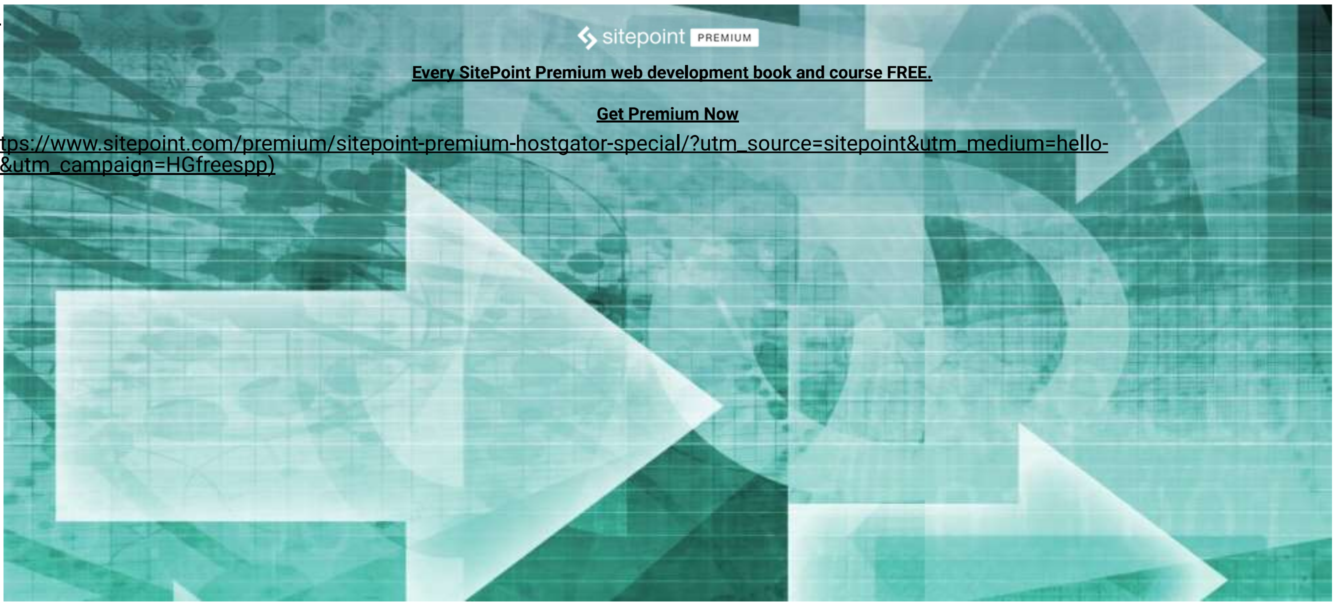


Streaming

Streaming in web refers to the mechanism of sending responses in chunks rather than sending them at once. In the traditional HTTP request / response cycle, a response is not transferred to the browser until it is fully prepared which makes users wait. This limitation is a remnant of the weak software and hardware of the past, but everything has changed a lot since then. Network systems and browsers are now powerful enough to handle transmission of content in a fast and efficient way. Imagine trying to watch a video and not being able to watch it until it's fully downloaded into the player's buffer. With the help of streaming, we can watch videos or listen to our favorite music quite efficiently as content is being loaded instantly while the rest of the data is being downloaded behind the scenes.

Streaming seems ideal for giving users the perception that a web-app is loading fast. Streaming of HTTP responses, however, is quite different from streaming of media content. Streaming of HTTP responses simply means sending a response in fixed or variable size chunks to the browser while the webserver is preparing the remainder. For example, you want to display a list of Hollywood movies on a single page. If you prepare the full response first and then send it to browser, the end-user will definitely feel the delay. But if you send 100 movies in one chunk and display them in the browser while you are preparing the HTML for the next 100 movies, the content will feel like it's loading quite fast.

HTTP responses don't consist just of renderable items – there are plenty of other things such as response status code, HTTP headers, cookies etc. that are essential parts of a response, but go unnoticed by end-users because they're never rendered. Without them, our content has no meaning at all. Instead of letting users wait, the bigger sites send the non-renderable information to the browser and once the browser starts receiving it, it starts rotating the loading indicator you've almost certainly seen when on slower connections. Stylesheet and JavaScript files remain unchanged for most of the time, so many sites also send them along with non-renderable content chunks, so that the browser starts fetching and executing them while the rest of the response is being prepared. This is quite a powerful technique for creating the illusion of speed. When the content of the `body` tag is generated, it is sent to the browser, and that content can be sent in chunks again, further propagating the illusion of speed.

Output Buffering

Output buffering is a mechanism in which instead of sending a response immediately to browser we buffer it somewhere so that we can send it at once when whole content is ready. This is the default way by which PHP sends responses to browsers. We all know that in order to send a response to the browser, we have to use statements like echo or print. Each time we are using echo we are basically telling PHP to send a response to the browser. But since PHP has output buffering enabled by default, that content gets buffered and not sent to the client. Output buffering is configured via `output_buffering` in `php.ini` and you can see its current configuration value by running `phpinfo()`. PHP's documentation (http://www.php.net/manual/en/outcontrol.configuration.php#ini.output-buffering) has the following information on output buffering:

> You can enable output buffering for all files by setting this directive to 'On'. If you wish to limit the size of the buffer to a certain size – you can use a maximum number of bytes instead of 'On', as a value for this directive (e.g., output_buffering=4096). As of PHP 4.3.5, this directive is always Off in PHP-CLI.

According to the above information, we can easily see that the default size of the PHP buffer under most configurations is 4096 bytes (4KB) which means PHP buffers can hold data up to 4KB. Once this limit is exceeded or PHP code execution is finished, buffered content is automatically sent to whatever back end PHP is being used (CGI, mod_php, FastCGI). Output buffering is `always Off` in PHP-CLI. We will see what this means soon.

It is also worth noting that even webservers may have buffering enabled, which means data fetched from the PHP back end can be buffered by webservers and might override PHP's settings. Now that we have a pretty solid understanding of streaming and output buffering, let's take a look at a quick example.

A simple example

Create a file `output.php` in your webserver's root directory so that it is accessible from the browser, e.g. `http://localhost/output.php`. Place the following code in `output.php`.

```
<?php
       echo "Hello ";
       sleep(5);
       echo "World!";
    ?>
```

Go ahead and access this file in your browser. You will not see any content until five seconds have passed, after which the whole "Hello World!" phrase appears. This is because of output buffering. Instead of sending the response to the browser when the first `echo` is executed, its contents are buffered. Since buffered content is sent to the browser if either the buffers get full or code execution ends, and since `Hello World!` is not enough to occupy more than 4KB of buffer size, the content is sent when code execution ends.

Now run the same example but this time from the console (the command line) with the following statement:

```
php /path/to/directory/output.php
```

As soon as you hit enter, you will see the word `Hello` appear, and after five seconds the word `World!` will appear, too. This is what "always off in PHP-CLI" meant. Because output buffering is off, the response is sent as soon as each `echo` is executed.

Now let's look at output buffering in a bit more detail.

Examples

In the last example, we have seen that because of output buffering in PHP, we don't get a response until PHP's execution has finished. This is not desirable since we want to send some content to the browser while we are preparing other responses. As we know, the default size of the output buffer in PHP is 4KB so if we want to send a response to the client we have to generate a response in chunks and each chunk must be 4KB in size. Let's see an example:

8KB chunk example

```
<?php
    $multiplier = 1;
    $size = 1024 * $multiplier;
    for($i = 1; $i <= $size; $i++) {
        echo ".";
    }
    sleep(5);
    echo "Hello World";
?>
```

Save the above code in a file in your webserver's root directory. If you run this example you will see that your browser's loading indicator didn't indicate that data is being received until after five seconds. Now, let's change `$multiplier` from `1` to `8` and refresh. If there are no specific settings configured, you'll notice the browser telling us it started to receive some data almost immediately. Not having to wait for five seconds to realize the page has started loading is very good for user experience.

You might be wondering why we have set `$multiplier` from `1` to `8`. The reason behind this is related to the webserver's buffers. Like we said above, at the first level there is the PHP buffering which we can check via the `output_buffering` PHP setting. Then, there might be PHP back end (CGI, mod_php, FastCGI) buffering and at the end there might be the webserver's buffering. Normally, both Nginx and Apache buffer content up to either `4KB` or `8KB` depending on the operating system being used. Normally, on 64bit operating systems, the limit is `8KB` and on 32bit operating systems it is `4KB`.

The flow of the above code is as follows, assuming `output_buffering` in PHP is set to `4KB`: in the loop when data up to `4KB` has been stored in the PHP buffer due to the `echo` statement, PHP automatically sends this data to it's back end (CGI, mod_php, FastCGI). `mod_php` doesn't buffer data and sends it straight to Apache. `CGI` and `FastCGI` normally buffer data upto `4KB` by default (depending on configuration) so when they receive it, their buffers get full too so the data is instantly sent to the webserver. The webserver in turns buffers data too, up to `4KB` or `8KB` depending on the operating system. Since I'm using a 64bit operating system, the buffering limit on my side is `8KB`. The server receives data of `4KB` but its buffer size is `8KB` so this will not result in a buffer overflow and no output is sent to the browser. When another `4KB` is prepared by the PHP loop, the aforementioned procedure is repeated but this time because of the already saved `4KB` in the server's buffer, the coming `4KB` will result in buffer overflow, causing it to clear and be sent to the browser.

Now go ahead and place following the code after `$size = 1024 * $multiplier;`:

```
$size -= 1;
```

Refresh, and you will see that this time the browser is not indicating that it has started getting content until five seconds have elapsed. This is because we are iterating the loop to `8KB - 1` times which will not result in a buffer overflow and no data will be sent to browser until after five seconds. Now place the following code before `sleep(5);`:

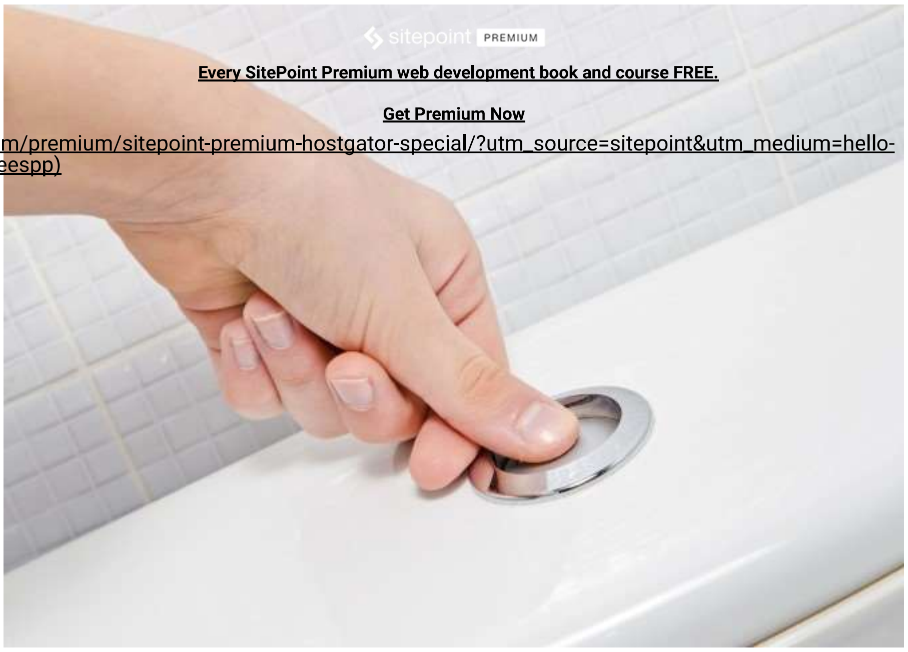
```
echo ".";
```

Refresh your browser one more time, and you will see that this time the browser is indicating that it has started receiving content without the five second delay. We are iterating the loop to `8KB - 1` times which means that the first 4096 bytes have been pushed towards the server and buffered. When the loop finishes, PHP's buffers will have 4095 bytes but `echo ".";` after the loop is helping us fill the buffer with 4096 bytes, resulting in sending the buffer content to upper layers and thus to the browser.

There is one caveat that you should be aware of. PHP's `output_buffering` setting has two possible values. One is to indicate whether it is `On` and the second is it to indicate the maximum size of the buffer. If `output_buffering` is set to `1`, then you might not be able to see your content or browser loading indicator rotating until PHP code execution is finished. This is because having `output_buffering` on `1` means that we have enabled it but haven't specified a maximum size, so in this case PHP buffers can hold data up to the number in the `memory_limit` setting.

ob_flush and flush

We are now familiar with the concept of output buffering and streaming in PHP, and we also know how to send responses in chunks to the browser. However, you might be wondering whether or not there is a better way of sending content in chunks. It is just not feasible to generate `8KB` chunks just to send data to the client in advance because normal web pages don't have much content and `8KB` is certainly a decent amount of data to be sent in chunks. It's also not beneficial to send useless data as that will only increase latency. It turns out that there are some built-in methods that we can use to overcome this problem.



`ob_flush()` and `flush` are PHP's built-in methods which are used to send the data to the upper layers. Buffered data is not sent to the upper layers unless the buffers are full or PHP code execution is finished. To send data even when buffers are not full and PHP code execution is not finished we can use `ob_flush` and `flush`.

Now let's see an example:

```
<?php
       $multiplier = 1;
       $size = 1024 * $multiplier;
       for($i = 1; $i <= $size; $i++) {
           echo ".";
       }
       sleep(5);
       echo "Hello World";
   ?>
```

In the example above, place the following lines before `sleep(5);`

```
ob_flush();
   flush();
```

Save the file and access it in the browser. As soon as you ask the browser to fetch the web page, you will see that the browser is indicating that it has started to receive content. That's exactly what we want, because we don't have to worry about generating content in `8KB` chunks and we can easily stream content to the browser without having to wait for the whole content to be generated. You can try different multipliers to get a more solid grip on these concepts.

There are, however, some caveats that you should be aware of. The above code will work fine in Apache with `mod_php`. It will even work without the `for` loop. As soon as `ob_flush()` and `flush()` are executed, the browser will start indicating that some content is coming. However, `ob_flush()` and `flush()` might not work with Nginx out of the box because of the way Nginx processes requests. In order for `ob_flush` and `flush` to work seamlessly in Nginx you can use following configuration:

```
fastcgi_buffer_size   1k;
    fastcgi_buffers       128 1k;  # up to 1k + 128 * 1k
    fastcgi_max_temp_file_size 0;
    gzip off;
```

You can find out more about this in this post (http://www.justincarmony.com/blog/2011/01/24/php-nginx-and-output-flushing/).

Streaming with Ajax

Now that we have seen how to send content in chunks in a standard HTTP request / response cycle, let's see how to do the same for Ajax requests. Ajax requests are a nice and elegant way of getting data without reloading the full page. We associate a callback with an Ajax request and that callback gets executed once all content is received. This means that we cannot stream content in Ajax requests. Luckily, we have `XMLHTTPRequest 2`, which is the next version of the Ajax API and supported (http://caniuse.com/#feat=xhr2) in the latest browsers. This new version has a lot of cool features such as cross-origin requests, uploading progress events and support for uploading / downloading binary data. Progress events are used to tell the user how much data we have uploaded, and we can also get downloaded data in chunks. Let's see an example:

Create an HTML file with the following code:

```
<html>
      <head>
        <title>Ajax Streaming Test</title>
      </head>
      <body>
        <center><a href="#" id="test">Test Ajax Streaming</a></center>
        <script type="text/javascript">
          document.getElementById('test').onclick = function() {
            xhr = new XMLHttpRequest();
            xhr.open("GET", "response.php", true);
            xhr.onprogress = function(e) {
              alert(e.currentTarget.responseText);
            }
            xhr.onreadystatechange = function() {
              if (xhr.readyState == 4) {
                console.log("Complete = " + xhr.responseText);
              }
            }
            xhr.send();
          };
        </script>
      </body>
   </html>
```

Now load this file in browser and click the link. An Ajax request is initiated to fetch data from `response.php` and we are listening to the `onprogress` event. Whenever a new chunk arrives, we output it in an alert.

Now put the following code in `response.php` and save it in the same folder, relative to the above HTML file.

```
<?php for ($i = 1; $i <= 10; $i++): ?>
      <?php sleep(1); ?>
      Count = <?php echo "$i\n"; ?>
      <?php ob_flush(); flush(); ?>
   <?php endfor; ?>
```

As you can see, we run a loop ten times, pausing for one second on each run and then echoing some content. This content gets sent to the upper layers with the flushes. Now go ahead and click on `Test Ajax Streaming`. If all goes well, you'll notice `Count = 1` getting displayed in an alert. When you dismiss the alert, you'll see another alert with `Count = 1 \n Count = 2`. When you dismiss that one, you'll see `Count = 1 \n Count = 2 \n Count = 3` in another alert, and so on until 10. When the entire Ajax request is successfully completed, you will see the complete output in the console. We have just implemented streaming in Ajax requests and we can easily update our interface accordingly, giving end-users an outstanding experience.

Note: there are some browser incompatibilities. Test the above code in both Chrome and Firefox. Firefox will behave exactly as demonstrated in terms of output, but Chrome will display an empty alert first, and then continue as expected. Keep this edge case in mind when implementing streaming!

Conclusion

Streaming is an awesome way of sending content to the user in order to simulate speed. But like everything, streaming is not a silver bullet and it has it's own shortcomings. The following are some situations where streaming is not an ideal solution:

1 Handling exceptions: A status code is essential for browsers to determine the success / failure of every request. Since we have sent the status code in advance along with headers, cookies etc. if at some point an exception occurs on the server side, the server will not be able to convey it to browser since the status code is already sent.

2 Sending small chunks of data is inefficient as networks prefer a small number of chunks with bigger responses instead of a high number of chunks with smaller responses (http://en.wikipedia.org/wiki/Nagle%27s_algorithm). In order to send content in chunks, we have to choose the chunk size carefully.

I hope this article helped you grasp the basics of streaming and buffers. Please let us know of other experiments you came up with after reading this, if any, and of course, leave your feedback in the comments below. Would you like to see more examples? More explanations? Let us know!

Was this helpful?

More: ajax (https://www.sitepoint.com/tag/ajax/), apache (https://www.sitepoint.com/tag/apache/), Async (https://www.sitepoint.com/tag/async/), asynchronous (https://www.sitepoint.com/tag/asynchronous/), buffering (https://www.sitepoint.com/tag/buffering/), flush (https://www.sitepoint.com/tag/flush/), nginx (https://www.sitepoint.com/tag/nginx/), PHP (https://www.sitepoint.com/tag/php/), streaming (https://www.sitepoint.com/tag/streaming/)

Meet the author
Imran Latif (https://www.sitepoint.com/author/ilatif/) (https://twitter.com/ilatif_bwp) (https://plus.google.com/106912362610440726708/) (https://www.facebook.com/imran.latif.988) (https://github.com/ilatif)

Imran Latif is a Web Developer and Ruby on Rails and JavaScript enthusiast from Pakistan. He is a passionate programmer and always keeps on learning new tools and technologies. During his free time he watches tech videos and reads tech articles to increase his knowledge.

20 Comments **SitePoint** Login

Recommend 3 Share Sort by Best

Join the discussion…

Christophe Coevoet • 2 years ago
There is a better way to handle the Nginx buffers. Instead of disabling the buffers in your config (which might impact other pages where the server buffer could be useful), it would be better to disable it only for the streamed response. This is done by sending a X-Accel-Buffering header with the value "no"
8 • Reply • Share ›

Bruno Skvorc SitePoint Staff → Christophe Coevoet • 2 years ago
Thanks for the input!
• Reply • Share ›

Bob Dole → Bruno Skvorc • 2 years ago
Not to offend, but i think... you are using a rail gun to hunt ants...

The propper use for this would be a different approach, at least for me, i would dump a raw page (with all their data) then change tokens of it then send it... but i guess thats a crappy way to do those things... so in fact... i see a big disavange on using buffered output... but am not a fan of it... maybe there is a great use for it... just we didn´t discovered yet... (maybe running some javascrip on the browser, and then send feedback to the middle of the still running php script?)

Who knows... session processing is a mistery, for those like me...

Btw thanks for the article, very educational, but as i said... it leave many misteries... maybe a second part of it would be great Bruno