PHP with MySQL: cookies, sessions, Super Globals, Debugging, Error and exception handling

Cookies are text files stored on the client computer and they are kept of use tracking purpose. PHP transparently supports HTTP cookies.

There are three steps involved in identifying returning users −

- Server script sends a set of cookies to the browser. For example name, age, or identification number etc.
- Browser stores this information on local machine for future use.
- When next time browser sends any request to web server then it sends those cookies information to the server and server uses that information to identify the user.

This chapter will teach you how to set cookies, how to access them and how to delete them.

The Anatomy of a Cookie

Cookies are usually set in an HTTP header (although JavaScript can also set a cookie directly on a browser). A PHP script that sets a cookie might send headers that look something like this −

```
HTTP/1.1 200 OK
Date: Fri, 04 Feb 2000 21:03:38 GMT
Server: Apache/1.3.9 (UNIX) PHP/4.0b3
Set-Cookie: name=xyz; expires=Friday, 04-Feb-07 22:03:38 GMT;
                 path=/; domain=tutorialspoint.com
Connection: close
Content-Type: text/html
```

As you can see, the Set-Cookie header contains a name value pair, a GMT date, a path and a domain. The name and value will be URL encoded. The expires field is an instruction to the browser to "forget" the cookie after the given time and date.

If the browser is configured to store cookies, it will then keep this information until the expiry date. If the user points the browser at any page that matches the path and domain of the cookie, it will resend the cookie to the server.The browser's headers might look something like this −

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.6 (X11; I; Linux 2.2.6-15apmac ppc)
Host: zink.demon.co.uk:1126
Accept: image/gif, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Cookie: name=xyz
```

A PHP script will then have access to the cookie in the environmental variables $_COOKIE or $HTTP_COOKIE_VARS[] which holds all cookie names and values. Above cookie can be accessed using $HTTP_COOKIE_VARS["name"].

Setting Cookies with PHP

PHP provided **setcookie()** function to set a cookie. This function requires upto six arguments and should be called before <html> tag. For each cookie this function has to be called separately.

```
setcookie(name, value, expire, path, domain, security);
```

Here is the detail of all the arguments −

- **Name** − This sets the name of the cookie and is stored in an environment variable called HTTP_COOKIE_VARS. This variable is used while accessing cookies.
- **Value** − This sets the value of the named variable and is the content that you actually want to store.
- **Expiry** − This specify a future time in seconds since 00:00:00 GMT on 1st Jan 1970. After this time cookie will become inaccessible. If this parameter is not set then cookie will automatically expire when the Web Browser is closed.
- **Path** − This specifies the directories for which the cookie is valid. A single forward slash character permits the cookie to be valid for all directories.
- **Domain** − This can be used to specify the domain name in very large domains and must contain at least two periods to be valid. All cookies are only valid for the host and domain which created them.
- **Security** − This can be set to 1 to specify that the cookie should only be sent by secure transmission using HTTPS otherwise set to 0 which mean cookie can be sent by regular HTTP.

Following example will create two cookies **name** and **age** these cookies will be expired after one hour.

```
<?php
   setcookie("name", "John Watkin", time()+3600, "/","", 0);
   setcookie("age", "36", time()+3600, "/", "",  0);
?>
<html>

   <head>
      <title>Setting Cookies with PHP</title>
   </head>

   <body>
      <?php echo "Set Cookies"?>
   </body>

</html>
```

Accessing Cookies with PHP

PHP provides many ways to access cookies. Simplest way is to use either $_COOKIE or $HTTP_COOKIE_VARS variables. Following example will access all the cookies set in above example.

```
<html>

   <head>
      <title>Accessing Cookies with PHP</title>
   </head>

   <body>

      <?php
         echo $_COOKIE["name"]. "<br />";

         /* is equivalent to */
         echo $HTTP_COOKIE_VARS["name"]. "<br />";

         echo $_COOKIE["age"] . "<br />";

         /* is equivalent to */
         echo $HTTP_COOKIE_VARS["age"] . "<br />";
      ?>

   </body>
</html>
```

You can use **isset()** function to check if a cookie is set or not.

```
<html>

   <head>
      <title>Accessing Cookies with PHP</title>
   </head>

   <body>

      <?php
         if( isset($_COOKIE["name"]))
            echo "Welcome " . $_COOKIE["name"] . "<br />";

         else
            echo "Sorry... Not recognized" . "<br />";
      ?>

   </body>
</html>
```

Deleting Cookie with PHP

Officially, to delete a cookie you should call setcookie() with the name argument only but this does not always work well, however, and should not be relied on.

It is safest to set the cookie with a date that has already expired −

```
<?php
   setcookie( "name", "", time()- 60, "/","", 0);
   setcookie( "age", "", time()- 60, "/","", 0);
?>
<html>

   <head>
      <title>Deleting Cookies with PHP</title>
   </head>

   <body>
      <?php echo "Deleted Cookies" ?>
   </body>

</html>
```

Variables

Scope can be defined as the range of availability a variable has to the program in which it is declared. PHP variables can be one of four scope types −

- Local variables
- Function parameters
- Global variables
- Static variables.

Global Variables

In contrast to local variables, a global variable can be accessed in any part of the program. However, in order to be modified, a global variable must be explicitly declared to be global in the function in which it is to be modified. This is accomplished, conveniently enough, by placing the keyword **GLOBAL** in front of the variable that should be recognized as global. Placing this keyword in front of an already existing variable tells PHP to use the variable having that name. Consider an example −

Live Demo

```
<?php
   $somevar = 15;

   function addit() {
      GLOBAL $somevar;
      $somevar++;

      print "Somevar is $somevar";
   }

   addit();
?>
```

This will produce the following result −

```
Somevar is 16
```

PHP – Sessions

An alternative way to make data accessible across the various pages of an entire website is to use a PHP Session.

A session creates a file in a temporary directory on the server where registered session variables and their values are stored. This data will be available to all pages on the site during that visit.

The location of the temporary file is determined by a setting in the **php.ini** file called **session.save_path**. Before using any session variable make sure you have setup this path.

When a session is started following things happen −

- PHP first creates a unique identifier for that particular session which is a random string of 32 hexadecimal numbers such as 3c7foj34c3jj973hjkop2fc937e3443.
- A cookie called **PHPSESSID** is automatically sent to the user's computer to store unique session identification string.
- A file is automatically created on the server in the designated temporary directory and bears the name of the unique identifier prefixed by sess_ ie sess_3c7foj34c3jj973hjkop2fc937e3443.

When a PHP script wants to retrieve the value from a session variable, PHP automatically gets the unique session identifier string from the PHPSESSID cookie and then looks in its temporary directory for the file bearing that name and a validation can be done by comparing both values.

A session ends when the user loses the browser or after leaving the site, the server will terminate the session after a predetermined period of time, commonly 30 minutes duration.

Starting a PHP Session

A PHP session is easily started by making a call to the **session_start()** function.This function first checks if a session is already started and if none is started then it starts one. It is recommended to put the call to **session_start()** at the beginning of the page.

Session variables are stored in associative array called **$_SESSION[]**. These variables can be accessed during lifetime of a session.

The following example starts a session then register a variable called **counter** that is incremented each time the page is visited during the session.

Make use of **isset()** function to check if session variable is already set or not.

Put this code in a test.php file and load this file many times to see the result −

Live Demo

```
<?php
   session_start();

   if( isset( $_SESSION['counter'] ) ) {
      $_SESSION['counter'] += 1;
   }else {
      $_SESSION['counter'] = 1;
   }

   $msg = "You have visited this page ".  $_SESSION['counter'];
   $msg .= "in this session.";
?>

<html>

   <head>
      <title>Setting up a PHP session</title>
   </head>

   <body>
      <?php  echo ( $msg ); ?>
   </body>

</html>
```

It will produce the following result −

```
You have visited this page 1in this session.
```

Destroying a PHP Session

A PHP session can be destroyed by **session_destroy()** function. This function does not need any argument and a single call can destroy all the session variables. If you want to destroy a single session variable then you can use **unset()** function to unset a session variable.

Here is the example to unset a single variable −

```
<?php
   unset($_SESSION['counter']);
?>
```

Here is the call which will destroy all the session variables −

```
<?php
   session_destroy();
?>
```

Turning on Auto Session

You don't need to call start_session() function to start a session when a user visits your site if you can set **session.auto_start** variable to 1 in **php.ini** file.

Sessions without cookies

There may be a case when a user does not allow to store cookies on their machine. So there is another method to send session ID to the browser.

Alternatively, you can use the constant SID which is defined if the session started. If the client did not send an appropriate session cookie, it has the form session_name=session_id. Otherwise, it expands to an empty string. Thus, you can embed it unconditionally into URLs.

The following example demonstrates how to register a variable, and how to link correctly to another page using SID.

Live Demo
```
<?php
   session_start();
   
   if (isset($_SESSION['counter'])) {
      $_SESSION['counter'] = 1;
   }else {
      $_SESSION['counter']++;
   }
   
   $msg = "You have visited this page ".  $_SESSION['counter'];
   $msg .= "in this session.";
   
   echo ( $msg );
?>

<p>
   To continue  click following link <br />
   
   <a  href = "nextpage.php?<?php echo htmlspecialchars(SID); ?>">
</p>
```

It will produce the following result −

```
You have visited this page 1in this session.
To continue click following link
```

The **htmlspecialchars()** may be used when printing the SID in order to prevent XSS related attacks.

Error and Exception handling

Error handling is the process of catching errors raised by your program and then taking appropriate action. If you would handle errors properly then it may lead to many unforeseen consequences.

Its very simple in PHP to handle an errors.

Using die() function

While writing your PHP program you should check all possible error condition before going ahead and take appropriate action when required.

Try following example without having **/tmp/test.xt** file and with this file.

```
<?php
   if(!file_exists("/tmp/test.txt")) {
      die("File not found");
   }else {
      $file = fopen("/tmp/test.txt","r");
      print "Opend file sucessfully";
   }
   // Test of the code here.
?>
```

This way you can write an efficient code. Using above technique you can stop your program whenever it errors out and display more meaningful and user friendly message.

Defining Custom Error Handling Function

You can write your own function to handling any error. PHP provides you a framework to define error handling function.

This function must be able to handle a minimum of two parameters (error level and error message) but can accept up to five parameters (optionally: file, line-number, and the error context) −

Syntax

```
error_function(error_level,error_message,
error_file,error_line,error_context);
```

Sr.No	Parameter & Description

1	**error_level** Required - Specifies the error report level for the user-defined error. Must be a value number.
2	**error_message** Required - Specifies the error message for the user-defined error
3	**error_file** Optional - Specifies the file name in which the error occurred
4	**error_line** Optional - Specifies the line number in which the error occurred
5	**error_context** Optional - Specifies an array containing every variable and their values in use when the error occurred

Possible Error levels

These error report levels are the different types of error the user-defined error handler can be used for. These values cab used in combination using **|** operator

Sr.No	Constant & Description	Value
1	**.E_ERROR** Fatal run-time errors. Execution of the script is halted	1
2	**E_WARNING** Non-fatal run-time errors. Execution of the script is not halted	2
3	**E_PARSE** Compile-time parse errors. Parse errors should only be generated by the parser.	4
4	**E_NOTICE** Run-time notices. The script found something that might be an error, but could also happen when running a script normally	8
5	**E_CORE_ERROR** Fatal errors that occur during PHP's initial start-up.	16
6	**E_CORE_WARNING** Non-fatal run-time errors. This occurs during PHP's initial start-up.	32

7	**E_USER_ERROR** Fatal user-generated error. This is like an E_ERROR set by the programmer using the PHP function trigger_error()	256
8	**E_USER_WARNING** Non-fatal user-generated warning. This is like an E_WARNING set by the programmer using the PHP function trigger_error()	512
9	**E_USER_NOTICE** User-generated notice. This is like an E_NOTICE set by the programmer using the PHP function trigger_error()	1024
10	**E_STRICT** Run-time notices. Enable to have PHP suggest changes to your code which will ensure the best interoperability and forward compatibility of your code.	2048
11	**E_RECOVERABLE_ERROR** Catchable fatal error. This is like an E_ERROR but can be caught by a user defined handle (see also set_error_handler())	4096
12	**E_ALL** All errors and warnings, except level E_STRICT (E_STRICT will be part of E_ALL as of PHP 6.0)	8191

All the above error level can be set using following PHP built-in library function where level cab be any of the value defined in above table.

```
int error_reporting ( [int $level] )
```

Following is the way you can create one error handling function −

```
<?php
   function handleError($errno, $errstr,$error_file,$error_line) {
      echo "<b>Error:</b> [$errno] $errstr - $error_file:$error_line";
      echo "<br />";
      echo "Terminating PHP Script";

      die();
   }
?>
```

Once you define your custom error handler you need to set it using PHP built-in library **set_error_handler** function. Now lets examine our example by calling a function which does not exist.

```
<?php
```

```
   error_reporting( E_ERROR );

   function handleError($errno, $errstr,$error_file,$error_line) {
      echo "<b>Error:</b> [$errno] $errstr - $error_file:$error_line";
      echo "<br />";
      echo "Terminating PHP Script";

      die();
   }

   //set error handler
   set_error_handler("handleError");

   //trigger error
   myFunction();
?>
```

Exceptions Handling

PHP 5 has an exception model similar to that of other programming languages. Exceptions are important and provides a better control over error handling.

Lets explain there new keyword related to exceptions.

- **Try** − A function using an exception should be in a "try" block. If the exception does not trigger, the code will continue as normal. However if the exception triggers, an exception is "thrown".
- **Throw** − This is how you trigger an exception. Each "throw" must have at least one "catch".
- **Catch** − A "catch" block retrieves an exception and creates an object containing the exception information.

When an exception is thrown, code following the statement will not be executed, and PHP will attempt to find the first matching catch block. If an exception is not caught, a PHP Fatal Error will be issued with an "Uncaught Exception ...

- An exception can be thrown, and caught ("catched") within PHP. Code may be surrounded in a try block.
- Each try must have at least one corresponding catch block. Multiple catch blocks can be used to catch different classes of exceptions.
- Exceptions can be thrown (or re-thrown) within a catch block.

Example

Following is the piece of code, copy and paste this code into a file and verify the result.

```
<?php
   try {
      $error = 'Always throw this error';
      throw new Exception($error);

      // Code following an exception is not executed.
      echo 'Never executed';
   }catch (Exception $e) {
      echo 'Caught exception: ',  $e->getMessage(), "\n";
   }

   // Continue execution
   echo 'Hello World';
?>
```

In the above example $e->getMessage function is used to get error message. There are following functions which can be used from **Exception** class.

- **getMessage()** − message of exception
- **getCode()** − code of exception
- **getFile()** − source filename
- **getLine()** − source line
- **getTrace()** − n array of the backtrace()
- **getTraceAsString()** − formated string of trace

Creating Custom Exception Handler

You can define your own custom exception handler. Use following function to set a user-defined exception handler function.

```
string set_exception_handler ( callback $exception_handler )
```

Here **exception_handler** is the name of the function to be called when an uncaught exception occurs. This function must be defined before calling set_exception_handler().

Example

```
<?php
   function exception_handler($exception) {
      echo "Uncaught exception: " , $exception->getMessage(), "\n";
   }

   set_exception_handler('exception_handler');
   throw new Exception('Uncaught Exception');

   echo "Not Executed\n";
?>
```

DEBUGGING

Programs rarely work correctly the first time. Many things can go wrong in your program that cause the PHP interpreter to generate an error message. You have a choice about where those error messages go. The messages can be sent along with other program output to the web browser. They can also be included in the web server error log.

To make error messages display in the browser, set the **display_errors** configuration directive to **On**. To send errors to the web server error log, set **log_errors** to On. You can set them both to On if you want error messages in both places.

PHP defines some constants you can use to set the value of **error_reporting** such that only errors of certain types get reported: E_ALL (for all errors except strict notices), E_PARSE (parse errors), E_ERROR (fatal errors), E_WARNING (warnings), E_NOTICE (notices), and E_STRICT (strict notices).

While writing your PHP program, it is a good idea to use PHP-aware editors like **BBEdit** or **Emacs**. One of the special special features of these editors is syntax highlighting. It changes the color of different parts of your program based on what those parts are. For example, strings are pink, keywords such as if and while are blue, comments are grey, and variables are black.

Another feature is quote and bracket matching, which helps to make sure that your quotes and brackets are balanced. When you type a closing delimiter such as }, the editor highlights the opening { that it matches.

There are following points which need to be verified while debugging your program.

- **Missing Semicolons** − Every PHP statement ends with a semicolon (;). PHP doesn't stop reading a statement until it reaches a semicolon. If you leave out the semicolon at the end of a line, PHP continues reading the statement on the following line.
- **Not Enough Equal Signs** − When you ask whether two values are equal in a comparison statement, you need two equal signs (==). Using one equal sign is a common mistake.
- **Misspelled Variable Names** − If you misspelled a variable then PHP understands it as a new variable. Remember: To PHP, $test is not the same variable as $Test.
- **Missing Dollar Signs** − A missing dollar sign in a variable name is really hard to see, but at least it usually results in an error message so that you know where to look for the problem.
- **Troubling Quotes** − You can have too many, too few, or the wrong kind of quotes. So check for a balanced number of quotes.
- **Missing Parentheses and curly brackets** − They should always be in pairs.
- **Array Index** − All the arrays should start from zero instead of 1.

Moreover, handle all the errors properly and direct all trace messages into system log file so that if any problem happens then it will be logged into system log file and you will be able to debug that problem.