

OUTPUT BUFFERING

Practical PHP Programming

One feature we've consistently been asked to cover in depth is output buffering, so this issue **Paul Hudson** is looking at it in some detail.

Output buffering was introduced in the original release PHP 4, yet for one reason or another it has still yet to become commonly used in websites. This is a great shame, because without output buffering, PHP sends the output of your scripts to your web server as soon as it's ready – this might be line by line or code block by code block.

As you can imagine, the need to send lots of little bits of data is incredibly slow, however much more annoying is the fact that you're restricted in the order you can send data. Output buffering solves this problem by enabling you to buffer up your output and send it to output when you're ready to do so, or even to not send it at all, if you so decide.

We had lots of requests for PHP tutorials on particular topics, and this was the most common. Many people didn't seem to understand the basic OB concepts, so I've broken this tutorial down into bite-size chunks to make each point clear by itself before moving on.

OB Advantages

As most people who work with cookies and other HTTP headers will know, it's often quite a pain to order your output properly. In HTTP, you always need to send header data before content data, which means that if you want to set a cookie half-way through a script, you're in trouble. Luckily, output buffering comes to the rescue by letting you 'send' cookies at any point your script – it stores these cookies separately to the HTML data then sends them together at the end, in the correct order. The bulking together of data also provides quite a performance improvement because there's no longer any need to send it a few kilobytes at a time – PHP stores up all the output of your script until you instruct it to send, at which point all data is sent in one chunk.

The most popular advantage to output buffering is that you can compress content before you send it. Due to the fact that HTML is lots of very simple, repeating text elements, and that normal written text on a web site is also very easy to compress,

compressing your pages can make a big dent in the amount of bandwidth your site (and your visitor!) uses. Because compression requires full knowledge of what it is compressing, you need output buffering.

One of the least-used but most powerful advantages is that output buffers are stackable, meaning that you can have several buffers working on top of each other, allowing you to build your output up over multiple buffers.

Performance considerations

If you're not using content compression, output buffering is very unlikely to affect the speed of your web server by any great amount – if anything, it should help it serve pages *faster* because of the optimised data sending. Content compression does take up a little CPU time on both the server and on clients visiting your site, but it's pretty small.

On the up-side, content compression should decrease the amount of bandwidth you use by 40-60%, which means your server will spend less time sending data across the network. The compression level you achieve depends entirely on the kind of content you serve up – if you have lots of pictures, which content compression won't affect, your compression level will be lower; if you're sending lots of XML, which is a naturally repeating format that is very easy to compress, your compression level will be much higher. It's important to remember that only the output of your PHP script will be compressed – images, CSS files *etc* are all served as normal.

Getting started

You can enable output buffering in one of two ways, one of which seems easier at first but is likely to cause hassle in the long term. The 'easy' option is to edit your `php.ini` file to enable output buffering for all scripts – this might sound great, but it will mean your scripts will break on other PHP installations, and also means you have no way to not have output buffering for a script. The second option, which is much smarter, is to use the set of output buffering function calls on a script-by-script basis.

The `ob_start()` function is used to create a new output buffer, and you can immediately start writing to it by printing out content as normal. Once you have a buffer open, there are two ways to close it: `ob_end_flush()` and `ob_end_clean()`, both of which end the buffer, but do so in slightly different ways. The former ends the buffer and sends all data to output, and the latter ends the buffer without sending it to output, effectively wiping out any information you saved in there. Every piece of text outputted while an output buffer is open is placed into that buffer as opposed to being sent to output. Consider the following script:

```
<?php
ob_start();
print "In first buffer!\n";
ob_end_flush();
ob_start();
print "In second buffer!\n";
ob_end_clean();
ob_start();
print "In third buffer!\n";
?>
```

That script will output **"In first buffer"** because the first text is placed into a buffer then flushed with `ob_end_flush()`. The **"In second buffer"** won't be printed out, though, because it's placed into a buffer which is cleaned using `ob_end_clean()` and

Make your mark

Brainstorms 'R' Us

Would you like to get your name in the mag and learn about stuff you're most interested in?

We're always looking out for ideas for new *Linux Format* PHP tutorials, and where better to look than to you, our readers? If, while reading past issues of *LXF's* PHP tutorials, you've thought "I wish they'd covered XYZ in more depth...", or "I really want to know how to use...", then now's the time to get your voice heard!

Send an email to paul.hudson@futurenet.co.uk with your ideas – all the good suggestions that you send in will be covered in future issues. So far, the topics we have covered in some depth include MySQL, XML, CLI, GUIs, media generation, templates, and more.

If you're short of ideas, you're certainly welcome to write in or post on the forums at www.linuxformat.co.uk with comments – we're passionate about improving the overall quality of our tutorials!

not sent to output. Finally, the script will print out **"In third buffer"** because PHP automatically flushes open output buffers when it reaches the end of a script.

Stacking buffers

The functions `ob_end_flush()` and `ob_end_clean()` are complemented by `ob_flush()` and `ob_clean()` – these do the same jobs as their longer cousins, with the difference that they don't end the output buffer. Instead, these functions send the content to output or clean the buffer (respectively), leaving it open for more text. We'll be looking at how you can use these functions to re-use your buffers later on, but for now it's important to understand that you can stack buffers up upon each other to make them even more useful.

Consider the following script:

```
<?php
ob_start();
print "In first buffer!\n";
ob_start();
print "In second buffer!\n";
ob_clean();
?>
```

In that script, we call `ob_start()` twice without closing either of the buffers, and so the end result is that **"In first buffer"** is printed out by itself. If you thought that **"In second buffer"** would be printed out too or that neither lines of text would appear, you haven't grasped quite how buffer stacking works!

The first buffer is started and filled with **"In first buffer"**, then a second buffer is started on top of the first buffer, leaving the first buffer still intact and containing **"In first buffer"**. At this point, we can no longer write to the first buffer, because the second buffer is top of the stack. The new buffer is filled with **"Hello second"**, and finally `ob_clean()` is called, wiping the second buffer, *but leaving the first one intact*.

Flushing stacked buffers

When you stack your output buffers up, data you flush is moved up one level in the stack as opposed to being sent directly to output. This makes more sense with some code, so here you go:

```
<?php
ob_start();
print "In first buffer!\n";
ob_start();
print "In second buffer!\n";
```



```

<< ob_end_flush();
    print "In first buffer\n";
    ob_end_flush();
?>

```

That script will output the following:

```

In first buffer
In second buffer
In first buffer

```

What happens there is that the second buffer gets flushed into the first buffer where it left off, as opposed to directly to output – it literally gets copied into the parent buffer. The first buffer then gets **“In first buffer”** added to it, then flushed to output. Take a look at this following script:

```

<?php
    ob_start();
    print "In first buffer\n";
    ob_start();
    print "In second buffer\n";
    ob_end_flush();
    print "In first buffer\n";
    ob_end_clean();
?>

```

It appears to be the same as the previous script, with the only difference being the last line – **ob_end_clean()** is used rather than **ob_end_flush()**. This time the output is nothing at all, because the second buffer gets flushed into the first buffer, then the first buffer gets cleaned, which means the clients receives none of the text.

As long as you keep in mind that output buffers are stacked up like blocks, which means you can't write to any one of the stack of buffers, this functionality will work in your favour. Using this method it's very easy to progressively build up your content by opening up new buffers as needed, flushing in content to a parent buffer as you go.

Reusing buffers

Given that **ob_flush()** and **ob_clean()** leave the current output buffer open for further writing, there's a potentially big performance boost just waiting to be taken advantage of – by not closing and re-opening buffers all the time, this next script could be rewritten...

```

<?php
    ob_start();
    print "In first buffer!\n";
    ob_end_flush();
    ob_start();
    print "In second buffer!\n";
    ob_end_clean();
    ob_start();
    print "In third buffer!\n";
?>

```

...like this...

```

<?php
    ob_start();
    print "In first buffer!\n";
    ob_flush();
    print "In second buffer!\n";
    ob_clean();
    print "In third buffer!\n";
?>

```

In the new script, the buffer is first flushed and left open, then cleaned and still left open, until finally being automatically closed and flushed by PHP as the script terminates. By not needing to create and end output buffers as the script executes, thereby reusing the same buffer each time, that script executes about 60% faster – this is a substantial difference, as I'm sure you'll agree.

Reading buffers

While writing to and flushing buffers is a boon by itself, you can also *read back* the contents of output buffers, effectively receiving a copy of all the output it holds. This clever functionality is contained in one simple call to **ob_get_contents()**, which takes no parameters and returns a string of all the content it contains. Reading your output back from a buffer is more useful than you might at first think, but it does take a little experimenting to get quite right.

Last issue, for example, we used output buffering and **ob_get_contents()** to write a static page cache – the modified date of the PHP script was compared against the modified date of the cached page, and if the script was newer, it would execute and output its content into an output buffer, which was then retrieved and written to a file.

The key advantage to retrieving output buffering is that you can make one script do many things with almost no change. For example, if you have a script that tracks the location of a package while it's being shipped around the world, the default configuration might have it send its data directly to output for web browsers. However, by using output buffering it literally is a tiny change to make that same script send its output to email, or to an SMS number – the possibilities are endless.

Combining reading output buffering with flushing means that you can save your output to a buffer, read it back in, pass it through various functions to alter the data, then send it back to output – you really have much, much more flexibility, and there are many clever ways you can take advantage of this.

Other OB functions

There are two utility functions that give you information on your current output buffering situation, and these are **ob_get_level()** and **ob_get_length()**. The **ob_get_level()** function is particularly useful as it tells you the buffer stack level you're at – literally how many buffers you have open currently. By default this will return 0 because you have no buffers open, but this number increases as you add more buffers. **ob_get_level()** function is particularly helpful if you want to recursively work with and close open buffers, because you can loop down from **ob_get_level()** to 0.

On the other hand, there's **ob_get_length()**, which returns the size in bytes of the current output buffer. Note that this is not the total length of all buffers, but only the length of the current buffer – you need to use this in combination with **ob_get_level()** to get the total buffer lengths while flushing.

Compressing output

When visitors come to your site, the HTTP request they send for a page also includes a lot of other information about that visitor. For example, it sends the name of the web browser they are using, the last page they were at, and what kind of content encoding they can accept. The content encoding is what we're interested in, because browsers that support compressed (*gzipped*) HTML say so in the HTTP request, which means that the web server can *gzip* the content before it sends it. A key feature of this system is that if a client *doesn't* say that it supports

