Advanced PHP: Web Applications - Sessions and Authentication

Tobias Ratschiller

Table of Contents

1. Web Applications

To understand the implications of Web application concepts, you need to differentiate between *applications* and single *scripts*. A script is a utility, and as such doesn't have any context. It doesn't know about other scripts in your system. An application, however, is designed to perform more advanced tasks. An application needs to *maintain state* and execute *transactions*. As most applications are made for user, they need to manage different classes of users and therefore need to authenticate them.

2. Sessions

Session management is a mechanism to maintain state about a series of requests from the same user across some period of time.

The problem is, that HTTP has no mechanism to maintain state; HTTP is a context-free or stateless protocol. Individual requests aren't related to each other. The Web server (and thus PHP) can't easily distinguish between single users and doesn't know about user sessions. Therefore, we need to find our own way to identify a user and associate session data (that is, all the data you want to store for a user) with the user. We use the term *session* for a single visit of a user. For example, a typical online shopping session might include logging in, putting an item into the shopping cart, going to the checkout page, entering address and credit card data, submitting the order, and closing the browser window.

To associate session data with a user, you need a *session identity number* - a key that ties the user to his data. But, as mentioned, HTTP lacks a mechanism to identify users; what should you use, then, to brand the user?

PHP's Built-In Session Library

Luckily, PHP 4.0 has basic session management built in, which frees you from the task of inventing session IDs, serializing and storing session data. While it's very easy and straightforward to use and may suffice for your needs, it lacks some of the advanced features that the PHPLib provides.

Figure 1. A basic example of using PHP's built-in sessions

```
session_start();
print($counter);
$counter++;
session_register("counter");
```

This example displays the session ID and a counter that increments each time you access the page. Of course, this example is different from a normal page counter - the session (and thus the counter) is tied to one specific user. With PHP's default configuration, the session cookie has a lifetime of `0`; if you close the browser and reopen it, the counter restarts from zero, as the cookie has been deleted.

Let's take a closer look at the PHP 4 session functions. PHP's session management library offers the characteristics every session management library needs to have:

- It stores session data on the server. Because the library uses different storage modules, you can keep the data in plain text files, shared memory, or databases. This reflects exactly what we've explained about storage media - where exactly data is being kept is not really important (well, as long the performance of the medium is sufficient).
- It uses a cryptographically random session ID to identify a user.
- It saves the session ID (and only the session ID) on the client side using cookies, `GET/POST`, or the script path. (The PHP library provides all of these methods; we show how to use them a little later.)
- If the user has disabled cookies, the application can use other means of session propagation.

A Session's Life

A PHP 4 session is started by calling `session_start()` or implicitly as soon as you register a session variable with `session_register()`. On start-up, PHP checks whether a valid session ID exists (passed by one of the methods outlined in the later section "Session ID Propagation"). If there's no session ID, a new ID is created. This is also the case when the session is refused and marked as invalid because the HTTP referrer for the page comes from a non-local site and `extern_referer_check` (note the single "r") is enabled in the PHP configuration. This introduces some additional security, as it prevents users coming from other PHP sites taking over a session (which is still highly improbable, however, due to the algorithm used for the generation of the session ID).

If a valid ID exists, the frozen variables of that session are reactivated and introduced back to the global namespace. It's as easy to handle session variables as it is to handle `GET/POST` variables: if you register a variable named `foo`, `$foo` is made accessible automatically after calling `session_start()`. Because the `serialize()` function was improved in PHP 4, it's also feasible to treat objects (classes) as session variables.

All variables you want to preserve across page requests need to be registered to the session library with the `session_register()` function. Note that this function takes the *name* of a variable as argument, not the variable *itself*. To register the variable `$foo`, you'd use this:

```
        session_register("foo");
```

This code:

```
session_register($foo);
```

would produce something meaningful only if `$foo` was the name of another variable:

```
$bar = "This is a string";

$foo = "bar";

session_register($foo);
```

You can use `session_unregister()` to remove variables from the session library.

As with real life, it's not always easy to tell when a session's life ends - unless it's a violent death, forced by `session_destroy()`. If the session is to die of old age, different configurations need to be taken into consideration. If you propagate the session ID via cookies, the default cookie lifetime is `0`, meaning that it will be deleted as soon as the user closes the browser. You can influence the cookie's lifetime with the configuration value `lifetime`. Because the server doesn't know whether the cookie still exists on the client side, PHP has another lifetime variable that determines how long *after the last access* to this session the data should be destroyed: `gc_maxlifetime`. But performing such a cleanup of old sessions (called *garbage collection*) on every page request would cause considerable overhead. Therefore, you can specify with what probability the garbage collection routine should be invoked. If `gc_probability` is `100`, the cleanup will be performed on every request (that is, with a probability of 100%); if it's `1` as by default, old sessions will be removed with a probability of 1% per request.

If you don't use cookies but pass the session ID via `GET` or `POST` instead, you need to pay special attention to the garbage collection routines. People might bookmark URLs containing the session ID, so you need to make sure that sessions are cleaned up often - if the session data still exists when the user accesses the page with the session ID at a later time, he'll simply resume the previous session instead of starting with a new session, which may not be your intention. A value of `10` to `20` for `gc_probability` would better fit this scenario than the default value of `1`.

Storage Modules

To read and save session data, PHP uses storage modules, thus abstracting the back end of the library. There are currently three storage modules available: `files`, `mm` and `user`. By default, PHP uses the `files` module to save the session data to disk. It creates a text file named after the session ID in `/tmp`. In the previous example, the content of this file would look like this, which is a serialized representation of the variable:

```
counter|i:4;
```

You probably won't ever need to access this file directly.

Tip

Serializing means the transformation of variables to a byte-code representation that can be stored anywhere as a normal string. Without this feature, it wouldn't be possible, for example, to store PHP arrays into a database. Serializing data is very useful for preserving data across requests - an important facet of a session library. You can use `serialize()` and `deserialize()`, but note that in PHP 3 these functions don't work correctly on objects (classes); class functions will be discarded.

If you need higher performance, the `mm` module is a viable alternative; it stores the data in shared memory and is therefore not limited by the hardware I/O system. The last module, `user`, is used internally to realize user-level callback functions that you define with `session_set_save_handler()`.

The real power lies in the capacity to specify user callbacks as storage modules. Because you can write your functions to handle sessions while still being able to rely on the standardized PHP API, you can store sessions wherever and however you want - in a database like MySQL, XML files, on a remote FTP server (okay, the latter wouldn't make much sense, but you get the idea).

The function `session_set_save_handler()` takes six strings as arguments, which must be your callback functions. The syntax of the function is as follows:

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

Tip

To leave out one argument, pass an empty string (`""`) to `session_set_save_handler()`.

The functions are defined as follows:

```
bool open (string save_path, string sess_name);
```

This function is executed on the initialization of a session; you should use it to prepare your functions, to initialize variables, or the like. It takes two strings as arguments. The first is the path where sessions should be saved. This variable can be specified in php.ini or by the

```
session_save_path()
```

function – you can use this variable as a joker and use it for module-specific configuration.

The second argument is the session's name, by default `PHPSESSID`. The

```
open()
```

function should return `true` on success and `false` on error.

bool close ();

This function is executed on shutdown of a session. Use it to free memory or to destroy your variables. It takes no arguments and should return `true` on success and `false` on error.

```
mixed read (string sess_id, );
```

This important function is called whenever a session is started. It must read out the data of the session identified with `sess_id` and return it as a serialized string. If there's no session with this ID, an empty string `""` should be returned. In case of an error, `false` should be returned.

bool write (string sess_id, , string value);

When the session needs to be saved, this function is invoked. The first argument is a string containing the session's ID; the second argument is the serialized representation of the session variables. This function should return `true` on success and `false` on error.

```
bool destroy (string sess_id, );
```

When the developer calls

```
session_destroy(),
```

this function is executed. It should destroy all data associated with the session

```
sess_id
```

and return `true` on success and `false` on error.

```
        bool gc (int max_lifetime, );
```

This function is called on a session's start-up with the probability specified in `gc_probability`. It's used for garbage collection; that is, to remove sessions that weren't updated for more than `gc_maxlifetime` seconds. This function should return `true` on success and `false` on error.

Session ID Propagation

PHP 4 sessions support the following methods of passing the session ID:

- Cookies (default)
- `GET/POST`
- Hidden in the URL, either done manually or by automatic URL rewriting

Cookies are the default way to pass the session ID between pages. If you're happy with cookies, you don't have to worry about any special configuration.

Another common way is to pass the ID is with `GET/POST`. Your URL would then be similar to `script.php3?<session-name>=<session-id>`. You can create such URLs by using the global constant `SID`:

```
printf('<a href="script.php?%s">Link</a>', SID);
```

Automatic URL rewriting is one of the *very* cool new features of PHP 4. To enable it, you need to configure PHP with `--enable-trans-id` and recompile it. Then the session ID in the form `<session-name>=<session-id>` will be added to all relative links within your PHP-parsed pages. While this is a handy feature, it should be used with caution on high-performance sites. PHP has to look at each individual page, analyze it to see whether it contains relative links, and eventually add the ID to the links. This obviously introduces a performance penalty. Cookies, on the other hand, are set only once, avoiding the overhead of URL rewriting.

3. Authentication

Session management gives you a base to build on. Sophisticated web applications aren't possible without a way to maintain state.

But you'll soon see that you need another technique again and again: authentication. For example, if your users can access some parts of your web site only after having registered themselves with the system, you need authentication.

The Login Process

To be able to authenticate a user, you need his or her identity - you ask the user for an identification. Identification is a statement of who the user is: this can be a username, a customer number or anything else, as long as it is unique among your user base.

After you got the the information of who the user claims to be, you need to verify the claim. To be able, you need additional data from the user (aside from the claimed identity): an authentication element, commonly a password. To protect against eavesdropping, a trusted path - a secure communication channel - is necessary to transmit the password. On the web, this could be SSL.

The authentication element is checked against an authentication database. If the authentication returns that the user's identification is correct, the system completes the login process and associates the user's identity and access control information with the user session. In trivial applications, the access control information may merely consist of a flag that the current user is successfully authenticated. In more complex situations, the system may also associate a security level or permission level, defining what the current user is allowed to do in the application (for example, there may a super-user class, or a read-only user class). Depending on the level of needed security, the system needs to log successful or failed login attempts; for example, the C2 Security Standard requires systems to audit all login events.

HTTP-Authentication

HTTP provides a method for user authentication: HTTP Basic Authentication. On pages that require authentication, the web server replies with a special header to the client:

```
HTTP/1.1 401 Authorization Required

WWW-Authenticate: Basic realm="Protected Area"
```

Upon this, the browser will pop up a modal dialog box asking for username and password. Unfortunately, this type of authentication really merits the "basic" prefix - there are a number of drawbacks:

- To log users out, you have to apply tricks.
- To log users out after a defined idle time, you have to apply more tricks.
- If you need groups of user (what we called "permission levels" earlier), you need your own application logic to filter individual users to groups.
- It's impossible to brand the login process, the gray pop up dialog will always be the same, regardless of the web site.
- Novice users are generally scared of these dialogs. And you can't provide any help, as you can't modify the dialog.
- HTTP Basic Authentication over PHP is only possible with the module version.

All this leads to the conclusion, that it might be wiser to use something else, except of the most basic scenarios.

PHP Authentication

PHP native authentication makes it possible to use arbitrary login screens and authentication procedures. You can write your own authentication library which uses the PHP session management functions we have introduced earlier, or you can use the PHPLib. PHPLib can be found at http://phplib.netuse.de/ and it is ”"[...] a toolkit for PHP3 developers supporting them in the development of Web applications"”. The main idea of PHPLib is, that there is a collection of modules and methods written in PHP for the PHP application programmer's disposal, including session management and authentication. If you want to use PHPLib's authentication library, you also need to use its session management functions, as its `Session` class is the foundation of all the `Auth` class.

The PHPLib uses a object oriented approach, which is usually intimidating to new users. If you regularly develop larger applications, however, you're advised to look at the PHPLib; if nothing else, it will get you a very profound idea how to solve the most common problems in web application development.

To use PHPLib's `Auth` class, you write a new class which extends the base class and provides your own validation functions and login form. A very simple example, which compares username and password against a hard coded value, could look like the code shown in figure 2.

Figure 2. PHPLib's authentcation

```
class My_Auth extends Auth
{
    // The class name (serialization helper)
    var $classname        = "My_Auth";

    // DB details to use
    var $database_class = "DB_Session";
    var $database_table = "auth_user";

    // Arbitrary value used in uniqid generation
    var $magic = "Foobarbaz";

    function auth_validatelogin()
    {
        // Variables from login form
        global $username, $password;

        // Save username
        $this->auth["uname"] = $username;

        // Dummy compare
        if($username == "kris" && $password == "test")
        {
            // Validation successful, store uid
            $auth["uid"] = $username;

            // Return user name
            return($username);
        }

        return(false);
    }
}
```

In real life, you'll probably want to authenticate against a database with valid logins and provide your own login form in the `auth_loginform()` function. Now the only thing left is adding an appropriate `page_open()` call to the page you want to protect:

```
page_open(array("sess" => "My_Session", "auth" => "My_Auth"));
```

The first time a user requests a page, he or she will be redirected to a login form and asked to provide username and password. If the authentication is successful, the script continues to be executed below the page_open() call, otherwise PHPLib will stop processing with an appropriate error message (which you can also provide yourself).

In the next step, you could use the `Perm` class to provide permission management facilities. We'd like to refer you to the PHPLib documentation for more details though; now that you've got a basic understanding of how authentication works, it shouldn't be too hard to extend your application yourself.

Tobias Ratschiller is a New Media Consultant in Italy, specializing in the creation of large scale dynamic websites. He has provided consulting and implementation work for some of the world's largest websites and has contributed to several PHP titles. Together with Till Gerken, he's currently writing a book titled Advanced Web Application Development with PHP, which will be published in May 2000 by New Riders. Apart from that, he teaches at seminars about usability, user interface design and content management systems. Tobias runs http://phpwizard.net/.