File And Directory Manipulation In PHP (part 1)

By icarus

2003−08−07

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/PHP/File_Manipulation/

Rank And File

One of the most basic things you can do with any programming language is create and manipulate data structures. These structures may be transient − in−memory variables like arrays or objects − or more permanent − disk−based entities like files or database tables; however, the ability to create, modify and erase them in a convenient and consistent manner is of paramount importance to any programmer.

Like all widely−used programming languages, PHP has the very useful ability to read data from, and write data to, files on the system. But the language doesn't just stop there − PHP comes with a full−featured file−and−directory−manipulation API that allows you (among other things) to view and modify file attributes, read and list directory contents, alter file permissions, retrieve file contents into a variety of native data structures, and search for files based on specific patterns. This file manipulation API is both powerful and flexible − two characteristics that will endear it to any developer who's ever had to work with file manipulation commands.

Over the course of this tutorial, I'm going to introduce you to this filesystem API, demonstrating some of the common (and not−so−common) tasks you can perform with it. Note that this is not intended to be an exhaustive reference to all the file functions in PHP − there's a very good manual available for that − but rather an introduction to the functions you'll find yourself using most frequently, together with examples of how they can be used. If you've been using PHP for a while, some of this material will already be familiar to you; new users, on the other hand, should find most of it interesting.

Let's get started!

Handle With Care

I'll begin with something simple − opening a file and reading its contents. Let's assume that we have a text file called "mindspace.txt", containing the following random thoughts:

```
We're running out of space on planet Earth.

Scientists are attempting to colonize Mars.
```

```
I have a huge amount of empty real estate in my mind.

Imagine if I could rent it to the citizens of Earth for a nominal monthly fee.


Would I be rich? Or just crazy?
```

Now, in order to read this data into a PHP script, I need to open this file and assign it a file handle − I can then use this file handle to interact with the file and extract its contents into a PHP variable. Take a look:

```
<?php

// set file to read
$filename = "mindspace.txt";

// open file
$fh = fopen($filename, "r") or die("Could not open file!");

// read file contents
$data = fread($fh, filesize($filename)) or die("Could not read file!");

// close file
fclose($fh);

// print file contents
echo $data . " −− ALL DONE −−";

?>
```

And when you run this script, PHP should return the contents of the file "mindspace.txt", with a message at the end.

A quick explanation: in order to read data from an external file, PHP requires you to define a file handle for it with the fopen() function. I've done this in the very first line of the script above.

```
<?php

// set file to read
$filename = "mindspace.txt";

// open file
$fh = fopen($filename, "r") or die("Could not open file!");

?>
```

You can specify a full path to the file as well:

```
<?php
```

```
// set file to read
$filename = "/home/me/mindspace.txt";

// open file
$fh = fopen($filename, "r") or die("Could not open file!");

?>
```

Notice the second argument to fopen() − it's a string indicating the "mode" in which the file is to be opened. A number of modes are available − read, write, append and so on − and I'll discuss them in detail a little further along.

If the fopen() function is successful, it returns a file handle (stored in $fh) which can be used for further interaction with the file. This file handle is provided to the fread() function, which uses it to read the contents of the file.

```
<?php

// read file contents
$data = fread($fh, filesize($filename)) or die("Could not read file!");

?>
```

The second argument to fread() is the number of bytes to be read. In this case, I've used the filesize() function to obtain the size of the file in bytes and provide that number to the fread() function; you can set an arbitrary value here if you like, but be warned that reading will automatically stop once the end of the file is reached.

Once the file contents have been read into a variable with fread(), the fclose() function is used to close the file and destroy the file handle created by fopen(). This is not strictly necessary, but it is a good habit to develop as it avoids loose ends cluttering up your script.

```
<?php

// close file
fclose($fh);

?>
```

Different Strokes

An alternative way to accomplish the same thing is to use the fgets() function in combination with the feof() function, as demonstrated below:

```
<?php

// set file to read
$filename = "mindspace.txt";

// open file
$fh = fopen ($filename, "r") or die("Could not open file");

// read file
while (!feof($fh))
{
$data = fgets($fh);
echo $data;
}

// close file
fclose ($fh);

echo "−− ALL DONE −−";

?>
```

The fgets() function works slightly differently from the fread() function, in that it reads a file line by line until the end of the file is reached (the feof() function is used to test for the end−of−file marker).

There are also a couple of other methods of reading data from a file − the very cool file() function, which reads the entire file into an array with one fell swoop, assigning each line as an element of the array. The following example demonstrates:

```
<?php

// set file to read
$filename = "mindspace.txt";

// read file into array
$data = file($filename) or die("Could not read file!");

// loop through array and print each line
foreach ($data as $line)
{
```

```
echo $line;
}

// print file contents
echo "−− ALL DONE −−";

?>
```

As you can see, this example assigns the contents of the file "mindspace.txt" to the array variable $data via the file() function. Each element of the array variable now corresponds to a single line from the file. Once this has been done, it's a simple matter to run through the array and display its contents with the "foreach" loop.

Don't want the data in an array? Try the new file_get_contents() function, which reads the entire file into a string,

```
<?php

// set file to read
$filename = "mindspace.txt";

// read file into string
$data = file_get_contents($filename) or die("Could not read file!");

// print file contents
echo $data . "−− ALL DONE −−";

?>
```

or use the readfile() function, which reads a file and dumps it directly to the standard output device (in the case of PHP, usually the Web browser):

```
<?php

// set file to read
$filename = "mindspace.txt";

// read and output file
readfile($filename) or die("Could not read file!");

echo "−− ALL DONE −−";

?>
```

Weapon Of Choice

If you've used C before, you're probably already familiar with the "include" directive that appears near the beginning of every C program. Since we're talking about reading files, it's appropriate to bring in PHP's

equivalent − the include() and require() functions, which come in handy when you need to read an external file into your PHP script.

Consider the following simple example, which illustrates:

```
<html>
<head>
<title>Weapons Worth Dying For</title>
<style>
h1,h3,li { font-family:Verdana; }
</style>
</head>
<?php
```

// this time, Bond's going to need the cigarette−lighter gun... require("./gun.php4");

// the new car−copter...
include("./car.php4");

// and of course, the gold watch with the hidden GPS locator require("./watch.php4");

?>

<body>
<h3>So, James, here's your check list for the mission.</h3>
<ol type="a">
The gun: <?php echo "$gun"; ?>
The car: <?php echo "$car"; ?>
The watch: <?php echo "$watch"; ?>

<h3>Oh yes...and remember never to let them see you cry. Good luck, 007!</h3> </body> </html>

Now, if you try to access this page as it, you'll get a bunch of error messages warning you about missing files. So you need to create the files "gun.php4", "car.php4" and "watch.php4":

```
<?php
// gun.php4
$gun = "AK−47";
?>
```

<?php
// car.php4
$car = "BMW G8";

?>

<?php
// watch.php4
$watch = "Rolex SAW−007";
?>

And this time, when you access the primary page, PHP should automatically include the specified files, read the variables $gun, $watch and $car from them, and display them on the page.

Files named in the include() and require() functions are searched for in a set of default locations. These locations are specified via PHP's "include_path" configuration directive, which may be set globally in the main "php.ini" configuration file, or on a per−script basis using the
ini_set() function call.

A quick note on the difference between the include() and require() functions − the require() function returns a fatal error if the named file cannot be found and halts script processing, while the include() function returns a warning but allows script processing to continue.

An important point to be noted is that when a file is require()−d or include()−d, the PHP parser leaves "PHP mode" and goes back to regular "HTML mode". Therefore, all PHP code within the included external files needs to be enclosed within regular PHP <?...?> tags.

A very useful and practical application of the include() function is to use it to include a standard footer or copyright notice across all the pages of your Web site, like this:

```
<html>

<head>

<title></title>

</head>
```

<body>

...your HTML page...

<?php include("footer.html"); ?>

</body>
</html>

where "footer.html" contains

```
<font size=−1 face=Arial>This material copyright Melonfire, 2003. All
rights reserved.</font>
```

Now, this footer will appear on each and every page that contains the
include() statement above − and, if you need to change the message, you only need to edit the single file named "footer.html"!

Note also that PHP also offers the require_once() and include_once() functions, which ensure that a file which has already been read is not read again. This can come in handy if you have a situation in which you want to eliminate multiple reads of the same include file, either for performance reasons or to avoid corruption of the variable space.

A Little Brainwashing

Obviously, reading a file is no great shakes − but how about writing to a file? Well, this next script does just that:

```
<?php

// set file to write
$filename = "matrix.txt";

// open file
$fh = fopen($filename, "w") or die("Could not open file!");

// write to file
fwrite($fh, "Welcome to The Matrix, Neo") or die("Could not write to file");

// close file
fclose($fh);

?>
```

Now, once you run this script, it should create a file named "matrix.txt", which contains the text above.

As you can see, in order to open a file for writing, you simply need to use the same fopen() function, but with a different mode ("w" for write).

```
<?php
```

```
// open file
$fh = fopen($filename, "w") or die("Could not open file!");

?>
```

A number of different modes are available for the fopen() function − you've already seen two, here are a few more:

MODE WHAT IT DOES
−−
r Opens a file in read mode

w Opens a file in write mode; existing file contents are truncated

a Opens a file in append mode; existing file contents are preserved

You can add a "+" to any of the modes above to open the file for simultaneous read and write. In case the named file does not exist in any of the write modes, a new file with the specified name is created.

Once the file has been opened, you can write text to it simply by specifying the data to be written as an argument to the fwrite() function.

```
<?php

// write to file
fwrite($fh, "Welcome to The Matrix, Neo") or die("Could not write to file");

?>
```

Close the file, and you're done!

You can also write to a file with the new file_put_contents() function, which accepts a string as input and writes it to the named file with minimal fuss or muss.

```
<?php

// set file to write
$filename = "/tmp/matrix.txt";

// write to file
file_put_contents($filename, "Welcome to The Matrix, Neo") or die("Could not write to file");

?>
```

Weather Balloon

Let's consider some more realistic examples. This next piece of code connects to a MySQL database, retrieves a result set and either displays it in a browser or writes it to a file based on a configuration variable.

```
<?php

// user−defined output handler
function myOutputHandler($buf)
{
global $output;

// either dump the buffer to a file
if ($output != "www")
{
$fp = fopen ("weather.html", "w");
fwrite($fp, $buf);
fclose($fp);
}
// ... or return it for printing to the browser
else
{
return $buf;
}
}

// start buffering the output
// specify the callback function
ob_start("myOutputHandler");

// output format − either "www" or "file"
$output = "www";

// send some output
?>

<html>
<head><basefont face="Arial"></head>
<body>

<?
// open connection to database
$connection = mysql_connect("localhost", "joe", "nfg84m") or die ("Unable to connect!");
mysql_select_db("weather") or die ("Unable to select database!");

// get data
$query = "SELECT * FROM weather";
$result = mysql_query($query) or die ("Error in query: $query. " . mysql_error());

// if a result is returned
if (mysql_num_rows($result) > 0)
{
```

```
// iterate through resultset
// print data
while (list($temp, $forecast) = mysql_fetch_row($result))
{
echo "Outside temperature is $temp";
echo "<br>";
echo "Forecast is $forecast";
echo "<p>";
}
}
else
{
echo "No data available";
}

// close database connection
mysql_close($connection);

// send some more output
?>

</body>
</html>

<?php

// end buffering
// this will invoke the user−defined callback
ob_end_flush();
?>
```

This example uses PHP's output buffering functions to create a custom output buffer handler. When ob_end_flush() is called, PHP will invoke the user−defined function myOutputHandler(), and will pass the entire contents of the buffer to it as a string. It is now up to the function to decide what to do with the buffer − either print it to the Web browser or write it to a static HTML file for later use.

Sometimes, overwriting file contents is not exactly what you need − in some cases, what you really want to do is append to an existing file. This is common when dealing with log files, as in the example below:

```
<?php

// set log file
function writeLog($msg)
{
// set file to write
$filename = "error.log";

// open file
$fh = fopen($filename, "a") or die("Could not open log");
```

```
// create the data string to be written
$str = date("[Y−m−d h:i:s] ", mktime()) . $msg . "\r\n";

// write to log
fwrite($fh, $str) or die("Could not write to log");

// close file
fclose($fh);
}

writeLog("NOTIFY: Initiating database connection");

// open connection to database
$connection = mysql_connect("localhost", "joe", "nfg84m") or
writeLog("FATAL: Unable to connect!");
mysql_select_db("weather") or writeLog("FATAL: Unable to select database!");

// get data
$query = "SELECT * FROM books";
$result = mysql_query($query) or writeLog("FATAL: Error in query: $query. " . mysql_error());

// if a result is returned
if (mysql_num_rows($result) > 0)
{
// iterate through resultset
// print data
while (list($title, $author) = mysql_fetch_row($result))
{
echo "$title − $author <br>";
}
}
else
{
echo "No data available";
}

// close database connection
mysql_close($connection);

writeLog("NOTIFY: Terminating database connection");

?>
```

In this case, every call to the writeLog() function will add to the existing log file contents, rather than overwriting them.

How about copying a file? With the new file_get_contents() and file_put_contents() functions − both of which are binary−safe − it's a snap!

```
<?php
```

```
// set source file
$src = "/bin/ls";

// set destination file
$dst = "/tmp/list";

// read source
$contents = file_get_contents($src) or die ("Could not read source file");

// write destination
file_put_contents($dst, $contents) or die("Could not write destination file");

?>
```

A Matter Of Existence

PHP also comes with a bunch of functions that allow you to test the status of a file − for example, find out whether it exists, whether it's empty, whether it's readable or writable, and whether it's a binary or text file. Of these, the most commonly used operator is the file_exists() function, which is used to test for the existence of a specific file.

Here's an example which asks the user to enter the path to a file in a Web form, and then returns a message displaying whether or not the file exists:

```
<html>

<head>

</head>

<body>

<?php
// if form has not yet been submitted
// display input box
if (!$_POST['file'])
{
?>
<form action="<?=$_SERVER['PHP_SELF']?>" method="post">
Enter file path <input type="text" name="file">
</form>
<?
}
// else process form input
else
{
// check if file exists
// display appropriate message
if (file_exists($_POST['file']))
{
```

```
echo "File exists!";
}
else
{
echo "File does not exist!";
}
}
?>

</body>
</html>
```

There are many more such functions − here's a brief list, together with an example that builds on the one above to provide more information on the file specified by the user.

is_dir() − returns a Boolean indicating whether the specified path is a directory

is_file() − returns a Boolean indicating whether the specified file is a regular file

is_link() − returns a Boolean indicating whether the specified file is a symbolic link

is_executable() − returns a Boolean indicating whether the specified file is executable

is_readable() − returns a Boolean indicating whether the specified file is readable

is_writable() − returns a Boolean indicating whether the specified file is writable

And here's a script that demonstrates how you could use these operators to obtain information on any file on your system:

```
<html>

<head>

</head>

<body>

<?php
// if form has not yet been submitted
// display input box
if (!$_POST['file'])
{
?>
<form action="<?=$_SERVER['PHP_SELF']?>" method="post">
Enter file path <input type="text" name="file">
</form>
<?
}
// else process form input
else
```

```
{
$output = "Filename: <b>" . $_POST['file'] . "</b><br>";

// check if file exists
// display appropriate message
if (file_exists($_POST['file']))
{
// is it a directory?
if (is_dir($_POST['file']))
{
$output .= "File is a directory <br>";
}

// is it a file?
if (is_file($_POST['file']))
{
$output .= "File is a regular file <br>";
}

// is it a link?
if (is_link($_POST['file']))
{
$output .= "File is a symbolic link <br>";
}

// is it executable?
if (is_executable($_POST['file']))
{
$output .= "File is executable <br>";
}

// is it readable?
if (is_readable($_POST['file']))
{
$output .= "File is readable <br>";
}

// is it writable?
if (is_writable($_POST['file']))
{
$output .= "File is writable <br>";
}

}
else
{
$output .= "File does not exist! <br>";
}

echo $output;
}
```

```
?>

</body>
</html>
```

And here's what the output might look like:

Filename: /home/me/addresses.dat
File is a regular file
File is executable
File is readable
File is writable

Permission Granted

In addition to obtaining basic information on a file, PHP also comes with a set of functions designed to provide information on file sizes, permissions and modification times. Here's a list:

filesize() − gets size of file

filemtime() − gets last modification time of file

filemtime() − gets last access time of file

fileowner() − gets file owner

filegroup() − gets file group

fileperms() − gets file permissions

filetype() − gets file type

Consider the following example, which asks for a file name and displays the above information for that file:

```
<html>

<head>

</head>

<body>

<?php
// if form has not yet been submitted
// display input box
if (!$_POST['file'])
{
?>
<form action="<?=$_SERVER['PHP_SELF']?>" method="post">
```

```
Enter file path <input type="text" name="file">
</form>
<?
}
// else process form input
else
{
echo "Filename: <b>" . $_POST['file'] . "</b><br>";

// check if file exists
if (file_exists($_POST['file']))
{
// print file size
echo "File size: <b>" . filesize($_POST['file']) . " bytes</b><br>";

// print file owner
echo "File owner: <b>" . fileowner($_POST['file']) . "</b><br>";

// print file group
echo "File group: <b>" . filegroup($_POST['file']) . "</b><br>";

// print file permissions
echo "File permissions: <b>" . fileperms($_POST['file']) . "</b><br>";

// print file type
echo "File type: <b>" . filetype($_POST['file']) . "</b><br>";

// print file last access time
echo "File last accessed on: <b>" . date("Y−m−d",
fileatime($_POST['file'])) . "</b><br>";

// print file last modification time
echo "File last modified on: <b>" . date("Y−m−d",
filemtime($_POST['file'])) . "</b><br>";

}
else
{
echo "File does not exist! <br>";
}

}
?>

</body>
</html>
```

Here's the output:

Filename: /apps/inventory_control/reports.php

File size: 1503 bytes
File owner: 512
File group: 100
File permissions: 33188
File type: file
File last accessed on: 2003−07−31
File last modified on: 2003−07−25

Note that the various file functions discussed in this section only return valid values if the underlying operating system supports them. On Windows, for example, it's quite likely that the fileowner(), filegroup() and
fileperms() functions will return null values.

In Stat We Trust

An alternative way of obtaining file information is via the stat() function, which retrieves detailed status information on the named file. The return value from stat() is an array consisting of the following elements (take a look at the PHP manual page for the stat() function for more information on what each array element represents):

ARRAY WHAT IT
KEY MEANS
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
dev device number
ino inode number
mode inode protection mode
nlink number of links
uid file owner's user ID
gid file owner's group ID
rdev device type
size file size (in bytes)
atime file's last access timestamp
mtime file's last modify timestamp
ctime file's last change timestamp
blksize filesystem block size
blocks file's allocated blocks

Consider the following example, and its output, which illustrate how stat()
works:

```
<?php

// if form has not yet been submitted

// display input box

if (!$_POST['file'])

{

?>
```

```
            <form action="<?=$_SERVER['PHP_SELF']?>" method="post">

            Enter file path <input type="text" name="file">

            </form>

            <?

            }

            // else process form input

            else

            {

            echo "Filename: <b>" . $_POST['file'] . "</b><br>";
```

```
// check if file exists
if (file_exists($_POST['file']))
{
// get file status
$stats = stat($_POST['file']);

// iterate over array and print information
foreach ($stats as $k=>$v)
{
if (!is_numeric($k))
{
echo "$k: <b>$v</b><br>";
}
}

}
else
{

echo "File does not exist! <br>";

}
}
?>

</body>
</html>
```

Here's an example of the output:

Filename: /home/web/apps/library/index.php
dev: 2049
ino: 767302
mode: 33188
nlink: 1
uid: 512

gid: 100
rdev: 21884
size: 1503
atime: 1059626194
mtime: 1059118092
ctime: 1059118095
blksize: 4096
blocks: 8

Note that stat() will return −1 for those attributes which are unsupported by the underlying operating system.

A Short Break

And that's about all we have time for. Over the last few pages, I offered you a gentle introduction to PHP's file manipulation functions, showing you alternative ways to read and write files on the file system. I also showed you how to obtain detailed status information on any file, including data on attributes like file size, permissions, ownership, et al.

This isn't all, though. PHP also comes with a whole bunch of other, more specialized file and directory functions, which let you copy, delete and rename files; scan directories; work with remote files over HTTP and FTP; perform pattern matches on file names; and read and write to processes instead of files. All that, and more, in the concluding segment of this tutorial − so make sure you come back for that!

Note: Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!