

Programming in PHP X52.9224

Instructor : David Mintz <dmintz@davidmintz.org>

http://davidmintz.org/php_course/

Note: For complete technical documentation for PHP, see <http://php.net/manual/en>

Session Five

Outline: Handling file uploads; HTTP header and output control functions; HTTP cookies; maintaining state with HTTP sessions; configuring PHP via Apache

POST method uploads

Says the PHP manual (almost verbatim): This feature lets people upload both text and binary files. PHP is capable of receiving file uploads from any RFC-1867-compliant browser (which includes Netscape Navigator 3 or later, Microsoft Internet Explorer 3 with a patch from Microsoft, or later without a patch).

Relevant php.ini settings:

```
file_uploads enable/disable uploads (on or off)
upload_max_filesize max filesize, default 2M
upload_tmp_dir temporary directory for uploaded files (system default if not specified).
post_max_size maximum size of POST data that PHP will accept.
max_input_time maximum number of seconds each script may spend parsing request data
```

Example File Upload Form

To accept file uploads you will need to display a form like the following:

```
<!-- The data encoding type, enctype, MUST be specified as below -->
<form enctype="multipart/form-data" action="your_upload_handler.php"
method="POST">
  <!-- MAX_FILE_SIZE must precede the file input field -->
  <input type="hidden" name="MAX_FILE_SIZE" value="30000" />
  <!-- Name of input element determines name in $_FILES array -->
  Send this file: <input name="userfile" type="file" />
  <input type="submit" value="Send File" />
</form>
```

The MAX_FILE_SIZE hidden field (measured in bytes) must precede the file input field, and its value is the maximum filesize accepted. This is an advisory to the browser; PHP also checks it.

Fooling this setting on the browser side is quite easy, so never rely on files with a greater size being blocked by this feature. The PHP settings for maximum-size, however, cannot be fooled. This form element should always be used as it saves users the trouble of waiting for a big file being transferred only to find that it was too big and the transfer failed.

Your file upload form must have the attribute `enctype="multipart/form-data"` in order for the file upload to work.

The global `$_FILES` array exists as of PHP 4.1.0 (You would use `$HTTP_POST_FILES` instead if you were stuck with an older PHP version). These arrays will contain all the uploaded file information.

The contents of `$_FILES` from the example form is as follows. Note that this assumes the use of the file upload name *userfile*, as used in the example script above. This can be any name.

`$_FILES['userfile']['name']` The original name of the file on the client machine.

`$_FILES['userfile']['type']` The mime type of the file, if the browser provided this information. An example would be "image/gif".

`$_FILES['userfile']['size']` The size, in bytes, of the uploaded file.

`$_FILES['userfile']['tmp_name']` The temporary filename of the file in which the uploaded file was stored on the server.

`$_FILES['userfile']['error']` The error code associated with this file upload. This element was added in PHP 4.2.0

Files will by default be stored in the server's default temporary directory, unless another location has been given with the `upload_tmp_dir` directive in `php.ini`

Example: Validating file uploads

See also the function documentation entries for `is_uploaded_file()` and `move_uploaded_file()` for further information. The following example will process the file upload that came from a form.

```
<?php
// In PHP versions earlier than 4.1.0, $HTTP_POST_FILES
// should be used instead of $_FILES.

$uploaddir = '/var/www/uploads/';
$uploadfile = $uploaddir . basename($_FILES['userfile']['name']);
echo '<pre>';
if (move_uploaded_file($_FILES['userfile']['tmp_name'], $uploadfile)) {
    echo "File is valid, and was successfully uploaded.\n";
} else {
```

```

    echo "Possible file upload attack!\n";
}

echo 'Here is some more debugging info: ';
print_r($_FILES);

print "</pre>";
?>

```

The PHP script which receives the uploaded file should implement whatever logic is necessary for determining what should be done with the uploaded file. You can, for example, use the `$_FILES['userfile']['size']` variable to throw away any files that are either too small or too big. You could use the `$_FILES['userfile']['type']` variable to throw away any files that didn't match a certain type criteria. As of PHP 4.2.0, you could use `$_FILES['userfile']['error']` and plan your logic according to the error codes. Whatever the logic, you should either delete the file from the temporary directory or move it elsewhere.

If no file is selected for upload in your form, PHP will return `$_FILES['userfile']['size']` as 0, and `$_FILES['userfile']['tmp_name']` as none.

The file will be deleted from the temporary directory at the end of the request if it has not been moved away or renamed.

HTTP Headers

When I point my Mozilla browser at `http://nytimes.com`, my browser sends this request to `nytimes.com`:

```

GET / HTTP/1.1
Host: nytimes.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.4)
Gecko/20030624
Accept: text/xml,
application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/
x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

```

This is an *HTTP request* (in accordance with the HTTP 1.1 Protocol) The first line is called the Request line. It specifies the *method* (GET), the resource location (/ (forward slash), representing the default page in the server's document root directory), and the protocol. The `Host` header identifies the host that the browser believes itself to be contacting -- because a single physical server can and often does support multiple websites; this header cooperates with virtual hosting. Next comes the `User-Agent` header, with which the browser identifies itself And so on.

When `nytimes.com` responds, it sends response headers, which precede the actual document body transmitted across the network:

```

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1

```

```
Date: Tue, 19 Oct 2004 14:09:02 GMT
Content-type: text/html
Set-cookie: RMID=832fa3630a3c41751ffe5f27; expires=Wednesday, 19-Oct-2005
14:09:02 GMT; path=/; domain=.nytimes.com
Set-cookie: tpopunder_orbitz33-nyt4=1098194942; expires=Wednesday, 01-Dec-
2004 03:59:59 GMT; path=/; domain=.nytimes.com
Set-cookie: spopunder=1; path=/; domain=.nytimes.com
Cache-control: no-cache
Pragma: no-cache
Transfer-encoding: chunked
```

The New York Times' web server's response adheres to the HTTP 1.1 protocol by first sending a response line that identifies the protocol it's speaking, followed by a response status code, followed by a short description of the status code.

For an explanation of all this and other stuff, see, for example, <http://en.wikipedia.org/wiki/Http>; <http://www.ietf.org/rfc/rfc2616.txt>; Chris Shiflett, *The HTTP Handbook* (Developer's Library, ISBN 0-672-32454-7; <http://shiflett.org/books/http-developers-handbook>)

When you send output by means of a PHP page, the response header generation is performed automatically by the web server, with cooperation from PHP. Thus when I ask my browser to display the index page of a virtual host that I have configured on my local machine, the response headers preceding the response body are

```
HTTP/1.1 200 OK
Date: Tue, 19 Oct 2004 13:57:15 GMT
Server: Apache/2.0.50 (Win32) mod_ssl/2.0.50 OpenSSL/0.9.7c PHP/5.0.1
X-Powered-By: PHP/5.0.1
Content-Length: 1392
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1
```

even if my `index.php` does nothing about header output.

However, PHP gives you an opportunity to manipulate response headers yourself, and thereby achieve some powerful effects.

The `header()` function lets you add your own http headers to the response, or override the default headers. The function's signature is

```
void header ( string string [, bool replace [, int http_response_code]])
```

The optional second argument indicates whether this header should replace any existing identical header, or be added to the headers. It defaults to false. The optional third argument is an HTTP response code -- for communicating with the browser at the HTTP protocol level.

```
<?php
header("X-Powered-By: The Almighty Me 3.46");
header("Content-type: text/plain");
echo "Hello you silly world at " . date('r');
?>
```

Response, both headers and body:

```
HTTP/1.1 200 OK
Date: Tue, 19 Oct 2004 15:31:36 GMT
Server: Apache/2.0.50 (Win32) mod_ssl/2.0.50 OpenSSL/0.9.7c PHP/5.0.1
X-Powered-By: The Almighty Me 3.46
Content-Length: 56
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/plain; charset=ISO-8859-1
```

Hello you silly world at Tue, 19 Oct 2004 11:31:37 -0400

The Location header

Of particular interest is the "Location: " header, with which you can ask the browser to request a another URL. If you set this, PHP also takes care of sending the proper Redirect/302 http status code. You sometime do not want a POST request to be repeated (think placing an order for a product, or posting a message on a BBS), and a standard technique for discouraging duplicate submission is to redirect the browser to a confirmation page, which the user can reload harmlessly.

```
<?php
// form data processed successfully , redirect
header("Location: http://example.com/confirm.php");
exit;
?>
```

Two points about the above:

Just about any browser will comply the the Location header, but if your script is accessed by a program that is not a browser, it will ignore the Redirect status code and the Location header, and keep downloading data. Therefore whenever you redirect, you should explicitly `exit()` afterwards and not send any more output, unless you have compelling reason to do otherwise.

According to HTTP 1.1, the Location header requires an absolute URI , meaning protocol, host, path to resource, etc, not just a relative URI. You can figure out your hostname and the path to your current directory on the fly (i.e., avoid hard-coding, and make your code reasonably portable from one environment to another):

```
<?php
header("Location: http://" . $_SERVER['HTTP_HOST']
      . dirname($_SERVER['PHP_SELF'])
      . "/" . $page);
?>
```

The Content-type header

If you are serving content that is not of the default text/html content type, (e.g., an image, a MS Excel spreadsheet, a PDF document) you will need to set the `Content-type` header so that the browser will know how to to handle it (or at least take a shot at trying).

You can also use the `Content-disposition` header encourage the browser to display a Save as... dialogue so that the user will save the resource to disk instead of seeing it displayed in the browser. Hence:

```
<?php
// We'll be outputting a PDF so set the content type
header('Content-type: application/pdf');

// it will be called downloaded.pdf on their machine
header('Content-Disposition: attachment; filename="downloaded.pdf"');

// The PDF source is in original.pdf
readfile('original.pdf');
?>
```

We say "encourage" rather than force the browser to pop up a Save... dialogue, because not all browsers behave the same. Some versions of MSIE don't cooperate.

See <http://us2.php.net/manual/en/function.header.php> for details.

Output Buffering

By definition, *the response headers precede the body*. Therefore you cannot send any headers after your PHP program has sent any output to the browser. If you do, the call to `header()` will fail and trigger a warning:

```
<?php //redirection.wrong.php
echo "Hello world";
header("Location: http://example.com/someplace/else");
exit;
?>
```

Output (assuming `error_reporting` is verbose enough, `display_errors` is on, and buffering has not been turned on):

```
Hello world
Warning: Cannot modify header information - headers already sent by (output
started at C:\data\docs\php_course\examples\week5\Redirect.wrong.php:3) in
C:\data\docs\php_course\examples\week5\Redirect.wrong.php on line 4
```

The way to get around this, if you need to, is *output buffering*. You can control output buffering behavior with `php.ini` settings, or manipulate it at runtime with the `ob_XXX` functions.

Here is a (mindless) example of how buffering enables you to decide at runtime whether to redirect the browser even after you have said "echo 'something'" or otherwise sent output to the browser:

```
<?php
// start buffering output
ob_start();
echo "Hello world...<br />";
echo "Shall we re-direct? Let's toss a coin.";

$redirect = rand(0,1);
```

```

if ($redirect) {
    header(
        "Location: http://$_SERVER[HTTP_HOST]"
        . dirname($_SERVER['PHP_SELF'])
        . "/redirection_target.php"
    );
    ob_end_clean(); // discard buffer
    exit;
} else {
    ob_end_flush(); // stop buffering and flush buffer contents
    echo "...no, I decided to stay here";
}
?>

```

Another case where you might use buffering is when you need to capture the output of a function that insists on printing its output. To save the `print_r()` output in a string so you could write it to a file or email it somewhere, you would

```

<?php
ob_start(); // start buffering
print_r($foo); // do what you have to do
$string = ob_get_clean(); // now the print_r output is in $string
?>

```

Buffering makes other interesting stuff possible, such as output compression, with which you can send large documents in compressed form and vastly reduce the bandwidth the transfer consumes. See <http://www.php.net/manual/en/ref.outcontrol.php>

```

flush -- Flush the output buffer
ob_clean -- Clean (erase) the output buffer
ob_end_clean -- Clean (erase) the output buffer and turn off output buffering
ob_end_flush -- Flush (send) the output buffer and turn off output buffering
ob_flush -- Flush (send) the output buffer
ob_get_clean -- Get current buffer contents and delete current output buffer
ob_get_contents -- Return the contents of the output buffer
ob_get_flush -- Flush the output buffer, return it as a string and turn off output buffering
ob_get_length -- Return the length of the output buffer
ob_get_level -- Return the nesting level of the output buffering mechanism
ob_get_status -- Get status of output buffers
ob_gzhandler -- ob_start callback function to gzip output buffer
ob_implicit_flush -- Turn implicit flush on/off
ob_list_handlers -- List all output handlers in use
ob_start -- Turn on output buffering
output_add_rewrite_var -- Add URL rewriter values
output_reset_rewrite_vars -- Reset URL rewriter values

```

Cookies

HTTP is what is known as a *stateless* protocol. Each http client request/server response transaction is a new ballgame; the server has no clue whether the current client is the same client who requested x or y at some time in the past. HTTP Cookies are a mechanism for storing data in

the remote browser and thus tracking or identifying return users. The underlying protocol is still stateless; cookies are a means of simulating state.

The server (or a program running on it) does not actually set any cookies. Rather, it sends a `Set-Cookie` header to the browser, which in turn either accepts the cookie, or rejects it, depending on the browser settings and user decision.

This is the format of the HTTP cookie header that sends a new piece of data which is to be stored by the client for later retrieval, according to http://wp.netscape.com/newsref/std/cookie_spec.html:

`Set-Cookie: NAME=VALUE; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure`

NAME=VALUE

This string is a sequence of characters excluding semi-colon, comma and white space. If there is a need to place such data in the name or value, some encoding method such as URL style %XX encoding is recommended, though no encoding is defined or required.

This is the only required attribute on the Set-Cookie header.

expires=DATE

The expires attribute specifies a date string that defines the valid life time of that cookie. Once the expiration date has been reached, the cookie will no longer be stored or given out.

The date string is formatted as:

Wdy, DD-Mon-YYYY HH:MM:SS GMT

This is based on RFC 822, RFC 850, RFC 1036, and RFC 1123, with the variations that the only legal time zone is GMT and the separators between the elements of the date must be dashes.

expires is an optional attribute. If not specified, the cookie will expire when the user's session ends.

domain=DOMAIN_NAME

When searching the cookie list for valid cookies, a comparison of the domain attributes of the cookie is made with the Internet domain name of the host from which the URL will be fetched. If there is a tail match, then the cookie will go through path matching to see if it should be sent. "Tail matching" means that domain attribute is matched against the tail of the fully qualified domain name of the host. A domain attribute of "acme.com" would match host names "anvil.acme.com" as well as "shipping.crate.acme.com".

Only hosts within the specified domain can set a cookie for a domain and domains must have at least two (2) or three (3) periods in them to prevent domains of the form: ".com", ".edu", and "va.us". Any domain that fails within one of the seven special top level domains listed below only require two periods. Any other domain requires at least three. The seven special top level domains are: "COM", "EDU", "NET", "ORG", "GOV", "MIL", and "INT".

The default value of domain is the host name of the server which generated the cookie response.

path=PATH

The path attribute is used to specify the subset of URLs in a domain for which the cookie is valid. If a cookie has already passed domain matching, then the pathname component of the URL is compared with the path attribute, and if there is a match, the cookie is considered valid and is sent along with the URL request. The path `"/foo"` would match `"/foobar"` and `"/foo/bar.html"`. The path `"/"` is the most general path.

If the path is not specified, it is assumed to be the same path as the document being described by the header which contains the cookie.

secure

If a cookie is marked secure, it will only be transmitted if the communications channel with the host is a secure one. Currently this means that secure cookies will only be sent to HTTPS (HTTP over SSL) servers.

If secure is not specified, a cookie is considered safe to be sent in the clear over unsecured channels.

Syntax of the Cookie HTTP Request Header

When requesting a URL from an HTTP server, the browser will match the URL against all cookies and if any of them match, a line containing the name/value pairs of all matching cookies will be included in the HTTP request. Here is the format of that line:

```
Cookie: NAME1=OPAQUE_STRING1; NAME2=OPAQUE_STRING2 ...
```

Naturally, PHP does not make you send raw http cookie headers yourself, but instead provides a convenient high-level interface to cookies. The `setcookie()` function implements cookie-setting in a way that is not-so-coincidentally reminiscent of the specification above:

```
bool setcookie ( string name [, string value [, int expire [, string path [, string domain [, bool secure]]]]])
```

`setcookie()` defines a cookie to be sent along with the rest of the HTTP headers. Like other headers, cookies must be sent before any output from your script. otherwise, `setcookie()` will fail and return FALSE. If `setcookie()` successfully runs, it will return TRUE. -- although this does mean the user necessarily accepted the cookie.

All the arguments except the name argument are optional. You may also replace an argument with an empty string (`""`) in order to skip that argument. Because the expire argument is integer, it cannot be skipped with an empty string, use a zero (0) instead. The table found at <http://us2.php.net/setcookie> explains in detail each parameter of the `setcookie()` function. Briefly:

- *name* is the only obligatory argument

- *value* is a string - defaults to empty string (ergo sets cookie to false). this is the data you want the cookie to hold
- *expire* is a Unix timestamp -- you can set it with `time() + n seconds`, or `mkdate()`. defaults to zero, meaning cookie expires when the user closes the browser
- *path* is the path within and beneath which the cookie is valid within your site
- domain essentially means you can pass is `.mysite.com` to make it valid at `eexample.com` and `xample.mysite.com`
- *secure* is a boolean that means whether HTTPS is required

Recall our earlier discussion about output buffering. You can use it to send output prior to calling this function, at the cost of the overhead of all of your output to the browser being buffered in the server until you send it. Again, you can do this by calling `ob_start()` and `ob_end_flush()` in your script, or by setting the `output_buffering` configuration directive on in your `php.ini` or web server configuration files.

Example:

```
<?php
setcookie('php_test_cookie','your test cookie value');
echo "Hello world";
?>
```

First time I access the above page (in this session), my request headers are:

```
GET /examples/week5/cookie_setter.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.4)
Gecko/20030624
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
```

And the server response:

```
HTTP/1.1 200 OK
Date: Tue, 19 Oct 2004 21:30:31 GMT
Server: Apache/2.0.50 (Win32) mod_ssl/2.0.50 OpenSSL/0.9.7c PHP/5.0.1
X-Powered-By: PHP/5.0.1
Set-Cookie: php_test_cookie=your+test+cookie+value
Content-Length: 14
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=ISO-8859-1
```

Hello world.

My browser is configured to ask me if I want to accept the cookie, and I say yes. Next time I request this or any page on my localhost domain during the course of this session, the requests headers look like:

```
GET /examples/week5/cookie_setter.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.4)
Gecko/20030624
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,video/x-mng,image/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: php_test_cookie=your+test+cookie+value
Cache-Control: max-age=0
```

That's all there is to setting cookies.

If you want to *get* the cookies that are being sent to your script from the browser, PHP makes your life easy by storing them in the auto-global array `$_COOKIE`. You consult it just like any other PHP array.

```
<?php
if (array_key_exists('php_test_cookie',$_COOKIE)) {
    echo "<p>looks like you accepted my cookie.</p>";
    if ($_COOKIE['php_test_cookie']) {
        echo "<p>and it says: <em>{$_COOKIE['php_test_cookie']}</em>";
    }
}
}
```

Sessions

Cookies make it possible to simulate state by getting the client to store data which persists across multiple requests. But there are limits (per the cookie specification) to how much data you can stuff in a cookie, and how many cookies you can expect the client to accept. It's also possible that a user whose client is asking for confirmation of each cookie will accept some of your cookies and reject others, and put your application in a strange state. And you cede control of the data to the client when you rely on the client to store it for you. Cookies don't identify a person, but rather a combination of browser and computer.

HTTP sessions are a mechanism that implements state by putting the responsibility for data persistence on the server rather than on the client. A client accessing your site is assigned an unique id, the so-called session id, which usually is either stored in a cookie on the user side or is propagated in the URL. What the session id identifies is the data on the server side corresponding to that client.

You start a session by calling `session_start()`:

bool session_start (void)

session_start() either creates a new session or resumes the current one based on the current session id that's being passed via a request, such as GET, POST, or a cookie. This function always returns TRUE.

Note that if you are using cookie-based sessions -- and we will shortly talk about how you control this -- then the call to session_start() must precede any output to the browser. Why? It probably won't surprise you to learn that unless PHP is explicitly configured otherwise, session_start() sets a Set-cookie http header that asks the browser to accept a cookie holding the session id. In other words, when my browser fetches a PHP page that starts a session, the response headers include:

```
Set-Cookie: PHPSESSID=f1568b4649d1e11143c8c1299afdaf85; path=/
```

The default session storage implementation is filesystem-based. PHP stores session data in files located in whatever php's session.save_path configuration variable is set to. On the Windows box where I am composing this, that is C:/xampp/tmp; and in there is a file called, not coincidentally, sess_f1568b4649d1e11143c8c1299afdaf85. This is where PHP is stashing my session data between page requests.

You as PHP programmer set and get session variables by accessing the auto-global array \$_SESSION. A trivial example of setting some session variables.

```
<?php
session_start();
$_SESSION['your_name'] = 'David';
$_SESSION['lucky_number'] = rand(1,100);
$_SESSION['age'] = 46;
$_SESSION['likes_beer'] = true;
?>
```

After a client accesses the page that executes the above code, these session variables are automatically visible to subsequent pages accessed by that same client in the course of the session.

Behind your back, what actually happens is that PHP creates a string representation of the \$_SESSION array and writes it in your session file at the end of each request. When you say session_start(), PHP looks for the session file and, if it finds it, reads that same data from disk and uses it to populate \$_SESSION. If it doesn't find the session file, it creates an empty one.

The conversion between an in-memory data structure (such as an array) into a data stream of byte values (so that the data can be written to disk or sent over the network) is called *serialization*. The reverse process is called *deserialization*. After running the above snippet in our browser, our session file contains:

```
your_name|s:5:"David";lucky_number|i:76;age|i:46;likes_beer|b:1;
```

This reflects PHP's method of serializing data. (Apparently, at least with scalar data, it records the name, data type and value of each variable and delimits each with a semicolon.)

The way you delete an individual session variable is with the `unset()` function.

```
<?php
    session_start();
    unset($_SESSION[$variable_name]);
?>
```

Unsetting the entire `$_SESSION` variable is not recommended. Note that `unset()` is not specifically a session handling function; it is used for unsetting any variable.

The way to wipe out all the session data is with `session_destroy()`. The session will still be around, but it won't have any data in it. To completely obliterate the session, you should also unset the session id cookie by using `setcookie()` to set it to an empty string, with an expiration date that is in the past:

```
<?php
if (isset($_COOKIE[session_name()])) {
    setcookie(session_name(), '', time()-42000, '/');
}
?>
```

Cookies, URL, or both?

PHP has a plethora of configuration settings and functions for controlling how sessions are implemented. If your user refuses the cookie containing the session id, you can still propagate the session id from page to page by appending it to every URL -- and PHP provides a means to automate this known as trans-sid (transparent session identification, presumably)..

Or you can configure PHP to use only cookies and explicitly disallow url-rewriting. The relevant settings are

- boolean `session.use_trans_sid`. Determines whether transparent sid support is enabled or not. Defaults to 0 (disabled). If you enable this, and your visitor declines your `session_id` cookie, URLs will automatically be rewritten in the form `page.php?PHPSESSID=0df0e5451079b0316bcf5b0cfc1bcf97`. Now `page.php` will be able to pick up the session with or without a session id cookie set on the user side. However, **URL-based session management entails a security risk**. It is easier to leak session ids and have a session hijacked this way than with cookies, many argue.
- boolean `session.use_cookies`. Specifies whether the session module will use cookies to store the session id on the client side. Defaults to 1 (enabled).
- boolean `session.use_only_cookies`. Specifies whether the session module will only use cookies to store the session id on the client side. Defaults to 0 (disabled, for backward compatibility). Enabling this setting prevents attacks involving passing session ids in URLs. This setting was added in PHP 4.3.0.

See <http://www.php.net/manual/en/ref.session.php> and friends for the full story about PHP sessions.

Configuring PHP via Apache's configuration files

We have talked about controlling PHP's behavior by modifying `php.ini` and by using `ini_set()`. Because PHP and Apache play so nicely together, there is a third way: putting PHP configuration directives in Apache's configuration files. The syntax for setting non-boolean values is

```
php_value name value
```

For setting booleans:

```
php_flag name on|off
```

If you have control over the server and can edit Apache's master configuration file `httpd.conf`, then you can enter config directives right there and restart the server for them to take effect.

If you cannot or do not want to use `httpd.conf`, and if your `httpd.conf` has per-directory runtime configuration enabled, then you can use this technique by putting config directives in a file usually called `.htaccess`, and putting `.htaccess` in the directory where you want it to take effect – note that it will also affect all of that directory's descendants.

Not every configuration setting is settable in every context. For a complete list of what can be modified where, see <http://php.net/manual/en/ini.php#ini.list>. You should also be aware of how these various interfaces to PHP configuration may override one another. For directives that can be set in any context, an `httpd.conf` setting will override `php.ini`; `.htaccess` overrides `httpd.conf`; and `ini_set()` will override everything else.

One useful application of all this flexibility is making your code portable from one environment to another. For example, your `include_paths` on your development and production machines may well be different. If you set `include_path` appropriately via a `.htaccess` file on each system, you should be able.

Session Security

The way PHP sessions work out of the box is not particularly secure. If you plan to store sensitive data in sessions, you should take extra security measures -- even more so if you are running on a shared (as opposed to dedicated) server, where anyone with an account on your host can try to peruse anyone's else's serialized session data. For starters, you can disable `session.use_trans_sid`, enable `session.use_only_cookies`, and require your application to use SSL. See <http://shiflett.org/articles/the-truth-about-sessions> for a good article on session security.

Alternative Session Handlers

We've only been discussing PHP's filesystem-based session management. PHP supports user-defined session storage functions. Other session management techniques involve shared memory and databases. The latter is what high-end, high-traffic sites generally rely on. If you are curious, see <http://www.php.net/manual/en/function.session-set-save-handler.php>

Call `session_start()` first

You cannot get or set session data without first calling `session_start()`. It's easy to forget this, so we will say it again: *before you can get or set session variables in `$_SESSION`, you must call `session_start()`.*

We said earlier that `session_start()` sends a Set-cookie http header to the browser. This means that unless you are buffering your output, *you have to call `session_start()` before sending any output to the browser.*

If your session handling relies on cookies and the browser refuses to cooperate by accepting the session id cookie, sessions won't work. That is why you sometimes see notices on some web applications saying you have to accept a cookie.

Assignment

Coding

For this assignment you will borrow the user authentication code from David Sklar *Learning PHP 5* and put it to work. You will be submitting seven files: `formhelpers.php`; `login.php`; `logout.php`; `auth.php`; `1.php`; `2.php`; and `3.php`. If you are impatient, you can save a little time by packing them up into a zip file and uploading them to the class site in one shot; otherwise, upload them one by one. In either case, be sure to name your files according to these instructions. All of these files will reside in the same folder (subdirectory).

Download the code from <http://examples.oreilly.com/learnphp5/> and unpack the zipfile.

Open example 6-29 in your code editor, and save it as `formhelpers.php` in the directory where you want to install your assignment.

Open Example 8-14 in your code editor and save it as "`login.php`" in the same directory. Modify it so that it if and only if the user logs in successfully, `login.php` displays hypertext links to pages `1.php`, `2.php`, `3.php` and `logout.php`

If these code examples are not wrapped in opening and closing PHP tags, be sure to provide them.

Create the three PHP pages called `1.php`, `2.php` and `3.php`, each with a link to the two others -- a crude navigation bar. Also create a link to `logout.php` on each.

Now the actual coding: implement logic that checks whether a session variable called 'username' is set.

If it is not set, *redirect the browser to login.php and exit*. Use something like the following to effect the redirection, so that it will work without modification when I run your code on my machine:

```
header("Location: http://" . $_SERVER['HTTP_HOST']  
      . dirname($_SERVER['PHP_SELF'])  
      . "/login.php");
```

If it is set, display a message that says "Current user: " followed by the value of the session variable 'username'.

Store the code that implements this logic in a separate file called auth.php and use a `require()` statement to load it in each of pages 1.php, 2.php, 3.php and logout.php

Remember that the first thing auth.php should do is `session_start()`.

What should happen is: I fire up my browser and try to access 1.php (or 2.php, or 3.php). I end up at login.php. I enter a bad username and/or password and I get a failed login message. I enter a good username and password, and get a greeting. Now I can freely browse among pages 1.php, 2.php and 3.php.

Now you need to provide a way for users to log out. Create a page called logout.php which loads auth.php (as indicated earlier) and then unsets `$_SESSION['username']` (the `unset()` function will do; so will `session_destroy()`). Display a message that says "you have logged out," and display a link to login.php inviting them to log back in.

Congratulations, you have built a minimal but functional authentication system.

Optional exercises

After completing the assignment as indicated above, come up with ways to

- (a) dynamically generate a basic navigation bar (for example, you could hard-code the paths and page titles of each page in an array, and iterate through it comparing each element to `$_SERVER['PHP_SELF']`. The current page would be "grayed out" and rest would be hypertext links.)
- (b) make all the pages in this directory well-formed, valid HTML documents with a couple of `include()` calls (instead of manually copying-and-pasting blocks of HTML).

Prep your system for MySQL

Next week we will start working with MySQL 4.1 and PHP's `mysqli` interface to MySQL. Please make sure you have `mysqli` support installed and enabled per the instructions at http://davidmintz.org/php_course/forum/index.php?a=topic&t=17 and ask for help if you have any trouble.

Reading

Sklar *Learning PHP 5* Chapter 8; Chapter 13 "Uploading Files in Forms"

The PHP Manual: Cookies, Sessions, Output Buffering, and Handling File Uploads:

- <http://www.php.net/manual/en/ref.outcontrol.php>
- <http://www.php.net/manual/en/ref.http.php>
- <http://www.php.net/manual/en/ref.session.php>
- <http://www.php.net/manual/en/features.file-upload.php>

Last updated 2005-03-22 12:05 pm