Basic File Handling in PHP

Introduction

Files are stored in directories on a hard drive, and because they retain their data after the computer is shut down, they are a persistent storage mechanism, instead of temporary storage such as RAM.

As a server-side programming language, PHP allows you to work with files and directories stored on the Web server.

This is very useful, because it means your PHP scripts can store information outside the scripts themselves.

This chapter is all about the PHP functions for working with the file system.

What We Will Learn

In this lecture you will learn:

1. How to include the code from one script file inside another script file
2. How to open files
3. How to close files
4. How to read data from files
5. How to write data to files

Files

Everything on your hard drive is stored as a file of one kind or another.

A file is nothing more than an **ordered sequence of bytes** stored on a hard disk or other storage media.

A **directory** is a **special type of file** that holds the names of the files and directories inside the folder (sometimes denoted as subdirectories or subfolders) and pointers to their storage areas on the media.

Including Files

`include()` allows you to include the code contained in another library or script file inside a php file, just as if the code had been copied and pasted into the php file.

Here's a simple example. Say you have created a simple function called `animalSpeak()` that returns the sound made by either a cat, a dog, or a mouse:

```
function animalSpeak( $animal ) {
 $sounds = array ( “cat” = > “meow”, “dog” = > “woof”, “mouse” = >“squeak” );

 return $sounds[$animal];
}

echo animalSpeak( “mouse” ) . “ < br / > ”; // Displays “squeak”
echo animalSpeak( “cat” ) . “ < br / > ”; // Displays “meow”
```

Including Files

Here, the function and the code that uses it are both contained within the **same script file**.

This is all very well, but what if you wanted to call `animalSpeak()` from within a different script or application?

You'd have to copy and paste the code that defines the `animalSpeak()` function from one script to the other — clearly not the best approach, because you'd end up with duplicate versions of `animalSpeak()` lying about.

This is where the **`include()`** function comes in. By storing the `animalSpeak()` function definition in a **separate file**, you can use **`include()`** to include the function from within as many other script files as you like.

Including Files

For example, you might store it in a file called `animal_functions.php`:

```
<?php

function animalSpeak( $animal ) {
  $sounds = array ( “cat” = > “meow”, “dog” = > “woof”, “mouse” = >“squeak” );

  return $sounds[$animal];
 }

?>
```

Notice that you need to place the **< ?php** and **? >** tags around any PHP code in the included file. If you don't do this, the included file will be treated as HTML markup, and sent straight to the browser.

Including Files

You might then use the function from within a `mouse.php` script file in the same folder, as follows:

```
<?php
 include( “animal_functions.php” );
 echo animalSpeak( “mouse” ) . “ < br / > ”; // Displays “squeak”;
?>
```

You can then use the same function inside a `cat.php` script file:

```
<?php
 include( “animal_functions.php” );
 echo animalSpeak( “cat” ) . “ < br / > ”; // Displays “meow”;
?>
```

Including Files

include() is very useful when building large websites

It enables you to write code once and then reuse it wherever you want

Imagine a website with multiple pages, where all pages have similar header and footer sections

you can put header and footer code in separate php files and then include them in every page you want

We will see an example for this later in this lecture.

Getting information about files

PHP provides some functions that enable you to access useful file information.

For example, you can use **`file_exists()`** to discover whether a file exists before attempting to open it:

```
file_exists( “myfile.txt” )
```

`file_exists()` returns true if the file at the specified path exists, or false otherwise.

Getting information about files

In a similar fashion, you can use the **`filesize()`** function to determine the size of a file on the hard disk.

Just as with `file_exists()`, this function takes a filename as an argument:

```
filesize( “myfile.txt” )
```

This returns the size of the specified file in bytes, or false upon error.

Opening a File

Usually, to work with a file from within your PHP script, you first need to open the file.

When you open a file, you create a file handle. A file handle is a pointer associated with the open file that you can then use to access the file's contents.

When you're finished with the file, you close it, which removes the file handle from memory.

Some PHP functions let you work directly with a file without needing to open or close it.

Opening a File with fopen()

The **fopen()** function opens a file and returns a file handle associated with the file.

The first argument passed to `fopen()` specifies the name of the file you want to open, and the second argument specifies the mode , or how the file is to be used.

For example:

```
$handle = fopen( “data.txt”, “r” );
```

The first argument can be just a filename **(“ data.txt “)**, in which case PHP will look for the file in the current directory, or it can be a **relative (” ../data.txt “)** or **absolute (“ /myfiles/data.txt “)** path to a file.

Remember Paths?

Usually, the current directory is the same directory as the script.

Within a relative path, two dots (..) refer to the immediate parent directory.

For example, **data.txt** points to a file called data.txt in the **current directory**, and **../data.txt** points to a file called data.txt in the **directory above the current directory**.

../../../data.txt backs **up the directory tree three levels** before looking for the data.txt file.

Opening a File with fopen()

The second argument to `fopen()` tells PHP how you're going to use the file. It can take one of the following string values:

Value	Description
r	Open the file for reading only. The file pointer is placed at the beginning of the file.
r+	Open the file for reading and writing. The file pointer is placed at the beginning of the file.
w	Open the file for writing only. Any existing content will be lost. If the file does not exist, PHP attempts to create it.
a	Open the file for appending only. Data is written to the end of an existing file. If the file does not exist, PHP attempts to create it.
a+	Open the file for reading and appending. Data is written to the end of an existing file. If the file does not exist, PHP attempts to create it.

Opening a File with fopen()

If there was a problem opening the file, fopen() returns false rather than a file handle resource.

Operations on files and directories are prone to errors, so you should always allow for things to go wrong when using them. It's good practice to use some form of error checking procedure so that if an error occurs (perhaps you don't have necessary privileges to access the file, or the file doesn't exist), your script will handle the error gracefully. For example:

```
if ( !( $handle = fopen( “data.txt”, “r” ) ) )

   die( “Cannot open the file” );
```

Closing a File with fclose()

Once you've finished working with a file, it needs to be closed.

You can do this using **`fclose()`**, passing in the open file's handle as a single argument, like this:

```
fclose( $handle );
```

Although PHP should close all open files automatically when your script terminates, it's good practice to close files from within your script as soon as you're finished with them because it frees them up quicker for use by other processes and scripts — or even by other requests to the same script.

Reading from files using fread()

Now that you know how to open and close files, it's time to take a look at reading and writing data in a file.

The **`fread()`** function can be used to read a string of characters from a file. It takes two arguments: **a file handle** and the **number of characters to read**.

The function reads the specified number of characters (or less if the end of the file is reached) and returns them as a string. For example:

```
$handle = fopen( “data.txt”, “r” );
$data = fread( $handle, 10 );
```

This code reads the first ten characters from data.txt and assigns them to `$data` as a string.

Reading from files using fread()

After `fread()` has finished, the **file pointer**, which holds the **current position** in the file, moves forward in the file by the **number of characters read**.

So after the previous example code runs, the file pointer moves forward to ten characters after the start of the file.

If you repeat the same call to `fread()`, you'll get the **next** ten characters in the file. If there are less than ten characters left to read in the file, `fread()` simply reads and returns as many as there are.

Reading from files using fgetc()

If you want to read only one character at a time you can use the `fgetc()` function.

`fgetc()` takes a single argument — a file handle — and **returns just one character** from the file it points to; it returns **false when it reaches the end of the file**:

```
$one_char = fgetc( $handle );
```

However, `fgetc()` is slow when working with large files. It is faster to read a bunch of characters at once using `fread()`, or one of the other file - reading functions discussed later.

Testing for the End of a File

The **`feof()`** function serves a single, simple purpose:

It returns true when the file pointer has reached the end of the file (or if an error occurs) and returns false otherwise.

It takes just **one argument** — the file handle to test.

Notice that `feof()` only returns true once the script has tried to read one or more characters past the last character in the file

Testing for the End of a File

```
// hello_world.txt contains the characters “Hello, world!”

$handle = fopen( “hello_world.txt”, “r” );
$hello = fread( $handle, 13 );

echo $hello . “ < br / > ”; // Displays “Hello, world!”
echo feof( $handle ) . “ < br / > ”; // Displays “” (false)

$five_more_chars = fread( $handle, 5 );

echo $five_more_chars . “ < br / > ”; // Displays “” (or possibly a newline)
echo feof( $handle ) . “ < br / > ”; // Displays “1” (true)

fclose( $handle );
```

Testing for the End of a File

`feof()` is useful with `fread()` or `fgetc()` in a while loop when you don't know how long the file is:

```
// hello_world.txt contains the characters “Hello, world!”

$handle = fopen( “hello_world.txt”, “r” );
$text = “”;

while ( !feof( $handle ) ) {
 $text .= fread( $handle, 3 ); // Read 3 chars at a time
}

echo $text . “ < br / > ”; // Displays “Hello, world!”
fclose( $handle );
```

Reading One Line at a Time

Often it's useful to read text from a file one line at a time. A line is a nice manageable chunk of text to process or display.

To read a line of text from an open file, call the **`fgets()`** function, passing in the file handle. The function reads from the **current file pointer** to the **end of the current line**, and **returns** the read characters as a **string** (or false if there was a problem, such as the end of the file being reached).

Note that any end-of-line character (or characters) at the end of the line is also included in the string.

Reading One Line at a Time

The following example uses `fgets()` to read and display a three-line text file, one line at a time. The while loop exits when `fgets()` returns false (which means it's reached the end of the file):

```
/* milton.txt contains:
 The mind is its own place, and in it self
 Can make a Heav'n of Hell, a Hell of Heav'n.
 What matter where, if I be still the same,
*/

$handle = fopen( "milton.txt", "r" );
$lineNumber = 1;

while ( $line = fgets( $handle ) ) {
 echo $lineNumber++ . ": $line < br / > ";
}
fclose( $handle );
```

Output

```
1: The mind is its own place, and in it self
2: Can make a Heav'n of Hell, a Hell of Heav'n.
3: What matter where, if I be still the same,
```

Writing to a file using fwrite()

You can use the **`fwrite()`** function to write data to a file. It requires **two arguments**: **a file handle** and **a string** to write to the file.

The function writes the contents of the string to the file, **returning the number of characters written** (or false if there's an error). For example:

```
$handle = fopen( “data.txt”, “w” );
fwrite( $handle, “ABCxyz” );
```

Writing to a file using fwrite()

The first line opens the file data.txt for writing, which erases any existing data in the file. (If the file doesn't exist, PHP attempts to create it.)

The second line writes the character string “ ABCxyz ” to the beginning of the file.

As with `fread()`, the file pointer moves to the position after the written string; if you repeat the second line, `fwrite()` appends the same six characters again, so that the file contains the characters “ ABCxyzABCxyz ”.

A Simple Visitor Counter

One very popular use for Web scripts is a visitor hit counter, which is used to show how many times a Web page has been visited and therefore how popular the Web site is.

Hit counters come in different forms, the simplest of which is a text counter.

In a text counter, the current value of the counter is stored in a file and increased each time the page is visited.

Let's create a simple script to do that.

The script

```
<!DOCTYPE html>
 <head>
 <title>Hit counter</title>
 <link rel=”stylesheet” type=”text/css” href=”css/style.css” />
 </head>
 <body>

 <h1>This is my home page!</h1>

<?php

$counterFile = "count.dat";

if ( !file_exists( $counterFile ) ) {
 if ( !( $handle = fopen( $counterFile, "w" ) ) ) {
      die( "Cannot create the counter file." );
 } else {
      fwrite( $handle, 0 );
      fclose( $handle );
 }
}
```

The script

```
if ( !( $handle = fopen( $counterFile, "r" ) ) ) {
 die( "Cannot read the counter file." );
}

$counter = (int) fread( $handle, 20 );
fclose( $handle );

$counter++;

echo "<h1>You're visitor No. $counter.</h1>";

if ( !( $handle = fopen( $counterFile, "w" ) ) ){
 die( "Cannot open the counter file for writing." );
}

fwrite( $handle, $counter );
fclose( $handle );

?>

 </body>
</html>
```

Give it a try

Save this script as `hit_counter.php` in a folder called `counter` on your webserver and give it a try.

Make sure you have the correct permissions that allows your script to create a new file in this folder. For example on Linux you would need to set the permissions for counter folder to 777, for this script to work.

Then open the hit_counter.php file in your browser.

You can access this file here.

How it works?

After displaying a page header, the script stores the filename of the file that will hold the hit count:

```
$counterFile = “count.dat“;
```

How it works?

Next, the script checks to see if the counter file exists. If it doesn't, it is created by opening the file for writing, writing a zero to it (thereby initializing the hit count to zero), then closing it:

```
if ( !file_exists( $counterFile ) ) {
 if ( !( $handle = fopen( $counterFile, “w” ) ) ) {
    die( “Cannot create the counter file.” );
 } else {
    fwrite( $handle, 0 );
    fclose( $handle );
 }
}
```

How it works?

Next the counter file is opened for reading:

```
if ( !( $handle = fopen( $counterFile, “r” ) ) ) {
    die( “Cannot read the counter file.” );
}
```

How it works?

The script now uses the file handle to read the hit counter value from the open file.

As you can see, the script calls `fread()` to read up to 20 bytes from the data file (enough to store a very large integer):

```
$counter = (int) fread( $handle, 20 );
```

Because `fread()` returns a string value, and the counter needs to be an integer value, the return value is cast(converted) into an integer using (int).

How it works?

The call to `fclose()` closes the file referenced by the file handle `$handle`, freeing up the file for reading or writing by other processes:

```
fclose( $handle );
```

After closing the data file, the script increments the counter and tells the visitor how many times the page has been accessed:

```
$counter++;
echo “<h1>You’re visitor No. $counter.</h1>”;
```

How it works?

Next the script writes the new counter value back to the data file. To do this it opens the file in **write mode (w)**, then calls `fwrite()` to write the `$counter` variable's value to the file, followed by `fclose()` to close the open file again:

```
if ( !( $handle = fopen( $counterFile, “w” ) ) ){
 die( “Cannot open the counter file for writing.” );
}

fwrite( $handle, $counter );
fclose( $handle );
```

Multiple pages

Now although this works perfectly for a single page in our website, what if wanted to have the same hit counter on every page of the website?

Should we copy the same code inside hit_counter.php for every new page?

Of course not! The whole purpose of PHP is to write reusable code.

So let's assume we have 2 other pages in our website, contact page and about page.

We want the same counter to appear on all those pages, but the heading on each page is different.

Reuse Code!

There are parts of our hit_counter.php code that are unchanged on all pages.

```
<!DOCTYPE html>
 <head>
 <title>Hit counter</title>
 <link rel=”stylesheet” type=”text/css” href=”css/style.css” />
 </head>
 <body>
 <div class="container>
 <h1>This is my home page!</h1>

<?php

$counterFile = "count.dat";

if ( !file_exists( $counterFile ) ) {
 if ( !( $handle = fopen( $counterFile, "w" ) ) ) {
      die( "Cannot create the counter file." );
 } else {
      fwrite( $handle, 0 );
      fclose( $handle );
 }
}
```

Reuse Code

```
if ( !( $handle = fopen( $counterFile, "r" ) ) ) {
 die( "Cannot read the counter file." );
}

$counter = (int) fread( $handle, 20 );
fclose( $handle );

$counter++;

echo "<h1>You're visitor No. $counter.</h1>";

if ( !( $handle = fopen( $counterFile, "w" ) ) ){
 die( "Cannot open the counter file for writing." );
}

fwrite( $handle, $counter );
fclose( $handle );

?>
  </div>
 </body>
</html>
```

So let's put these commonly used parts in separate files and use include() function to insert those lines in every file that we want.

I will separate header, footer and the hit counter and put them in separate php files, since they are going to be present in all pages.

header.php

I like to separate the header part and put in a new file called header.php

This way, if I want to change any part of the header, I won't have to change multiple files. I'll just change the header.php

header.php

```
<!DOCTYPE html>
 <head>
 <title>Hit counter</title>
 <link rel="stylesheet" type="text/css" href="css/style.css" />
 </head>
 <body>
```

footer.php

Exactly the same applies to the footer part of my page. It is same for all my pages.

footer.php

```
</div>
 </body>
</html>
```

hit_counter2.php

The main functionality that we were after, was the visitor counting. Let's put that into a new php file and use it everywhere.

hit_counter2.php

```
<?php

$counterFile = "count.dat";

if ( !file_exists( $counterFile ) ) {
 if ( !( $handle = fopen( $counterFile, "w" ) ) ) {
      die( "Cannot create the counter file." );
 } else {
      fwrite( $handle, 0 );
      fclose( $handle );
 }
}
.
.
.
?>
```

This file includes all of the code related to controlling and reading the counter, but we won't print the counter value in here. Instead we will do that in each individual page.

Home page

Now that we have all the components of our pages, we can create them one-by-one

Let's start with the `index.php` which is our home page.

`index.php`

```php
<?php

include("hit_counter2.php");
include("header.php");

echo "<h1>This is my home page</h1>";
echo "<h1>You're visitor No. $counter.</h1>";

include("footer.php");
?>
```

The counter value is read from the counter file in hit_counter2.php. here we are just printing that out.

Contact page

We can use the exact same code for out contact page, except that we want a different heading to show on this page.

contact.php

```
<?php

include("hit_counter2.php");
include("header.php");

echo "<h1>This is my contact page</h1>";
echo "<h1>You're visitor No. $counter.</h1>";

include("footer.php");
?>
```

About page

Again the same ...

about.php

```
<?php

include("hit_counter2.php");
include("header.php");

echo "<h1>This is my about page</h1>";
echo "<h1>You're visitor No. $counter.</h1>";

include("footer.php");
?>
```

Well done!

This is only a very simple example.

We can do a lot more with php file handling and include functionality.

you can download the full source code for this example here.