.cache_limiter (string)

Scope: `PHP_INI_ALL`; Default value: `nocache`

This directive determines whether session pages are cached and, if so, how. Five values are available:

- `none`: This setting disables the transmission of any cache control headers along with the session-enabled pages.
- `nocache`: This is the default setting. This setting ensures that every request is first sent to the originating server before a potentially cached version is offered.
- `private`: Designating a cached document as private means that the document will be made available only to the originating user. It will not be shared with other users.
- `private_no_expire`: This is a variation of the `private` designation, resulting in no document expiration date being sent to the browser. This was added as a workaround for various browsers that became confused by the `Expire` header sent along when this directive is set to `private`.
- `public`: This setting deems all documents as cacheable, even if the original document request requires authentication.

session.cache_expire (integer)

Scope: `PHP_INI_ALL`; Default value: `180`

This directive determines the number of seconds that cached session pages are made available before new pages are created. If `session.cache_limiter` is set to `nocache`, this directive is ignored.

session.use_trans_sid (0|1)

Scope: `PHP_INI_SYSTEM | PHP_INI_PERDIR`; Default value: `0`

If `session.use_cookies` is disabled, the user's unique SID must be attached to the URL in order to ensure ID propagation. This can be handled explicitly by manually appending the variable `$SID` to the end of each URL, or handled automatically by enabling this directive. Not surprisingly, if you commit to using URL rewrites, you should enable this directive to eliminate the possibility of human error during the rewrite process.

session.hash_function (0|1)

Scope: `PHP_INI_ALL`; Default value: `0`

The SID can be created using one of two well-known algorithms: MD5 or SHA1. These result in SIDs consisting of 128 and 160 bits, respectively. Setting this directive to `0` results in the use of MD5, while setting it to `1` results in the use of SHA1.

session.hash_bits_per_character (integer)

Scope: PHP_INI_ALL; Default value: 4

Once generated, the SID is converted from its native binary format to some readable string format. The converter must know whether each character comprises 4, 5, or 6 bits, and looks to session.hash_bits_per_character for the answer. For example, setting this directive to 4 will result in a 32-character string consisting of a combination of the characters 0 through 9 and a through f. Setting it to 5 results in a 26-character string consisting of the characters 0 through 9 and a through v. Finally, setting it to 6 results in a 22-character string consisting of the characters 0 through 9, a through z, A through Z, "-", and ",". Example SIDs using 4, 5, and 6 bits follow, respectively:

```
d9b24a2a1863780e996e5d750ea9e9d2
fine57lneqkvvqmele7h0h05m1
rb68n-8b7Log62RrP4SKx1
```

session.gc_maxlifetime (integer)

Scope: PHP_INI_ALL; Default value: 1440

This directive determines the duration, in seconds, for which a session is considered valid. Once this limit is reached, the session information will be destroyed, allowing for the recuperation of system resources. By default, this is set to the unusual value of 1440, or 24 minutes.

url_rewriter.tags (string)

Scope: PHP_INI_ALL; Default value: a=href,area=href,frame=src,input=src,form=fakeentry

When session.use_trans_sid is enabled, the SID will automatically be appended to HTML tags located in the requested document before sending the document to the client. However, many of these tags play no role in initiating a server request (unlike a hyperlink or form tag); you can use url_rewriter.tags to tell the server exactly to which tags the SID should be appended. For example:

```
url_rewriter.tags a=href, frame=src, form=, fieldset=
```

Key Concepts

This section introduces many of the key session-handling tasks, presenting the relevant session functions along the way. Some of these tasks include the creation and destruction of a session, designation and retrieval of the SID, and storage and retrieval of session variables. This introduction sets the stage for the next section, in which several practical session-handling examples are provided.

Starting a Session

Remember that HTTP is oblivious to both the user's past and future conditions. Therefore, you need to explicitly initiate and subsequently resume the session with each request. Both tasks are done using the session_start() function.

session_start()

```
boolean session_start()
```

The function `session_start()` creates a new session or continues a current session, based upon whether it can locate a SID. A session is started simply by calling `session_start()` like this:

```
session_start();
```

Note that the `session_start()` function reports a successful outcome regardless of the result. Therefore, using any sort of exception handling in this case will prove fruitless.

■ **Note** You can eliminate execution of this function altogether by enabling the configuration directive `session.auto_start`. Keep in mind, however, that this will start or resume a session for every PHP-enabled page.

Destroying a Session

Although you can configure PHP's session-handling directives to automatically destroy a session based on an expiration time or probability, sometimes it's useful to manually cancel out the session yourself. For example, you might want to enable the user to manually log out of your site. When the user clicks the appropriate link, you can erase the session variables from memory, and even completely wipe the session from storage, done through the `session_unset()` and `session_destroy()` functions, respectively. Both functions are introduced in this section.

session_unset()

```
void session_unset()
```

The `session_unset()` function erases all session variables stored in the current session, effectively resetting the session to the state in which it was found upon creation (no session variables registered). Note that this will not completely remove the session from the storage mechanism. If you want to completely destroy the session, you need to use the function `session_destroy()`.

session_destroy()

```
boolean session_destroy()
```

The function `session_destroy()` invalidates the current session by completely removing the session from the storage mechanism. 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.

Retrieving and Setting the Session ID

Remember that the SID ties all session data to a particular user. Although PHP will both create and propagate the SID autonomously, there are times when you may wish to both retrieve and set this SID manually. The function `session_id()` is capable of carrying out both tasks.

session_id()

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

The function `session_id()` can both set and get the SID. If it is passed no parameter, the function `session_id()` returns the current SID. If the optional `sid` parameter is included, the current SID will be replaced with that value. An example follows:

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

This results in output similar to the following:

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

Creating and Deleting Session Variables

It was once common practice to create and delete session variables via the functions `session_register()` and `session_unregister()`, respectively. These days, however, the preferred method involves simply setting and deleting these variable just like any other, except that you need to refer to it in the context of the `$_SESSION` superglobal. For example, suppose you wanted to set a session variable named `username`:

```
<?php
   session_start();
   $_SESSION['username'] = "jason";
   echo "Your username is ".$_SESSION['username'].".";
?>
```

This returns the following:

```
Your username is jason.
```

To delete the variable, you can use the `unset()` function:

```
<?php
   session_start();
   $_SESSION['username'] = "jason";
   echo "Your username is: ".$_SESSION['username']."<br />";
   unset($_SESSION['username']);
   echo "Username now set to: ".$_SESSION['username'].".";
?>
```

This returns:

```
Your username is: jason.
Username now set to: .
```

Encoding and Decoding Session Data

Regardless of the storage media, PHP stores session data in a standardized format consisting of a single string. For example, the contents of a session consisting of two variables, namely `username` and `loggedon`, is displayed here:

```
username|s:5:"jason";loggedon|s:20:"Feb 16 2006 22:32:29";
```

Each session variable reference is separated by a semicolon, and consists of three components: the name, length, and value. The general syntax follows:

name|s:*length*:"*value*";

Thankfully, PHP handles the session encoding and decoding autonomously. However, sometimes you might wish to execute these tasks manually. Two functions are available for doing so: `session_encode()` and `session_decode()`, respectively.

session_encode()

```
boolean session_encode()
```

The function `session_encode()` offers a particularly convenient method for manually encoding all session variables into a single string. You might then insert this string into a database and later retrieve it, finally decoding it with `session_decode()`, for example.

Listing 18-1 offers a usage example. Assume that the user has a cookie containing that user's unique ID stored on a computer. When the user requests the page containing Listing 18-1, the user ID is retrieved from the cookie. This value is then assigned to be the SID. Certain session variables are created and assigned values, and then all of this information is encoded using `session_encode()`, readying it for insertion into a PostgreSQL database.

Listing 18-1. *Using session_encode() to Ready Data for Storage in a PostgreSQL Database*

```
<?php
   // Initiate session and create a few session variables
   session_start();
   $_SESSION['username

   // Set the variables. These could be set via an HTML form, for example.
   $_SESSION['username'] = "jason";
   $_SESSION['loggedon'] = date("M d Y H:i:s");

   // Encode all session data into a single string and return the result
   $sessionVars = session_encode();
   echo $sessionVars;
?>
```

This returns the following:

```
username|s:5:"jason";loggedon|s:20:"Feb 16 2006 22:32:29";
```

Keep in mind that `session_encode()` will encode all session variables available to that user, not just those that were registered within the particular script in which `session_encode()` executes.

session_decode()

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

Encoded session data can be decoded with `session_decode()`. The input parameter `session_data` represents the encoded string of session variables. The function will decode the variables, returning them to their original format, and subsequently return `TRUE` on success and `FALSE` otherwise. As an example, suppose that some session data was stored in a PostgreSQL database, namely each SID and the variables `$_SESSION['username']` and `$_SESSION['loggedon']`. In the following script, that data is retrieved from the table and decoded:

```
<?php
   // Start the session and retrieve the session ID
   session_start();
   $sid = session_id();

   $conn=pg_connect("host=localhost dbname=corporate
                     user=website password=secret")
                     or die(pg_last_error($conn));

   // Retrieve the user data
   $query = "SELECT data FROM usersession WHERE sid='$sid'";
   $result = pg_query($conn, $query);
```

```
    $sessionVars = pg_fetch_result($result,0,'data');
    session_decode($sessionVars);

    echo "User ".$_SESSION['username']." logged on at ".$_SESSION['loggedon'].".";
?>
```

This returns:

```
User jason logged on at Feb 16 2006 22:55:22.
```

Keep in mind that this is not the preferred method for storing data in a nonstandard media! Rather, you can define custom session handlers, and tie those handlers directly into PHP's API. How this is accomplished is demonstrated later in this chapter.

Practical Session-Handling Examples

Now that you're familiar with the basic functions that make session handling work, you are ready to consider a few real-world examples. The first example shows you how to create a mechanism that automatically authenticates returning registered site users. The second example demonstrates how session variables can be used to provide the user with an index of recently viewed documents. Both examples are fairly commonplace, which should not come as a surprise given their obvious utility. What may come as a surprise is the ease with which you can create them.

■ **Note** If you're unfamiliar with the PostgreSQL server and are confused by the syntax found in the following examples, consider reviewing the material found in Chapter 30.

Auto-Login

Once a user has logged in, typically by supplying a username and password combination that uniquely identifies that user, it's often convenient to allow the user to later return to the site without having to repeat the process. You can do this easily using sessions, a few session variables, and a PostgreSQL table. Although there are many ways to implement this feature, checking for an existing session variable (namely `$username`) is sufficient. If that variable exists, the user can pass transparently into the site. If not, a login form is presented.

■ **Note** By default, the `session.cookie_lifetime` configuration directive is set to `0`, which means that the cookie will not persist if the browser is restarted. Therefore, you should change this value to an appropriate number of seconds in order to make the session persist over a period of time.

Listing 18-2 offers the PostgreSQL table, which is called users for this example. This table contains just a few items of information pertinent to a user profile; in a real-world scenario, you would probably need to expand upon this table to best fit your application requirements.

Listing 18-2. *The users Table*

```
CREATE table users (
   userid serial,
   name varchar(25) NOT NULL,
   username varchar(15) NOT NULL,
   pswd varchar(15) NOT NULL,
   CONSTRAINT users_pk PRIMARY KEY(userid)
);
```

Listing 18-3 contains the snippet used to present the login form to the user if a valid session is not found.

Listing 18-3. *The Login Form (login.html)*

```
<p>
    <form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>">
        Username:<br /><input type="text" name="username" size="10" /><br />
        Password:<br /><input type="password" name="pswd" SIZE="10" /><br />
        <input type="submit" value="Login" />
    </form>
</p>
```

Finally, Listing 18-4 contains the login employed to execute the auto-login process.

Listing 18-4. *Verifying Login Information Using Sessions*

```
<?php
   session_start();
   // Has a session been initiated previously?
   if (! isset($_SESSION['name'])) {
      // If no previous session, has the user submitted the form?
      if (isset($_POST['username']))
      {
         $username = $_POST['username'];
         $pswd = $_POST['pswd'];

         // Connect to the PostgreSQL database
         $conn=pg_connect("host=localhost dbname=corporate
                           user=website password=secret")
               or die(pg_last_error($conn));

         // Look for the user in the users table.
         $query = "SELECT name FROM users
                   WHERE username='$username' AND pswd='$pswd'";
```

```
        $result = pg_query($conn, $query);
        // If the user was found, assign some session variables.
        if (pg_num_rows($result) == 1)
        {
           $_SESSION['name'] = pg_fetch_result($result,0,'name');
           $_SESSION['username'] = pg_fetch_result($result,0,'username');
           echo "You're logged in. Feel free to return at a later time.";
        }
      // If the user has not previously logged in, show the login form
      } else {
         include "login.html";
      }
   // The user has returned. Offer a welcoming note.
   } else {
       $name = $_SESSION['name'];
       echo "Welcome back, $name!";
    }
?>
```

At a time when users are inundated with the need to remember usernames and passwords for every imaginable type of online service, from checking e-mail to library book renewal to reviewing a bank account, providing an automatic login feature when the circumstances permit will surely be welcomed by your users.

Recently Viewed Document Index

How many times have you returned to a Web site, wondering where exactly to find that great PHP tutorial that you nevertheless forgot to bookmark? Wouldn't it be nice if the Web site were able to remember which articles you read, and present you with a list whenever requested? This example demonstrates such a feature.

The solution is surprisingly easy, yet effective. To remember which documents have been read by a given user, you can require that both the user and each document be identified by a unique identifier. For the user, the SID satisfies this requirement. The documents can be identified really in any way you wish, although for the purposes of this example, we'll just use the article's title and URL, and assume that this information is derived from data stored in a database table named `articles`, which is created in Listing 18-5. The only required task is to store the article identifiers in session variables, which is done in Listing 18-6.

Listing 18-5. *The articles Table*

```
create table articles (
   articleid SERIAL,
   title varchar(50) NOT NULL,
   content text NOT NULL,
   CONSTRAINT articles_pk PRIMARY KEY(articleid)
);
```

Listing 18-6. *The Article Aggregator*

```
<?php
   // Start session
   session_start();
   // Retrieve requested article id
   $articleid = $_GET['articleid'];

   // Connect to server and select database
   $conn=pg_connect("host=localhost dbname=corporate
                     user=website password=secret")
         or die(pg_last_error($conn));

   // Create and execute query
   $query = "SELECT title, content FROM articles WHERE articleid='$articleid'";
   $result = pg_query($conn, $query);

   // Retrieve query results
   list($title,$content) = pg_fetch_row($result, 0);

   // Add article title and link to list
   $articlelink = "<a href='article.php?articleid=$articleid'>$title</a>";
   if (! in_array($articlelink, $_SESSION['articles']))
      $_SESSION['articles'][] = "$articlelink";

   // Output list of requested articles
   echo "<p>$title</p><p>$content</p>";
   echo "<p>Recently Viewed Articles</p>";
   echo "<ul>";
   foreach($_SESSION['articles'] as $doc) echo "<li>$doc</li>";
   echo "</ul>";
?>
```

The sample output is shown in Figure 18-1.

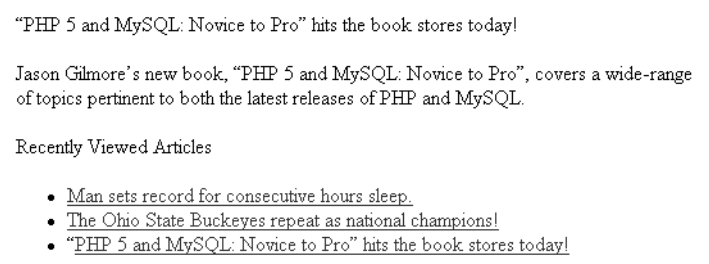

Figure 18-1. *Tracking a user's viewed documents*

Creating Custom Session Handlers

User-defined session handlers offer the greatest degree of flexibility of the three storage methods. But to properly implement custom session handlers, you must follow a few implementation rules, regardless of the chosen handling method. For starters, the six functions in the following list must be defined, each of which satisfies one required component of PHP's session-handling functionality. Additionally, parameter definitions for each function must be followed, again regardless of whether your particular implementation uses the parameter. This section outlines the purpose and structure of these six functions. In addition, it introduces `session_set_save_handler()`, the function used to magically transform PHP's session-handler behavior into that defined by your custom handler functions. Finally, this section concludes with a demonstration of this great feature, offering a PostgreSQL-based implementation of these handlers. You can immediately incorporate this library into your own application, rendering a PostgreSQL table as the primary storage location for your session information.

- `session_open(`*`$session_save_path, $session_name`*`)`: This function initializes any elements that may be used throughout the session process. The two input parameters *`$session_save_path`* and *`$session_name`* refer to the configuration directives found in the `php.ini` file. PHP's `get_cfg_var()` function is used to retrieve these configuration values in later examples.
- `session_close()`: This function operates much like a typical handler function does, closing any open resources initialized by `session_open()`. As you can see, there are no input parameters for this function. Keep in mind that this does not destroy the session. That is the job of `session_destroy()`, introduced at the end of this list.
- `session_read(`*`$sessionID`*`)`: This function reads the session data from the storage media. The input parameter *`$sessionID`* refers to the SID that will be used to identify the data stored for this particular client.
- `session_write(`*`$sessionID, $value`*`)`: This function writes the session data to the storage media. The input parameter *`$sessionID`* is the variable name, and the input parameter *`$value`* is the session data.
- `session_destroy(`*`$sessionID`*`)`: This function is likely the last function you'll call in your script. It destroys the session and all relevant session variables. The input parameter *`$sessionID`* refers to the SID in the currently open session.
- `session_garbage_collect(`*`$lifetime`*`)`: This function effectively deletes all sessions that have expired. The input parameter *`$lifetime`* refers to the session configuration directive `session.gc_maxlifetime`, found in the `php.ini` file.

Tying Custom Session Functions into PHP's Logic

After you define the six custom handler functions, you must tie them into PHP's session-handling logic. This is accomplished by passing their names into the function `session_set_save_handler()`. Keep in mind that these names could be anything you choose, but they must accept the proper number and type of parameters, as specified in the previous section, and must be passed into the `session_set_save_handler()` function in this order: open, close, read, write, destroy, and garbage collect. An example depicting how this function is called follows:

```
session_set_save_handler("session_open", "session_close", "session_read",
                         "session_write", "session_destroy",
                         "session_garbage_collect");
```

The next section shows you how to create handlers that manage session information within a PostgreSQL database. Once defined, you'll see how to tie the custom handler functions into PHP's session logic using `session_set_save_handler()`.

Custom PostgreSQL-Based Session Handlers

You must complete two tasks before you can deploy the PostgreSQL-based handlers:

1. Create a database and table that will be used to store the session data.
2. Create the six custom handler functions.

Listing 18-7 offers the PostgreSQL table `sessioninfo`. For the purposes of this example, assume that this table is found in the database `sessions`, although you could place this table where you wish.

Listing 18-7. *The PostgreSQL Session Storage Table*

```
CREATE TABLE sessioninfo (
   SID CHAR(32) NOT NULL,
   expiration INT NOT NULL,
   value TEXT NOT NULL,
   CONSTRAINT sessioninfo_pk PRIMARY KEY(SID)
);
```

Listing 18-8 provides the custom PostgreSQL session functions. Note that it defines each of the requisite handlers, making sure that the appropriate number of parameters is passed into each, regardless of whether those parameters are actually used in the function.

Listing 18-8. *The PostgreSQL Session Storage Handler*

```
<?php
   /*
   * pg_session_open()
   * Opens a persistent server connection and selects the database.
   */

   function pg_session_open($session_path, $session_name) {

      $conn=pg_connect("host=localhost dbname=corporate
                        user=website password=secret");

   } // end pg_session_open()
```

```
/*
* pg_session_close()
* Doesn't actually do anything since the server connection is
* persistent. Keep in mind that although this function
* doesn't do anything in this particular implementation, it
* must nonetheless be defined.
*/

function pg_session_close() {

   return 1;

} // end pg_session_close()

/*
* pg_session_select()
* Reads the session data from the database
*/

function pg_session_select($SID) {

   $query = "SELECT value FROM sessioninfo
             WHERE SID = '$SID' AND
             expiration > ". time();

   $result = pg_query($query);

   if (pg_num_rows($result)) {

      $row=pg_fetch_assoc($result);
      $value = $row['value'];
      return $value;

   } else {

      return "";

   }

} // end pg_session_select()

/*
* pg_session_write()
* This function writes the session data to the database.
* If that SID already exists, then the existing data will be updated.
*/
```

```
function pg_session_write($SID, $value) {

   $lifetime = get_cfg_var("session.gc_maxlifetime");
   $expiration = time() + $lifetime;

   $query = "INSERT INTO sessioninfo
             VALUES('$SID', '$expiration', '$value')";
   $result = pg_query($query);

   if (! $result) {

      $query = "UPDATE sessioninfo SET
                expiration = '$expiration',
                value = '$value' WHERE
                SID = '$SID' AND expiration >". time();

     $result = pg_query($query);

  }

} // end pg_session_write()

/*
* pg_session_destroy()
* Deletes all session information having input SID (only one row)
*/

function pg_session_destroy($SID) {

   $query = "DELETE FROM sessioninfo
             WHERE SID = '$SID'";

   $result = pg_query($conn, $query);

} // end pg_session_destroy()

/*
* pg_session_garbage_collect()
* Deletes all sessions that have expired.
*/

function pg_session_garbage_collect($lifetime) {

   $query = "DELETE FROM sessioninfo
             WHERE sess_expiration < ". time() - $lifetime;
```

```
        $result = pg_query($query);

        return pg_affected_rows($result);

    } // end pg_session_garbage_collect()

?>
```

Once these functions are defined, they can be tied to PHP's handler logic with a call to `session_set_save_handler()`. The following should be appended to the end of the library defined in Listing 18-8:

```
session_set_save_handler("pg_session_open", "pg_session_close",
                         "pg_session_select",
                         "pg_session_write",
                         "pg_session_destroy",
                         "pg_session_garbage_collect");
```

To test the custom handler implementation, start a session and register a session variable using the following script:

```
<?php
    INCLUDE "pgsessionhandlers.php";
    session_start();
    $_SESSION['name'] = "Jason";
?>
```

After executing this script, take a look at the sessioninfo table's contents using the `psql` client:

```
corporate=# select * from sessioninfo;
+--------------------------------------+-------------------+-------------------+
| SID                                  | expiration        | value             |
+--------------------------------------+-------------------+-------------------+
| f3c57873f2f0654fe7d09e15a0554f08     | 1068488659        | name|s:5:"Jason"; |
+--------------------------------------+-------------------+-------------------+
1 row in set (0.00 sec)
```

As expected, a row has been inserted, mapping the SID to the session variable `"Jason"`. This information is set to expire 1,440 seconds after it was created; this value is calculated by determining the current number of seconds after the Unix epoch, and adding 1,440 to it. Note that although 1,440 is the default expiration setting as defined in the `php.ini` file, you are free to change this value to whatever you deem appropriate.

Note that this is not the only way to implement these procedures as they apply to PostgreSQL. You are free to modify this library as you see fit.

Summary

This chapter covered the gamut of PHP's session-handling capabilities. You learned about many of the configuration directives used to define this behavior, in addition to the most commonly used functions that are used to incorporate this functionality into your applications. The chapter concluded with a real-world example of PHP's user-defined session handlers, showing you how to turn a PostgreSQL table into the session-storage media.

The next chapter addresses another advanced but highly useful topic: templating. Separating logic from presentation is a topic of constant discussion, as it should be; intermingling the two practically guarantees you a lifetime of application maintenance anguish. Yet actually achieving such separation seems to be a rare feat when it comes to Web applications. It doesn't have to be this way!