# Chapter Five

# 5. PHP File Inclusion, Cookies and Sessions, Upload and Download

# 5.1. File inclusions

You can include the content of a PHP file into another PHP file before the server executes it. There are two PHP functions which can be used to include one PHP file into another PHP file.

* **include()** Function
* **require()** Function

This is a strong point of PHP which helps in creating functions, headers, footers, or elements that can be reused on multiple pages. This will help developers to make it easy to change the layout of complete website with minimal effort. If there is any change required then instead of changing thousands of files just change included file.

**include () Function**

The **include()** function takes all the text in a specified file and copies it into the file that uses the include function. If there is any problem in loading a file then the include () function generates a warning but the script will continue execution.

Assume you want to create a common menu for your website. Then create a file menu.php with the following content.

*|<a href="index.php">Home</a>||*

*<a href="feedback.php">Feedback</a>||*

*<a href="login.php">Login</a>|*

Now create as many pages as you like and include this file to create header. For example now your test.php file can have following content.

*<html>*

*<body>*

*<?php include("menu.php"); ?>*

*<p>This is an example to show how to include PHP file!</p>*

*</body>*

*</html>*

This will produce the following result:

*|Home||Feedback||Login|*

*This is an example to show how to include PHP file.*

You can include menu.php file in as many as files you like!

**require() Function**

The require() function takes all the text in a specified file and copies it into the file that uses the require function. If there is any problem in loading a file then the require() function generates a fatal error and halt the execution of the script.

So there is no difference in **require()** and **include()** except they handle error conditions. It is recommended to use the **require()** function instead of **include(),** because scripts should not continue executing if files are missing or misnamed.

You can try using above example with **require()** function and it will generate same result. But if you will try following two examples where file does not exist then you will get different results.

*<html> <body>*

*<?php include("xxmenu.php"); ?>*

*<p>This is an example to show how to include wrong PHP file!</p>*

*</body> </html>*

This will produce the following result:

*This is an example to show how to include wrong PHP file!*

Now lets try same example with **require**() function.

*<html> <body>*

*<?php require("xxmenu.php"); ?>*

*<p>This is an example to show how to include wrong PHP file!</p>*

*</body> </html>*

This time file execution halts and nothing is displayed.

**NOTE**: You may get plain warning messages or fatal error messages or nothing at all. This depends on your PHP Server configuration.

**PHP require\_once()**

require\_once() statement can be used to include a php file in another one, when you may need to include the called file more than once. If it is found that the file has already been included, calling script is going to ignore further inclusions.

If a.php is a php script calling b.php with require\_once() statement, and does not find b.php, a.php stops execution causing a fatal error.

Syntax

*require\_once('name of the calling file with path');*

Example :

*<?php*

*echo "today is:".date("Y-m-d");*

*?>*

The above file is x.php

The above file x.php, is included twice with require\_once() statement in the following file y.php. But from the output you will get that the second instance of inclusion is ignored, since require\_once() statement ignores all the similar inclusions after the first one.

*<?php*

*require\_once('x.php');*

*require\_once('x.php');*

*?>*

Output: *today is:2016-05-27*

If a calling script does not find a called script with require\_once statement, it halts the execution of the calling script.

**PHP include\_once()**

The include\_once() statement can be used to include a php file in another one, when you may **NOT** need to include the called file more than once. If it is found that the file has already been included, calling script is going to ignore further inclusions.

If a.php is a php script calling b.php with include\_once() statement, and does not find b.php, a.php executes with a warning, excluding the part of the code written within b.php.

Syntax

*include\_once('name of the called file with path');*

Example :

*<?php*

*echo "today is:".date("Y-m-d");*

*?>*

The above file is x.php

The above file x.php, is included twice with include\_once() statement in the following file y.php. But from the output you will get that the second instance of inclusion is ignored,  since include\_once() statement ignores all the the similar inclusions after the first one.

*<?php*

*include\_once('x.php');*

*include\_once('x.php');*

*?>*

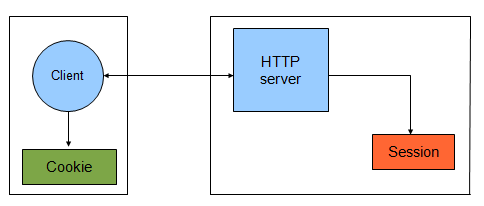
Output: *today is:2016-05-27*

If a calling script does not find a called script with include\_once statement, it halts the execution of the calling script.

﻿**Cookies**

A client can visit and load a website several times. If so, there should be certain mechanism to remember the previous instances of it being requested by a client. This leads to persistency of files or data.

As discussed in IP-I, http is a stateless protocol. It remembers nothing about previous transfers.



**A cookie** is a packet of information sent from the server to client, and then sent back to the server each time. Or 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.

**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 up to 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.

The following example will create two cookies **name** and **age**. These cookies will expire after an hour.

*<?php*

*setcookie("name", "abebe", 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. The 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>*

## 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, the following actions take place:

* 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 and then registers 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:

*<?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>*

**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();*

*?>*

* 1. **Upload Files**

Web applications allow visitors to upload files to and from their local computer. The files that are uploaded and downloaded may be simple text files or more complex file types, such as images, documents, or spreadsheets.

Files are uploaded through an HTML form using the “**post**” method and **enctype** attribute with value of “**multipart/form-data**,” which instructs the browser to post multiple sections – one for regular form data and one for the file contents. The file input field creates a browser button for the user to navigate to the appropriate file to upload

*<form method=”post” action=” ” enctype= multipart/form-data >*

*<input type="file" name="picture\_file" />*

*</form>*

The MAX\_FILE\_SIZE (uppercase) attribute of a hidden form field specifies the maximum number of bytes allowed in the uploaded file and it must appear before the file input field.

When the form is posted, information for the uploaded file is stored in the $\_FILES auto global array.

The $\_FILES[] array contains five elements:

* 1. **// Contains the error code associated with the file** $\_FILES['picture\_file']['error']
  2. **// Contains the temporary location of the file**

$\_FILES['picture\_file']['tmp\_name'] contents

* 1. **// Contains the name of the original file**$\_FILES['picture\_file']['name']
  2. **// Contains the size of the uploaded file in bytes**

$\_FILES['picture\_file']['size']

* 1. **// Contains the type of the file**

$\_FILES['picture\_file']['type']

Example:-The following HTML code below creates an upload form. This form is having method attribute set to **post** and **enctype** attribute is set to **multipart/form-data**

*<html><body><h3>File Upload:</h3>*

*Select a file to upload: <br />*

*<form action="<?php $\_PHP\_SELF ?>" method="post" enctype="multipart/form-data">*

*<input type="file" name="photo" required=””/><br />*

*<input type="submit" value="Upload File" />*

*</form></body></html>*

*<?php*

*$filename=$\_FILES['photo']['name'];*

*$filetmpname=$\_FILES['photo']['tmp\_name'];*

*$target="uploadedfiles/".$\_FILES['photo']['name'];*

*if(!file\_exists("uploadedfiles"))*

*mkdir("uploadedfiles");*

*if( copy($\_FILES['photo']['tmp\_name'], $target) or die( "Could not copy file!"))*

*echo "file uploded successfully";*

*else*

*die("unable to upload!");*

*?>*

*<h2>Uploaded File Info:</h2>*

*<ul><li>Sent file: <?php echo $\_FILES['photo']['name']; ?>*

*<li>File size: <?php echo $\_FILES['photo']['size']; ?> bytes*

*<li>File type: <?php echo $\_FILES['photo']['type']; ?>*

*</ul><a href=<?php echo $target; ?>>Click here to open</a></body></html>*

Example2: Inserting the above file to a database: First create a table with two columns such as id(int auto increment) and file\_path (varchar(50)).

*<?php*

*$server="localhost";*

*$dbuser="root";*

*$dbpass="";*

*$dbname="bookstore";*

*$connection = mysql\_connect($server, $dbuser, $dbpass) or die("Couldn't make connection.");*

*$db = mysql\_select\_db($dbname, $connection) or die("Couldn't select database");*

*?><html><body><h3>File Upload:</h3>*

*Select a file to upload: <br />*

*<form action="<?php $\_PHP\_SELF ?>" method="post" enctype="multipart/form-data">*

*<input type="file" name="photo" required=""/><br />*

*<input type="submit" value="Upload File" name="upload" />*

*</form></body></html>*

*<?php*

*if(isset($\_POST['upload']))*

*{*

*$filename=$\_FILES['photo']['name'];*

*$filetmpname=$\_FILES['photo']['tmp\_name'];*

*$target="uploadedfiles/".$\_FILES['photo']['name'];*

*if(!file\_exists("uploadedfiles"))*

*mkdir("uploadedfiles");*

*//uploading a file into uploadedfiles folder*

*if(copy($\_FILES['photo']['tmp\_name'], $target) or die( "Could not copy file!"))*

*{*

*//inserting a file to a file table in bookstore database*

*$result = mysql\_query("INSERT INTO file(file\_path) VALUE ('$target')");*

*if(!$result)*

*{*

*echo "File uploaded successfully";*

*}*

*//retrieving records from file table*

*$result = mysql\_query("select \* from file");*

*echo "<table border=1><tr><th>Id</th><th>File</th></tr>";*

*while($row=mysql\_fetch\_array($result))*

*{*

*echo "<tr><th>".$row['id']."</th><th><a href='".$row['file\_path']."'>".$row['file\_path']."</a></th></tr>";*

*}*

*echo "</table>";*

*}else*

*echo "Unable to upload files";*

*}*

*?>*

*</body></html>*