PHP: Processing XHTML Forms

Introduction

A Web form is a common method for allowing the user to provide input. As described in *XHTML: Forms*, there are two parts to a form: the user interface, and a script to process the input and ultimately do something meaningful with it. This document explains how PHP can be used to process form-based input. Consult the *XHTML: Forms* document for details about form syntax, submission method types (i.e., `GET` and `POST`), and the types of form widgets.

Form setup

The first step in handling user input via Web forms is to create the user interface. The form is typically defined in an XHTML document (e.g., a file that ends with `.html`) unless the form itself has dynamic content. The `action` attribute of the `form` element should be the URL of the PHP script that will be processing the form data.

Consider the following XHTML markup that defines a form with selection widgets, text input widgets, and a hidden field. Note that the `action` attribute of the `form` element designates the PHP script `processform.php` as the recipient of the form data.

```
<form method="post" action="processform.php">
  <!-- Radio buttons, none pre-selected -->
  <p>How would you rate your skill in programming?<br />
    <input type="radio" name="skill" value="beg" />Beginner
    <input type="radio" name="skill" value="int" />Intermediate
    <input type="radio" name="skill" value="adv" />Advanced
    <input type="radio" name="skill" value="sup" />Super-hacker</p>

  <!-- Radio buttons, one pre-selected -->
  <p>How many hours do you spend programming each week?<br />
    <input type="radio" name="hours" value="0-10" />0-10<br />
    <input type="radio" name="hours" value="11-20" checked="checked" />11-20<br />
    <input type="radio" name="hours" value="21-30" />21-30<br />
    <input type="radio" name="hours" value="30+" />30+</p>

  <!-- Checkboxes, several pre-selected -->
  <p>I agree to...<br />
    <input type="checkbox" name="cheaplabor" value="yes" checked="checked" />work for $1.50/hour.<br />
    <input type="checkbox" name="longdays" value="yes" checked="checked" />work 12 hours per day.<br />
    <input type="checkbox" name="late" value="yes" />show up late every day.<br />
    <input type="checkbox" name="usecomments" value="yes" checked="checked" />comment my code.
    <br /></p>

  <!-- Menu, one selected, multiple selections allowed -->
  <select name="state[]" size="5" multiple="multiple">
    <option value="al">Alabama</option>
    <option value="ak">Alaska</option>
    <option value="as">American Samoa</option>
    <option value="az">Arizona</option>
    <option value="ar">Arkansas</option>
    <option value="ca" selected="selected">California</option>
    <option value="other">Some other state</option>
  </select>

  <!-- Text box and password box -->
  <p>Username: <input type="text" name="username" /></p>
  <p>Password: <input type="password" name="passwd" /></p>
```

```
  <!-- Text area -->
  <textarea name="comments" rows="5" cols="40"></textarea>

  <!-- Hidden field -->
  <input type="hidden" name="promotion_code" value="x3g9kf43" />

  <!-- Submit button -->
  <p><input type="submit" value="Submit the Data" /></p>
</form>
```

Now that the user interface has been established, the PHP script can be created to process the data.

Receiving form data

Recall that form data is submitted in name-value pairs, which are derived from the form widgets' `name` and `value` attributes. The standard method for accessing this data is by accessing one of the predefined associative arrays named `$_POST` and `$_GET`, depending on the form submission method used. The syntax is `$_POST['name']`, where *name* corresponds to the name attribute of a given form widget. Here is a complete PHP script that lists the name-value pairs in a table.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Programming is Fun!</title>
  </head>
  <body>

    <h1>Form Results</h1>
    <table border="1">
      <tr><th>Field</th><th>Value</th></tr>
    <?php
      print "<tr><td>Skill</td><td>{$_POST['skill']}</td></tr>\n";
      print "<tr><td>Hours spent programming</td><td>{$_POST['hours']}</td></tr>\n";
      print "<tr><td>Programming languages used</td><td>{$_POST['proglang']}</td></tr>\n";
      print "<tr><td>Work for $1.50/hr?</td><td>{$_POST['cheaplabor']}</td></tr>\n";
      print "<tr><td>Work 12-hour days?</td><td>{$_POST['longdays']}</td></tr>\n";
      print "<tr><td>Show up late?</td><td>{$_POST['late']}</td></tr>\n";
      print "<tr><td>Comment your code?</td><td>{$_POST['usecomments']}</td></tr>\n";
      print "<tr><td>State of residence</td><td>{$_POST['state']}</td></tr>\n";
      print "<tr><td>Username</td><td>{$_POST['username']}</td></tr>\n";
      print "<tr><td>Password</td><td>{$_POST['passwd']}</td></tr>\n";
      print "<tr><td>Comments</td><td>{$_POST['comments']}</td></tr>\n";
      print "<tr><td>Promotion code</td><td>{$_POST['promotion_code']}</td></tr>\n";
    ?>
    </table>
  </body>
</html>
```

If a value for a form widget was not specified – for example, if no text was supplied for the text box named `username` – the value for that entry in the `$_GET` or `$_POST` associative array will be the empty string. The empty string is simply the result of trying to print the value of an undefined key in an associative array.

Menus with multiple selections

When using a menu widget that allows multiple selections, an empty pair of brackets must be added to the `name` attribute of the `select` element. Recall the menu definition from the previous XHTML markup:

```
<select name="state[]" size="5" multiple="multiple">
  ...
</select>
```

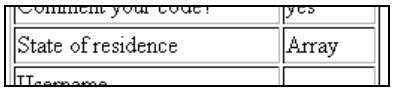
If the brackets are omitted, only the bottom-most selected menu choice will be available via the `$_GET` or `$_POST` associative arrays. For example, suppose that three menu options were selected: Alabama, Alaska, and Arkansas.

Without the brackets, the value of `$_POST['state']` is the string `"Arkansas"`. However, if the `name` attribute's value for the `select` element is changed to `state[]`, the value stored in `$_POST['state']` is an array with three string elements. There are several things to note here:

- When multiple menu options are selected, the form data sent to the server actually contains multiple name-value pairs. For example, the URL encoding for a GET request may look something like the following:

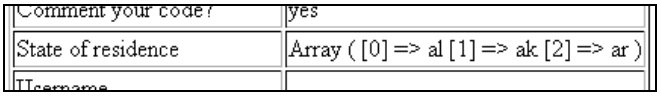
```
...&state=alabama&state=alaska&state=arkansas...
```

- The value of `$_POST['state']` will be an array even if only one item in the menu is selected. In this case `$_POST['state']` will be an array with one element.
- Arrays will be covered in a later document. However, there are two concepts worth mentioning at this point.
    1. If you call `print()` with an array, the output will be `Array`. Here is how the output may appear in a browser:



    2. To display the contents of an array, use `print_r()` – meaning "print recursively". The output format is rather crude, but it allows you to see the contents of the array. The modified line of PHP code and example browser output follows:

```
print "<tr><td>State of residence</td><td>";
print_r($_POST['state']);
print "</td></tr>\n";
```



- The requirement of modifying the `name` attribute seems to be unique to PHP. For example, Perl's CGI interface automatically converts the multiple selections into an array.

Useful string functions

This section lists common PHP functions that manipulate strings in the context of XHTML forms. For more information as well as examples of these functions, use the PHP online reference by visiting http://www.php.net/*function*, where *function* is the name of the PHP function of interest.

Function	Description
`addslashes($str)`	Replaces single quotes, double quotes, and backslashes in string `$str` with their escaped equivalents (i.e., `\'`, `\"`, and `\\` respectively).
`crypt($str)`	Returns an encrypted version of string `$str` using the standard Unix DES-based encryption algorithm.
`htmlentities($str)`	Returns a new string where the characters in string `$str` that are illegal in XHTML (e.g., `&`, `<` and `"`) with their legal equivalents (e.g., `&`, `<`, and `&quot;` respectively).
`html_entity_decode($str)`	Returns a new string where the ampersand-escaped XHTML entities in string `$str` are replaced with their text equivalents.

`htmlspecialchars($str)`	Returns a new string where ampersands, single quotation marks, double quotation marks, less-than characters, and greater-than characters in string `$str` are replaced with their legal XHTML equivalents (e.g., `&`, `&#039;`, `&quot;`, `<`, and `>` respectively). *NOTE*: This function only performs a subset of the replacements handled by `htmlentities()`.
`nl2br($str)`	Returns a new string where XHTML line breaks (i.e., ` `) are placed before the newline characters in string `$str`. *NOTE*: The newline characters are not replaced.
`rawurldecode($str)`	Returns a new string where percent-escaped entities in URL string `$str` are replaced with their text non-alphabetic equivalents.
`rawurlencode($str)`	Returns a new string where non-alphabetic symbols in string `$str` are replaced with their percent-escaped equivalents.
`strip_tags($str)`	Returns a new string where all XHTML tags have been removed from string `$str`.
`stripslashes($str)`	Returns a new string where escaped characters (e.g., `\"`) in string `$str` are replaced with their text equivalents (e.g., `"`).

Cross-site scripting

"Cross-site scripting (XSS) is a type of computer security vulnerability typically found in web applications which allow code injection by malicious web users into the web pages viewed by other users" (Wikipedia). There are several types of cross-site scripting, but this section will cover the simplest method – DOM-based XSS.

The Document Object Model (DOM) is a standard for representing XML or HTML documents as a hierarchy of objects. For example, a browser window contains an XHTML document. This document, denoted by the `html` element, contains a `body` element, which in turn contains other XHTML elements. In DOM-based XSS, the vulnerability is exploited through client-side scripts (e.g., JavaScript) via the user's Web browser. These scripts access various parts of the DOM in ways that the user (and the Website's authors) did not intend. The most common example of a DOM-based XSS vulnerability is to have the user click on a link which takes the user to an unexpected site.

Suppose you are writing PHP code to handle forum posts. Your page has two purposes: to display existing comments/posts by retrieving content from a database, and to provide a Web form to take input from the user (i.e., allowing the user to post to the forum). To keep the coding simple, suppose you decide to store the content of the form input *as is* into the database. Most of the time, this method will not cause many problems; however, consider the following "post":

```
<a href="#" onclick="document.location.href='http://www.downloadvirus.com'">Sort posts by date</a>
```

A casual user will simply see a link labeled "Sort posts by date." Given that the user would likely see such a link in a forum, he/she would assume that clicking on the link would perform the desired operation. However, when the link is clicked, the JavaScript assignment statement (i.e., the value of the link's `onclick` attribute) takes effect. Two components of the DOM hierarchy are used in the left-hand side of the assignment: the document object and the location object, where the location object contains information about the URL of the document. The location object has member named `href` that holds the URL of the current Website. If changed, the browser will navigate to the given URL. Thus when the user clicks the link, the user's Web browser navigates to "http://www.downloadvirus.com", which would presumably cause unexpected events to occur (e.g., downloading a virus).

If your simplistic PHP code simply queries the database and prints the content of the post, your page opens up the potential for DOM-based XSS. An easy solution to this problem is to "sanitize" the result of the database query via the `htmlentities()` function. Instead of printing the malicious post as the attacker intended (i.e., XHTML markup), the XHTML characters will be replaced by their entity equivalents:

```
&lt;a href=&quot;#&quot; onclick=&quot;document.location.href=&#039;http://www.downloadvirus.com&#039;
&quot;&gt;Sort posts by date&lt;/a&gt;
```

The above when rendered as XHTML will *appear* as XHTML on the Web page, but will not *function* as the attacker intended.

Session Management Using Cookies

Cookies are small files that exist on the client's computer, which store information that a Web site can access. Cookies allow state information to be stored locally, meaning that context between pages can exist in a separate file rather than being passed as part of the URL. Sessions can be thought of as "server-side cookies:" the information is stored on the Web server because of the potential for cookies to be altered on the client's machine. Sessions are implemented via cookies, where a cookie holds a value that allows the server to identify a particular client.

Here are some general attributes about cookies:

- Data can be stored for long periods of time. For example, cookie data can exist after the user closes his/her Web browser.
- Synchronization of data within Web server clusters, where multiple servers handle requests, is not needed.
- Information can be accessed on the client-side via JavaScript.

Likewise, here are some general attributes about sessions:

- Information is stored on the server and is thus protected from client manipulation.
- Session data does not need to be transmitted with each page request; only the session identifier is required.
- Session sizes are limited by the server rather than the user's Web browser.

Using cookies

Cookies have a minimum of three attributes: name, value, and expiration time (i.e., time after which the cookie's data will no longer be valid). Use the `setcookie()` function to provide values for these three attributes:

```
setcookie(username, "dknuth", time() + 86400);
```

The above example creates a cookie named `username` with a value of `dknuth`, and will expire in 86400 seconds (24 hours) after the page has been loaded. The `setcookie()` function has three additional parameters:

- *Availability path*. The directory in which the cookie is available. The default availability path is `/`, meaning the cookie is available over the entire site. An availability path of `/foo/` means the cookie will only be available in the `foo` directory and its subdirectories.
- *Domain*. The subdomain in which the cookie is available. For example a domain of `secure.programmingisfun.net` will make the cookie available there but not in `www.programmingisfun.net`. To make a cookie available in both domains, a domain of `.programmingisfun.net` should be used. The default value for the domain is the empty string, meaning that the cookie is available from any domain.
- *Secure connection*. An indicator of whether the cookie must only be sent through a secure HTTPS connection or not. The default value is zero, indicating HTTP or HTTPS are acceptable; a value of one indicates that only HTTPS can be used.

Cookie data is accessed via the superglobal `$_COOKIE`, where the *key* is the cookie name.

```
if(isset($_COOKIE['username']))
  print "Welcome, $_COOKIE[username]!\n";
else
  print "Welcome, guest!\n";
```

The structure of HTTP requires that cookie information be part of the header information, as opposed to the "body" information, which actually contains the page markup. When setting a cookie, the Web server includes header data for that cookie; therefore, calls to `setcookie()` must occur *before* you being sending page markup.
Cookie data is sent to the Web server each time a user visits a particular page. If your PHP code sets a cookie, that cookie will not be available on the first rendering of the page. The reason is because the cookie data is not part of the HTTP request for that page. However, the next time that page is loaded, the cookie data *will* be part of the HTTP request, and therefore the cookie can be accessed.

Suppose you have the following PHP code and that the cookie named username does not exist.

```
<?php
  setcookie("username","dknuth",time()+3600);
```

```
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
    <title>Programming is Fun!</title>
  </head>
  <body>
    <p>Welcome,
    <?php
      if (isset($_COOKIE['username']))
        print "$_COOKIE[username]. ";
      else
        print "guest. ";
    ?>
    Enjoy your stay!</p>
  </body>
</html>
```

The text displayed in the browser will be “Welcome, guest.” However, if the page is reloaded, the text displayed will be “Welcome, dknuth” because the cookie has been sent in the request (i.e., reload) of the page.

To delete a cookie, re-set the cookie using an empty string as the value and a negative number as the expiration time.

```
setcookie(username, "", -1);   // Delete the cookie named 'username'
```

Using sessions

To begin a session, call the `session_start()` function. This function checks to see if the visitor sent a cookie with a session ID. If such a cookie was sent, the session data is loaded into the superglobal `$_SESSION`; otherwise, a new session file and corresponding cookie are created.

Just as cookies are accessed through the superglobal `$_COOKIE`, session variables are accessed through the superglobal `$_SESSION`:

```
session_start();
…
print "Welcome, $_SESSION['username']!";
```

Unlike cookies, session data is available as soon as it is set.

To add session data – analogous to setting a cookie – assign a value to the `$_SESSION` variable. The following example assigns the string value `"PHP"` to the `favlang` session variable.

```
$_SESSION['favlang'] = "PHP";
```

A session lasts until the user closes his/her Web browser. To explicitly end a session, the `$_SESSION` array must be cleared and the session data on the Web server must be removed. Here is an example:

```
session_start();
$_SESSION = array();
session_destroy();
```

Note that the `session_start()` call is necessary to have any subsequent PHP code affect the users session. Without this call the `$_SESSION` array will already be empty, and the `session_destroy()` call will not have any effect because the PHP code does not know that a session is in progress.

Example

For a simple example of how to use the PHP session mechanism, take a look at the following three files:

1. http://web.eecs.utk.edu/~bvz/cgi-bin/pizzaSession.php

2. http://web.eecs.utk.edu/~bvz/cgi-bin/pizzaSession1.php
3. http://web.eecs.utk.edu/~bvz/cgi-bin/pizzaPhpSession.html

These three files manage a sample set of html form elements. The html page is the entry page and pizzaSession.php handles this page. It starts a PHP session and saves the form element information in the $_SESSION table. It then generates an html page that echos the form data and asks the user to press a re-confirm button. When the user does so, pizzaSession1.php gets invoked and accesses the $_SESSION table in order to re-echo the form data.

Limitations

One unfortunate limitation of PHP's session mechanism is that $_SESSION tables do not work if you use multiple servers. Each server keeps its own individual copy of the $_SESSION table, so if one server serves the first page and another server serves the second page, then the script on the second server will not be able to access the $_SESSION information that is stored on the first server. The unique session id however will still be available however, and you can use that as a retrieval key into a database. So you can always store the session information in a database and then retrieve it using the session id as a key. Another limitation of PHP's session id is that it often is stored as a cookie on the user's browser, and so a malicious user could try to tamper with the session id. There are methods for trying to build in additional security, but they are beyond the scope of this course.