Stream Me Up, Scotty (part 2)

By Vikram Vaswani

Table of Contents

Skinning Cats

In the first part of this article, I took a look at the new FTP commands built into PHP4, and demonstrated how they could be used to build a fully Web−based FTP client. But there's more than one way to skin a cat − which is why the second part of this article will explore an alternative technique of performing remote file management, this time without resorting to PHP's FTP commands.

Over the next few pages, I'll be explaining how your PHP scripts can interact with a filesystem to display files and directories, copy and move data, and transfer files via the Web browser. Keep reading!

Looking Around

Let's start at the top. Before you can get around to copying and moving files, you first need to be able to see them. And that's where PHP's directory commands come in − they provide you with an easy way to view and manipulate the contents of a specific directory on the system.

In order to access the contents of a directory, you first need to open a "handle" to the directory with the opendir() function; you can then read and manipulate the contents with readdir(). Once you're done, it's good programming practice to close the handle you've just opened with closedir().

The following example demonstrates these three steps:

```
<? // get directory handle $hook = opendir("/tmp"); // read
directory and echo list while (readdir($hook)) { $file =
readdir($hook); echo "$file<br>"; } // close directory
closedir($hook); ?>
```

And you should see something like this:

```
.. index.html .XF86Setup686 something.doc file.jpg
```

A more professional approach is to write it the way the manual recommends.

```
<? // get directory handle $hook = opendir("/tmp"); // read
directory and echo list while (($file =
readdir($hook))!==false) { echo "$file<br>"; } // close
directory closedir($hook); ?>
```

The Object−Oriented Approach

Another approach to obtain a directory listing is to use the PHP "dir" class, which creates a directory object with its own properties and functions. Take a look:

```
<? // get directory handle $hook = dir("/tmp"); // display
location echo "<b>Current path is $hook−>path</b><br>"; //
read directory and echo list while ($file=$hook−>read()) {
echo "$file<br>"; } // close directory $hook−>close(); ?>
```

In this case, three functions are available − read(), close() and rewind() − together with two properties − $path, which records the current location in the filesystem, and $handle, which stores the resource ID for the directory handle.

And if you take a close look at the output from the two examples above, you'll see that PHP typically also displays the parent and current directories in the listing (look for the . and .. symbols.) As you'll see when I get to the composite example, it's much neater to filter these two items out of the directory listing.

```
<? // get directory handle $hook = dir("/tmp"); // display
location echo "<b>Current path is $hook−>path</b><br>"; //
read directory and echo list while ($file=$hook−>read()) { if
($file != "." &$file != "..") { echo "$file<br>"; } } // close
directory $hook−>close(); ?>
```

And the output is:

```
index.html .XF86Setup686 something.doc file.jpg
```

Much neater!

Finally, there's the chdir() function, which simply changes to a specific directory. You can combine this with what you've just seen to create a very simple file browser − take a look:

```
<html> <head> <basefont face="Arial"> </head> <body> <? // if
form has not been submitted, display form if (!$submit) { ?>
<center> <form action=<? echo $PHP_SELF; ?> method=post> Enter
location: <input type=text name=location size=15> <input
type=submit name=submit value="Go"> </form> </center> <? }
else { // change to directory $res = chdir($location); // if
chdir() successful if ($res) { // get directory handle $hook =
dir($location); // display location echo "<b>Current path is
$hook−>path</b><br>"; // read directory and echo list while
($file=$hook−>read()) { if ($file != "." &$file != "..") {
echo "$file<br>"; } } // close directory $hook−>close(); }
else { // display error echo "<b>Unable to locate directory
$location</b><br>"; } } ?> </body> </html>
```

The Truth About Files And Directories

Now, let's move to files. As you've seen, the functions above simply display a combined listing of files and sub−directories in the selected directory. How about separating the two?

Well, PHP comes with a bunch of functions which let you distinguish between files and directories, and even between different types of files. The following snippet uses the is_dir() function to display directories in a different colour:

```
... <? // get directory handle $hook = opendir($location); //
read directory and echo list while (($file = readdir($hook))
!== false) { if ($file != "." &$file != "..") { // set up full
path $path = $location . "/" . $file; // check for directory
if (is_dir($path)) { echo "<font color=Red>$file</font><br>";
} else { echo "<font color=Green>$file</font><br>"; } } } //
close directory closedir($hook); ?> ...
```

In this case, the is_dir() function is used to identify whether or not a directory is being listed. PHP also offers the is_file() and is_link() function to identify files and links respectively.

You can also obtain information on the listed files − size, permissions, ownership and the like. For example, the following code snippet shows you how to use the is_readable(), is_writeable() and is_executable() functions to display file properties, and the filesize() function to obtain the size of the file in bytes.

```
... <? // get directory handle $hook = opendir($location); //
read directory and echo list while (($file = readdir($hook))
!== false) { if ($file != "." &$file != "..") { // set up full
path $path = $location . "/" . $file; echo "<table border=1
cellspacing=2>"; // set up attribute list $att = ""; if
(is_readable($path)) { $att .= "r"; } if (is_writeable($path))
{ $att .= "w"; } if (is_executable($path)) { $att .= "x"; } //
check for directory if (is_dir($path)) { // print file
information echo "<tr><td width=200><font
color=red>$file</font></td><td width=100>0 bytes</td><td
width=100>$att</td></tr>"; } else { // get file size $size =
filesize($path); // print file information echo "<tr><td
width=200>$file</td><td width=100>$size bytes</td><td
width=100>$att</td></tr>"; } } } // close directory
closedir($hook); echo "</table>"; ?> ...
```

To Create And Destroy

Next up, a little file and directory manipulation. If you'd like to create directories, PHP provides the mkdir() function, which accepts a file path and a mode as parameters, So, if I wanted to create a directory called "/tmp/me" with permissions 755, I would use something like this:

```
<? if (mkdir ("/tmp/me", 0755)) { echo "Successful!"; } else {
echo "Unsuccessful"; } ?>
```

And to reverse the process, there's always the rmdir() function, used to delete directories. Note that rmdir() will fail if the directory permissions don't allow deletion, or if the directory contains one or more files.

```
<? if (rmdir ("/tmp/me")) { echo "Successful!"; } else { echo
"Unsuccessful − check permissions and ensure directory is
empty!"; } ?>
```

You can delete files with the unlink() function, which accepts a filename as parameter. <? if (unlink("/tmp/dummyfile")) { echo "Successful!"; } else { echo "Unsuccessful!"; } ?>

And finally, you can copy files with the copy() function,

```
<? if (copy("/tmp/source", "/tmp/destination")) { echo
"Successful!"; } else { echo "Unsuccessful!"; } ?>
```

and rename (or move) them with the rename() function.

```
<? if (rename("/tmp/oldfile", "/etc/newfile")) { echo
"Successful!"; } else { echo "Unsuccessful!"; } ?>
```

Upsa−daisy!

PHP also allows you to upload files via HTTP, assuming you have an RFC−1867 compliant browser (if you're using a relatively recent version of either Internet Explorer or Netscape Navigator, you're OK). There are two components to this: the form which accepts the file upload, and the server−side script which checks the file and decides where to put it.

Let's start with the form first.

```
<html> <head> </head> <body> <center> <form
enctype="multipart/form−data" action="upload.php4"
method=post> Enter file name <input name="upfile" type="file">
<br> <input type="submit" value="Beam Me Up, Scotty"> </form>
</center> </body> </html>
```

There are two points of note here: the form's encoding type of "multipart/form−data", and the file browse button <input name="upfile" type="file">, which displays a directory browser and lets you select a file from your system. Once you're done, hit the Submit button and let the script "upload.php4" take over.

Before I take you through "upload.php4", though, you should be aware that when a file is uploaded in this manner, PHP typically creates four variables, each of them containing the name of the file field as prefix. In the example above, these variables would be:

$upfile_name − the original name of the file

$upfile − the temporary name assigned to the file by PHP once it has been successfully uploaded

$upfile_size − the size of the file

$upfile_type − the MIME file type

You should also note that once the file is uploaded, it is stored in the temporary file area on the remote computer, and your script needs to move it out of there to your desired destination directory. If this does not happen, the file will be automatically deleted.

As stated in the first part of this article, these variables are also available in the $HTTP_POST_FILES array, and it is recommended that you use this array to access the variables above (rather than accessing them directly) for greater security.

Remote Control

Given all that information, let's take a look at "upload.php4"

```
<? // get some information echo "Filename: $upfile_name<br>";
echo "Temporary filename: $upfile<br>"; echo "File size:
$upfile_size bytes<br>"; echo "File type: $upfile_type<br>";
// if upload successful if ($upfile) { echo "Upload
successful!<br>"; // copy file to new location if
(copy($upfile, "/tmp/uploads/" . $upfile_name)) { echo "File
copy successful!<br>"; } } // else display error else { echo
"Upload unsuccessful!<br>"; } ?>
```

Be warned: you should enforce strict rules about what can and can't be uploaded when using such a system in a production environment. Failure to do this would open up a security hole which would allow users to upload Perl scripts, C binaries and PHP documents to the server, and perhaps even execute them remotely.

A good way to avoid this is to use the $upfile_type variable to decide which files get uploaded, and which get rejected. For example,

```
<? if ($upfile_type == "text/plain" || $upfile_type ==
"text/html" || $upfile_type == "image/gif" || $upfile_type ==
"image/jpeg") { // file upload code } else { echo "Permission
denied!"; } ?>
```

Similarly, you can use the $upfile_size variable to reject files which are too large for comfort.

The Application

Now that you know the various techniques, it's time to build a simple file manager. You can download the complete source code from here − I'll just give you a quick tour through the various sections.

"index.php" sets up the main interface for the applications. On the left side is a list of available functions, while the right side contains a listing of files and directories. Directories can be entered by clicking on them, while files located within the Web server document root can be downloaded by clicking on them (files outside the server root cannot be downloaded). The script "index.php" can accept the parameter $dir, which indicates the directory to be displayed.

All this takes place via two functions, links() and filelist(), which you can see in the file "links.php" − the former sets up the links on the left side, while the latter uses the opendir() and other file information functions to obtain information on the size and permissions of files in the directory. Directories are set up as active hyperlinks − once a directory is clicked, the script "index.php" is called once again, but with a different $dir value.

At any point, you can access the functions on the left side − "upload.php" accepts a file for upload, "mkdir.php" allows you to create a new directory in the current locations, "rmdir.php" reverses the process, and "rmfile.php" allows you to delete one or more files. All these scripts accept a single parameter, $dir, which tells them where their respective actions are to be performed.

It's also possible to download files by turning them into clickable hyperlinks − the application checks to first make sure that the files are in the Web server DocumentRoot, sets a flag if they are, and then turns each file into a hyperlink. Again, there are security implications here − so be careful!

And once you're done, it might be worthwhile to go back to the equivalent file browser we built in the first part of this article. If you compare the two approaches, other things being equal, you'll notice that the HTTP file browser is much faster than the FTP file browser, and also allows more flexibility when manipulating files.

Hopefully, this exercise has helped you gain a clearer idea of PHP's file manipulation capabilities, together with an understanding of what is and what isn't possible. I'm off − but I'll see you soon!