Output Buffering with PHP

By Harish Kamath

Table of Contents

Off The Beaten Track

You don't have to look at the numbers any more to gauge PHP's popularity − a quick scan of the manual pages provides more than enough evidence that developers all over the world are taking the language to their hearts. As the manual's index demonstrates, a huge number of extensions are available for the language, many created and supported by independent programmers. Individually, each performs a specialized function; collectively, they have turned PHP into one of the most full−featured Web programming languages available today.

The bad thing about all this choice, though, is that it gets harder to separate the wheat from the chaff. And that's where today's article comes in − over the next few pages, I'm going to introduce you to some little−known, but nevertheless very useful functions, which offer you never−before−seen control over the output generated by your PHP scripts, and provide some interesting twists on the standard way of writing PHP code. Keep reading!

The Matrix Awaits

Output control, you say? What's that? And more importantly, why do I care?

As the name suggests, PHP's output control functions provide a way for you, the developer, to exercise some degree of control over how the output generated by a particular PHP script is handled.

Why do you need this? Perhaps the best way to demonstrate is with an example.

```
<?php

// send a header
header ("Pragma: no−cache");

// send some output
print "Welcome to the Matrix, Neo";

// send another header
// this will generate an error
header ("Content−Type: text/html");

?>
```

Here's what you'll see when you browse to the script:

```
Welcome to the Matrix, Neo
Warning: Cannot add header information − headers already sent
by (output
started at /usr/local/apache/htdocs/x1.php:7) in
/usr/local/apache/htdocs/x1.php on line 11
```

In this case, PHP barfs because I've attempted to send a HTTP header to the browser after the script has generated a line of output. If you know anything about HTTP, you know that this is a big no−no − as the PHP manual succinctly puts it, "...you can not normally send headers to the browser after data has already been sent...".

So where does that leave you? Well, either you make it a point to ensure that all sensitive code − like the code that generates HTTP headers − always appears right at the top of your scripts...or you get creative with the output control functions available to you in PHP.

```
<?php

// start buffering the output
```

```
ob_start();

// send a header
header ("Pragma: no−cache");

// print some output
print "Welcome to the Matrix, Neo";

// send another header
// in this case, since the output is being stored in a buffer
// and not
being printed, no error will be generated header
("Content−Type:
text/html");

// print the contents of the buffer
ob_end_flush();

?>
```

Now, if you run this script, you'll see the output

```
Welcome to the Matrix, Neo
```

with no nasty error messages spoiling the view.

How did this happen? Well, it's fairly simple. In this case, rather than having PHP send output directly to the standard output device (the browser) as the script gets executed, I've chosen to define a special output buffer which stores all the output generated by the script during its lifetime. When I do this, the output of the script is never seen by the user unless I explicitly make the contents of this buffer visible via a call to PHP's output control API (discussed in detail on the next page).

When you compare the two examples above, the advantage of using this technique should immediately become clear. By using a memory buffer to manage how the output of my script appears, I can do things which would otherwise be difficult or tedious − including sending HTTP headers to the browser *after* a script has started generating output, or (as you will see) displaying different Web pages on the basis of conditional tests within my script.

Start Me Up

Let's take a closer look at the output control functions available in PHP. Everything begins with the ob_start() function, which actually initializes an output buffer for use within the script.

```
<?php

// start buffering the output
ob_start();

?>
```

The ob_start() function doesn't really need much explanation. Call it, and an output buffer opens up, ready to intercept and suck in whatever output is generated by the script.

As you will see in subsequent examples, it's possible to use this output buffer in combination with a user−defined handler to reprocess and modify the output of the script as it enters the buffer.

Once a buffer has been defined, the script proceeds to execute as usual. When you've decided that it's time to dispay the contents of the buffer to the user, you can simultaneously end output buffering and send the contents of the current buffer to the browser via a call to ob_end_flush().

```
<?php

// print the contents of the buffer
ob_end_flush();

?>
```

Thus, the ob_start() and ob_end_flush() functions act as a wrapper around your script, controlling the display of script output as per your whims and fancies.

Melting Down

Of course, there aren't just two functions available to you − the output control API has a few other cards up its sleeve as well. Consider the following example:

```
<?php

// run at script start
function page_init()
{
// start buffering the output
ob_start();
}

// run at script end
function page_exit()
{
global $error;

// if an error occurred
// erase all output generated so far
// and display an error message
if ($error == 1)
{
ob_end_clean();
print_error_template();
}
// no errors?
// display output
else
{
ob_end_flush();
}
}

// print error page
function print_error_template()
{
echo "<html><head><basefont face=Arial></head><body>An error
occurred. Total system meltdown now in
progress.</body></html>"; }


page_init();

// script code goes here
echo "This script is humming like a well−oiled machine";
```

```
// if an error occurs
// set the global $error variable to 1
// uncomment the next line to see what happens if no error
occurs $error
= 1;

page_exit();

?>
```

In this case, depending on whether or not a particular error condition is met, PHP will either display the contents of the buffer, or wipe it clean via the ob_end_clean() function.

Today's Forecast

As you saw on the previous page, you can display the contents of the current buffer at any time via a call to ob_end_flush(). If, however, you don't want to display the buffered output to the user, but instead want to do something else with it − write it to a file, for example − you have a couple of options.

The first − and simplest − involves using ob_get_contents() to extract the current contents of the output buffer to a PHP variable, and then processing that variable to get the results you desire. Here's a quick example:

```
<?php

// start buffering the output
ob_start();

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

// 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

// now decide what to do with the buffered output
if ($output == "www")
{
// either print the contents of the buffer...
ob_end_flush();
}
else
{
// ... or write it to a file
$data = ob_get_contents();
$fp = fopen ("weather.html", "w");
fwrite($fp, $data);
fclose($fp);
ob_end_clean();
}

?>
```

In this case, the ob_get_contents() function is used to retrieve the current contents of the output buffer, and write it to a file.

Alternatively, you might want to use a callback function that is invoked every time output is caught by the buffer. The name of this user−defined callback function must be specified as an argument during the initial call to ob_start(), and the function itself must be constructed to accept the contents of the buffer as function argument.

If that sounded way too complicated, the next example, which rewrites the one above to use this technique, should make it clearer.

```
<?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();
?>
```

In this case, 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 tp the function to decide what to do with the buffer − in this case, I've used the $output variable to decide whether or not to write it to a file. You can just as easily write a function to process it in some other way − for example, run a search−and−replace operation on the buffer, write it to a database, or email it to a specific address.

Making It Simpler

In case you're wondering, it isn't really necessary for you to call ob_start() and ob_end_flush() at the beginning and end of every script if you don't want to. Once you make the decision to use output buffering in your scripts, you'll find it quite annoying to include two extra lines of code at the top and bottom of each and every script in your application.

Well, there's a simple solution − you can have PHP automatically transfer all script output to an output buffer by explicitly switching buffering on for *all* scripts , either via a configuration directive in the "php.ini" configuration file, or via a call to ini_set().

```
output_buffering = On
```

This is sometimes referred to as implicit buffering, since it's handled automatically by the PHP engine.

If you'd like to restrict the output buffer to a specific size, you can do so by specifying the size as a numeric value to the configuration drirective above.

It's important to keep in mind, though, that while implicit buffering is a fairly cool thing, there are a couple of caveats which come with it. First, implicit output buffering adds to the work PHP has to do when processing a script, which could end up degrading performance. Second, since all the output of the script is stored in a separate buffer, debugging script errors becomes far more complicated.

The Real World

So that's the theory − now how about tossing it into the real world and seeing how it works? Take a look at the following PHP script, which demonstrates how output buffering, together with a series of judicious calls to PHP's error−handling functions, can be used to provide a simple and efficient wrapper for catching and handling errors in a script.

```
<?php
// use an output buffer to store page contents
ob_start();
?>

<html>
<head><basefont face="Arial"></head>
<body>
<h2>News</h2>
<?php

// custom error handler
function e($type, $msg, $file, $line)
{
// read some environment variables
// these can be used to provide some additional debug
information
global $HTTP_HOST, $HTTP_USER_AGENT, $REMOTE_ADDR,
$REQUEST_URI;

// define the log file
$errorLog = "error.log";

// construct the error string
$errorString = "Date: " . date("d−m−Y H:i:s", mktime()) .
"\n";
$errorString .= "Error type: $type\n";
$errorString .= "Error message: $msg\n";
$errorString .= "Script: $file($line)\n";
$errorString .= "Host: $HTTP_HOST\n";
$errorString .= "Client: $HTTP_USER_AGENT\n";
$errorString .= "Client IP: $REMOTE_ADDR\n";
$errorString .= "Request URI: $REQUEST_URI\n\n";

// log the error string to the specified log file
error_log($errorString, 3, $errorLog);

// discard current buffer contents
// and turn off output buffering
ob_end_clean();
```

```
// display error page
echo "<html><head><basefont face=Arial></head><body>";

echo "<h1>Error!</h1>";

echo "We're sorry, but this page could not be displayed
because
of an internal error. The error has been recorded and will be
rectified
as soon as possible. Our apologies for the inconvenience. <p>
<a
href=/>Click here to go back to the main menu.</a>";

echo "</body></html>";

// exit
exit();
}

// report warnings and fatal errors
error_reporting(E_ERROR | E_WARNING);

// define a custom handler
set_error_handler("e");

// attempt a MySQL connection
$connection = @mysql_connect("localhost", "john", "doe");
mysql_select_db("content");

// generate and execute query
$query = "SELECT * FROM news ORDER BY timestamp DESC";
$result = mysql_query($query, $connection);

// if resultset exists
if (mysql_num_rows($result) > 0)
{
?>

<ul>

<?php
// iterate through query results
// print data
while($row = mysql_fetch_object($result))
{
?>
<li><b><?=$row->slug?></b>
<br>
<font size=−1><i><?=$row−>timestamp?></i></font>
<p>
<font size=−1><?php echo substr($row−>content, 0, 150); ?>...
<a
href=story.php?id=<?=$row−>id?>>Read more</a></font>
<p>
<?php
}
?>
</ul>
<?php
}
else
{
echo "No stories available at this time";
}

// no errors occured
// print buffer contents
ob_end_flush();
?>

</body>
</html>
```

In this case, the first thing I've done is initialized the output buffer via a call to ob_start() − this ensures that all script output is placed in a buffer, rather than being displayed to the user. This output may be dumped to the standard output device at any time via a call to ob_end_flush().

Now, whenever an error occurs, the custom error handler, cleverly named e(), will first flush the output buffer, then send a custom error template to the browser and terminate script execution. So, even if there was a Web page being constructed on the fly when the error occurred, it will never see the light of day, as it will be discarded in favour of the custom error template. If, on the other hand, the script executes without any errors, the final call to ob_end_flush will output the fully−generated HTML page to the browser.

Zip Zap Zoom

With everyone and their grandma now online, Web site are making an effort to provide users with more and better content. However, with bandwidth and download time still issues, it's essential that Web developers ensure this content is made available to the end−user as quickly as possible.

PHP's output control API offers an interesting solution to the problem, making it possible to compress the output sent to the client on−the−fly to reduce download time. Of course, the client needs to support this − but if it does, using this feature can make your Web site more efficient and your users happier.

Here's an example:

```
<?php

ob_start("ob_gzhandler");

?>
<html>
<head></head>
<body>

<!-- content goes here -->
<!-- content goes here -->

</html>
</body>

<?php

ob_end_flush();

?>
```

In this case, I've specified a callback function for the output buffer. For once, this callback is not a user−defined handler; rather, it is a PHP function named ob_gzhandler() whose sole purpose is to determine whether the client can accept compressed data (via the Content−Encoding header). If it can, ob_gzhandler() compresses the contents of the output buffer and sends it to the client, which then decodes and renders it; if not, the data is sent as is, in uncompressed form.

Endgame

And that's about it. In this article, you learned a little bit about PHP's output control functions, using them to buffer the output of your script until you decide it's safe to display it to the user. You also learned how to erase the contents of the output buffer, retrieve the contents as a string for further processing, and apply user−defined callbacks to the buffer. Finally, this article demonstrated some applications of this technology in a real−world environment, using PHP's output control API to gracefully handle script errors and compress data sent to requesting clients to minimize download time.

More information on the material discussed in this article can be found at:

The PHP manual pages on output buffering, http://www.php.net/manual/en/ref.outcontrol.php

A discussion of HTTP headers, http://www.garshol.priv.no/download/text/http−tut.html

Till next time...stay healthy!

Note: All examples in this article have been tested on Linux/i586 with Apache 1.3.20 and PHP 4.1.1. 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!