14.3 Sessions

Cookies are very useful, but they come with certain annoyances and limitations. The most prominent limitation of cookies is that they rely upon the browser to faithfully send data back and forth with the server. In general the browser should not and cannot be trusted to store any important data, since the user might clear his cookies or could edit them to tamper with their data. Here are some of the limitations of cookies:

- Cookies are able to store only short amounts of text data; there is a maximum total cookie size before the data being sent overwhelms the maximum allowed HTTP header size and causes the overall request to fail.
- Cookies can take up lots of bandwidth if a large amount of information is stored in them, since this information is sent back and forth on every request to the site.
- Cookies rely on trusting the browser to send back information unaltered. A malicious or errant browser or user could delete, damage, or tamper with cookie data.
- Unless the web server is a secure encrypted site, the contents of cookies are sent in plain text format over the web, so they could be read by others on the same network. This could allow potential attackers to steal cookie data containing sensitive information.

For these reasons and others, a more sophisticated overall system is needed. Recall that the underlying motivation for most usage of cookies is to retain a "memory" of a user's interaction with a given web site over a period of time. In general the length of this interaction is relatively short, consisting of several page requests during one continuous sitting. This abstract concept of a set of related page requests from one client to a server is called a *session*.

session

A series of related page requests from a client to a server over a short period of time.

Really the problem with our use of cookies is that the data is being stored on the client's machine. A more robust system would store the data on the web server itself, remembering the client when that person connected on each page request. Sensitive information like user names and passwords should be stored on a server anyway rather than in cookies sent to each client. The only problem with this system is that a server isn't really able to tell which user is which without cookies.

Most modern server-side web programming languages, such as PHP, have adopted a system of implementing support for session data using a hybrid approach between client-side cookies and server-side saved data. In this section we'll discuss how sessions work, how they are implemented, and how to program with sessions in PHP.

14.3.1 Lifecycle of a Session

In most server-side programming languages that provide session support, the server sets a cookie on the client containing a single piece of information called a *session ID* number. Each client is given a unique session ID. The session ID is sent from the client to the server on each page request. The session ID is a lookup key that helps the server look up information it has saved associated with that particular client and his/her current session. The server-side session data storage is like a large associative array, keyed by session IDs, that maintains of data about each user's current session. It is a bit like a server-side set of cookies.

The basic communication between the client and server for establishing a session is as follows:

- The client arrives at the site and requests its first page. (Or, perhaps, it requests a page that makes the server decide to initiate a session, such as submitting a Login form.)
- The server creates a unique session ID for this user. The server also saves any data it wishes to associate with that user into its private session storage area, such as their login status, permission level, user account data, etc., keyed by that session ID.
- The server sends back the page response, along with a cookie containing the user's session ID. (In PHP, these session ID cookies have names like `PHPSESSID`, though you don't need to know that in order to program with sessions in PHP.)
- On future page requests, the user sends back the same session ID cookie, and the server uses it to look up the user's associated data. The server will remember that this is the same user and can display customized messages such as, "Welcome back, Joe!", etc.
- Eventually, once the user has not requested any pages for a while with that session ID, the ID is timed out and the session data is discarded by the server. (The default timeout for sessions on PHP is 1440 seconds (24 minutes) with no page activity from that user.)

Figure 14.5 summarizes the steps for establishing a session as described above.

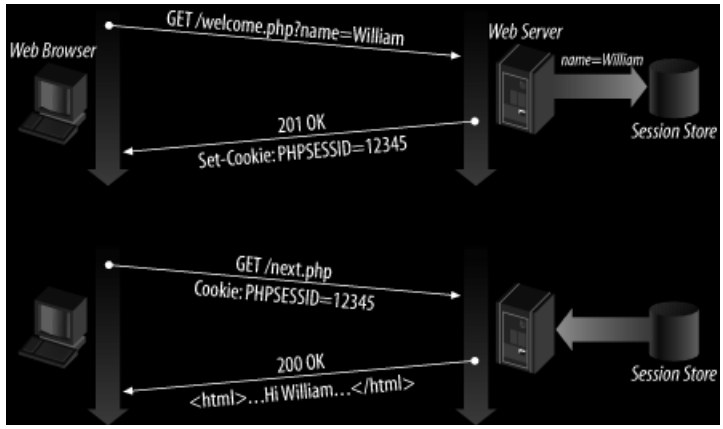

Figure 14.5 Establishing a session

When implemented in this way, sessions offer the following benefits over cookies:

- Sessions can store more data. Client-side cookies can be only up to a few kilobytes at most, but you can store large amounts of session data without a problem.
- Sessions save bandwidth. When lots of data is stored in cookies, that data must be sent back and forth to and from the server on each page request. With sessions, only the session ID, a reasonably small integer, is sent back and forth with each request.
- Session data is more secure. While cookie data can be intercepted and viewed by third parties, session data is stored on the server and is generally safe as long as the attacker does not have direct access to the server's file system.

For this system to work well, session IDs must be large randomly generated numbers, and they must be difficult to guess. If not, one malicious user could guess the session ID of another and could access the site as though they were that user. Luckily the system for generating session IDs in modern languages is relatively difficult to crack. Sessions are not without their security risks, though. A variety of attacks on web sites involve various ways of trying to get access to users' session ID numbers in order to perform malicious actions. We will discuss this in more detail in the Security chapter.

14.3.2 Sessions in PHP

PHP has several global functions for managing sessions, summarized in Table 14.4. The most prominent function is `session_start`, which begins a session for the current user. Many of the other functions do not need to be used in order to get a fully working session. The `session_start` function must be called before your PHP script produces any HTML output.

Function	Description
`session_destroy()`	removes all data associated with current session
`session_id()`	returns session ID number for current session
`session_regenerate_id()` `session_regenerate_id(boolean)`	replace current session ID with a new one; if `TRUE` is passed, also wipes any data associated with the old one
`session_save_path()` `session_save_path("directory")`	gets/sets the folder name on the server where session data is stored (can also be configured in PHP.ini)
`session_start()`	begins a new session for the current client; if a session is already in progress, has no effect
`session_status()`	returns information about whether session support is enabled or disabled on the current server
`session_unset()`	frees all session variables currently registered

Table 14.4 PHP session functions

When you call `session_start`, several things happen. The server looks at the current cookies sent by the client, if any, to determine whether a session already exists for this user. If not, a new random session ID is created and a session data storage is set up for the session. If there is already a session for the user, any data associated with that session is loaded for use. If your pages use sessions, then `session_start` should be the first function you call on those pages.

Once you start your session, PHP stores data associated with the session in a superglobal associative array named `$_SESSION`. As with cookies and request query parameters, the keys of this array are session variable names and the values associated with the keys are the values for those session variables. The idea is that once you start a session, you can store any data you like into `$_SESSION`, and on later page views, you will be able to access the array and its contents will still be there. This is in contrast to how normal variables work in PHP; variables other than `$_SESSION` will be thrown out once the current page request is completed, but data stored in `$_SESSION` will persist across many page views. Example 14.12 shows the syntax for adding, removing, and examining session variables.

```
$_SESSION["name"] = value;              # store a session variable
$_SESSION["name"]                       # access a session variable
if (isset($_SESSION["name"])) { ... }   # check existence of session variable
unset($_SESSION["name"]);               # delete a session variable
```

Example 14.12 Syntax template for accessing session data in PHP

Keep in mind that session data is not permanent on the server. In general, a session will time out after around 24 minutes of inactivity. Therefore `$_SESSION` is not a place for storing your sole copy of any important permanent server data about the current user. Rather, use `$_SESSION` for convenient access to temporary data about the current session of page views, and for anything more crucial or long-lived, back it up into a server data file or database. For example, if the user creates a new user

account, you can use session variables to remember the user name the user chose, but also create a new record in your Users database table so that the user name doesn't disappear after the session.

Sometimes you want to immediately discard a session rather than waiting for it to time out, such as if the session is remembering login information and the user indicates that he/she wants to log out. To get rid of a session, call `session_destroy` to discard any data associated with the session. If you want to get rid of the session and immediately begin a new session for some reason, you must tell PHP to regenerate you a new session ID. Example 14.13 demonstrates the process.

```
session_destroy();              # delete current session
session_regenerate_id(TRUE);    # reset to a new session ID for future sessions
session_start();                # start new session
```

Example 14.13 Destroying a session and starting a new one

Example Page: Power Animals

Let's look at an example page that uses session variables to randomly assign the visitor a "power animal" from a list of animals, and also tracks the number of times the user has visited the page. Once the user has been assigned his/her random power animal, that animal should stay associated with that user for the rest of the session. Each time the page is reloaded, the site should tell the user how many times he/she has visited the page.

We'll store the user's power animal and number of page views as session variables named `"poweranimal"` and `"views"` in the `$_SESSIONS` global array. After telling PHP to start a session by calling `session_start`, we'll check whether these session variables are set. If not, we know that this is the user's first visit to the page, so we'll initialize the variables by choosing a random animal and setting the number of page views to 1. If the user has visited the page before, the session variables will already be set, so we don't need to choose a random power animal. We can just use the power animal already chosen on a previous page view for that user, and increment the user's number of page views by 1. Example 14.14 shows this PHP code.

```
<?php
$animals = array("bee", "llama", "octopus", "rabbit", "squirrel", "yak");
session_start();

# check whether the user has ever visited the page before in this session
if (!isset($_SESSION["poweranimal"]) || !isset($_SESSION["views"])) {
  # new user; set up session data, choose random power animal
  $_SESSION["poweranimal"] = $animals[rand(0, count($animals) - 1)];
  $_SESSION["views"] = 1;
} else {
  $_SESSION["views"]++;   # returning user
}
?>
```

Example 14.14 Power animal page, initial PHP code

The HTML content to use this PHP code would appear in the same file, immediately below the PHP code to manage the session. We will insert the session variables' values into the HTML content using PHP expression blocks. The power animal's name will be printed and also used as the base for displaying images for each animal. (Assume that we have already set up files such as bee.png, rabbit.png, llama.png, and so on in the same folder as our PHP files.) The first time the user visits the

page, they are welcomed as a new user and shown their power animal. On subsequent reloads of the page (by pressing the browser's Refresh button), the site remembers the user's power animal and displays the same animal every time, along with an increasing count of the number of page views for that user. Example 14.15 shows the HTML code portion of the PHP file and two page outputs.

```
<!DOCTYPE html>
<html>
  <body>
    <h1>Power Animal Finder</h1>

    <?php if ($_SESSION["views"] == 1) { ?>
      <p>Welcome to our site, new visitor!</p>
    <?php } else { ?>
      <p>Welcome back! This is your visit #<?= $_SESSION["views"] ?>.
    <?php } ?>

    <p>Your power animal is the <?= $_SESSION["poweranimal"] ?>!</p>
    <p><img src="<?= $_SESSION["poweranimal"] ?>.png" alt="power animal" /></p>
  </body>
</html>
```

Example 14.15 HTML code for power animal page

Now suppose we want to give a user the ability to start over, choosing a new power animal and resetting the number of page views back to 1. The user could do this themselves by deleting their cookies or restarting their browser, but that's a clunky solution. Let's provide a "Start Over" option on the page that lets the user click a button to achieve such a reset.

To implement the reset feature, we'll insert a form at the end of the HTML content that submits itself back to the same page file (a blank `action` attribute on the `form` tag achieves this). The form will have a Reload button and a "Start Over?" checkbox. Example 14.16 shows the HTML for it.