3

PHP and the Web

Terms You’ll Need to Understand

- Server-side
- Client-side
- Hypertext Transfer Protocol (HTTP)
- GET request
- POST request
- Superglobal array
- HTTP header
- Cookie
- Session
- Session identifier

Techniques You’ll Need to Master

- Distinguishing between server-side and client-side
- Handling form data using superglobal arrays
- Working with cookies
- Persisting data in sessions
Server-side Versus Client-side

One of the keys to understanding PHP’s role in the Web is to understand how the Web works at a fundamental level. This generally involves a basic understanding of HTTP, Hypertext Transfer Protocol. To examine the basic operation of the Web, consider a typical HTTP client, your Web browser. When you visit a URL such as http://example.org/, your browser sends an HTTP request to the web server at example.org. The simplest example of this request is as follows:

```
GET / HTTP/1.1
Host: example.org
```

The web server’s responsibility is to respond to this request, preferably with the resource that is desired (the document root in this example). An example of a response is as follows:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 419

<html>
<head><title>Example Web Page</title></head>
<body>
<p>You have reached this web page by typing &quot;example.com&quot;, &quot;example.net&quot;, or &quot;example.org&quot; into your web browser.</p>
<p>These domain names are reserved for use in documentation and are not available for registration. See <a href="http://www.rfc-editor.org/rfc/rfc2606.txt">RFC 2606</a>, Section 3.</p>
</body>
</html>
```

As you should notice, the majority of this response is the actual content, the HTML. When your browser receives this response, it will render the web page (see Figure 3.1). Once a page is rendered, you can disconnect your computer from the Internet, and this won’t cause a problem until your browser needs to send another HTTP request.

Where does PHP fit into this process? PHP’s role is best explained as an aid to the web server while it is generating the HTTP response. Thus, by the time the web server sends the response, PHP’s job is done. Its output is included in the response. Because PHP’s activity takes place on the server, it is an example of a server-side technology.

By contrast, any processing that takes place after the browser has received the response is referred to as client-side. JavaScript is a popular choice for client-side scripting. You’re probably familiar with using JavaScript or at least seeing it when you view the source of a web page. This is a distinguishing characteristic. What you see when you view the source of a web page is the content of the HTTP request. This content can be generated on the server, so just as PHP can be used to generate HTML, it can also generate JavaScript.
JavaScript executes on the client. Thus, interacting with PHP is much more difficult because it requires another HTTP request to be sent. After all, PHP’s job is done, and the web server is quietly awaiting the next request. By the time JavaScript executes, there isn’t even a connection between the Web client (your browser) and the web server anymore.

If you find yourself having trouble determining whether you can pass data from PHP to JavaScript or from JavaScript to PHP, it would be wise to review this section a few times. A clear understanding of the environment in which PHP operates, and the distinction between client-side and server-side technologies, is important.

**HTML Forms**

One task with which you should already be familiar is processing HTML forms. Forms provide a convenient way for users to send data to the server, and this makes the Web much more interactive. PHP makes processing these forms easy for developers; the form data is available in the `$_GET` and `$_POST` superglobal arrays, depending on the method used in the form (which in turn affects the request method used by the browser). In addition, `$_REQUEST` is a method-agnostic array that you can use to access form data (basically a merge of both `$_GET` and `$_POST`).

Superglobal arrays are available in every scope, which makes them convenient to use. For example, you might use them in a function without having to declare them as global, and there is no need to ever pass them to a function. They are always available.

For versions of PHP prior to 4.1.0, you must use a different set of arrays because `$_GET`, `$_POST`, and `$_REQUEST` are not available. Instead, you must use `$_HTTP_GET_VARS` and `$_HTTP_POST_VARS` (for `$_GET` and `$_POST`, respectively). There is no equivalent for `$_REQUEST` (where both arrays are merged), and these are also not superglobals, so you must use them similar to standard arrays.
To illustrate how form data is passed, consider the following form:

```html
<form action="/process.php" method="post">
   <input type="text" name="answer" />
   <input type="submit" />
</form>
```

Figure 3.2 shows how this form appears in a Web browser.

![Figure 3.2 - A browser renders an HTML form.](image)

If a user enters C for the answer and submits the form, an HTTP request similar to the following is sent to the web server:

```http
POST /process.php HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 8

answer=C
```

As a PHP developer, you can reference this value as `$_POST['answer']` because the request method (indicated on the first line of the HTTP request) is POST.

By contrast, if the method of the form specifies the use of a GET request, the request is similar to the following:

```http
GET /process.php?answer=C HTTP/1.1
Host: example.org
```

Rather than passing the form data as the content of the request, it is passed as the query string of the URL. In this situation, you can reference `$_GET['answer']` to get the user's answer.
One important point about HTML forms is that the result of any form element is a single name/value pair in the request. This is true for hidden form elements, radio buttons, checkboxes, and all other types. For example, consider the following form:

```html
<form action="/process.php" method="post">
  <input type="hidden" name="answer" value="C" />
  <input type="submit" />
</form>
```

Figure 3.3 shows how this form appears in a Web browser. Unlike the previous example, the user is only presented with the submit button. As long as the user uses this form to send the POST request, the value of `$_POST["answer"]` will always be `C`. The actual request sent by the browser is identical to the previous example, thus it is impossible to discern the type of HTML form used to generate a request by only observing the request.

The behavior of some form elements can be confusing. Notably, elements such as check boxes and radio buttons, because of their Boolean nature, are only included in the request if selected. When selected, their value is determined by the value attribute given in the HTML markup. Thus, the corresponding variable in PHP might or might not be set, and you might want to use `isset()` on these types of elements to determine this.

There is also the special case in which multiple form elements are given the same name, such as in the following example:

```html
<form action="/process.php" method="post">
  <input type="text" name="answer" />
  <input type="text" name="answer" />
  <input type="submit" />
</form>
```
The browser will send a request similar to the following (assuming that the user answers C and A, respectively):

```
POST /process.php HTTP/1.1
Host: example.org
Content-Type: application/x-www-form-urlencoded
Content-Length: 8
answer=C&answer=A
```

If you reference `$_POST['answer']`, you will notice that its value is A. Where did the first answer go? As PHP processes the form for you and assigns variables within the superglobal arrays, values can be overwritten. If this is not the desired behavior, there is a simple naming convention you can use instead:

```
<form action="/process.php" method="post">
  <input type="text" name="answer[]" />
  <input type="text" name="answer[]" />
  <input type="submit" />
</form>
```

By adding `[]` to the end of the form element name, you are asking PHP to create an array for this particular element. Assuming that the same answers as before (C and A, respectively) are entered before submitting the form, `$_POST['answer']` is now an enumerated array, and the output of `print_r($_POST['answer'])` is as follows:

```php
Array
  [
    [0] => C
    [1] => A
  ]
```

So, you now have both values preserved conveniently in an array.

**Cookies**

What are cookies? When described as entities, which is how cookies are often referenced in conversation, you can be easily misled. Cookies are actually just an extension of the HTTP protocol. Specifically, there are two additional HTTP headers: `Set-Cookie` and `Cookie`. The operation of these cookies is best described by the following series of events:

1. Client sends an HTTP request to server.
2. Server sends an HTTP response with `Set-Cookie: foo=bar` to client.
3. Client sends an HTTP request with `Cookie: foo=bar` to server.
4. Server sends an HTTP response to client.
Thus, the typical scenario involves two complete HTTP transactions. In step 2, the server is asking the client to return a particular cookie in future requests. In step 3, if the user's preferences are set to allow cookies, and if the cookie is valid for this particular request, the browser requests the resource again but includes the cookie.

Hopefully this simple explanation already makes it clear why you cannot determine whether a user's preferences are set to allow cookies during the first request. When you set a cookie in your PHP code, whether by using `setcookie()` or `header()`, all you are doing is modifying the HTTP response to include a `Set-Cookie` header. You cannot, during the time that you are generating this response, determine how the browser will react. After all, the browser won't even receive the response (and the `Set-Cookie` header) until PHP has finished executing.

The `Set-Cookie` header, at a minimum, contains the name and value of the cookie. For example,

```
Set-Cookie: foo=bar
```

Other attributes can be included to modify when the cookie is to be sent in a subsequent request. These optional attributes are as follows:

- **domain**—Restricts requests for which the cookie is sent to those that are within the specified domain or in subdomains. The default is the domain of the current resource.
- **expires**—A date after which the cookie is no longer valid and should be deleted. The default is to persist the cookie in memory only, expiring it as soon as the browser ends.
- **path**—Only requests for resources within the specified path include the cookie. The default is no path restrictions.
- **secure**—An attribute with no value that indicates that the cookie should only be sent in requests sent over a secure connection, such as SSL.

An example of a `Set-Cookie` header with all optional attributes is as follows:

```
Set-Cookie: foo=bar; domain=example.org; expires=Mon, 26 Jul 2004 12:34:56 GMT; path=/; secure
```

The `Cookie` header included in subsequent requests contains only the name and value of the cookie:

```
Cookie: foo=bar
```

The attributes included in the `Set-Cookie` header are only used to determine whether the cookie should be included in the request at all. If included, only the name and value are given. In PHP, cookies sent in the request are made available in the `$COOKIE` superglobal array (for PHP versions prior to 4.1.0, cookies are available in the `$HTTP_COOKIE_VARS` array).
Sessions

One common use of cookies, and one of the main reasons behind their inception, is to maintain state. Stated differently, cookies allow you to associate separate HTTP transactions together by identifying a specific client.

If you set a cookie with a unique identifier, you can store information about the client on the server, and on the next request from that same client, you can use the cookie to identify the client and fetch the data that you stored. This technique is known as session management, and it relies on the ability to maintain state.

PHP makes all of this easy with its built-in sessions. To initiate PHP’s sessions, simply include the following function call on any PHP page:

```php
session_start();
```

If you are using the default `php.ini`, this function requires PHP to manipulate some HTTP headers, so you must call it prior to any output. After you have called this function, you can simply use the `$_SESSION` superglobal array to store and access session variables. (For PHP versions prior to 4.1.0, `$_HTTP_SESSION_VARS` must be used instead.) For example, the following code sets a session variable named `foo`:

```php
$_SESSION['foo'] = 'bar';
```

PHP takes care of propagating the session identifier (the unique identifier used to distinguish each client from any other) in a cookie or on the URL, depending on your `php.ini` settings, and it also takes care of storing and retrieving the session data.

Quite a few directives in `php.ini` affect sessions. The most notable ones are as follows:

- `session.save_path`—This indicates the directory in which PHP will store session data.
- `session.use_cookies`—This is a Boolean that indicates whether PHP will use cookies to propagate the session identifier.
- `session.use_only_cookies`—This is a Boolean that indicates whether PHP will only check cookies for a session identifier (and not the URL).
- `session.name`—The name of the session (also used as the name of the session identifier).
- `session.auto_start`—This is a Boolean that indicates whether PHP should always enable session management, allowing you to avoid the call to `session_start()`.
- `session.cookie_lifetime`, `session.cookie_path`, `session.cookie_domain`—These correspond to the attributes used in the `Set-Cookie` header for the session identifier.
Exam Prep Questions

- `session.use_trans_sid`—This is a Boolean that indicates whether PHP should dynamically choose whether to propagate the session identifier via cookies or the URL, depending on the user's preferences. If cookies are enabled, PHP will use a cookie; otherwise, it will use the URL. On the first page, PHP will use both methods since it cannot yet determine whether the user's preferences allow cookies (recall the previous discussion on cookies).

By default, PHP stores session data on the filesystem. If you want to modify this behavior, you can create your own session-handling functions for opening, closing, reading, writing, deleting, and garbage collection. To instruct PHP to use your functions for these session-related tasks, use `session_set_save_handler()` as follows:

```php
session_set_save_handler('myopen', 'myclose', 'myread', 'mywrite', 'mydelete', 'mygarbage');
```

This gives you complete flexibility over the behavior of the session management features, and you still use sessions the same way (`session_start()` and using `$_SESSION`). Thus, any existing code that uses standard session features will still work as expected.

### Exam Prep Questions

1. Is it possible to pass data from PHP to JavaScript?
   - A. No, because PHP is server-side, and JavaScript is client-side.
   - B. No, because PHP is a loosely typed language.
   - C. Yes, because JavaScript executes before PHP.
   - D. Yes, because PHP can generate valid JavaScript.

   Answer D is correct. JavaScript, like HTML, can be dynamically generated by PHP. Answers A and B are incorrect because the answer is yes. Answer C is incorrect because PHP executes before JavaScript.

2. Is it possible to pass data from JavaScript to PHP?
   - A. Yes, but not without sending another HTTP request.
   - B. Yes, because PHP executes before JavaScript.
   - C. No, because JavaScript is server-side, and PHP is client-side.
   - D. No, because JavaScript executes before PHP.

   Answer A is correct. Although your instincts might lead you to believe that you cannot pass data from JavaScript to PHP, such a thing can be achieved with another HTTP request. Answer B is incorrect because PHP executing before JavaScript is not what makes this possible. This is actually the characteristic that might lead you to believe (incorrectly) that the answer is no. Answers C and D are incorrect because the answer is yes, but also because the explanations given are false.
3. Which types of form elements can be excluded from the HTTP request?
   A. text, radio, and check box
   B. text, submit, and hidden
   C. submit and hidden
   D. radio and check box

Answer D is correct. When not selected, both radio buttons and check boxes are excluded from the HTTP request. Answer A and B are incorrect because text boxes are always included in the request. Answer C is incorrect because hidden form elements are always included.

4. When processing the form, what is the difference between a hidden form element and a nonhidden one, such as a text box?
   A. The hidden form element does not have a name.
   B. There is no difference.
   C. The hidden form element does not have a value.
   D. The hidden form element is excluded from the request.

Answer B is correct. When processing a form, each form element is simply a name/value pair within one of the superglobal arrays. Answers A and C are incorrect because hidden form elements can (and should) have both a name and a value. Answer D is incorrect because hidden form elements are only excluded from the user's view, not from the HTTP request.

5. Which of the following form element names can be used to create an array in PHP?
   A. foo
   B. [foo]
   C. foo[]
   D. foo[bar]

Answer C is correct. PHP will create an enumerated array called foo that contains the values of all form elements named foo[] in the HTML form. Answers A, B, and D are incorrect because any subsequent form elements of the same name will overwrite the value in previous elements.
6. When an expiration date is given in a `Set-Cookie` header, what is the resulting behavior in subsequent requests?

   A. If the expiration date has expired, the cookie is not included.
   B. The behavior is the same; the expiration date is included in the Cookie header, and you can access this information in the `_COOKIE` superglobal array.
   C. The cookie persists in memory until the browser is closed.
   D. The cookie is deleted and therefore not included in subsequent requests.

   Answer A is correct. Answer B is incorrect because only the name and value of the cookie are included in the Cookie header. Answer C is incorrect because setting an expiration date causes a cookie to either be deleted (if the date has expired) or written to disk. Answer D is incorrect because the cookie is only deleted if the date has expired, which isn’t necessarily the case.

7. If you set a cookie with either `setcookie()` or `header()`, you can immediately check to see whether the client accepted it.

   A. True, you can check the `_COOKIE` superglobal array to see if it contains the value you set.
   B. True, but only if `register_globals` is enabled.
   C. False, you can only use `setcookie()` if you need to test for acceptance. Using `header()` does not work.
   D. False, you must wait until you receive another HTTP request to determine whether it includes the `Cookie` header.

   Answer D is correct. The response that contains the Set-Cookie header is not sent until PHP finishes executing, so you cannot test for acceptance prior to this. Answers A and B are incorrect because the answer is false. Answer C is incorrect because using `setcookie()` and `header()` both result in the same thing: A `Set-Cookie` header is included in the response.

8. Why must you call `session_start()` prior to any output?

   A. Because it is easy to forget if not placed at the top of your scripts.
   B. Because you can no longer access the session data store after there has been output.
   C. Because `session_start()` sets some HTTP headers.
   D. Because calling `session_start()` causes the HTTP headers to be sent.

   Answer C is correct. Answer A is incorrect because this is a technical necessity, not a best practice. Answer B is incorrect because accessing the session data store is completely independent of whether there has been any output. Answer D is incorrect because you can set other HTTP headers after a call to `session_start()`.
9. Which of the following represents the proper way to set a session variable?
   A. $SESSION['foo'] = 'bar';
   B. session_start();
   C. session_set_save_handler ('myopen', 'myclose', 'myread', 'mywrite', 'mydelete', 'mygarbage');
   D. $foo = $SESSION['foo'];

   Answer A is correct. Answer B is incorrect because `session_start()` only activates PHP sessions for the current script. Answer C is incorrect because `session_set_save_handler()` allows you to override PHP's default session mechanism with your own custom functions. Answer D is incorrect; session data is being used as the value of a regular variable and is not being manipulated in any way.

10. Which of the following functions allows you to store session data in a database?
    A. session_start();
    B. session_set_save_handler();
    C. mysql_query();
    D. You cannot store session data in a database.

    Answer B is correct. You can use `session_set_save_handler()` to override PHP's default session-handling functions and store session data any way you want. Answer A is incorrect because `session_start()` only activates PHP sessions for the current script. Answer C is incorrect because `mysql_query()` only executes a query with MySQL and does not affect the behavior of PHP's session mechanism. Answer D is incorrect because this statement is false.