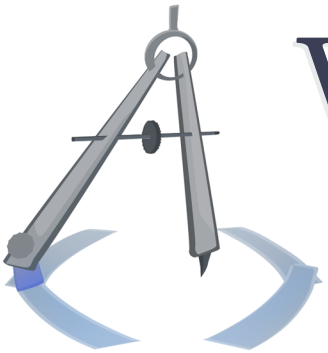


Website Architecture

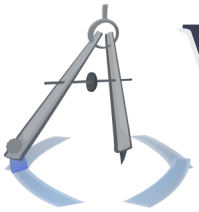
Lezione 11

Sessions

Introdues the concept of sessions, which allow persistent data to be stored on the server for each site user.

Michael Serritella

Summer 2010



Intro to sessions

Using sessions in PHP, you can keep variables persistently across different page views by the same user. They offer the same kind of persistence as cookies - particularly because they are built on top of cookies! Sessions are extremely useful for dynamic sites and offer a level of persistence just below a database, for when data is important to keep across page views but not important enough to keep between.. site views (?) (or, as the name suggests, "sessions").

Concept

Using sessions, you can store variables across page views which correspond to a particular visitor. You will be able to access the variables just as easily as with cookies, GET, or POST information. There is a superglobal array called `$_SESSION` which stores all of the variables that you want to save. You turn the session on or off via function calls, and a cookie is used to bootstrap this process.

If all of the data is stored on the server, then how are cookies used? Let's zoom out a little. Let's look at the old way we might have done this, using cookies more simply. If we want to store data between page views, the user has to send it to us every time as part of the request (as the cookie data is embedded within a HTTP request header). So, they're not giving it to us every time anymore. We want to keep it on the server. But now the problem is identifying each page request as a distinct, familiar user. We know that request information is unreliable, so we can't track the IP address or the referrer (e.g. "If he's coming from 'somesite.com/memberPage/Bob', then he must be Bob!").

The most prominent alternative is the idea of token passing. We give the user a token - something unique - and they show it to us every time in order to prove their identity. In computer science, all data are essentially integers, so why not use a large integer? Sure; let's do that. This number is called the **Session ID**, abbreviated to **sessid** by people who have seen too many movies or who have never engineered software for use by another person.

So how would this work? We keep a set of data for each user, with each set identified uniquely by the session ID. Maybe each session has its own folder in a filesystem; maybe each of them is an entry in a database. It doesn't matter; the point is that this is a viable scheme. Most commonly, it is implemented with a single variable in the cookie that is called **PHPSESSID** (sigh).

Advantages

Using sessions as opposed to cookies has many advantages. We will go over some of them now from a basic perspective.

Network economy

Using cookies, all of the data in the cookie has to be sent with every request to the applicable domain. Even if you're just requesting a JPG or a CSS file. This can easily get out of hand, and it is plainly inappropriate for requesting resources which do not use the cookie (if it doesn't need the cookie data to make an informed decision, then what's the point?). This is very well mitigated, at least, by the use of sessions. The cookie overhead is minimal, which is great for most requests and still OK for requests which don't use the cookie (as long as the request still fits in one packet, it's fine).

Privacy

Using cookies, the data is visible both on the user's hard disk and over the network connection, since it is being sent over the wire. This would be a candy store for someone sniffing around your computer, and it's a possible exploit for remote hackers who are well motivated. With sessions, however, all data is kept on the server side, so privacy is greatly improved.

Tolerance of variable complexity

Using PHP sessions, variables of basically any size and complexity can be stored within `$_SESSION`. That includes objects.. arrays.. arrays of arrays.. etc. Thus, you are encouraged to use data structures with a higher level of abstraction, allowing more natural software-engineering choices. With cookies, you could hardly justify this.

Disadvantages?

These aren't disadvatages vs. cookies, exactly, but perhaps they are weak points of the scheme in general or vs. a more complex tool, like a database.

In their default implementation, the session data is stored on the same Web server as PHP. So there is a RAM issue and even a slight disk-performance issue. That is configurable, but it may take a little effort. Even more accurately, you can say that sessions may cause a strain in the RAM usage of the server, wherever they are stored. No matter what its persistent storage scheme is, the `$_SESSION` data is stored all at once and retrieved

all at once; it is a package deal. Most people don't consider this, but you may want to consider making session data as compact and relevant as possible. Every page which uses sessions will load & construct `$_SESSION`, even if it uses only one variable in there. Just be aware of this; don't go nuts if you have a performance-sensitive site.

As we will see later, sessions open their own security holes, though they are still preferable to cookies. They are not a complete solution for privacy or authentication.

Protocol & procedure

Now, how does this work in PHP? What data is sent back and forth? How do we get started? In this section, we look at these questions, and we will see the PHP code necessary to use sessions - how to turn them on, turn them off, retrieve data, etc.

Token passing methodology

Before we do that, we should look at the way we actually do this token passing. There are two options in PHP. There is the cookie method, which we have seen, where the session ID is given as a cookie variable. This is the preferred method.

The other method is by passing it as a GET variable. PHP can do this "automatically", and it is an option if you suspect that the user does not allow cookies. What PHP actually does is add an "output filter", which postprocesses the text you write via `print()`. It buffers the text, checks everything to see whether it's an `<a>` tag, etc., and then rewrites the URIs to contain an extra GET variable.

The latter, sometimes called "transparent SID support", actually opens up more security holes and is significantly less efficient for the server. For example, what if someone sees your most recent URIs, either from a referrer, from your browser history, or from a link you mistakenly post on a forum? It's trouble. You should basically be aware of this method so you know how to avoid it.

Configuration options

The sessions module is configurable, and we will mention the toggling of options from time to time. You should see the **documentation** for these options. Set them as you would set any configuration option, such as with `ini_set()`.

It is recommended that you assert **session.use_only_cookies** and make sure to deassert **session.use_trans_sid**, for instance. If you're allowed to set these in php.ini files and you have access to those files on your server, then use that method. Else, make sure to set these configuration options at the top of your scripts, before you use session functions. This is a common theme with the other configuration options related to sessions.

Starting a session

OK, so let's start a session. In the most common case, this is easy enough: use `session_start()` (**docs**). Make sure to call this before you start writing the text of your page.

Once you start a session, you either create a new session ID (and thus new session variables) or reuse the ID given in the request (and associated data). Thus, this is called in every page that uses sessions. It is *not only for starting a session*; it is also for *continuing a session*.

You can retrieve the session ID with `session_id()` (**docs**). You can also use this function to set an arbitrary session ID, and a related function is `session_regenerate_id()` (**docs**). Resetting the session ID is a sort of annoying process, so be sure to read the comments on the documentation, as users have experimented with PHP's flaws and given solutions.

On top of that, you can have an additional layer of abstraction: you can have multiple sessions at once per each user, distinguished by different session names. These names are ef-

fectively the name of the cookie variable used to store the session ID; default is **PHPSESSID**. Use `session_name()` (docs) for this purpose, and check its page comments.

Accessing data

Once you have started or continued an existing session, accessing data is very simple. You can treat the `$_SESSION` variable like any other array - add elements to it; search its keys; do whatever. You may read from and write to this variable.

Example:

PHP

```php
<?php

session_start();
print("Hey - this is from last time: ".$_SESSION["fromLastTime"]."\n");
$_SESSION["forNextTime"] = array(1, 3, 5);

?>
```


Committing data

We can save data for next time using `$_SESSION`, but when does it really get written to persistent storage? It turns out that, by default, this data gets written when the script terminates. The data is locked in the meantime, so other scripts can't access it. You probably don't want this kind of coarse-grain locking. You probably want to commit your changes and let go of this information as soon as possible. To do so, use `session_write_close()` (docs) or its less-idiotically-named alias, `session_commit()` (docs).

Apparently, one of the motivations for this function goes back to the days of HTML frames. If each frame uses the session data, users would experience each frame loading in serial! This was because only one frame had a lock on the session data at a time (which is apparently still true) and it didn't let go of the lock until the page was finished writing (which you can now change).

Closing & destroying the session

You may want to terminate the session from the server side. This is actually a two-step process. One: You have to destroy the data in `$_SESSION`. Two: You should probably make sure that the client discards the session ID. To accomplish the former, use `session_destroy()` (docs), and be sure to read its documentation and comments. To accomplish the latter, you will need to unset the cookie variable which held the session ID using `setcookie()`. This is shown in the `session_destroy()` documentation.

Example uses

Now, when would you want to use a session? What exactly does it mean to have data that is persistent but not quite important enough for a database[1]? Let's see a few examples.

User login This is the canonical example. Your site may have its own user-account system, and sessions are the standard way of keeping track of whether the user is logged in. If a user has logged in, you typically store some flags to that effect, along with the user's account name, in `$_SESSION`.

Shopping carts Shopping carts have to be the 2nd-most-popular use of sessions. Keep an array of objects which represent the items in the cart, for instance. Boom.

Browsing history Increasingly, you will find that mega-stores like Amazon and eBay will keep track of your most-recently-viewed products, maybe to offer suggestions of different products. This browsing history is (ostensibly) kept in the same manner as with PHP sessions; even if they don't use PHP, the principles are the same.

[1] Using a database involves (more) persistent disk storage and thus consumption of space, more communication overhead, and taking resources away from other database tasks.

Implementation & lower-level details

The knowledge presented so far is good enough for maybe 73% of sites. But often enough, you will want to know more, such as how to customize the storage engine, how to customize how complex objects are stored, or how to resolve strange client-side issues.

Consequences of using cookies

Sessions are built on top of cookies. Therefore, they inherit the advantages and disadvantages of cookies in general.

Detecting if cookies are disabled

On your site, sessions may be used for convenience (e.g. browsing history) or for mandatory functionality (e.g. user-account login). If they are necessary and you are using cookie-based sessions, you should give some kind of message to the user if they have cookies disabled. Ideally, right?

It turns out that there's no direct test to see if cookies are disabled. You have to probe around. The basic way of doing this is to try to set a cookie value and then check if it exists later. Perhaps you can set the test value on every page of your site and then the login page checks to see if the test value exists; else it reports an error to the user. Or you can design something more nuanced. But the basic method is to test & check.

Cookie parameters

Cookies can be parameterized for a specific domain and subdirectory, and they can have expiration dates. The same applies to the cookies which are used for sessions. See `session_set_cookie_params()` (docs) for options.

Sending of the Set-Cookie header

When you call `session_start()`, the `Set-Cookie` header is sent. If you happen to call `session_start()` twice in a script, then the header may be written twice, and then the client has to handle this properly when it receives two versions of the header within the same response. Some rare clients apparently have

a problem with this, so at the very least, you should be aware of this problem. You will see why this is relevant when you see some of the security problems with sessions - namely session fixation.

Session expiration

PHP has a garbage collector in its session module, which may discard session data if it has not been written within a customizable period of time. See the **session.gc_maxlifetime** configuration option, among others, to change this. Or you can keep updating some data in `$_SESSION` every once in a while.

Custom saving & restoring of objects

By default, the session module will handle saving & restoring objects of user-created classes, and it works well most of the time[2]. It just writes out the state of your object into a string and then rebuilds the object from the string when the session is restored. This process is called **serialization**; serialization does not imply string data - it could be any binary data - but this is

[2]The documentation admits that it cannot handle objects with circular references; this is usually not a problem.

often the case. It is conceivable that you don't want to serialize everything. Perhaps some fields of your object are not important or do not need to be saved from one page view to the next.

In order to customize this, implement the magic methods `__sleep()` and `__wakeup()`; see the documentation **here**.

Customizing the storage engine

So PHP serializes your data into a string. Then what? By default, the string gets stored in a file. If you want, you can store this file in a particular place, or you can even store it using another media entirely. You can store it in a database, for example, which may afford you some flexibility, like the ability to keep session data shared across a distributed architecutre (quite advanced for us now, but keep it in mind).

If you use the default file storage, you can change the file's directory fairly simply; use `session_save_path()` (**docs**). One user posted a solid suggestion in the comments:

```php
<?php

session_save_path
    (realpath(dirname($_SERVER['DOCUMENT_ROOT']).'../someSessionDir'));

?>
```

If you want to completely customize the way that session data is stored, use the function **`session_set_save_handler()`** (**docs**).

Sessions on shared servers

If you are on a shared server *and you don't have your own domain* (e.g. "whatever.com/~oneuser", "whatever.com/~anotheruser"), what if your site uses sessions and so does another site on that server? A user may visit both. In that case, you have some server-side and some client-side issues to worry about.

By default, PHP may be configured to save sessions in a generic directory, like "/tmp". If many sites use the same directory for sessions, it's possible to have a "collision" of the same session IDs or for a malicious programmer to access sessions from other

sites on the same server. You should use `session_save_path()` as directed above; put the session directory somewhere in your own personal webspace.

From the client side, the user would probably have one **PHPSESSID** variable in his/her cookie for "whatever.com". You should distinguish your session's variable name within the cookie by using `session_name()`; pick something unique to your site name. And limit the cookie to your subdirectory, why don't you.

Security & correctness issues

Sessions are overall a win, but they open up a few of their own security problems, the two largest of which are **session fixation** and **cross-site request forgery** (called **XSRF** or **CSRF**).

You can view much more about these vulnerabilities online, especially at the Open Web Application Security Project (**1** (**1b**) and **2**).

The most important thing to know here is that PHP's documentation acknowledges session fixation and recommendations SSL to fix it, which is ridiculous. Look at OWASP for an actually well-engineered solution.