WITH PHP WITH PHP WITH PHP WITH PHP WITH PHP WITH PHP WITH PHP
LOGGING
WITH PHP
WITH PHP WITH PHP WITH PHP WITH PHP WITH PHP WITH PHP WITH PHP

By icarus

Table of Contents

Timberrrrrrrr!

I have to warn you not to be misled by the title of this article. It isn't really about cutting down trees with PHP.

I know that PHP is the Swiss Army knife of Web scripting languages, but I don't think its developers have come up with a way to have it reduce our tree cover (at least not yet). Rather, this article is about a different sort of logging − the sort which involves writing messages to files or devices on a regular basis, and using these messages to "do something useful", be it generating a report or building an audit trail for activity tracking.

Over the next few pages, I'm going to take a quick look at the various mechanisms available to log script activity via PHP, demonstrating both built−in functions and add−on classes that allow you to add logging functionality to your applications. I'll also spend some time on the PHP functions and configuration options related to logging, demonstrating how, for example, PHP errors can be sent to the Web server error log or emailed to an administrator when they are over a particular priority level.

My primary audience for this tutorial is the novice to intermediate PHP user − if you're an advanced user, you probably already know enough about logging with PHP, and might prefer to skip the rest of this article in favour of more nutritious fare. Everyone else − keep reading!

The Bare Necessities

Logging data to a file in PHP can be as simple or as complex as you want to make it. Break it down, though, and it all comes down to these three simple lines of code:

```
<?php

// open file
$fd = fopen($filename, "a");

// write string
fwrite($fd, $str . "\n");

// close file
fclose($fd);

?>
```

Fundamentally, logging data to a file consists of three steps:

1. Open the target file (or create it if it doesn't already exist);

2. Append your data to the end of the file;

3. Close the file.

You can encapsulate this as a function,

```
<?php

function logToFile($filename, $msg)
{
// open file
$fd = fopen($filename, "a");

// write string
fwrite($fd, $msg . "\n");

// close file
fclose($fd);
}

?>
```

and then use it liberally within your code as and when required.

```
<?php

function logToFile($filename, $msg)
{
// open file
$fd = fopen($filename, "a");

// write string
fwrite($fd, $msg . "\n");

// close file
fclose($fd);
}


$v = "Mary had a little lamb";
if (!is_numeric($v)) { logToFile("my.log", "Non−numeric
variable
encountered"); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { logToFile("my.log", "No fish
available"); }

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn)
{
logToFile("my.log", "Could not connect to database");
die("Could not connect to database");
}

?>
```

You can make the function a little more professional by having it automatically append the date and time of the log message to the log file as well:

```
<?php

function logToFile($filename, $msg)
{
// open file
$fd = fopen($filename, "a");

// append date/time to message
$str = "[" . date("Y/m/d h:i:s", mktime()) . "] " . $msg;

// write string
fwrite($fd, $str . "\n");

// close file
fclose($fd);
}

?>
```

Here's an example of the output:

```
[2002/11/25 06:02:42] Non−numeric variable encountered
[2002/11/25
06:02:42] No fish available [2002/11/25 06:02:43] Could not
connect to
database
```

Turning The Tables

While the most common destination for log data is a text file, it's quite possible that you might want to send your log messages elsewhere as well. A frequently−used alternative to a text file, especially when the number of writes isn't too high, is an SQL database, where log messages are appended as records to the end of a table.

In order to illustrate this, consider the following example of an SQL table used to store log messages:

```
CREATE TABLE `log`
(
`date` TIMESTAMP NOT NULL,
`type` TINYINT NOT NULL,
`msg` VARCHAR(255) NOT NULL
);
```

And here's the rewritten logToDB() function, this time built around PHP's MySQL database functions and the table above:

```
function logToDB($msg, $type)
{

// open connection to database
$connection = mysql_connect("localhost", "joe", "pass") or die
("Unable to connect!");
mysql_select_db("logdb") or die ("Unable to select
database!");

// formulate and execute query
$query = "INSERT INTO log (date, type, msg) VALUES(NOW(),
'$type', '$msg')";
mysql_query($query) or die ("Error in query: $query. " .
mysql_error());

// close connection
mysql_close($connection);
}

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { logToDB("No fish available", 14);
}
```

If you don't estimate seeing a very large number of writes, using a database to store logs can offer significant advantages over a regular file, purely from the point of view of easier retrieval and sorting. Storing log messages in this manner makes it possible to easily retrieve ordered subsets of the log data, either by date or message type.

Another option might be to send log messages to a specified email address via PHP's mail() function − for example, alerting a sysop whenever a script encounters errors. Here's an example:

```
function logToMail($msg, $address)
{

// append date/time to message
$str = "[" . date("Y/m/d h:i:s", mktime()) . "] " . $msg;

mail($address, "Log message", $str);

}

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn) { logToMail("Could not connect to database",
"webmaster@my.domain.name.com"); }
```

Turning Up The Heat

If the thought of rolling your own code doesn't appeal to you, you can also use PHP's built−in error_log() function, which logs data both to a file and elsewhere. The error_log() function needs a minimum of two arguments: the error message to be logged, and an integer indicating where the message should be sent. There are three possible integer values in PHP 4.x:

0 − send the message to the system logger ("syslogd" on *NIX, and the Event Log on Windows NT);

1 − send the message to the specified email address;

3 − send the message to the specified file;

Here's a trivial example which demonstrates how this works:

```
<?php
// set a variable
$temp = 101.6;

// test it and log an error
// this will only work if
// you have "log_errors" and "error_log" set in your php.ini
file if
($temp > 98.6) {
error_log("Body temperature above normal.", 0);
}

?>
```

Now, if you look at the system log file after running the script, you'll see something like this:

```
[28−Feb−2002 15:50:49] Body temperature above normal.
```

You can also write the error message to a file,

```
<?php

// set a variable
$temp = 101.6;

// test it and log an error
if ($temp > 98.6)
{
```

```
error_log("Body temperature above normal.", 3, "a.out");
}

?>
```

or send it out as email.

```
<?php

// set a variable
$temp = 101.6;

// test it and log an error
if ($temp > 98.6)
{
error_log("Body temperature above normal.", 1,
"administrator@this.body.com"); }

?>
```

It's possible to combine this error logging facility with a custom error handler to ensure that all script errors get logged to a file. Here's an example which demonstrates this:

```
<?php

// custom handler
function eh($type, $msg, $file, $line)
{
// log all errors
error_log("$msg (error type $type)", 0);

// if fatal error, die()
if ($type == E_USER_ERROR)
{
die($msg);
}
}


// report all errors
error_reporting(E_ALL);

// define custom handler
set_error_handler("eh");
```

```
// let's now write some bad code

// this will trigger a warning, since the file doesn't exist
include("common.php"); ?>
```

And here's the output that gets logged to the system log file:

```
[28−Feb−2002 16:15:06] Failed opening 'common.php' for
inclusion
(include_path='.;') (error type 2)
```

Biting Into A PEAR

As if all that wasn't enough, PHP offers one more alternative − a special Log class that comes courtesy of PEAR, the PHP Extension and Application Repository (http://pear.php.net). In case you didn't know, PEAR is an online repository of free PHP software, including classes and modules for everything from data archiving to XML parsing. This Log class is maintained by Jon Parise, and the latest version can be downloaded from http://pear.php.net/package−info.php?package=Log

Like the error_log() function, the PEAR Log class allows logging to a variety of different destinations − system logger, text file, email address, database and even MCAL. Here's an example demonstrating basic usage:

```
<?php

// include class
include("Log.php");

// create Log object
$l = &Log::singleton("file", "my.log");

// test it
$v = "Mary had a little lamb";
if (!is_numeric($v)) { $l−>log("Non−numeric variable
encountered",
PEAR_LOG_WARNING); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { $l−>log("No fish available",
PEAR_LOG_ERR);
}

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn) { $l−>log("Could not connect to database",
PEAR_LOG_CRIT);
}

?>
```

The first step here is to include() the Log class:

```
// include class
include("Log.php");
```

Once that's done, a new Log object can be created. This object constructor requires, as its first parameter, a string indicating the kind of log to open − current valid values are 'console', 'syslog', 'sql', 'file', and 'mcal'. Depending on the log type, additional data may be provided − the name of the file to use in case of a file logger, for example.

```
// create Log object
$l = &Log::singleton("file", "my.log");
```

The Log class internally defines eight priority levels for logging − here's the list, culled directly from the source code of the class:

```
$priorities = array(
PEAR_LOG_EMERG => 'emergency',
PEAR_LOG_ALERT => 'alert',
PEAR_LOG_CRIT => 'critical',
PEAR_LOG_ERR => 'error',
PEAR_LOG_WARNING => 'warning',
PEAR_LOG_NOTICE => 'notice',
PEAR_LOG_INFO => 'info',
PEAR_LOG_DEBUG => 'debug'
);
```

Each log message that you send to the logger can be flagged with a specific priority, and the logger can be set up to log only those messages matching or exceeding a specific priority level (by default, all messages are logged). This can be clearly seen from the test code below, in which each message to the logger includes a priority level:

```
$v = "Mary had a little lamb";
if (!is_numeric($v)) { $l->log("Non-numeric variable
encountered",
PEAR_LOG_WARNING); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { $l->log("No fish available",
PEAR_LOG_ERR);
}

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn) { $l->log("Could not connect to database",
PEAR_LOG_CRIT);
}
```

Here's what the log file looks like:

```
Nov 26 06:47:15 [warning] Non−numeric variable encountered

Nov 26 06:47:15 [error] No fish available

Nov 26 06:47:17 [critical] Could not connect to database
```

If you'd like to tell the logger to only log messages above a certain priority level − for example, critical and above − this priority level should be specified as the fifth argument to the object constructor. Consider the following revision of the previous example:

```
<?php

// include class
include("Log.php");

// create Log object
$l = &Log::singleton("file", "my.log", NULL, array(),
PEAR_LOG_ERR);

// test it
$v = "Mary had a little lamb";
if (!is_numeric($v)) { $l−>log("Non−numeric variable
encountered",
PEAR_LOG_WARNING); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { $l−>log("No fish available",
PEAR_LOG_ERR);
}

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn) { $l−>log("Could not connect to database",
PEAR_LOG_CRIT);
}

?>
```

In this case, only messages flagged as PEAR_LOG_ERR and above will be written to the specified log file.

Destination Unknown

The Log class also supports sending log messages to the console, the system logger, an SQL database or a user−specified email address. Take a look:

```
<?php

// include class
include("Log.php");

// create Log object
$l = &Log::singleton("console");

// test it
$v = "Mary had a little lamb";
if (!is_numeric($v)) { $l−>log("Non−numeric variable
encountered",
PEAR_LOG_WARNING); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { $l−>log("No fish available",
PEAR_LOG_ERR);
}

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn) { $l−>log("Could not connect to database",
PEAR_LOG_CRIT);
}

?>
```

In this case, log messages are directed to the text console (if you're running this script via a Web server, the console is the browser window). Here's the output:

```
Nov 26 22:05:25 [warning] Non−numeric variable encountered
Nov 26 22:05:25 [error] No fish available
Nov 26 22:05:27 [critical] Could not connect to database
```

Messages can also be sent to an SQL database,

```
<?php

// include class
```

```
include("Log.php");

// create Log object
// second argument is table name
// fourth argument is PHP::DB compatible DSN for database
access $l =
&Log::singleton("sql", "log_table", "", array('dsn' =>
'mysql://joe:pass@localhost/test'));

// test it
$v = "Mary had a little lamb";
if (!is_numeric($v)) { $l->log("Non-numeric variable
encountered",
PEAR_LOG_WARNING); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { $l->log("No fish available",
PEAR_LOG_ERR);
}

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn) { $l->log("Could not connect to database",
PEAR_LOG_CRIT);
}

?>
```

which can then be queried to retrieve subsets of the log messages, sorted by date or priority level.

```
mysql> SELECT * FROM log_table WHERE priority >= 4;
+----------------+-------+----------+-----------------------------------+
| logtime | ident | priority | message |
+----------------+-------+----------+-----------------------------------+
| 20021126074936 | | 4 | Non-numeric variable encountered |
| 20021126074936 | | 4 | Non-numeric variable encountered |
| 20021126074937 | | 4 | Non-numeric variable encountered |
+----------------+-------+----------+-----------------------------------+
3 rows in set (0.05 sec)
```

Finally, log messages can also be directed to a specified email address, as in the following example:

```
<?php

// include class
include("Log.php");
```

```
// create Log object
$l = &Log::singleton("mail", "admin@host.com");

// test it
$v = "Mary had a little lamb";
if (!is_numeric($v)) { $l->log("Non-numeric variable
encountered",
PEAR_LOG_WARNING); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { $l->log("No fish available",
PEAR_LOG_ERR);
}

$conn = @mysql_connect("localhost", "joe", "pass");
if (!$conn) { $l->log("Could not connect to database",
PEAR_LOG_CRIT);
}

?>
```

You can customize the mail message by adding an array containing a custom From: and Subject: line to the object constructor, as below:

```
<?php

// include class
include("Log.php");

// create Log object
$l = &Log::singleton("mail", "admin@host.com", NULL,
array('from' => 'L.
Ogger', 'subject' => 'Log message'));

// test it
$v = "Mary had a little lamb";
if (!is_numeric($v)) { $l->log("Non-numeric variable
encountered",
PEAR_LOG_WARNING); }

$a = array("chocolate", "strawberry", "peach");
if (!in_array('fish', $a)) { $l->log("No fish available",
PEAR_LOG_ERR);
}

$conn = @mysql_connect("localhost", "joe", "pass");
```

```
if (!$conn) { $l->log("Could not connect to database",
PEAR_LOG_CRIT);
}

?>
```

Artificial Intelligence

You can also configure PHP to automatically log all script errors to a specific file via the special "log_errors" configuration directive in "php.ini". This directive, when set to true, logs all PHP errors to either a user−specified log file (the value of this file must be specified in the "error_log" configuration directive, also set via "php.ini"), or to the Web server error log if no log file is specified.

Consider the following example, which demonstrates:

```
<?php

// turn on automatic error logging
ini_set('log_errors', true);

// include non−existent file
include("non.existent.file.php");
?>
```

In this case, I'm manually instantiating an error by attempting to include a file which does not exist. Obviously, PHP will barf and display an error screen containing the following:

```
Warning: Failed opening 'non.existent.file.php' for inclusion
(include_path='.;/usr/local/php/includes') in error.php on
line 10
```

The same error also appears in the server's error log:

```
[Tue Nov 26 12:49:31 2002] [error] PHP Warning: Failed opening
'non.existent.file.php' for inclusion
(include_path='.;/usr/local/php/includes') in error.php on
line 10
```

You can turn off PHP error display, and only have the error message appear in the log file, by setting the "display_errors" variable (also accessible via "php.ini") to false. Consider the following variant of the previous example, which demonstrates:

```
<?php

// turn on automatic error logging
ini_set('log_errors', true);
```

```
// turn off error display
ini_set('display_errors', false);

// include non-existent file
include("non.existent.file.php");
?>
```

In this case, though an error takes place, it is never displayed to the user, but merely gets logged to the server's error log.

On *NIX systems, setting the "error_log" variable to the special value "syslog" logs all errors via the standard "syslog" daemon.

Big Brother Is Watching

Finally, let's wrap things up with a couple of examples that show how the various techniques demonstrated above can be used to build logs and audit trails for a Web application.

In the first example, every time a Web page is displayed, a log entry is made in an "access.log" file. This log entry is a comma−separated list of values containing the URL requested, the client browser identification string, and a timestamp.

```
<?php
// create log string
$str = date("Y/m/d h:i:s", mktime()) . "," .
$_SERVER['REQUEST_URI'] .
"," . $_SERVER['HTTP_USER_AGENT'] . "\n";

// write to file
error_log($str, 3, "access.log");

// rest of page here
?>
```

Here's a snippet from the access log:

```
2002/11/26 09:07:38,/alpha.php,Mozilla/4.0 (compatible; MSIE
5.0;
Windows 95) 2002/11/26 09:07:38,/alpha.php,Mozilla/4.0
(compatible; MSIE
5.0; Windows 95) 2002/11/26
09:10:31,/home/hello.php,Mozilla/4.0
(compatible; MSIE 5.0; Windows 95) 2002/11/26
09:17:38,/index.php,Mozilla/4.0 (compatible; MSIE 5.0; Windows
95)
2002/11/26 09:17:38,/base/index.php,Mozilla/4.0 (compatible;
MSIE 5.0;
Windows 95)
```

Since this data is in a structured format, it can easily be analyzed and a "hit count" created for each URL in the file. This next script does exactly that:

```
<?php

// create array to hold unique URLs
$urlStats = array();
```

```
// read access log
$lines = file("access.log");

// iterate through access log
foreach ($lines as $l)
{
$data = explode(",", $l);
$ts = $data[0];
$url = $data[1];
$agent = $data[2];

// check to see if URL exists in array
// if it does, increment incidence count
// if it does not, create a new key with incidence count 1
if ($urlStats[$url])
{
$urlStats[$url]++;
}
else
{
$urlStats[$url] = 1;
}
}

// print list of unique URLs with count
print_r($urlStats);
?>
```

This script parses the "access.log" file, and creates a PHP associative array whose keys correspond to the URLs found in the file. The value associated with each key is an integer indicating the number of appearances the URL makes in the file. Once the entire file has been parsed, the $urlStats array contains a list of all the unique URLs in the access log, together with the number of times each has appeared. This data can then be used to generate a report of the most frequently−accessed URLs.

Consider this next example, which provides an API for adding, editing and deleting users to (from) a Web application. Each time the user database is edited, a separate audit() process tracks the change, logging both the nature of the change and information about the user initiating the change. This log data is stored in a separate SQL table, from where it can be retrieved for statistical reporting, user activity monitoring or debugging.

```
<?php
// assume that administrator has logged in to system to
perform
user−administration tasks // admin username is stored in a
session
variable by default // this is useful for audit purposes
```

```
session_start(); $_SESSION['LOGGED_IN_USER'] = "john";

// add a new user
function addUser($user, $pass, $perms)
{

// open connection to database
$connection = mysql_connect("localhost", "joe", "pass") or die
("Unable to connect!");
mysql_select_db("myapp") or die ("Unable to select
database!");

// formulate and execute query
$query = "INSERT INTO users (user, pass, perms)
VALUES('$user',
'$pass', '$perms')";
mysql_query($query) or die ("Error in query: $query. " .
mysql_error());

// log activity to audit database
audit("ADD_USER", $_SESSION['LOGGED_IN_USER'],
"$user:$pass:$perms", addslashes($query));

// close connection
mysql_close($connection);

}

// edit an existing user
function updateUser($user, $pass, $perms)
{

$connection = mysql_connect("localhost", "joe", "pass") or die
("Unable to connect!");
mysql_select_db("myapp") or die ("Unable to select
database!");

// formulate and execute query
$query = "UPDATE users SET pass = '$pass', perms = '$perms'
WHERE user = '$user'";
mysql_query($query) or die ("Error in query: $query. " .
mysql_error());

// log activity to audit database
audit("UPDATE_USER", $_SESSION['LOGGED_IN_USER'],
"$user:$pass:$perms", addslashes($query));

// close connection
mysql_close($connection);
}

// delete an existing user
function deleteUser($user)
{

$connection = mysql_connect("localhost", "joe", "pass") or die
("Unable to connect!");
mysql_select_db("myapp") or die ("Unable to select
database!");

// formulate and execute query
$query = "DELETE FROM users WHERE user = '$user'";
mysql_query($query) or die ("Error in query: $query. " .
mysql_error());

// log activity to audit database
audit("DELETE_USER", $_SESSION['LOGGED_IN_USER'], "$user",
addslashes($query));

// close connection
mysql_close($connection);

}

// generic audit function
// logs all activity to a database
function audit($op, $owner, $args, $msg)
{
$connection = mysql_connect("localhost", "root", "pass") or
die
("Unable to connect!");
mysql_select_db("trails") or die ("Unable to select
database!");

// formulate and execute query
$query = "INSERT INTO audit (timestamp, op, owner, args, msg)
VALUES (NOW(), '$op', '$owner', '$args', '$msg')";
mysql_query($query) or die ("Error in query: $query. " .
mysql_error());

}


addUser("joe", "joe", 3);
addUser("sarahh", "bsdfg49", 1);
updateUser("joe", "joe", 4);
deleteUser("sarahh");
```

```
addUser("sarah", "bsdfg49", 1);
?>
```

Here's a snippet from the audit table:

```
+---------------------+-------------+-------+
| timestamp | op | owner |
+---------------------+-------------+-------+
| 2002-11-26 08:28:05 | UPDATE_USER | john |
| 2002-11-26 08:28:05 | DELETE_USER | john |
| 2002-11-26 08:28:05 | ADD_USER | john |
| 2002-11-26 08:33:14 | ADD_USER | joe |
+---------------------+-------------+-------+
```

This audit table can then be queried to obtain detailed information on the activities performed by the various users, sorted by time or type of activity. For example,

```
mysql> SELECT timestamp, op, args FROM trails WHERE timestamp
>=
mysql> 2002-11-26
AND owner = 'joe';
+---------------------+-------------+------------------+
| timestamp | op | args |
+---------------------+-------------+------------------+
| 2002-11-26 08:33:29 | ADD_USER | joe:joe:3 |
| 2002-11-26 08:33:29 | ADD_USER | sarahh:bsdfg49:1 |
| 2002-11-26 08:33:29 | UPDATE_USER | joe:joe:4 |
| 2002-11-26 08:33:29 | DELETE_USER | sarahh |
| 2002-11-26 08:33:29 | ADD_USER | sarah:bsdfg49:1 |
+---------------------+-------------+------------------+
```

This is a somewhat trivial example, but it serves to demonstrate the concept of logging activity and using those logs to build an audit trail. While you can make this as complex as you want, tracking everything from user clicks to form input in order to gain a better understanding of how users navigate through and use your application, remember that every addition to the log affects the overall performance of your application; log too much data and your application will suffocate and die.

Closing Time

And that's about it for the moment. In this article, I offered you a brief overview of logging in PHP, demonstrating how you can use built−in functions, off−the−shelf libraries or your own code to easily and efficiently create logs of the activity taking place in your application. I demonstrated logging to a file, to a database, to an email address and to the console, and wrapped things up with a couple of simple examples that demonstrated how logs could be built and analyzed in a real−world application.

I hope you enjoyed reading this article as much as I enjoyed writing it. If you'd like to know more about the topics discussed in the previous pages, here are some links you should bookmark:

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

PHP mail functions, at http://www.php.net/manual/en/ref.mail.php

PHP MySQL functions, at http://www.php.net/manual/en/ref.mysql.php

PHP error logging functions, at http://www.php.net/manual/en/ref.errorfunc.php

The PHP Log class, at http://pear.php.net/package−info.php?package=Log

PEAR, at http://pear.php.net

Till next time...stay healthy!

Note: All examples in this article have been tested on Linux/i586 with PHP 4.2.3. 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!