Published in Dev Cafe

Firthous Follow

Aug 23, 2020 · 3 min read · Listen

How to compress API responses in PHP Lumen

Simply using the inbuilt PHP function will solve largely

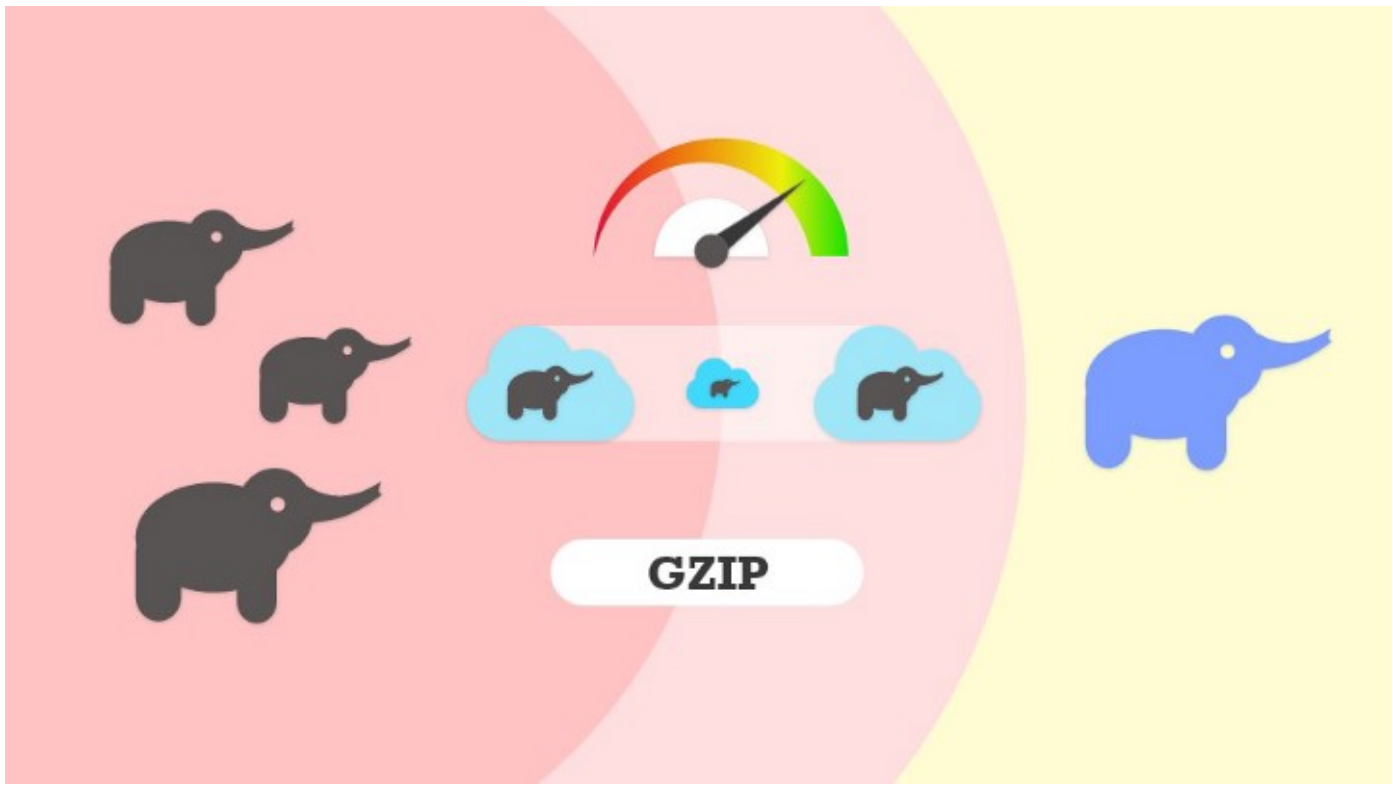


Illustration by author

Dealing with large data often we required to compress/decompress the data to reduce some bandwidth on client-server communications. Mostly to avoid large responses on ajax calls we required to follow compression techniques to maximize the application performance.

There are two popular algorithms available to achieve this GZip and Deflate. both of these algorithms are recognized by modern-day browsers. When REST APIs are mostly designed its responses in JSON format. Since JSON is text-based, so it can be compressed and served **based on our application requirement** either dynamically or statically

The Compress Method

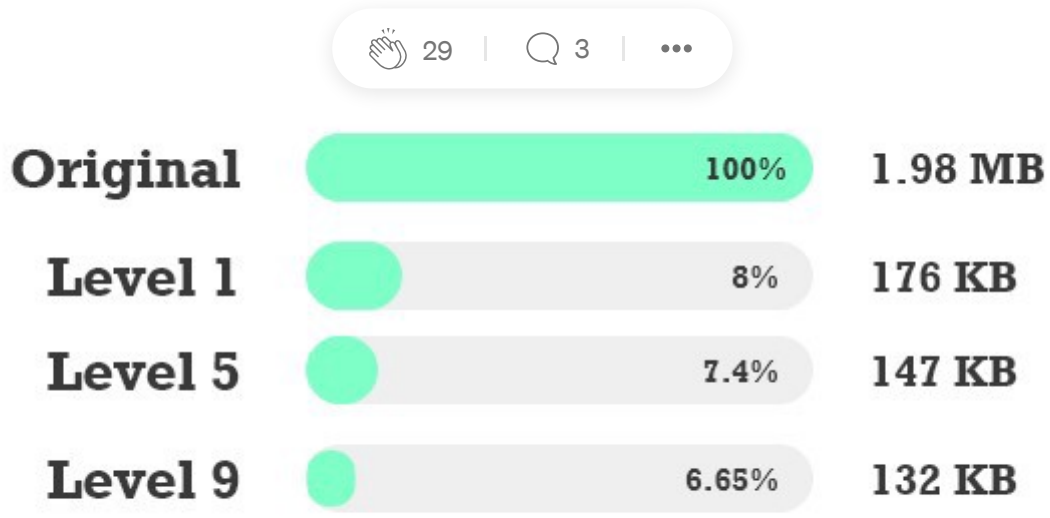
In this article, we going to use `gzencode()` inbuilt PHP function to compress the response data.

```
gzencode(string $data, int $level = -1, int $encoding_mode =
FORCE_GZIP ) : string
```

2. Level (optional)— The compression level 0 to 9, 0 means no compression and 9 means the highest level compression.
3. Algorithm (optional) — default `FORCE_ZIP` or `FORCE_DEFLATE`

Benchmark

Some benchmark tests on 1.98 MB of JSON strings data can be compressed at level 9 is around 132 KB.



Benchmark by author

Lumen Implementations

Laravel Lumen is one of the best solutions to building REST API based microservices in PHP. Here we going to implement simple middleware that will send the compressed response with necessary response headers.

Step 1: Create the middleware class

Create the new file `GZipMiddleware.php` in the folder `app/Http/Middleware`

In middleware, the function `gzencode()` was used number 9 of high-level compression with the default gzip algorithm. It also tells the browser the encoding method `` `Content-Encoding`=> `gzip` `` as in the response headers.

Step 2: Enable the middleware

We going to use only specific API methods that return large responses. So we need to activate in `$app->routeMiddleware()` function on the file located in the folder `bootstrap/app.php`

```
$app->routeMiddleware([
    'gzip' => App\Http\Middleware\GzipMiddleware::class,
]);
```

Enabled in route middleware and adding the newly added `GzipMiddleware` class with handy name `'gzip'` so easily we can use on any route methods.

Step 3: Add on the router

The final step simply adds the name we defined in the middleware to the methods that required a compressed response. Router configurations were located on `routes/web.php`

```
$router->get('/data', [
    'middleware' => 'gzip',
    'as' => 'data',
    'uses'=>'DataController@getData'
]);
```

Now every request to the method `/data` will produce a compressed response.

Pros & Cons

Based on our use cases either we use a cache mechanism or data can be stored and served in compression format for optimal usages. Using gzip on the fly dynamically has some pros and cons.

Pros

Cons

Compression techniques will use CPU cycles on both client and server machines it means response time slightly will be increased if it used dynamically on the server-side.

Conclusion

Compression techniques can be used either dynamically or statically. In PHP we have a handy and convenient function for doing compression. Gzip will help us to solve moving large data in the client-server applications with improved performance.