The hypertext transport protocol (HTTP) defines a simple series of procedures by which a client computer may make a request of a server computer, and the server computer may reply. The protocol is central to the everyday operation of the World Wide Web, in which it is used to request and provide files to browsers in response to URL entries and clicks on hyperlinks.

Bear in mind that in some cases, notably those involved in a multi-tier software application, the concepts of “client” and “server” can be somewhat muddled. Even if both machines involved in an HTTP transaction are in a data center and would be referred to in conversation as “servers,” the one making a request is the HTTP client and the one responding to that request is the HTTP server.

Because much of the work that PHP does has to do with Web browsers, the language has evolved a number of structures and mechanisms having to do with HTTP. As a PHP programmer, you should have a good understanding of these elements of the language. They are all the more important because other protocols, chiefly simple object access protocol (SOAP), covered in Chapter 5, ride (or can ride) on top of HTTP.

This chapter introduces you to the HTTP protocol as it pertains to PHP. It provides an overview of how the protocol works in the context of a simple page request, then builds on that foundation to show how it can be used to supply input values, arguments, effectively, to a server-side software program. From there, we'll see how the PHP language can be used to grab, examine, and manipulate values communicated by HTTP. There's also coverage of how to access cookies, which are small pieces of data stored on the client side, persistently if you like, through PHP.

You can approach HTTP on a couple of levels. You can treat it as a “black box” concept if you want, using well-documented procedures (for retrieving data from a form, saving cookies, etc.) without really knowing what goes on behind the metaphorical curtain.
Alternately, you can dig deeper into HTTP and really know what the protocol does in response to commands you issue in your PHP software. This chapter assumes you're interested in the details, and provides the recipes along the way.

The best way to learn about the real-life operation of HTTP is to install a network-monitoring program of the sort commonly called a sniffer. Sniffers can provide far more information than you need to learn about HTTP, though, and can also be quite costly. I recommend a specialized HTTP-monitoring program called HttpDetect, from Effetech. HttpDetect sits in the background, watching the network traffic for HTTP activity. When it spots some, it records the traffic to a table. You get a convenient view of the full HTTP request and response headers, as well as the data that was sent or received as part of the request. The HttpDetect product works equally well on clients and servers.

Effetech asks a reasonable registration fee for its excellent software; the fee is slightly higher if you plan to use HttpDetect commercially. If you're planning to do any sort of network programming that involves HTTP (which is to say, any but the simplest PHP work and certainly anything involving SOAP—the sorts of things this book deals with) you'll benefit tremendously by having HttpDetect in your toolkit. You can't troubleshoot what you can't examine, and HttpDetect exposes HTTP traffic clearly.


3.1 Understanding HTTP

HTTP has to do with making requests for documents, and receiving documents in response. The documents may be static, or may be custom generated in response to each request. HTTP requests can carry variable values, which enables the protocol to carry user-entered information from client to server. This information can then be employed by the server-side programs invoked by the requests.

The full HTTP 1.1 specification is defined by RFC 2616, which is available here: ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt.

3.1.1 A Simple Page Request and Response

The oldest and simplest use of HTTP is as a means of requesting a static document—one that is not generated specifically because of any particular HTTP request. Though this situation is usually not relevant to multi-tier PHP applications (except for such uses as splash screens and navigation menus), it's worth examining because the simple cases make it easier to see what's happening in the more elaborate situations.
Request

An HTTP transaction begins with a uniform resource locator (URL). The user of a browser enters a URL into the appropriate part of his or her browser window, or clicks a link (for which a URL is embedded in the hypertext markup language [HTML]). The browser (otherwise known, in the jargon of the HTTP specification, as the user agent) submits the URL to the server (which is itself specified in the URL) and waits for a response.

To expand on the concept of a request somewhat, let's consider an "ordinary" URL:

http://www.davidwall.com/

When appearing in the "Location" box (or "Address" box, or "URL" box, or whatever your browser calls it—if it's graphical and has a box at all), that URL is used in a GET request. There are two main HTTP methods (GET and POST) for requesting pages; GET is the more widely used one.

Examined fully, a browser's request for that URL looks like this:

GET /HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword,
application/x-shockwave-flash, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; APC)
Host: www.davidwall.com
Connection: Keep-Alive

That request was generated by Microsoft Internet Explorer 6.0, running under Windows 2000 Server.

Notice that the first line specifies the method of the request, GET, and the version of the protocol specification: HTTP 1.1. The slash (/) that follows GET specifies the file that is being requested, relative to the host's Web service root. The slash is actually a special case; it means the request is for the default document in the root directory. The default document is determined by the configuration of the Web server, and is usually index.html or something similar.

Other lines (called headers) in the request denote Multimedia Internet Mail Extension (MIME) types and language encodings that the browser can handle, and provide the server with some information about the client (the Web browser) with which this request originated. TCP/IP (together with Domain Name Service [DNS], or other internetworking protocols) uses the Host header to route the rest of the request to the right server.

There's a blank line at the end of an HTTP GET request.

Response

In answer to an HTTP request (a GET request in this case) comes an HTTP response. As is true of a request, a response is a series of special lines of text (the headers). Unlike a
request, though, a response (other than a response indicating an error condition) is followed by further data, the requested document.

The response to the HTTP GET request examined in the last section looks like this:

```plaintext
HTTP/1.1 200 OK
Date: Mon, 24 Nov 2003 04:03:21 GMT
Server: Apache/1.3.27
Last-Modified: Thu, 13 Nov 2003 00:21:26 GMT
ETag: "9b05b-3b60-3fb2ce86"
Accept-Ranges: bytes
Content-Length: 15200
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html
<...page data omitted...>
```

The most important part of that response is the first line, which indicates success—the request can be fulfilled. Success is indicated by the "200 OK" code. See the sidebar for other possible HTTP response codes, good and bad.

It's also important to note the Content-Type header, which is a MIME type specification indicating that the data that follows the response header is an HTML document. There are scores of MIME types, covering everything from plain text, to extensible markup language (XML), to streaming audio-visual media in various formats.

### HTTP Response Codes

The list of legal HTTP response codes is defined by RFC 2616 (available at ftp://ftp.rfc-editor.org/in-notes/rfc2616.txt), but in general you'll find it useful to know the general pattern they employ. Each series of response codes corresponds to a particular class of conditions:

- 200-series codes indicate success.
- 300-series codes indicate a redirection to a new URL.
- 400-series codes indicate a client-side error (like a request for a nonexistent document—the famous "404 Not Found" error).
- 500-series codes indicate a server-side error.

#### 3.1.2 A More Complex GET Request

Ordinary document retrievals, as accomplished by typing a URL into a browser window, are the simplest application of the HTTP GET method. A slightly more complex technique allows programmers much greater flexibility in allowing interaction between server-side programs and end users on the client side.
Consider this URL, taken from the presentation layer of the Currawong Accounting application discussed in Chapters 6 through 10:


That’s a GET request to a particular PHP program that provides a single name/value pair. The name/value pair can be referred to within the called program, and for that reason this technique allows variable information to move from the client to the server. It is a characteristic of a GET request that name/value pairs are communicated as part of the URL in that way.

The request, viewed in an HTTP monitor like HttpDetect, looks largely the same as a simple GET request:

GET /acct/presentation/editAccount.php?id=6 HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; APC)
Host: 192.168.1.201
Connection: Keep-Alive

The only difference is in the document requested:


That line calls a program called editAccount.php, which is stored in the /acct/presentation/ directory relative to the Web document root of the machine with IP address 192.168.1.201. The response looks unremarkable, as well, at least from an HTTP perspective. Here is the response header:

HTTP/1.1 200 OK
Date: Mon, 24 Nov 2003 05:34:21 GMT
Server: Apache/1.3.26 (Win32) mod_perl/1.25 mod_ssl/2.8.10 OpenSSL/0.9.6c DAV/1.0.3 AuthNuSphere/1.0.0
X-Powered-By: PHP/4.3.3
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

There’s nothing in the header to indicate a special response to the “id=6” name/value pair that appeared in the corresponding GET request, but because editAccount.php is designed to access the name/value pair and process the value, the page data differs depending on the value of id. You’ll see how to access name/value pairs in GET requests later in this chapter.
3.1.3 A POST Request

Slightly more complicated than HTTP GET requests, HTTP POST requests most often result from the submission of HTML forms. Consider an HTML document that contains an opening FORM tag that looks like this:

```html
<FORM ACTION="http://db2/test/post/postTest.php" METHOD="POST">
```

That opening FORM tag indicates that, when submitted, the contents of the form will go to the program postTest.php at the specified location, and that the HTTP POST method will be used to convey the form values to the server. Assuming that there is a single element in the form (disregarding the submit button itself), the HTTP POST request would resemble this:

```
POST /test/post/postTest.php HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
        application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword,
        application/x-shockwave-flash, */*
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; APC)
Host: db2
Content-Length: 16
Connection: Keep-Alive
Cache-Control: no-cache

textBox=Tribbles
```

The primary difference between an HTTP POST request and an HTTP GET request is that in a POST request, the name/value pairs are embedded in the body of the message, rather than being visible in the URL. It's not too hard to generate HTTP POST requests to order, but it's harder than just editing a URL, which is all that's required to generate a custom HTTP GET request. For that reason, POST requests are slightly more secure than GET requests.

3.2 Working with HTTP in PHP

HTTP in terms of its specification is interesting, but as PHP programmers planning to use the protocol as one of our tools in creating multi-tier software applications, we want to know how PHP interacts with HTTP. It's important to know how to receive and make use of GET and POST requests in a program, and make use of HTTP's other features.

3.2.1 Accessing GET Variables

In making reference to GET variables from within PHP programs, it makes sense to make sure that you really have some (this is especially relevant if you're designing a program
that’s meant to take input from both GET and POST requests—you’d never have both at the same time). PHP provides the $_SERVER array to, among other things, help you determine if you’re working with an HTTP GET request or an HTTP POST request.

The $_SERVER array contains a number of named elements, each defining a characteristic of the server under which the PHP program is running. The $_SERVER array contains such information as the type of HTTP server that invoked the program, the path and filename of the PHP program itself, and, most interesting to us right now, the HTTP request method used to invoke the program. The $_SERVER[‘REQUEST_METHOD’] element can be checked for “GET” and “POST” to see which method was used to call the program.

The code looks like this:

```php
if ($_SERVER[‘REQUEST_METHOD’] == ‘GET’) {
    // React to GET request
}
```

Nothing to it, really. You just check $_SERVER[‘REQUEST_METHOD’] for the string literal “GET” and react if you find it. Usually, that means you'll want to evaluate another automatically created global array, this one called $_GET, for elements named after the arguments sent in the GET request. To put it another way, the $_GET array has named elements for each of the name/value pairs included in the invoking URL. Note that another array, $HTTP_GET_VARS, exists in PHP for purposes of backward compatibility but has been deprecated in favor for $_GET.

The $_GET associative array is the key to accessing variables communicated to the server side via an HTTP GET request. When you send a series of name/value pairs to a PHP program as part of an HTTP GET request, the names become keys in the $_GET array and the values become the values that correspond to those keys. For example, if you sent this URL to a PHP program

```
```

then you’d be able to make reference to any of these values in getTest.php:

```php
$_GET[‘alpha’]
$_GET[‘beta’]
$_GET[‘gamma’]
```

The $_GET array is “superglobal,” meaning it’s available in all contexts (including within function definitions) in getTest.php. There is never a need to declare a superglobal array such as $_GET.

### 3.2.2 Accessing POST Variables

As is the case with HTTP GET requests, it’s good practice to make sure you’re working with an HTTP POST request before you start querying the superglobal array that’s supposed to hold the POST values.
The technique is identical: You examine the REQUEST_METHOD element of the $_SERVER array, checking in this case to see if it's equal to "POST". If it is, you proceed to pick named values out of the $_POST array, looking for them under names that correspond to those in your name/value pairs. In a POST request, the names correspond to the NAME attributes of the elements of the form that was submitted in generating the request. Again, note that another array, $HTTP_POST_VARS, exists in PHP for purposes of backward compatibility but has been deprecated in favor for $_POST. Here is typical code:

```php
if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    // React to POST request
}
```

Like the $_GET array, the $_POST array is a superglobal associative array accessible from any point in a PHP program without the need for declaration within functions. It contains the name/value pairs that arrived as part of an HTTP POST request, with the names as keys and the values as values.

Consider this simple HTML page, which contains a short HTML form:

postTestClientSimple.html

```html
<html>
<head>
<title>POST Tester</title>
</head>
<body>
<h1>Simple POST Submission Test Client</h1>
<form action="http://db2/test/postTest.php" method="POST">
<p>
<input type="text" size=15 name="textBox1" value="Latrodectus">
</p>
<p>
<input type="text" size=15 name="textBox2" value="Hasseltii">
</p>
<p>
<input type="submit" value="Submit">
</p>
</form>
</body>
</html>
```
When the Submit button is clicked, an HTTP POST request is sent to http://db2/test/postTest.php. The request looks something like this:

```
POST /test/postTest.php HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-powerpoint, application/vnd.ms-excel, application/msword,
application/x-shockwave-flash, */*
Referer: http://db2/test/postTestClientSimple.html
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.0; APC)
Host: db2
Content-Length: 16
Connection: Keep-Alive
Cache-Control: no-cache

textBox=Tribbles
```

The request is accepted by postTest.php, whose code is as follows:

```
postTest.php

function demoFunction() 
{ 
  echo "<P>Look Ma, no declarations!";
  echo "<P>In the function:";
  echo "<BR>";
  print_r($_POST);
}

print_r($_POST);

demoFunction();
```

Note that the `print_r()` function (a native PHP function that can meaningfully print out several different datatypes, including objects, and is therefore especially useful in debugging situations) is called twice. The first time is in the main body of the program, so it's not so surprising that $_POST is available for access. The second reference is inside `demoFunction()`, though, and there's no special global line to declare $_POST within the function scope. Nonetheless, $_POST is perfectly accessible.

Figure 3.1 shows the form submitted to postTest.php and its results.

### 3.2.3 Accessing Various HTML Form Elements

Much of the work you will do in PHP will center on accessing the entries users have made into HTML form elements of various kinds. This section documents strategies for interacting with selection lists (single and multiple selection), checkboxes, and radio buttons
whose user-specified contents are submitted by HTTP POST requests. The preceding section dealt with simple text boxes.

**Simple Selection Lists**

A simple selection list allows the user to choose one element from a series of values. When sent to the server in an HTTP POST request, the VALUE attribute of the selected element (which need not be the same as the visible label) is available as...

```
$_POST['elementName']
```

...where elementName is the NAME attribute of the selection list in the originating HTML document. Here is an example:

```php
selectClientSimple.php

<html>
<head>

<title>POST Tester</title>

</head>

</html>
```
<H1>Test Client</H1>

<FORM ACTION="http://db2/test/post/selectSimple.php" METHOD="POST">

<P>
<SELECT NAME="stateName">
<OPTION VALUE="Alabama">Alabama</OPTION>
<OPTION VALUE="Alaska">Alaska</OPTION>
<OPTION VALUE="Arizona">Arizona</OPTION>
<OPTION VALUE="Arkansas">Arkansas</OPTION>
<OPTION VALUE="California">California</OPTION>
<OPTION VALUE="Colorado">Colorado</OPTION>
</SELECT>

</P>

<P>
<INPUT TYPE="SUBMIT" VALUE="Submit">

</FORM>

</BODY>
</HTML>

That form submits to this PHP program, which shows the selected state:

selectSimple.php

echo "The selected element is...";

echo $_POST['stateName'];

Figure 3.2 illustrates the appearance of the two files as rendered in a Web browser.

**Multiple-Selection Selection Lists**

In HTML, selection lists that allow multiple simultaneous selections are nearly identical to their single-selection cousins (the addition of the single attribute MULTIPLE changes their behavior). In PHP, though, this

$_POST['elementName']

where elementName is again the NAME attribute of the selection list in the originating HTML document, contains a subarray. The elements of the subarray are the HTML VALUE attributes of the selected elements.
Figure 3.2: A simple selection list is easy to reference through the $_POST array.

This HTML document includes a multiple-selection selection list:

selectClientMultiple.html

<HTML>
<HEAD>
<TITLE>POST Tester</TITLE>
</HEAD>
<BODY>
<H1>Test Client</H1>
<FORM ACTION="http://db2/test/post/selectMultiple.php" METHOD="POST">

<P>
<SELECT NAME="stateName[]" MULTIPLE>
<OPTION VALUE="Alabama">Alabama</OPTION>
<OPTION VALUE="Alaska">Alaska</OPTION>
<OPTION VALUE="Arizona">Arizona</OPTION>
<OPTION VALUE="Arkansas">Arkansas</OPTION>
<OPTION VALUE="California">California</OPTION>
<OPTION VALUE="Colorado">Colorado</OPTION>
</SELECT>
</FORM>
</BODY>
</HTML>
That form submits to this program:

```php
selectMultiple.php
```

```php
echo "The value of \$_POST['stateName'] is...<BR>";

echo \$_POST['stateName'];

echo "<P>

echo "The selected elements are... <BR>";

foreach (\$_POST['stateName'] as $key => $value) {
    echo $value;
    echo "<BR>
}
```

In that program, the foreach loop goes through each element in the \$_POST['statename'] array, echoing the value of each element. Figure 3.3 shows how this looks in practice.

**Checkboxes**

The key procedure in determining which of a series of checkboxes is checked is to see which keys evaluate to true. The keys of checked checkboxes evaluate to true; the keys of unchecked checkboxes evaluate to false.

This example shows how to access checkbox values through \$_POST.

```html
checkboxClient.html
```

```html
<HTML>
<HEAD>

<TITLE>POST Tester</TITLE>

</HEAD>

<BODY>

</BODY>
```
Figure 3.3: Accessing a multiple-selection selection list is a little more complicated, but not hard.

```html
<H1>Test Client</H1>

<form action="http://db2/test/post/checkbox.php" method="POST">
  <p>
    <input type="checkbox" name="agree"> &nbsp I agree.<br>
    <input type="checkbox" name="donateMoney"> &nbsp I would like to donate money.
  </p>
  <p>
    <input type="submit" value="Submit">
  </p>
</form>
```

That form sends the contents of its form to this server-side program:

`checkbox.php`
3.2 Working with HTTP in PHP

```php
echo "The contents of $_POST are...<BR>";
print_r($_POST);

echo "<P>The value of \$_POST['agree'] is...<BR>";
echo \$_POST['agree'];

echo "<P>The value of \$_POST['donateMoney'] is...<BR>";
echo \$_POST['donateMoney'];

echo "<P>";

echo "The selected box(es) is/are...<BR>";

// Convert \$_POST array to $parameters for clarity.

foreach ($_POST as $key => $value)
{
    if ($key)
    {
        echo $key;
        echo "<BR>";
    }
}
```

Simply put, the HTML VALUE attributes of checkboxes that are checked show up in the \$_POST array, whereas the VALUE attributes of those that are not checked do not appear in the \$_POST array. Figure 3.4 shows how this works in practice.

**Radio Buttons**

In HTML, radio buttons are very similar to checkboxes, with the key difference that all radio buttons in a set have the same HTML NAME attribute. Because they are radio buttons, only one of the radio buttons in a set can be active at one time.

Figuring out which radio button the user chose can be easily determined from the \$_POST array. If you examine the following element,

\$_POST['elementName']

it will be equal to the HTML VALUE attribute of the selected radio button. This program shows an example of this strategy:

```php
radioClient.php

<HTML>
<HEAD>
```
Figure 3.4: Each checkbox has a unique NAME attribute, with respect to HTTP POST.

```html
<TITLE>POST Tester</TITLE>
</HEAD>

<BODY>

<H1>Test Client</H1>

<FORM ACTION="http://db2/test/post/radio.php" METHOD="POST">

<P>

<BR> <INPUT TYPE="RADIO" NAME="radioWaves" VALUE="Yes"> Yes.
<BR> <INPUT TYPE="RADIO" NAME="radioWaves" VALUE="Probably"> Probably.
<BR> <INPUT TYPE="RADIO" NAME="radioWaves" VALUE="Probably Not"> Probably Not.
<BR> <INPUT TYPE="RADIO" NAME="radioWaves" VALUE="No"> No.

<P>
```
In that program, a simple examination of \$_POST['radioWaves'] gives the HTML VALUE attribute of the selected radio button. The behavior of this program is shown in Figure 3.5.

### 3.3 Cookies

A key provision of the HTTP protocol is the ability of the server to store and retrieve small pieces of data on the client. These little pieces of data are called cookies, and they help compensate for a strategic limitation of HTTP: its statelessness.

HTTP is said to be stateless because, by default, there is no way to correlate two sequential request/response transactions involving the same client and same server. If Server A gets a request from Client B and responds to it as required, then a few seconds later gets another request from Client B, there's no way to know that the second request almost certainly came from the same user and perhaps should be dealt with accordingly. If the two pages were parts of a game, for example, it might be helpful to track users as they progressed from one page to the next, perhaps incorporating their cumulative score in each subsequent page.

Fortunately, HTTP provides for cookies. There are several ways to set and retrieve cookies. JavaScript and other client-side scripting languages can do it, as can Perl, Active Server Pages (ASP), and pretty much every other server-side scripting language. The next section shows how to do it in PHP.
Chapter 3: HTTP in PHP

3.3.1 Setting Cookies

To set a cookie with PHP, use the setcookie function. It's easy to use:

```php
setcookie('username', 'derektom');
```

That single line of code is the simplest way to define a cookie and give it a value—it's the logical equivalent of establishing a variable and assigning a value to it. The setcookie function sets up a cookie called username and gives it the value derektom.

Because that cookie-creating function call lacks an expiration specification, though, the cookie called username will be stored only in volatile memory and will live only as long as the client, the browser, typically, remains running. When the client program closes, the cookie is not written to persistent storage and therefore ceases to exist. It is not available the next time the client starts up.

In order to make a cookie persist beyond the time the client is shut down (or, for that matter, to make it expire before the client shuts down), you need to supplement
the setcookie call with an expiration specification. It should take the form of a number representing a quantity of milliseconds after midnight on 1 January 1970. You can get the current timestamp like this:

```php
$now = time();
```

That means you can calculate a number representing the "time to live" that you want and end up with the needed expiration timestamp.

```php
$now = time();
expirationDate = $now + 259200000; // 259200000 is 1000 * 60 * 60 * 24 * 3, or the number of milliseconds in three days.
setcookie('username', 'derektom', $expirationDate);
```

If you want to do so, you can put restrictions on the pages that can retrieve the cookie setcookie sets. If you do this,

```php
setcookie('username', 'derektom', $expirationDate, '/game/');
```
then only code contained in files whose URLs include /game/ will be able to retrieve the value of username (or even detect that username exists). A similar restriction takes effect when you qualify setcookie like this:

```php
setcookie('username', 'derektom', $expirationDate, $path, 'mmrpg.davidwall.com');
```

By using the fifth argument of setcookie, we ensure that only pages whose domain paths end in mmrpg.davidwall.com will be able to retrieve the cookie. Note that specifying 'mmrpg.davidwall.com' allows domains like a.mmrpg.davidwall.com and b.mmrpg.davidwall.com to access the cookie. If the setcookie call had specified '.mmrpg.davidwall.com' (with a leading dot), then only pages served from mmrpg.davidwall.com itself (not from any subsidiary domains) would be able to read the cookie.

The sixth setcookie argument can be set to 1. When it is set to 1, the cookie may be accessed only over a secure connection. When the sixth argument is set to 0, or not specified at all, the cookie may be accessed over a clear connection.

Note that you may have to use null arguments as placeholders. If you wanted to restrict the domain that could access a cookie but not specify an expiration date or path restriction, you would have to use this technique:

```php
setcookie('username', 'derektom', '', '', 'mmrpg.davidwall.com');
```

The empty quotation marks act as placeholders.

### 3.3.2 Retrieving Cookies

Retrieving a cookie in PHP is even easier than setting one. The key procedure is to examine the `$_COOKIE` global array, which is an associative array with keys that correspond to
cookie names. Say you'd established a cookie called username, as was done in Section 3.3.1. You could retrieve it like this:

```php
echo "The value of the username cookie is " . $_COOKIE['username'] . "" ;
```

If you were not sure whether the username cookie existed, you could use function isset to find out:

```php
if (isset($_COOKIE['username']))
{
    echo "The value of the username cookie is " . $_COOKIE['username'] . "" ;
}
```

Furthermore, because $_COOKIE is an associative array, it can be traversed like any other associative array. If you didn't know the names of all the cookies that had been set, this would find them all for you:

```php
foreach ($_COOKIE as $name => $value)
{
    echo "New cookie/value pair: $name = $value" ;
}
```

### 3.3.3 Deleting Cookies

The procedure for getting rid of a cookie that's already been set is similar to the procedure for creating a new cookie. The trick is, you set the expiration timestamp for a point in the past, which effectively expires the cookie right away.

If there were a cookie in $_COOKIE called username, and you no longer required its services, you could call setcookie() like this:

```php
$now = time();
$expirationDate = $now - 100000 ; // Minus an arbitrary number of milliseconds
setcookie('username', 'derektom', $expirationDate);
```

By subtracting a number (the exact number doesn't matter) from the value that results from the time function, we get a time in the past. The username cookie is therefore past its expiration point, and disappears from $_COOKIE.

### 3.4 Sessions

If you wanted to, you could manually adapt the cookie-management capabilities of HTTP and PHP to track a user throughout a session. That is, your PHP code could be set up so that every page requested by the user included a check for a cookie of a certain name, which, if it existed, would contain a unique value. If it didn't exist, it could be created and assigned a unique value. The unique value would serve to identify the user and would remain constant across multiple transactions.
Sessions are important because they allow you to provide continuity to the user. Once someone has logged in, he or she can remain logged in throughout the session. Preferences can be maintained, and data accumulated. Cookies are the difference between a simple-minded application that forgets its users after each interaction and an apparently more intelligent application that can keep track of them for extended periods.

Generating and keeping track of those unique values would be a major hassle, though. Because session tracking, otherwise known as state maintenance, is such a common requirement of browser-based software applications, PHP has some built-in mechanisms for the purpose that are easy to use.

### 3.4.1 Preparing the PHP Server for Session Tracking

Depending on how you configure your PHP server, it either will or will not automatically issue a unique identifier to every new site visitor it serves. That is, you can determine whether the PHP interpreter (the PHP executable that runs .php files) tracks sessions by default.

A setting in php.ini determines the behavior. If this line

```php
session.auto_start=1
```

appears in php.ini, the server will automatically issue session identifiers to every new client it encounters. If the same value is set equal to 0

```php
session.auto_start=0
```

then there is no automatic session tracking and, if it is needed, session tracking must be started manually, as covered in the next section.

### 3.4.2 Establishing a Session

If the php.ini file of a particular PHP server does not include activation of the `session.auto_start` variable, session tracking must be initiated manually. It is not a complicated process, requiring only a single line of code.

If you wish to initiate session tracking on a client, include this line in a PHP page served to that client:

```php
session_start();
```

The effect is the same as if `session.auto_start` had been enabled. All subsequent session-related techniques may be used in exactly the same way.

When a session is established (either automatically or explicitly), a session identifier springs into existence. A session identifier is a random sequence of letters and numbers. It is stored in two places: on the client in an element of the `$_COOKIE` array, and on the server in a file. Further data associated with the session (more on how to create that data appears in the next section) is stored on the server only, but the identifier is stored on both sides of the connection.
Because the identifier is stored on the client side as an element in the \$_COOKIE associative array, we can retrieve it in much the same way we'd retrieve any other value out of that array. You have to get the key of the identifier element in a special way, though. Here's the code:

```php
$sessionKey = session_name();
```

Then, we can use $sessionKey to get the right value out of the \$_COOKIE array:

```php
$identifier = \$_COOKIE[$sessionKey];
```

After that runs, $identifier will contain a long string of letters and numerals, and may be manipulated like any other string. There's usually no reason to do this, though, other than to inform yourself of how session tracking works on the client side.

What happens if the client is set up to reject cookies? PHP works around the limitation. The PHP interpreter will, for example, automatically modify HTML links to include session identifier values in HTTP GET form (that is, incorporated into the URL). The ACTION attributes of PHP-generated HTML forms are modified to include the session identifier, too.

### 3.4.3 Setting a Session Variable

Session variables are similar to cookies in that session variables, like cookies, are stored in a global array. The difference is that most session data (all except the single reference to the session identifier, as covered in the previous section) is stored on the server side, whereas all cookie data reside on the client.

To establish a session variable, simply establish a new element in the \$_SESSION associative array:

```php
\$_SESSION['username'] = 'derektom';
```

### 3.4.4 Retrieving a Session Variable

The element identified by the username key can be retrieved at any time in the future, as long as the session is active, which is to say, until the client, the browser, is closed. This kind of retrieval is routine:

```php
echo \$_SESSION['username'];
```

And, as is the case with any associative array, code like this is valid with \$_SESSION:

```php
foreach (\$_SESSION as $name => $value) {
    echo "New cookie/value pair: $name = $value";
}
```
3.5 Questions and Exercises

1. What are the relative advantages and disadvantages of HTTP GET and POST requests?

2. What are some of the risks associated with using an HTML HIDDEN field in conjunction with HTTP POST requests?

3. How can HTTP GET and POST compensate for cookies being disabled in browser settings?

4. What problems with session maintenance would you anticipate if an application were spread across multiple servers whose activities were regulated by a load balancer? How might you get around these problems?