PHP 101 (part 10): A Session In The Cookie Jar

Patience Pays

Now that you’ve used PHP with MySQL and SQLite, you probably think you know everything you need to get started with PHP programming. In fact, you might even be thinking of cutting down your visits to Zend.com altogether, giving up this series for something flashier and cooler…

Uh-uh. Big mistake.

You see, while built-in database support makes programming with PHP easy, it isn’t the only thing that makes PHP so popular. An easy-to-use XML API and new exception handling mechanism (in PHP 5), support for pluggable modules, and built-in session management are just some of the many other features that make PHP rock. And all these capabilities are going to be explored, in depth, right here in this very series, if you can just find it in yourself to hang around a little longer. So close your eyes, take a deep breath, and read on to find out all about this tutorial’s topic: sessions and cookies.

Party Time

Maybe you heard this at the last party you went to: “*HTTP is a stateless protocol, and the Internet is a stateless development environment*”.

No? Hmmm. Obviously, you don’t go to the right parties.

In simple language, all this means is that HTTP, the HyperText Transfer Protocol that

is the backbone of the Web, is unable to retain a memory of the identity of each client that connects to a Web site, and therefore treats each request for a Web page as a unique

and independent connection, with no relationship whatsoever to the connections that preceded

it. This “stateless environment” works great so long as you’re aimlessly surfing the Web,

but it can cause a serious headache for sites that actually depend on the data accumulated

in previous requests. The most common example is that of an online shopping cart – in

a stateless environment, it becomes difficult to keep track of all the items you’ve

shortlisted for purchase as you jump from one catalog page to another.

Obviously, then, what is required is a method that makes it possible to “maintain state”,

allowing client connections to be tracked and connection-specific data to be maintained.

And thus came about **cookies**, which allow Web sites to store client-specific information on the client system, and access the information whenever required. A cookie is simply a file, containing a series of variable-value pairs and linked to a domain. When a client

requests a particular domain, the values in the cookie file are read and imported into the

server environment, where a developer can read, modify and use them for different purposes.

A cookie is a convenient way to carry forward data from one client visit to the next.

Another common approach is to use a **session** to store connection-specific data; this session data is preserved on the server for the duration of the visit, and is destroyed on its conclusion. Sessions work by associating every session with a session ID (a unique identifier for the session) that is automatically generated by

PHP. This session ID is
stored in two places: on the client using a temporary cookie, and on the server in a
flat file or a database. By using the session ID to put a name to every request received,
a developer can identify which client initiated which request, and track and maintain
client-specific information in session variables (variable-value pairs which remain alive
for the duration of the session and which can store textual or numeric information).
Sessions and cookies thus provide an elegant way to bypass the stateless nature of the HTTP
protocol, and are used on many of today's largest sites to track and maintain information
for personal and commercial transactions. Typically, you use a session to store values that are required over the course of a single visit, and a cookie to store more persistent data that is used over multiple visits.
PHP has included support for cookies since PHP 3.0, and built-in session management since
PHP 4.0. Both these features are enabled by default, so you don't have to do anything
special to activate them. Instead, scroll down and take a look at your first session.

The First Session

One of the standard examples used to demonstrate how a session works is the hit counter application. This is a simple counter that initializes a variable the first time you visit

a Web page, and increments it each time you reload the page. The counter variable is stored

in a session, which means that if you browse to another site and then return, the last

saved value of the counter will be restored (so long as you didn’t destroy the session

by shutting down the browser in the interim).

Take a look at the code:

```
<?php
// initialize a session

session_start();
// increment a session counter

$_SESSION['counter']++;
// print value
echo "You have viewed this page " . $_SESSION['counter'] . " times";
?>
```

To see how this works, request the script above through your browser a few times. You will

notice that the counter increases by 1 on each subsequent page load. If you open up two

browser windows and request the same page in each one, PHP will maintain and

increment

individual session counters for each browser instance. The session ID is used to

identify

which client made which request, and recreate the prior saved environment for

each

individual session. This also means that if you visit one (or more) other Web sites

during

the same session and then return to the script above without shutting down your

browser

in the interim, your previous session will be retrieved and recreated for you.

Every session in PHP begins with a call to the `session_start()` function. This function checks to see whether a session already exists, and either restores it (if it does) or creates a new one (if it doesn't). Session variables can then be registered by adding keys

and values to the special `$_SESSION` superglobal array, and can be accessed at any time during the session using standard array notation. In the example above, a key named

`counter` has been added to the `$_SESSION` array. The first time a

session is created, this key will have the value 0. On every subsequent request for

the page

during the same session, the previous value of the counter will be retrieved and

incremented by 1.

If the example above doesn't work as advertised, check to make sure that the session.save_path variable in your php.ini file points to a valid temporary directory for your system. This value is hard-wired to /tmp

by default, so if you're trying the example on a Windows system, you will need to edit it to

C:\Windows\temp (or your system's temporary directory).

Remember Me

Here's another example, this one asking you to log in and then storing your login name and session start time as two session variables. This information is then used to display the total number of minutes the session has been active.

```
<?php
// initialize a session

session_start();

?>

<html>


<head></head>


<body>
<?php
```

```
if (!isset($_SESSION['name']) && !isset($_POST['name'])) {

    // if no data, print the form

?>

    <form action="<?php echo $_SERVER['PHP_SELF']?>" method="post">
                <input type="text" name="name">


        <input type="submit" name="submit" value="Enter your name">


    </form>

<?php

}

else if (!isset($_SESSION['name']) && isset($_POST['name'])) {
        // if a session does not exist but the form has been submitted

    // check to see if the form has all required values

    // create a new session

    if (!empty($_POST['name'])) {
        $_SESSION['name'] = $_POST['name'];

        $_SESSION['start'] = time();
            echo "Welcome, " . $_POST['name'] . ". A new session has been
activated for you. Click <a href=" . $_SERVER['PHP_SELF'] . ">here</a> to refresh
the page.";
        }

    else {

        echo "ERROR: Please enter your name!";

    }

}

else if (isset($_SESSION['name'])) {

    // if a previous session exists

    // calculate elapsed time since session start and now

    echo "Welcome back, " . $_SESSION['name'] . ". This session was activated
" . round((time() - $_SESSION['start']) / 60) . " minute(s) ago. Click <a
href=" . $_SERVER['PHP_SELF'] . ">here</a> to refresh the page.";
```

```
}

?>

</body>

</html>
```

In this example, the presence or absence of a session variable is used to decide

which

of the three possible screens to display. The session start time is also recorded in

`$_SESSION['start']` with the `time()` function, which returns

the total number of seconds between January 1 1970 and the current time. At a

later

stage, the value stored in `$_SESSION['start']` is compared with the most

current value of `time()` to calculate and display an (approximate)

display of elapsed time.

It’s important to note that the call to `session_start()` must appear first,

before any output is generated by the script (assuming you’re not using PHP’s

output

buffering feature, which you can read about at

http://www.php.net/manual/en/ref.outcontrol.php). This is because the PHP

session handler internally uses in-memory cookies to store session data, and the cookie

creation headers must be transmitted to the client browser before any output. If

you ever

see an error like this in one of your session-enabled pages:

```
Warning: Cannot send session cache limiter - headers already
sent (output started at ...)
```

it’s usually because somewhere, somehow, some output has found its way to the

browser before

`session_start()` was called. Even a carriage return or a blank space outside the

PHP tags surrounding `session_start()` can cause this error, so watch out for them.

As noted previously, every session has a unique session ID, which PHP uses to

keep track of

different clients. This session ID is a long alphanumeric string, which is

automatically

passed by PHP from page to page so that the continuity of the session is

maintained. To see

what it looks like, use the `session_id()` function, as in this simple example:

```php
<?php
// initialize a session

session_start();
// print session ID
```

```
echo "I'm tracking you with session ID " . session_id();
?>
```

When the user shuts down the client browser and destroys the session,

the `$_SESSION`

array will be flushed of all session variables. You can also explicitly destroy a

session – for

example, when a user logs out – by calling the `session_destroy()` function, as in

the following example:

```
<?php
// initialize a session

session_start();
// then destroy it

session_destroy();
?>
```

In case you were wondering if you read that right – yes, before you can call

`session_destroy()` to destroy a session, you must first call

`session_start()` to recreate it.

Remember that `$_SESSION` is a superglobal, so you can use it inside

and outside functions without needing to declare it as global first. The following

simple example illustrates this:

```
<?php
```

```
// initialize a session


session_start();
// this function checks the value of a session variable


// and returns true or false


function isAdmin() {


    if ($_SESSION['name'] == 'admin') {


        return true;
        }


    else {


        return false;


    }


}
// set a value for $_SESSION['name']


$_SESSION['name'] = "guessme";


// call a function which uses a session variable
// returns false here
echo isAdmin()."<br />";
// set a new value for $_SESSION['name']

$_SESSION['name'] = "admin";

// call a function which uses a session variable

// returns true here
echo isAdmin()."<br />";
?>
```

You can read more about sessions and session handling functions at http://www.php.net/manual/en/ref.session.php.

Rules Of The Game

A session works by using an in-memory cookie, which explains why it’s only active while the browser instance that created it is active; once the browser instance is terminated, the memory allocated to that instance is flushed and returned to the system, destroying the session cookie in the process. If you want longer-lasting cookies, you can use PHP’s built-in cookie functions to write data to the user’s disk as a cookie file, and read this data back as and when needed.

Before you start using cookies, there are a few things you should be aware of:

1.Since cookies are used to record information about your activities on a

particular
domain, they can only be read by the domain that created them
2.A single domain cannot set more than twenty cookies, and each cookie is limited
to a maximum size of 4 KB
3.A cookie usually possesses six attributes, of which only the first is mandatory.
Here they are:
•**name:** the name of the cookie

•**value:** the value of the cookie

•**expires:** the date and time at which the cookie expires

•**path:** the top-level directory on the domain from which cookie data can be accessed

•**domain:** the domain for which the cookie is valid

•**secure:** a Boolean flag indicating whether the cookie should be transmitted only over a secure HTTP connection

More information on cookies can be obtained from Netscape, the people who originally invented
them. Visit http://www.netscape.com/newsref/std/cookie_spec.html for the Netscape cookie
specification.

It's important to remember that, since cookies are stored on the user's hard drive, you as
the developer have very little control over them. If a user decides to turn off cookie
support in his or her browser, your cookies will simply not be saved. Therefore, avoid
writing code that depends heavily on cookies; and have a backup plan ready in case cookie

data cannot be retrieved from the client.

With that caveat out of the way, let’s look at some simple cookie-handling code in PHP.

Meeting Old Friends

PHP offers a single function for cookie manipulation: `setcookie()`. This function allows you to read and write cookie files, as demonstrated in the following example:

```
<?php
if (!isset($_COOKIE['visited'])) {

    // if a cookie does not exist

    // set it

    setcookie("visited", "1", mktime()+86400, "/") or die("Could not set cookie");
        echo "This is your first visit here today.";

}

else {

    // if a cookie already exists

    echo "Nice to see you again, old friend!";
}
?>
```

To see how this works, request the page above through your browser a couple of times. The
first time around, because no cookie has yet been set, the first message will be displayed.
On all subsequent attempts, because the cookie has already been set, the client will be
recognized and the second message will be displayed. Note that this works even if you
terminate the browser instance, restart it and visit the page again – a marked difference
from what happened in the session examples you saw earlier.

The `setcookie()` function accepts six arguments: the name of the cookie, its value, its expiry date, the domain, the path for which it is valid, and a Boolean value
indicating its security state. As noted previously, only the name and value are mandatory,
although the example above specifies both a top-level directory and an expiry date for
the cookie (1 day) with the `mktime()` function, which works like the `time()` function described previously.

Cookie values are automatically sent to PHP from the client, and converted to key-value pairs in the $_COOKIE variable, a superglobal array similar to $_SESSION. Values can then be retrieved using standard associative array notation, as in the example above. Note that, as with sessions, calls to setcookie() must take place before any output is generated by the script, or else you'll see an error like this:

```
Warning: Cannot add header information - headers already
sent by (output started at ... )
```

Form And Function

Here's another, slightly more complex example:

```
<?php
if (!isset($_POST['email'])) {

    // if form has not been submitted

    // display form

    // if cookie already exists, pre-fill form field with cookie value

?>

    <html>


    <head></head>
```

```
      <body>



   <form action="<?php echo $_SERVER['PHP_SELF']?>" method="post">


      Enter your email address: <input type="text" name="email" value="<?
php echo $_COOKIE['email']; ?>">
            <input type="submit" name="submit">


      <?php


      // also calculate the time since the last submission


      if ($_COOKIE['lastsave']) {


         $days = round((time() - $_COOKIE['lastsave']) / 86400);
                  echo "<br /> $days day(s) since last submission";


      }


      ?>


   </form>
   </body>

   </html>

<?php
}

else {

    // if form has been submitted

    // set cookies with form value and timestamp

    // both cookies expire after 30 days

    if (!empty($_POST['email'])) {

        setcookie("email", $_POST['email'], mktime()+(86400*30), "/");
        setcookie("lastsave", time(), mktime()+(86400*30), "/");

        echo "Your email address has been recorded.";
        }
    else {

        echo "ERROR: Please enter your email address!";

    }

}

?>

</body>

</html>
```

In this case, the value entered into the form is stored as a cookie called email, and automatically retrieved to pre-fill the form field on all subsequent requests. This technique is frequently used by Web sites that require the user to enter a login name
and password; by automatically pre-filling the username field in the login box with the
value used in the last successful attempt, they save the user a few keystrokes.
This example also demonstrates how you can set more than one cookie for a domain, by

calling `setcookie()` multiple times. In the example above, the time at which

the data was entered is stored as a second cookie, and used to calculate the time

elapsed

between successive entries.

To remove a cookie from the client, simply call `setcookie()` with the same

syntax you used to originally set the cookie, but an expiry date in the past. This

will

cause the cookie to be removed from the client system. Here's an example:

```
<?php
// delete cookie

setcookie("lastsave", NULL, mktime() - 3600, "/");
?>
```

Read more about cookies and the `setcookie()` function

athttp://www.php.net/manual/en/features.cookies.php andhttp://www.php.net/man

ual/en/function.setcookie.php.

Access Granted

As I said at the beginning of this tutorial, cookies and sessions are two different

ways

of making data "persistent" on the client. A session retains data for the duration

of the

session, while a cookie retains values for as long as you need it to. With that in

mind,

let’s now look at an example that uses them both.

The application here is a simple user authentication system, where certain pages can only

be viewed by users who successfully log in to the system. Users who have not been

authenticated with a valid password are denied access to these “special” pages. The list

of valid usernames and passwords is stored in a MySQL database, and PHP is used to verify

a user’s credentials and decide whether or not to grant access.

Assuming the MySQL database table looks like this

```
+-------+----------------------------------------------+
[redacted]
+-------+----------------------------------------------+
[redacted]
| harry | 6e74234b8b552685113b53c7bff0f386c8cef8cf     |
[redacted]
| sam   | bd17f8243e771a57cfbb06aa9a82bbf09fd2d90b     |
[redacted]
+-------+----------------------------------------------+
```

with a unique username field and a password field created with

the `SHA1()` function,

here’s the PHP script that does all the hard work:

```
<?php
```

```
if (isset($_POST['name']) || isset($_POST['pass'])) {

    // form submitted

    // check for required values

    if (empty($_POST['name'])) {
                die ("ERROR: Please enter username!");

    }

    if (empty($_POST['pass'])) {

        die ("ERROR: Please enter password!");

    }
        // set server access variables

    $host = "localhost";
        $user = "test";

    $pass = "test";

    $db = "db2";
    // open connection

    $connection = mysql_connect($host, $user, $pass) or die ("Unable to
connect!");

    // select database

    mysql_select_db($db) or die ("Unable to select database!");

    // create query

    $query = "SELECT * FROM users WHERE name = '" . $_POST['name'] . "' AND
pass = SHA1('" . $_POST['pass'] . "')";

    // execute query

    $result = mysql_query($query) or die ("Error in query: $query. " .mysql_error());

    // see if any rows were returned
```

```
if (mysql_num_rows($result) == 1) {
    // if a row was returned

    // authentication was successful

    // create session and set cookie with username

    session_start();

    $_SESSION['auth'] = 1;

    setcookie("username", $_POST['name'], time()+(84600*30));
    echo "Access granted!";

}

else {

    // no result

    // authentication failed

    echo "ERROR: Incorrect username or password!";

}
```

```
    // free result set memory

    mysql_free_result($result);



    // close connection

    mysql_close($connection);

}

else {

    // no submission

    // display login form

?>

    <html>

    <head></head>
        <body>
    <center>

    <form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>">

    Username <input type="text" name="name" value="<?
php echo$_COOKIE['username']; ?>">
       <p />

    Password <input type="password" name="pass">

    <p />

    <input type="submit" name="submit" value="Log In">

    </center>

    </body>

    </html>

<?php

}
```

`?>`

Here, the values entered into the login box are integrated into a MySQL SELECT query, which
is executed on the user table. If both username and password match, a single record will be
returned, indicating that authentication succeeded; if they don’t, no records will be
returned, indicating that authentication failed.

Assuming authentication succeeds, a session is initialized, the `$_SESSION['auth']` key is created and assigned a value of Boolean `true`, and the username is stored in a cookie for next time. The cookie will remain valid for 30 days, and will be used to
pre-fill the username field in the login box on the next login attempt.

Of course, this isn’t enough by itself. While the script above performs authentication and
initializes both a session and a cookie if the user’s credentials are validated, a security
check must also be carried out on each of the restricted pages. Without this check, any user
could bypass the login screen and simply type in the exact URL to each page to view it.

Since it is clear from the previous script that the session

variable `$_SESSION['auth']`

can only exist if the user’s credentials have been validated, it suffices to check for the presence of the `$_SESSION['auth']` variable at the top of each restricted page, and grant access if that check returns true. Here's how:

```
<?php
// start session
session_start();

if (!$_SESSION['auth'] == 1) {

    // check if authentication was performed

    // else die with error

    die ("ERROR: Unauthorized access!");

}

else {

?>
      <html>


    <head></head>


    <body>
    This is a secure page. You can only see this if $_SESSION['auth'] = 1

    </body>

    </html>

<?php

}
?>
```

Pretty neat, huh? Only authenticated users will be able to see this page, because only

their clients will have a session with the `$_SESSION['auth']` variable in it.

Everyone else will simply see an error message.

That’s about it for this tutorial. In Part Eleven,
I’ll be telling you all about SimpleXML, the new XML processing toolkit that comes bundled with PHP 5. Make sure you come back for that!