File And Directory Manipulation In PHP (part 2)

By icarus
2003-08-21

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/PHP/File_Manipulation2/

The Road Ahead

In the first part of this article, I introduced you to the basic filesystem functions available in PHP, showing you (among other things) how to read and write files on the file system and obtain detailed status information on any file, including data on attributes like file sizes and permissions.

That, however, was just the tip of the iceberg. PHP's filesystem API includes a number of specialized file and directory functions, which let you copy, delete and rename files; scan directories; work with uploaded files over HTTP; perform pattern matches on file names; and read and write to processes instead of files. So keep reading - we've got a long and interesting journey ahead of us!

Stripping It To The Bone

You may remember, from the first part of this article, how I used the
fgets() function to read the contents of a file and print it to the browser. In case you don't, here's a quick reminder:

```
 <?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 ($fh);
 fclose ($fh);

echo "-- ALL DONE --";

?>
```

PHP also offers the fgetss() function, which works just like the regular
fgets() function, except that it also strips out HTML and PHP code from the lines it reads. So, for example, if you had a file containing intermingled HTML code and ASCII text (as most HTML files are), like in the following example,

```
 <HTML>

 <HEAD>

 </HEAD>
<BODY>
<TABLE BORDER="0" CELLSPACING="0" CELLPADDING="0">

 <TR>

 <TD>

 <FONT FACE="Verdana, Arial" SIZE="2" COLOR="#66CC00">

 <B><A HREF="article.php?id=169"><IMG SRC="/images/sm_arrow.jpg"
 WIDTH=20 HEIGHT=20 ALT="" BORDER=0 ALIGN="ABSMIDDLE" HSPACE=4>Learning
 To SMILe</A></B> </FONT> </TD> </TR> <TR>
 <TD> <FONT FACE="Verdana, Arial" SIZE="1" COLOR="Black">

 Use SMIL to quickly build new Web-based multimedia applications | 20 Dec 2002

 </FONT>

 </TD>
```

```
</TR>

<TR>

<TD HEIGHT=25>

&nbsp;

</TD>

</TR>

<TR>

<TD>

<FONT FACE="Verdana, Arial" SIZE="2" COLOR="#66CC00">

<B><A HREF="article.php?id=168"><IMG SRC="/images/sm_arrow.jpg"
WIDTH=20 HEIGHT=20 ALT="" BORDER=0 ALIGN="ABSMIDDLE" HSPACE=4>Writing
A User Manual (part 1)</A></B> </FONT> </TD> </TR>
<TR> <TD> <FONT FACE="Verdana, Arial" SIZE="1"
COLOR="Black"> It ain't easy writing a user manual...but our cheat
sheet should help | 20

Dec 2002

</FONT>

</TD>

</TR>
```

```
 </TABLE>
</BODY>

 </HTML>
```

you could retrieve just the ASCII from it with the following example script:

```
 <?php
// set file to read

 $filename = "data.html";
// open file

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

 while (!feof($fh))

 {

 // strip out tags

 $data = fgetss($fh, 999);



 // if nothing present after stripping tags, move on

 // else print the cleaned line

 if (trim($data) != "")
```

```
    {

    echo trim($data) . "\r\n";

    }

    }
  // close file

    fclose ($fh);

  echo "-- ALL DONE --";

  ?>
```

Here' s what the output looks like:

Learning To SMILe
Use SMIL to quickly build new Web-based multimedia applications | 20 Dec 2002 &nbsp; Writing A User Manual (part 1) It ain't easy writing a user manual...but our cheat sheet should help | 20 Dec 2002

Fertile Fields

In addition to the simple fgets() function, PHP also offers the more-sophisticated fgetcsv() function, which not only reads in data from a file, but also parses each line and, using the comma (,) symbol as delimiter, splits the data on each line into fields for further processing. The return value from every call to fgetcsv() is an array containing the fields found.

An example might make this clearer. Consider the following CSV file,

```
john,67,John Doe,India

sue,32,Sue Me,New York

sarah,10,Sarah Whu,Korea

ramu,23,R Amulet,London
```

and this PHP script, which reads it and displays the information within it as an XML document:

```
  <?php
// open XML tags

  echo "<?xml version='1.0'?>";

  echo "<collection>";
// set file to read

  $filename = "users.txt";
// open file

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

  while (!feof($fh))

  {

  // create XML structure

  echo "<user>";

  $fields = fgetcsv($fh, 1000);

  echo "<username>" . $fields[0] . "</username>";

  echo "<fullname>" . $fields[2] . "</fullname>";

  echo "<age>" . $fields[1] . "</age>";

  echo "<city>" . $fields[3] . "</city>";
  echo "</user>";

  }
// close file

  fclose ($fh);
echo "</collection>";
?>
```

In this case, the comma-separated values in the input file are automatically parsed into an array, and can then be processed, or reassembled in any order you like, to create different output. Here' s what the script above results in:

<?xml version='1.0'?> <collection><user><username>john</username><fullname>John Doe</fullname><age>67</age><city>India</city></user><user><username>sue</username><fullname>Sue Me</fullname><age>32</age><city>New York</city></user><user><username>sarah</username><fullname>Sarah Whu</fullname><age>10</age><city>Korea</city></user><user><username>ramu</username><fullname>R Amulet</fullname><age>23</age><city>London</city></user></collection>

Configuring The System

If what you' re really after involves reading configuration variables in from a standard .INI file, you don' t need to write custom code to parse the file and read in the variable-value pairs. Instead, just use PHP' s parse_ini_file() function, which automatically takes care of this for you.

Consider the following sample .INI file,

```
[global]

printing = bsd

default case = lower

log file = /var/log/samba/log.%m

printcap name = /etc/printcap
```

```
  max log size = 50

  domain master = yes

  dns proxy = no
[temp]

  comment = Temporary file space

  path = /tmp

  read only = no

  public = yes

  create mask = 0777

  force group = nobody

  force user = nobody
```

and the PHP code to parse it:

```
  <?php
// set file to read

  $filename = "samba.ini";
// read INI file into array

  $data = parse_ini_file($filename);
// print array
```

```
print_r($data);

?>
```

A quick glance at the output shows that PHP has, indeed, read the file, parsed its contents, and converted the variable-value pairs into an associative array.

```
Array
(
[printing] => bsd
[default case] => lower
[log file] => /var/log/samba/log.%m
[printcap name] => /etc/printcap
[max log size] => 50
[domain master] => 1
[dns proxy] =>
[comment] => Temporary file space
[path] => /tmp
[read only] =>
[public] => 1
[create mask] => 0777
[force group] => nobody
[force user] => nobody
)
```

The only problem with the approach, however, is that variables with the same name from different sections will override each other; if there are multiple configuration variables with the same name, the output array will always contain only the last value. In order to illustrate, look what happens when I add a new section to the sample file above which repeats some of the variables from a previous section:

```
[global]

printing = bsd

default case = lower

log file = /var/log/samba/log.%m

printcap name = /etc/printcap

max log size = 50

domain master = yes

dns proxy = no
```

```
 dns proxy = no
[temp]

 comment = Temporary file space

 path = /tmp

 read only = no

 public = yes

 create mask = 0777

 force group = nobody

 force user = nobody



 [shared]

 comment = Shared area

 path = /shared

 printable = no

 create mode = 0770

 directory mode = 0770

 writeable = yes
```

Here' s the output:

```
Array
(
[printing] => bsd
[default case] => lower
[log file] => /var/log/samba/log.%m
[printcap name] => /etc/printcap
[max log size] => 50
[domain master] => 1
[dns proxy] =>
[comment] => Shared area
[path] => /shared
[read only] =>
[public] => 1
[create mask] => 0777
[force group] => nobody
[force user] => nobody
[printable] =>
[create mode] => 0770
[directory mode] => 0770
[writeable] => 1
)
```

As you can see, some of the variable-value pairs (from the "temp" section of the file) have been lost. PHP offers a solution to this problem by allowing a second, optional argument to parse_ini_file() - a Boolean indicating whether the namespaces of the various sections should be respected. When I add that to the script above,

```
 <?php

// set file to read



 $filename = "samba.ini";

// read INI file into array



 // process each section separately



 $data = parse_ini_file($filename, true);

// print array



 print_r($data);

?>
```

look how drastically the output changes:

```
Array
(
[global] => Array
(
[printing] => bsd
[default case] => lower
[log file] => /var/log/samba/log.%m
[printcap name] => /etc/printcap
[max log size] => 50
[domain master] => 1
[dns proxy] =>
)

[temp] => Array
(
[comment] => Temporary file space
[path] => /tmp
[read only] =>
[public] => 1
[create mask] => 0777
[force group] => nobody
[force user] => nobody
)

[shared] => Array
(
[comment] => Shared area
[path] => /shared
[printable] =>
[create mode] => 0770
[directory mode] => 0770
[writeable] => 1
)

)
```

With the addition of the second argument to parse_ini_file(), PHP now creates a nested array, with the outer array referencing the sections, and each inner one referencing the variables in each section.

The Right Path

In addition to functions that allow you to obtain information on file sizes, permissions and modification times, PHP also offers a number of functions designed to manipulate file and path names, and split a file path into its constituent components. The two most commonly-used ones here are the basename() function, which returns the filename component of a path, and the dirname() function, which returns the directory name component of a path

The following example demonstrates the basename() and dirname() components in action, by splitting a file path into its constituents:

```
 <?php

// set path



 $path = "/usr/local/apache/bin/httpd";

// print filename



 echo "File is " . basename($path) . "\r\n";

// print directory name
// print directory name

  echo "Directory is " . dirname($path) . "\r\n";

?>
```

Here' s the output:

File is httpd
Directory is /usr/local/apache/bin

You can also use the pathinfo() function to obtain this information - this function returns an associative array containing keys for directory name, file name and file extension. Take a look at this next script, which returns this information for the directory holding the currently executing script.

```
  <?php

// parse current file path

  $info = pathinfo($_SERVER['PHP_SELF']);

// print info

  print_r($info);

?>
```

Here' s the output:

```
Array
(
[dirname] => /dev/php
[basename] => fs.php
[extension] => php
)
```

Finally, you can use the realpath() function to translate relative paths into absolute ones, as below:

```
  <?php

// returns /usr/local/lib

  echo realpath("/usr/local/apache/bin/../../lib/");

// returns /lib
// returns /lib

  echo realpath("../../lib");

?>
```

You can obtain the script' s current working directory by combining the dirname() function with the special $_SERVER[' SCRIPT_FILENAME'] variable,

```
  <?php

echo "Current directory is " . dirname($_SERVER['SCRIPT_FILENAME']);

?>
```

or with the alternative getcwd() function, as below,

```
  <?php

// get current working directory

  echo "Current directory is " . getcwd();

?>
```

or even through creative use of the realpath() function, as below:

```
  <?php

echo "Current directory is " . realpath(".");

?>
```

Move It

In addition to offering you path information, PHP comes with a whole bunch of functions designed to simplify the task of moving, copying, renaming and deleting files on the filesystem. The first of these is the copy() function, which accepts two arguments, a source file and a destination, and copies the former to the latter. Here' s an example which demonstrates:

```
 <?php
// check to see if file exists

 // if so, back it up

 if (file_exists("matrix.txt"))

 {

 copy ("matrix.txt", "matrix.txt.orig") or die ("Could
 not copy file"); }
?>
```

Note that if the destination file already exists, copy() will usually overwrite it. A failure to copy the file will cause copy() to return false; success returns true.

A corollary to the copy() function is the rename() function, which can be used to both rename and move files. Like copy(), it too accepts two arguments, a source file and a destination file. Consider the following example, which renames a file,

```
 <?php
// check to see if file exists

 // if so, rename it

 if (file_exists("matrix.txt"))

 {

 rename ("matrix.txt", "neoworld.txt") or die ("Could
 not rename file"); }
?>
```

and this one, which simultaneously renames and moves a file.

```
 <?php
// check to see if file exists

 // if so, move and rename it

 if (file_exists("/home/john/newsletters"))

 {

 rename ("/home/john/newsletters", "/home/john/mail/lists")
 or die ("Could not move file"); }
?>
```

It' s possible to rename directories in the same manner as files - as illustrated in this next snippet:

```
 <?php
// check to see if directory exists

 // if so, rename it

 if (is_dir("Mail"))

 {

 rename ("Mail", "mail-jun-03") or die ("Could not rename
 directory"); }
?>
```

The rename() function comes in handy when you need to update files which are constantly being used by multiple

The rename() function comes in handy when you need to update files which are constantly being used by multiple processes. Instead of directly updating the target file with new data, rename() allows you to copy the original file contents into a new file, make your changes and then, once you' re happy with the result, simply rename() the new file to the old one.

The following example demonstrates:

```
 <?php

// read file



 $contents = file("/etc/inittab") or die ("Could not read file");

// add a line to it



 $contents[] = "x:5:respawn:/etc/X11/prefdm -nodaemon";

// write contents to temporary file



 $tmpfile = tempnam("/tmp", "inittab.tmp.");

// open file



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

foreach ($contents as $line)



 {



 fwrite ($fh, $line) or die ("Could not write file");



 }

// close file



 fclose ($fh);

// move temporary file over original



 rename($tmpfile, "/etc/inittab") or die ("Could not replace file");

?>
```

Note my use of the tempnam() function above - this function generates a unique file name, given a directory and a

Note my use of the tempnam() function above - this function generates a unique file name, given a directory and a filename prefix, and can help to avoid filename collisions between different processes.

When it comes time to delete files, PHP offers the unlink() function, which can be used to erase a file from the filesystem. Consider the following example, which demonstrates by deleting a specific file:

```
 <?php

// check to see if file exists

 // if so, erase it

 if (file_exists("error.log"))

 {

 unlink ("error.log") or die ("Could not delete file");

 }

?>
```

You can also use the unlink() function to iterate over a directory and remove all the files within it - this is demonstrated in an example coming up shortly.

Note that the unlink() function (and indeed, all other file manipulation
functions) will fail if the user ID under which the Web server is running does not have adequate permissions to delete or otherwise modify the named file(s).

Beam Me Up

Most often, you' ll find yourself using rename(), copy() and unlink() functions in the context of files uploaded to PHP via a Web browser - so-called "HTTP file uploads". Consider the following example, which
demonstrates:

```
 <html>

 <head>

 </head>

<body>
```

```
<?php

 if (!$_POST['submit'])

 {

 ?>

 <form enctype="multipart/form-data" action="<?=$_SERVER['PHP_SELF']?>"

 method="post">

 Description:

 <br>

 <input type="Text" name="desc" size="40">

 <p>

 File:

 <br>

 <input type="file" name="file">

 <p>

 <input type="submit" name="submit" value="Add Image">

 </form>

 <?
```

```
<?

}

else

{
// validate form data

if ($_FILES['file']['size'] == 0) { die("Bad upload!"); }
if ($_FILES['file']['type'] != "image/gif" && $_FILES['file']['type']
!= "image/jpeg" && $_FILES['file']['type'] !=

"image/pjpeg") { die("Invalid file format!"); }
if (!$_POST['desc']) { die("No description!"); }
// open connection to database

$connection = mysql_connect("localhost", "root", "secret")
or die ("Unable to connect!");

mysql_select_db("db1") or die ("Unable to select database!");
// formulate and execute query

$query = "INSERT INTO gallery (dsc) VALUES ('" . $_POST['desc']. "')";

$result = mysql_query($query) or die("Error in query: " . mysql_error());

$id = mysql_insert_id($connection);
// get file type and rename file to recordID.ext

if ($_FILES['file']['type'] == "image/gif") { $ext = ".gif";
}

if ($_FILES['file']['type'] == "image/jpeg") { $ext = ".jpg";
```

```
 }

 if ($_FILES['file']['type'] == "image/pjpeg") { $ext = ".jpg";

 }

$newFileName = $id . $ext;

// copy file to new location

 copy($_FILES['file']['tmp_name'], "/mydatadir/" . $newFileName);

// update database with new file name

 $query = "UPDATE gallery SET filename = '$newFileName' WHERE id = '$id'";

 $result = mysql_query($query) or die("Error in query: " . mysql_error());

}

 ?>

 </body>

 </html>
```

In this case, when a file is uploaded, it is automatically stored in a temporary directory by PHP, and its temporary filename is exposed via the "tmp_name" key of the $_FILES array. A copy() function can then be used to copy the uploaded file from its temporary location to its new location, and it can also be renamed along the way if needed (as in the above example). Once the script finishes executing, the temporary file is automatically deleted by PHP.

Since file uploads are fairly common in PHP, the language also offers two specialized functions designed specifically to assist you in the process of handling such uploaded files: the is_uploaded_file() function, which tests if a file was uploaded via the HTTP POST method, and the move_uploaded_file() function, which is used to move an uploaded file to a new location after verifying its integrity. Here is a rewrite of the previous example using these functions, in order to better illustrate how they can be used:

```
<html>

<head>

</head>
```

```
<body>
<?php

 if (!$_POST['submit'])

 {

 ?>

 <form enctype="multipart/form-data" action="<?=$_SERVER['PHP_SELF']?>"

 method="post">

 Description:

 <br>

 <input type="Text" name="desc" size="40">

 <p>

 File:

 <br>

 <input type="file" name="file">

 <p>

 <input type="submit" name="submit" value="Add Image">

 </form>
```

```
<?

}

else

{
// validate form data

if ($_FILES['file']['size'] == 0) { die("Bad upload!"); }
if ($_FILES['file']['type'] != "image/gif" && $_FILES['file']['type']
 != "image/jpeg" && $_FILES['file']['type'] !=

 "image/pjpeg") { die("Invalid file format!"); }
if (!$_POST['desc']) { die("No description!"); }
if (!is_uploaded_file($_FILES['file']['tmp_name'])) { die("Bad file!");
 }
// open connection to database

 $connection = mysql_connect("localhost", "root", "secret")
 or die ("Unable to connect!");

 mysql_select_db("db1") or die ("Unable to select database!");
// formulate and execute query

 $query = "INSERT INTO gallery (dsc) VALUES ('" . $_POST['desc']. "')";

 $result = mysql_query($query) or die("Error in query: " . mysql_error());

 $id = mysql_insert_id($connection);
// get file type and rename file to recordID.ext

 if ($_FILES['file']['type'] == "image/gif") { $ext = ".gif";
 }
 }

 if ($_FILES['file']['type'] == "image/jpeg") { $ext = ".jpg";

 }

 if ($_FILES['file']['type'] == "image/pjpeg") { $ext = ".jpg";

 }

$newFileName = $id . $ext;

// move file to new location

 move_uploaded_file($_FILES['file']['tmp_name'], "/mydatadir/" .

 $newFileName) or die("Could not move file!");

// update database with new file name

 $query = "UPDATE gallery SET filename = '$newFileName' WHERE id = '$id'";

 $result = mysql_query($query) or die("Error in query: " . mysql_error());

}

 ?>

 </body>

 </html>
```

Diving Into Directories

Thus far, most of the examples you' ve seen have dealt with individual files. However, you often find yourself faced with the task of iterating over one or more directories and processing the file list within each. In order to meet this requirement, PHP offers a comprehensive set of directory manipulation functions, which allow developers to read and parse an entire directory listing.

In order to demonstrate, consider the following simple example, which lists all the files in the directory "/bin":

```
<?php
```

```
  <?php
// initialize counter

  $count = 0;
// set directory name

  $dir = "/bin";
// open directory and parse file list

  if (is_dir($dir))

  {

  if ($dh = opendir($dir))

  {

  // iterate over file list

  // print filenames

  while (($filename = readdir($dh)) !== false)

  {

  if (($filename != ".") && ($filename != ".."))

  {

  $count ++;

  echo $dir . "/" . $filename . "\n";

  }
```

```
 }

 }

 // close directory

 closedir($dh);

 }

 }
echo "-- $count FILES FOUND --";
?>
```

You can combine the script above with the getcwd() function discussed earlier to obtain a list of all the files in the current working directory at any time - I' ll leave that to you to try out for yourself.

A Pattern Emerges

Want to list just the files matching a specific pattern? Use the neat little fnmatch() function, new in PHP 4.3, which matches strings against wildcard patterns. Here' s a quick example:

```
 <?php
// set directory name

 $dir = "/bin";
// set pattern

 $pattern = "e*";
// open directory and parse file list

 if (is_dir($dir))

 {
```

```
 if ($dh = opendir($dir))

 {

 // iterate over file list

 while (($filename = readdir($dh)) !== false)

 {

 // if filename matches search pattern, print it

 if (fnmatch($pattern, $filename))

 {

 echo $dir . "/" . $filename . "\n";

 }

 }

 // close directory

 closedir($dh);

 }

 }
?>
```

Here' s an example of the output:

Here' s an example of the output:

```
/bin/ed
/bin/egrep
/bin/echo
/bin/env
/bin/ex
```

The * wildcard matches one or more characters; if this is not what you want, you can also use the ? wildcard to match a single character.

An alternative to using fnmatch() with the opendir() and readdir() functions is the glob() function, also new to PHP 4.3 - this function searches the current directory for files matching the specified pattern and returns them as an array. Consider the following rewrite of the example above, which demonstrates:

```
 <?php

// set directory name



 $dir = "/bin";

// set pattern



 $pattern = "e*";

// change to named directory



 chdir($dir);

// find files matching pattern



 $files = glob($pattern);

// iterate over files array and print filenames



 foreach ($files as $f)



 {



 echo $dir . "/" . $f . "\r\n";



 }

?>
```

Since glob() looks in the current directory for files, remember to always
chdir() to the correct directory before executing glob().

Purging The Dead

Not only does PHP let you read directory contents, it also allows you to create and delete directories with the mkdir() and rmdir() functions respectively. Consider the following example, which creates a directory,

```
<?php
// create directory

mkdir("/tmp/stuff") or die ("Could not make directory");
?>
```

and its mirror image, which removes the newly-created directory.

```
<?php
// erase directory

rmdir("/tmp/stuff") or die ("Could not erase directory");
?>
```

As with the other filesystem manipulation functions, these functions too will fail if the user the Web server is running as lacks sufficient privileges to create and delete directories on the disk. Directories created with mkdir() will be owned by the process running the Web server.

It' s interesting also to note that the rmdir() function operates only if the directory to be deleted is completely empty. Try running it on a directory containing existing files, as below,

```
<?php
// create directory

mkdir("/tmp/stuff") or die ("Could not make directory");
// create sub-directory

mkdir("/tmp/stuff/personal") or die ("Could not make directory");
```

```
  mkdir("/tmp/stuff/personal") or die ("Could not make directory");

// remove parent directory


  rmdir("/tmp/stuff") or die ("Could not remove directory");

?>
```

and you' ll be rewarded with the following error:

Warning: mkdir(): File exists in /home/web/rmdir.php on line 4 Could not remove directory

Since it' s unlikely that you' ll find empty directories just waiting for your rmdir() in the real world, you' ll normally need to empty the directory manually prior to calling rmdir() on it. The following example demonstrates, by combining the unlink() function discussed previously with some recursive logic to create a function designed specifically to erase the contents of a directory (and its sub-directories) in one fell swoop:

```
  <?php

// recurses into a directory and empties it


  // call it like this: purge("/dir/")


  // remember the trailing slash!


  function purge($dir)


  {


  $handle = opendir($dir);


  while (false !== ($file = readdir($handle)))


  {


  if ($file != "." && $file != "..")


  {
    if (is_dir($dir.$file))

    {

    purge ($dir.$file."/");

    rmdir($dir.$file);

    }

    else

    {

    unlink($dir.$file);

    }

    }

    }

    closedir($handle);

    }
   ?>
```

Size Does Matter

If you' re looking for information on the total size of a partition or mount point, PHP offers the relatively-new disk_total_space() and
disk_free_space() functions, which return the total available space and total free space, in bytes, on a particular partition. Consider the following example, which demonstrates:

```
 <?php
// set partition

 $fs = "/";
// display available and used space

 echo "Total available space: " . round(disk_total_space($fs) / (1024*1024))
 . " MB\r\n"; echo "Total free space: " . round(disk_free_space($fs)
 / (1024*1024)) . " MB\r\n";
// calculate used space

 $disk_used_space = round((disk_total_space($fs) - disk_free_space($fs)) / (1024*1024));
 echo "Total used space: " . $disk_used_space . " MB\r\n";
// calculate % used space

 echo "% used space: " . round((disk_total_space($fs) -

 disk_free_space($fs)) / disk_total_space($fs) * 100) . " %";
?>
```

Here' s an example of what the output might look like:

Total available space: 7906 MB
Total free space: 4344 MB
Total used space: 3562 MB
% used space: 45 %

In Process

Just as PHP offers the fopen() and fclose() functions to open and close file handles, there' s also the popen() and pclose() functions, which can be used to open uni-directional handles to processes. Once a process handle has been created, data can be read from it or written to it using the standard fgets(), fputs(), fread() and fwrite() file functions.

Consider the following example, which demonstrates by opening a pipe to the "cat" command:

```
 <?php
```

```
// open handle to process

  $ph = popen("/bin/cat /etc/passwd", "r");
// read process output into variable

  $contents = fread($ph, 2096);
// display contents

  echo $contents;
// close process

  pclose($ph);
?>
```

As you can see, opening a pipe to a process and reading from it is very similar to opening and reading a file. As with files, the first step is to obtain a handle to the process with popen() - this handle serves as the foundation for all future communication. Once a handle has been obtained, data can be read from, or written to, the handle using the file input/output functions you' re already familiar with. The handle can be closed at any time with the pclose() function.

If you need bi-directional communication, PHP 4.3 also offers the new
proc_open() and proc_close() functions, which offers a greater degree of control over process communication.

Disk Full

And that' s about all I have. Over the course of the last few pages, I took you ever deeper into the waters of PHP' s filesystem API, demonstrating a number of its more arcane features and functions. I showed you how to strip out program code from a file while reading it, how to parse comma-separated data from a file into PHP structures, and how to read configuration variables into a PHP associative array.

Next, I introduced you to PHP' s file copy, move and delete functions, and demonstrated them in the context of a file upload application. I also showed you how to read and display the contents of a directory, and how to recursively iterate through a series of nested directories. Finally, I wrapped things up with a brief look at how to obtain disk usage reports for a mount point or partition, and explained how you could read data from processes just as you do with files.

While this tutorial did cover many of the common uses of PHP' s file and directory manipulation functions, it is by no means exhaustive. There are many more things you can do with PHP' s file functions - and you can get some great ideas (and learn a number of interesting things as well) by perusing the PHP manual pages for these functions, at the links below:

PHP' s file manipulation functions, at http://www.php.net/manual/en/ref.filesystem.php

PHP' s directory manipulation functions, at http://www.php.net/manual/en/ref.dir.php

And while you busy yourself with those links, I' m off on a short break. See you soon!

Note: Examples are illustrative only, and are not meant for a production environment. Melonfire provides no