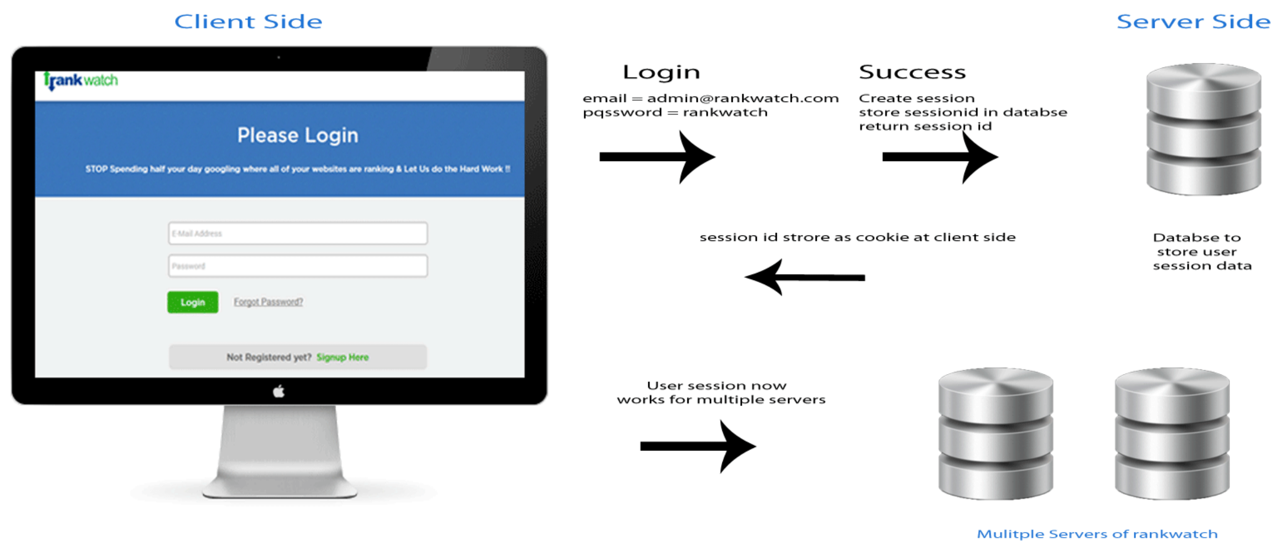
Working with Cookies and Sessions

medium.com/spacecode/working-with-cookies-and-sessions-794d5c071349

March 21, 2019

HTTP is a stateless protocol, which means that any data you have stored is forgotten when the page has been sent to the client and the connection is closed. Eventually, Netscape invented the cookie — a tiny bit of information that a website could store on the client's the machine that was sent back to the website that had written it, meaning that it was a secure way to store information across pages.


Client Side
Server Side
rankwatch
Please Login
STOP Spending half your day googling where all of your websites are ranking & Let Us do the Hard Work !!
E-Mail Address
Password
Login
Forgot Password?
Not Registered yet? Signup Here
Login
email = admin@rankwatch.com
pqssword = rankwatch
Success
Create session
store sessionid in databse
return session id
session id strore as cookie at client side
Databse to store user session data
User session now works for multiple servers
Mulitple Servers of rankwatch

PHP Session Working

Cookies earned a bad name at first because they allowed people to track how often a visitor came to their site and what they did while there, and many people believed that cookies signalled the end of privacy on the Web. Urban myths popped up saying that cookies could read any information from your hard drive, and people were encouraged to disable cookies across the board. The reality is that cookies are harmless, and fortunately for us, are now commonly accepted.

Sessions grew up from cookies as a way of storing data on the server side, because the inherent problem of storing anything sensitive on clients' machines is that they are able to tamper with it if they wish. In order to set up a unique identifier on the client to the server, and corresponds to a data file on the server.

COOKIES VERSUS SESSIONS

Both cookies and sessions are available to you as a PHP developer, and both accomplish the same task of storing data across pages on your site. However, there are differences between the two.

Cookies can be set to a long lifespan, which means that data stored in a cookie can be stored for months, if not years. Cookies, having their data stored on the client, work smoothly when you have a cluster of web servers, whereas sessions are stored on the server, meaning if one of your web servers handles the first request, the other web servers in your cluster will not have the stored information. Cookies can also be manipulated on the client side, using Javascript, whereas sessions cannot.

Sessions are stored on the server, which means the client does not have access to the information you store about them. This is particularly important if you store shopping baskets or other information you do not want your visitors to be able to edit by hacking their cookies. Session data, being stored on your server, does not need to be transmitted which each page, clients just need to send an ID, and the data is loaded from the local file.

Finally, sessions can be any size you want because they are held on your server, whereas many web browsers have a limit on how big cookies can be to stop rouge web sites chewing up gigabytes of data with meaningless cookie information. Sessions rely upon a client-side cookie to store the session identifier—without this, PHP must resort to placing the identifier in the URL, which is insecure. If a cookie is used, it is set to expire as soon as the user closes his browser.

Cookies versus sessions usually come down to one choice, do you want your data to work when your visitor comes back the next day? If so, then your only choice is cookies. If you are storing sensitive information, store it in a database and use the cookies to store an ID number to reference the data. If you do not need semi-permanent data, then sessions are generally preferred—they are a little easier to use, do not require their data to be sent in entirety with each page, and are also cleaned up as soon as your visitor closes his web browser.

USING COOKIES

The setcookie() call needs to be before the HTML form because of the way the web works. HTTP operates by sending all “header” information before it sends “body” information. In the header, it sends things like server type example: “Apache”, page size, and other important data. In the body, it sends the actual HTML you see on the screen. HTTP works in such a way that header data cannot come after body data—you must send all your header data before you send anybody data at all.

Cookies come into the category of header data. When you place a cookie using setcookie(), your web server adds a line in your header data for that cookie. If you try and send cookies after you have started sending HTML, PHP will flag a serious error and the cookies will not get placed.

There are two types to correct this:

1. Put your cookies at the top of your page. By sending them before you send anybody data, you avoid the problem entirely.
2. Enable output buffering in PHP. This allows you to send header information such as cookies whenever you like—even after body data.

The setcookie() function itself takes three main parameters: the name of the cookie, the value of the cookie, and the data the cookie should expire. For example:

```
setcookie("Name", $_POST['Name'], time() + 315360000)
```

In that example code, setcookie() sets a cookie called Name to the value set in a form element called Name. It uses time() + 315360000 as its third parameter, which is equal to the current time in seconds plus the number of seconds in a year so that the cookie is set to expire one year from the time it was set.

Once set, the Name cookie will be sent with every subsequent page request, and, PHP will make it available in $_COOKIE. Users can clear their cookie manually, either by using a special option in their web browser or just by deleting files.

The last three parameters of the setcookie() function allow you to restrict when it's sent, which gives you little more control:

1. Parameter four which is path allows you to set a directory in which the cookie is active.
2. Parameter five which is domain allows you to set a subdomain in which the cookie is active.
3. Parameter six which is secure lets you specify whether the cookie is active.

Once a cookie has been sent, it becomes available to use on the subsequent page looks through the $_COOKIE superglobal array variable. Using the previous call to setcookie(), subsequent page loads can have their Name value read like this:

USING SESSIONS

Sessions store temporary data about your visitors and are particularly good when you don't want that data to be accessible from outside of your server. They are an alternative to a cookie if the client has disabled cookie access on her machine because PHP can automatically rewrite URL to pass a session ID around for you.

STARTING A SESSION

A session is a combination of a server-side file containing all the data you wish to store, and a client-side cookie containing a reference to the server data. The file and the client-side cookie are created using the function session_start()—it has no parameter but informs the server that sessions are going to be used.

When you call session_start(), PHP will check to see whether the visitor sent a session cookie. If it did, PHP will load the session data. Otherwise, PHP will create a new session file on the server, and send an ID back to visitors to associate the visitor with the new file. Because each visitor has his own data locked away in his unique session file, you need to call session_start() before you try to read session variable—falling to do so will mean that you simply will not have access to this data. Furthermore, as session_start() needs to send the reference cookie to the user's computer, you need to have it before the body of your web page—even before any spaces.

ADDING SESSION DATA

All your session data is stored in the session superglobal array, $_SESSION, which means that each session variable is one element in that array, combined with its value. Adding the variable to this array is done in the same way as adding variables to an array, with the added bonus that session variables will still be there when your user browses to another page.

To set a session variable, use syntax like this:

```
$_SESSION['var'] = $val;
$_SESSION['firstname'] = 'Jim';
```

Older versions of PHP used the function session_register(); however, use of this function is strongly discouraged, as it will not work properly in default installations of PHP 5. If you have scripts that use session_register(), you should switch them over to using the $_SESSION superglobal, as it is more portable and easier to read.

Before you can add any variables to a session, you need to have already called the session_start() function.

READING SESSION DATA

Once you have put your data away, it becomes available in the $_SESSION superglobal array with the key of the variable name you gave it. Here is an example of setting data and reading it back out again:

Unlike cookies, sessions data is available as soon as it sets.

REMOVING SESSION DATA

Removing a specific value from a session is as simple as using the function unset(), just as you would for any other variable. It is important that you unset only specific elements of the $_SESSION array, not the $_SESSION array itself, because that would leave you unable to manipulate the session data at all. To extend the previous script to remove data, use this:

```
$_SESSION['foo'] = 'bar';
print $_SESSION[''foo];
unset($_SESSION['foo']);
```

ENDING A SESSION

A session lasts until your visitor closes their browser—if he/she navigates away to another page, then return to your site without having closed their browser, their session will still exist. You visitor’s session data might potentially last for days, as long as they keep browsing around your site, whereas cookies usually have a fixed lifespan.

If you want to explicitly end a user’s session and delete their data without them having to close their browser, you need to clear the $_SESSION array, then use the session_destroy() function. The session_destroy() function removes all session data stored on your hard disk, leaving you with a clean slate. To end a session and clear its data, use this code:

```
session_start();
$_SESSION = array();
session_destroy();
```

There are two important things to note here. First, session_start() is called so that PHP loads the user’s session, and second, we use an empty call to the array() function to make $_SESSION an empty array—effectively wiping it. If session_start() is not called, neither of the following two lines will work properly, so always call session_start().

CHECKING SESSION DATA

You can check whether a variable has been set in a user’s session using isset(), as you would a normal variable. Because of the $_SESSION, superglobal is only initialized once session_start() has been called, you need to call session_start() before using isset() on a session variable.

FILES VERSUS DATABASE

The session-handling system in PHP is actually quite basic at its core, simply storing and retrieving values from flat files based upon unique session IDs handed out when a session is started. While this system works very well for small-scale solutions, it does not work too well when multiple servers come to play. The problem is down to location, where should session data be stored?

If session data is stored in files, the files would need to be in a shared location somewhere—not ideal for performance or locking reason. However, if the data is stored in a database, that database could then be accessed from all machines in the web server cluster, thereby eliminating the problem. PHP's session storage system was designed to be flexible enough to cope with this situation.

To use your own solution in place of the standard session handlers, you need to call the function session_set_save_handler(), which takes several parameters. In order to handle sessions, you need to have your own callback functions that handle a set of events, which are:

1. Session open (called by session_start())
2. Session close (called at page end)
3. Session read (called after session_Start())
4. Session write (called when session data is to be written)
5. Session destroy (called by session_destroy())

To handle these six events, you need to create six functions with very specifc numbers of functions and return type. Then you pass these six functions into session_set_save_handler() in that order, and you are all set. This sets up all the basic functions, and prints out what gets passed to the function so you can see how the session operations work:

```
}
```

```
function sess_close() {
  print "Session closed.\n";
  return true;
 }

function sess_destroy ($sess_id) {
  print "Session destroy called.\n";
  return true;
 }

function sess_gc($sess_maxlifetime) {
  print "Session garbage collection called.\n";
  print "sess_maxlifetime: $sess_maxlifetime\n";
  return true;
 }

session_set_save_handler("sess_open", "sess_close", "sess_read", "sess_write",
"sess_destroy", "sess_gc");

session_start();
 $_SESSION['foo'] = 'bar';
 print "Some text \n";
 $_SESSION['baz'] = 'wombat';
```

That will give the following output:

```
Session opened.
Sess_path: /tmp
Sess_name: PHPSESSID
Session read.
Sess_ID: nksjfdio2334in1ln32
Some Text
Session value written.
Sess_ID: nksjfdio2334in1ln32
Data: foo|s:3 "bar"; baz| s:6: "wombat";
Session closed.
```

There are four important thing to note in that example:

1. You can, if you want, ignore the parameter passed into sess_open(). We are going to be using a database to store our session data, so we do not need the values at all.
2. Writing data comes just once, even though our two writes to the session are nonsequential—there is a print statement between them.

3. Reading data is done just once, and passes in the session ID.
4. All the functions return true except sess_read().

Item 1 is not true if you actually care about where the user asks you to save file. If you are using your own session filesystem, you might want to actually use $sess_path when it gets passed in—this is your call.

Item 2 and 3 are important, as they show that PHP only does its session reading and writing once. When it writes, it gives you the session ID to write and the whole content of that session, when it reads, it just gives you the session ID to read and expects you to return the whole session data value.

The last item shows that sess_read() is the one function that needs to return a meaningful value to PHP. All the others just need to return true, but reading data from a session needs to either return the data or return an empty string. ''.

What we are going to do is use MYSQL as our database system for session data using the same functions as those above—in essence, we are going to modify the script so that it actually works.

The ID field is not required, as it is not likely we will ever need to manipulate the database by hand.

Now, before you try this next code, you need to tweak two values in your php.ini file: session.gc_probability and session.gc_maxlifetime. The first one, in tandem with session.gc_divisor, sets how likely it is for PHP to trigger session clean up with each page request. By default, session.gc_probability is 1 and session.gc_divisor is 1000, which means it will execute session clean up once in every 1000 script. As we are going to be testing our script out, you will need to change session.gc_probability to 1000, giving us a 1000/1000 chance of executing the garbage collection routine. In other words, it will always run.

The second change to make is to lower session.gc_maxlifetime. By default, it is 1440 seconds, which is far too long to wait to see if our garbage collection routine works. Set this value to 20, meaning that when running our garbage collection script, we should consider everything older than 20 seconds to be unused and delectable. Of course, in production scripts, this value needs to be set back to 1440 so that people do not get their sessions timings out before they can even read a simple web page!

With that in mind, here's the new script!

```
mysql_connect("localhost", "phpuser", "jack837");
 mysql_select_db("phpdb");

function sess_open($sess_path, $sess_name) {
  return true;
 }

function sess_close() {
  return true;
 }

function sess_read($sess_id) {
  $result = mysql_query("SELECT data FROM session WHERE SessionID =
'$sess_id';");
  $currentTime = time();
  if (!mysql_num_rows($result)) {
   mysql_query("INSERT INTO sessions (SessionID, DateTouched) VALUES ('$sess_id',
$currentTime);");

return '';
  } else {
   extract(mysql_fetch_array($result), EXTR_PREFIX_ALL, 'sess');
   mysql_query("UPDATE session SET DateTouched = $currentTime" WHERE
SessionID = '$sess_id');

return $ses_Data;
  }
 }

function sess_write($sess_id, $data) {
  $currentTime = time();
  mysql_query("UPDATE session SET Data = '$data', DateTouched = $currentTime
WHERE SessionID = '$sess_id';");
  return true;
 }
```

```
fucntion sess_destroy($sess_id) {
  mysql_query("DELETE FROM session WHERE SessionID = '$sess_id';");
  reutnr true;
 }

function sess_gc($sess_maxlifetime) {
  $currentTime = time();
  mysql_query("DELETE FROM sessions DateTouched + $sess_maxlifetime <
$currentTime;");
  return true;
 }

session_set_save_handler("sess_open", "sess_close", "sess_read", "sess_write",
"sess_destroy", "sess_gc");
 session_start();

$_SESSION['foo'] = 'bar';
 $_SESSION['baz'] = 'wombat';
```

As that script starts, it forms a connection to the local SQL server, which is used through the script for the session-handling functions. When a session is read, sess_read() is called and given the session ID to read. This is used to query our session table—if the ID exists, its value is returned. If not, an empty session now is created with that session ID and an empty string is returned. The empty row is put in there so that we can later say UPDATE while writing and will not need to bother with whether the row exists already; we will know we created it when reading. The sess_write() function updates the session ID $sess_id so that it holds the data passed in with $data.

The last function of interest is sess_gc(), which is called randomly to handle deletion of old session information. We edited php.ini so that randomly means “every time” right now, and this function receives the lifespan in second of session data and deletes all rows that have not been read or updated in that time. We can tell how long it has been since a row was last read/written because both sess_read() and sess_write() update the DateTouched filed to the current time. Therefore, to tell whether or not a record was touched after the garbage collection time limit, we simply take DateTouched and add the time limit $sess_maxlifetime to it—if that value is under the current time, the session data is no longer valid.

It is interesting to note that you need not to use a database or files to store your session. As we have seen, you get to define the storage and retrieval method for your system, so if you really want, you could write your own extension called PigeonStore that sends and retrieves session data through pigeons. It really doesn’t matter, because PHP just calls the function you tell it to, what you do in there is to you, so use it wisely.