PHP 101 (part 5): Rank and File

Back to School

When you first started reading this series, I promised you that you'd have a whole lot of fun. If you're the cynical type, you may be feeling that I didn't keep my promise. After all, how much fun have you *really* had so far? All you've done is learn a bunch of theoretical rules, added and subtracted numbers from each other, learnt primitive decision-making and gone round and round in the circular funhouse of loops. Heck, if this wasn't a PHP tutorial, it would be kindergarten…

I hear you.

In this segment of our ongoing saga, I'm going to teach you how to do something that's definitely not for kids. It involves getting down and dirty with files on the disk: meeting them (shock!), reading their contents (shriek!) and (horror of horrors!) writing data to them. All of these exciting activities will take place under the aegis of PHP's very cool file manipulation API, which allows you 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.

Let's get started!

Handle With Care

I'll begin with something simple: opening a file and reading its contents. Let's assume that somewhere on your disk, hidden under /usr/local/stuff/that/should/be/elsewhere/recipes/, you have a text file containing the recipe for the perfect Spanish omelette. You now wish

to read the contents of this file into a PHP script.

In order to do this, there are three distinct steps to be

followed:

•Open the file and assign it a file handle.

•Interact with the file, via its handle, and extract its contents

into a PHP variable.

•Close the file.

Here’s a PHP script that does just that:

```
<?php

// set file to read

$file = '/usr/local/stuff/that/should/be/elsewhere/recipes/omelette.txt'or die('Could

not open file!');

// open file


$fh = fopen($file, 'r') or die('Could not open file!'); //fh 即 file handle


// read file contents


$data = fread($fh, filesize($file)) or die('Could not read file!');

// close file


fclose($fh);


// print file contents
```

```
echo $data;

?>
```

Run this script through your Web browser, and PHP should return the contents of the file.

Now let me explain each of the three steps above in detail:

Open the file and assign it a file handle

PHP needs a file handle to read data from a file. This file handle can be created with the `fopen()` function, which accepts two arguments: the name and path to the file, and a string indicating the “mode” in which the file is to be opened (`'r'` for read).

Three different modes are available for use with the `fopen()` function. Here’s the list:

`'r'` – opens a file in read mode

`'w'` – opens a file in write mode, destroying existing file contents

`'a'` – opens a file in append mode, preserving existing file contents

Interact with the file via its handle and extract its contents into a PHP variable

If the `fopen()` function is successful, it returns a file handle, `$fh`, which can be used for further interaction with the file. This file handle is used by the `fread()` function, which reads the file and places its contents into a variable.

The second argument to `fread()` is the number of bytes to be read. You can usually obtain this information through the `filesize()` function, which – who'd have guessed it?!- returns the size of the file in bytes.

Close the file

This last step is not strictly necessary as PHP closes the file automatically once it reaches the end of the script, but it's a good habit to develop. Explicitly closing the file with `fclose()` has two advantages: it ties up loose ends in your script, and it wins you lots of good karma from the PHP community.

You probably haven't see the `die()` function before, either. This function is mostly used as a primitive error-handling mechanism. In the event of a fatal error, such as the file path being invalid or the file permissions being such that PHP cannot read it, `die()` terminates script processing and optionally displays a user-specified error message indicating why it committed suicide.

Different Strokes

An alternative method of reading data from a file is the very cool `file()` function, which reads the entire file into an array (remember them?) with one line of code. Each element of the array then contains one line from the file. To display the contents of the file, simply iterate over the array in a `foreach()` loop and print each element.

The following example demonstrates:

```
<?php
// set file to read
$file = '/usr/local/stuff/that/should/be/elsewhere/recipes/omelette.txt'or die('Could not read file!');

// read file into array

$data = file($file) or die('Could not read file!');

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

    echo $line;

}
?>
```

In this example, the `file()` command opens the file, reads it into an array and closes the file – all in one, single, elegant movement. Each element of the array now corresponds to a line from the file. It’s easy to print the file’s contents now – just reach for that mainstay of array processing, the `foreach()` loop.

Don’t want the data in an array? Try the `file_get_contents()` function, new in PHP 4.3.0 and PHP 5.0, which reads the entire file into a string:

```
<?php
// set file to read
$file = '/usr/local/stuff/that/should/be/elsewhere/recipes/omelette.txt';

// read file into string

$data = file_get_contents($file) or die('Could not read file!');

// print contents

echo $data;
?>
```

Who am I kidding? I always use the one-line functions noted above instead of the three-line sequence of fopen(), fread() and fclose().

Laziness conquers all.

When Laziness is a Virtue

PHP also offers two very useful functions to import files into a PHP script: the include() and require() functions. These functions can be used to suck external files lock, stock and barrel into a PHP script, which is very handy if, for example, you have a modular application which has its code broken down across files in separate locations.

The best way to understand the utility of the include() and require() functions is with an example. Assume that on your Web site you have a standard menu bar at the top of every page, and a standard copyright notice in the bottom. Instead of copying and pasting the header and footer code on each individual page, PHP gurus simply create separate files for the header and footer, and import them at the top and bottom of each script. This also makes a change to the site design easier to implement: instead of manually editing a gazillion files, you simply edit two, and the changes are reflected across your entire site instantaneously.

Let’s see a real live example of this in action. Create the header in one file, called header.php:

```
<html>

<head>

<title><?php echo $page['title'];?></title>
</head>

<body>

<!-- top menu bar -->

<table width="90%" border="0" cellspacing="5" cellpadding="5">

<tr>

<td><a href="#">Home</a></td>

<td><a href="#">Site Map</a></td>
<td><a href="#">Search</a></td>

<td><a href="#">Help</a></td>

</tr>
```

```
</table>

<!-- header ends -->
```

Next, create the footer with the copyright notice in a second file,

`footer.php`:

```
<!-- footer begins -->

<br />
<center>Your usage of this site is subject to its published <a
href="tac.html">terms and conditions</a>. Data is copyright Big Company Inc,
1995-<?php echo date("Y", mktime()); ?></center>
</body>

</html>
```

Finally, create a script to display the main content of your site,

and `include()` the header and footer at appropriate places:

```
<?php
// create an array to set page-level variables
$page = array();

$page['title'] = 'Product Catalog';

/* once the file is imported, the variables set above will become available to it */
// include the page header


include('header.php');
?>





<!-- HTML content here -->
<?php
// include the page footer

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

Now, when you run the script above, PHP will automatically read in the header and footer files, merge them with the HTML content, and display the complete page to you. Simple, isn’t it?

Notice that you can even write PHP code inside the files being imported. When the file is first read in, the parser will look for

`<?php...?>` tags, and automatically execute the code inside it.

(If you're familiar with JavaScript, you can use this feature to replicate functionality similar to that of the `onLoad()` page event handler in JavaScript.)

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 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.

Writing to Ma

After everything you've just read, you've probably realized that reading a file is not exactly brain surgery. So let's proceed to something slightly more difficult – writing to a file.

The steps involved in writing data to a file are almost identical to those involved in reading it: open the file and obtain a file handle, use the file handle to write data to it, and close the file. There are two differences: first, you must `fopen()` the file in write mode (`'w'` for write), and second, instead of using the fread() function to read from the file handle, use the fwrite() function to write to it. Take a look:

```
<?php
// set file to write
$file = '/tmp/dump.txt';

// open file

$fh = fopen($file, 'w') or die('Could not open file!');
// write to file

fwrite($fh, "Look, Ma, I wrote a file!
") or die('Could not write to file');

// close file

fclose($fh);
?>
```

When you run this script, it should create a file named dump.txt in /tmp, and write a line of text to it, with a carriage return at the end. Notice that double quotes are needed to convert into a carriage

return.

The fopen(), fwrite() and fread() functions are all binary-safe,

which means you can use them on binary files without worrying about

damage to the file contents. Read more about many of the issues related

to binary-safe file manipulation on different platforms

athttp://www.php.net/manual/en/function.fopen.php.

If I’ve spoiled you by showing you the one-line shortcut functions

for file reads, let me damage you further by introducing you to the

file_put_contents() function, new in PHP 5.0, which takes a string and

writes it to a file in a single line of code.

```
<?php
// set file to write
$filename = '/tmp/dump.txt';

// write to file
file_put_contents($filename, "Look, Ma, I wrote a file!
") or die('Could not write to file');
?>
```

Bear in mind that the directory in which you’re trying to create the

file must exist before you can write to it. Forgetting this important step is a common cause of script errors.

Information is Power

PHP also comes with a bunch of functions that allow you to test the status of a file – for example to 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 (!isset($_POST['file'])) {

?>
        <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
        Enter file path <input type="text" name="file">

    </form>
<?php

}

// 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, followed by an example that builds on the previous one 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

•filesize() – gets size of file

•filemtime() – gets last modification time of file

•filamtime() – gets last access time of file

•fileowner() – gets file owner

•filegroup() – gets file group

•fileperms() – gets file permissions

•filetype() – gets file type

This script asks for a file name as input and uses the functions above to return information on it.

```
<html>

<head>

</head>

<body>
<?php

/* if form has not yet been submitted, display input box */
if (!isset($_POST['file'])) {
```

```
?>
        <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
        Enter file path <input type="text" name="file">

    </form>
<?php

}

// else process form input

else {

    echo 'File name: <b>'.$_POST['file']  .'</b><br />';
        /* check if file exists and display appropriate message */

    if (file_exists($_POST['file'])) {

        // print file size

        echo 'File size: '.filesize($_POST['file']).' bytes<br />';
                // print file owner

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

```
// print file group

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

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

// print file type

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

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

echo 'File last modified on: '.date('Y-m-d',filemtime($_POST['file'])).'<br />';
// is it a directory?

if (is_dir($_POST['file'])) {

    echo 'File is a directory <br />';

}

// is it a file?
```

```
if (is_file($_POST['file'])) {
            echo 'File is a regular file <br />';

}

// is it a link?

if (is_link($_POST['file'])) {

    echo 'File is a symbolic link <br />';
    }

// is it executable?

if (is_executable($_POST['file'])) {

    echo 'File is executable <br />';

}

// is it readable?

if (is_readable($_POST['file'])) {
            echo 'File is readable <br />';

}
```

```
        // is it writable?

        if (is_writable($_POST['file'])) {

            echo 'File is writable <br />';
            }

    }

    else {

        echo 'File does not exist! <br />';

    }

}

?>
</body>

</html>
```

And here’s what the output might look like:

```
File name: /usr/local/apache/logs/error_log

File owner: 0

File permissions: 33188

File last accessed on: 2004-05-26

File is a regular file

```

Breaking Eggs

So now you know how to read a file, write to it, and test its status. Let’s look at some examples of what you can do with this new-found power.

Let’s go back to my Spanish omelette recipe. Let’s suppose I’m feeling generous, and I decide that I’d like to hear what people *really* think about my culinary skills. Since I have a bunch of recipes that I’d like to share with people, and since they all look something like this:

```
SPANISH OMELETTE

- 1 chopped onion

- 1/2 chopped green pepper

- Salt and pepper to taste

1. Fry onions in a pan

3. Add tomatoes, green pepper, salt and pepper to taste

```

I need a quick way to convert them all into HTML so that they look presentable on my Web site. We’ve already established that I’m lazy, so fuggedaboutme re-creating the recipes in HTML. Instead, I’ll have PHP do the heavy lifting for me:

```
<html>

<head></head>
<body>
<?php

// read recipe file into array

$data = file('/usr/local/stuff/that/should/be/elsewhere/omelette.txt') or die('Could
not read file!');

/* first line contains title: read it into variable */

$title = $data[0];

// remove first line from array

array_shift($data);
?>
<h2><?php echo $title; ?></h2>
<?php

/* iterate over content and print it */

foreach ($data as $line) {
    echo nl2br($line);

}

?>
</body>


</html>
```

I’ve used the `file()` function to read the recipe into an array, and assign the first line (the title) to a variable. That title is then printed at the top of the page. Since the rest of the data is fairly presentable as is, I can simply print the lines to the screen one after the other. Line breaks are automatically handled for me by the extremely cool `nl2br()` function, which converts regular text linebreaks into the HTML equivalent, the `<br />` tag. The end result: an HTML-ized version of my recipe that the world can marvel at. Take a look:

```
&lt;html&gt;
[illegible]

[illegible]
&lt;/h2&gt;
[illegible]
- 1 chopped onion&lt;br /&gt;
[illegible]
- 1/2 chopped green pepper&lt;br /&gt;
[illegible]

[illegible]
METHOD:&lt;br /&gt;
[illegible]
2. Pour beaten eggs over onions and fry gently&lt;br /&gt;
[illegible]
4. Serve with toast or bread&lt;br /&gt;
[illegible]
&lt;/body&gt;
[illegible]
```

If the elegance and creative simplicity of my Spanish omelette recipe has left you speechless, I'm not surprised – many people feel that way. Until you get your voice back: Ciao… and make sure you come back to work through Part Six of PHP 101, which discusses creating your own reusable functions.