Minify and compress network payloads

- Measure
- Minification
- Data compression
- Dynamic compression
- Static compression

There are two useful techniques that can be used to improve the performance of your web page:

- Minification
- Data compression

Incorporating both of these techniques reduces payload sizes and in turn improves page load times.

Measure

Lighthouse displays a failed audit if it detects any CSS or JS resources on your page that can be minified.

Lighthouse Minify CSS Audit

Lighthouse Minify JS Audit

It also audits for any uncompressed assets.

Lighthouse: Enable text compression

Minification

Minification is the process of removing whitespace and any code that is not necessary to create a smaller but perfectly valid code file. Terser is a popular JavaScript compression tool and webpack v4 includes a plugin for this library by default to create minified build files.

- If you're using webpack v4 or greater, you should be good to go without doing any additional work.
- If you are using an older version of webpack, install and include `TerserWebpackPlugin` into your webpack configuration settings. Follow the documentation to learn how.
- If you are not using a module bundler, use `Terser` as a CLI tool or include it directly as a dependency to your application. The project documentation provides instructions.

Data compression

Compression is the process of modifying data using a compression algorithm. Gzip is the most widely used compression format for server and client interactions. Brotli is a newer compression algorithm which can provide even better compression results than Gzip.

Compressing files can significantly improve the performance of a webpage, but you rarely need to do this yourself. Many hosting platforms, CDNs and reverse proxy servers either encode assets with compression by default or allow you to easily configure them. Read the documentation for the tool that you are using to see if compression is already supported before attempting to roll out your own solution.

There are two different ways to compress files sent to a browser:

- Dynamically
- Statically

Both approaches have their own advantages and disadvantages which is covered in the next section. Use whichever works best for your application.

Dynamic compression

This process involves compressing assets on-the-fly as they get requested by the browser. This can be simpler than compressing files manually or with a build process, but can cause delays if high compression levels are used.

Express is a popular web framework for Node and provides a compression middleware library. Use it to compress any asset as it gets requested. Here is an example of an entire server file that uses it correctly:

```
const express = require('express');
const compression = require('compression');

const app = express();

app.use(compression());

app.use(express.static('public'));

const listener = app.listen(process.env.PORT, function() {
  console.log('Your app is listening on port ' + listener.address().port);
});
```

This compresses your assets using `gzip`. If your web server supports it, consider using a separate module like shrink-ray to compress via Brotli to achieve better compression ratios.

Use express.js to compress assets with gzip and Brotli.

Static compression

Static compression involves compressing and saving assets ahead of time. This can make the build process take longer, especially if high compression levels are used, but ensures that no delays happen when the browser fetches the compressed resource.

If your web server supports Brotli, use a plugin like BrotliWebpackPlugin with webpack to compress your assets as part of your build step. Otherwise, use CompressionPlugin to compress your assets with gzip. It can be included just like any other plugin in the webpack configurations file:

```
module.exports = {
  //...
  plugins: [
    //...
    new CompressionPlugin()
  ]
}
```

Once compressed files are part of the build folder, create a route in your server to handle all JS endpoints to serve the compressed files. Here is an example of how this can be done with Node and Express for gzipped assets.

```
const express = require('express');
const app = express();

app.get('*.js', (req, res, next) => {
  req.url = req.url + '.gz';
  res.set('Content-Encoding', 'gzip');
  next();
});

app.use(express.static('public'));
```

□□□ #3
□ □□□ □□□ 2026-10-31 22:37:37 UTC
□ cocobook-sync □□□ 2026-11-04 17:38:52 UTC