**Reduce JavaScript execution time**

May 2, 2019 • Updated Oct 4, 2019

Appears in: [Performance audits](https://web.dev/lighthouse-performance)

When your JavaScript takes a long time to execute, it slows down your page performance in several ways:

* **Network cost**

More bytes equals longer download times.

* **Parse and compile cost**

JavaScript gets parsed and compiled on the main thread. When the main thread is busy, the page can't respond to user input.

* **Execution cost**

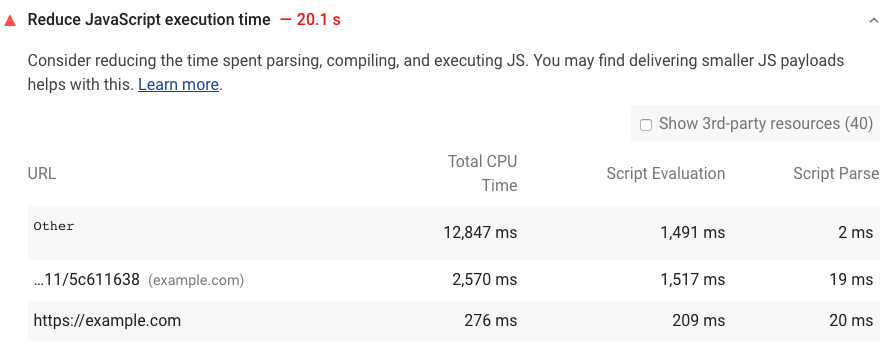
JavaScript is also executed on the main thread. If your page runs a lot of code before it's really needed, that also delays your [Time To Interactive](https://web.dev/interactive), which is one of the key metrics related to how users perceive your page speed.

* **Memory cost**

If your JavaScript holds on to a lot of references, it can potentially consume a lot of memory. Pages appear janky or slow when they consume a lot of memory. Memory leaks can cause your page to freeze up completely.

**How the Lighthouse JavaScript execution time audit fails** [**#**](https://web.dev/bootup-time/?utm_source=lighthouse&utm_medium=devtools#how-the-lighthouse-javascript-execution-time-audit-fails)

[Lighthouse](https://developers.google.com/web/tools/lighthouse/) shows a warning when JavaScript execution takes longer than 2 seconds. The audit fails when execution takes longer than 3.5 seconds:



To help you identify the biggest contributors to execution time, Lighthouse reports the time spent executing, evaluating, and parsing each JavaScript file that your page loads.

See the [Lighthouse performance scoring](https://web.dev/performance-scoring) post to learn how your page's overall performance score is calculated.

**How to speed up JavaScript execution** [**#**](https://web.dev/bootup-time/?utm_source=lighthouse&utm_medium=devtools#how-to-speed-up-javascript-execution)

* [Only send the code that your users need by implementing code splitting](https://web.dev/reduce-javascript-payloads-with-code-splitting).
* [Minify and compress your code](https://web.dev/reduce-network-payloads-using-text-compression).
* [Remove unused code](https://web.dev/remove-unused-code).
* [Reduce network trips by caching your code with the PRPL pattern](https://web.dev/apply-instant-loading-with-prpl).

For other ways to improve page load, check out the [Performance audits landing page](https://web.dev/lighthouse-performance).

**Reduce JavaScript payloads with code splitting**

Nov 5, 2018

Appears in: [Fast load times](https://web.dev/fast)

[](https://web.dev/authors/houssein/)

[*Houssein Djirdeh*](https://web.dev/authors/houssein/)

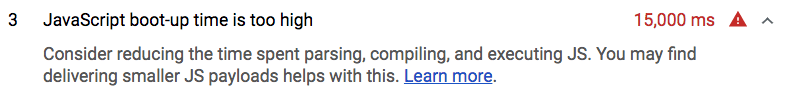
* [Twitter](https://twitter.com/hdjirdeh)
* [GitHub](https://github.com/housseindjirdeh)
* [Glitch](https://glitch.com/@housseindjirdeh)
* [Blog](https://houssein.me/)

Nobody likes waiting. [**Over 50% of users abandon a website if it takes longer than 3 seconds to load**](https://www.thinkwithgoogle.com/intl/en-154/insights-inspiration/research-data/need-mobile-speed-how-mobile-latency-impacts-publisher-revenue/).

Sending large JavaScript payloads impacts the speed of your site significantly. Instead of shipping all the JavaScript to your user as soon as the first page of your application is loaded, split your bundle into multiple pieces and only send what's necessary at the very beginning.

**Measure** [**#**](https://web.dev/reduce-javascript-payloads-with-code-splitting/#measure)

Lighthouse displays a failed audit when a significant amount of time is taken to execute all the JavaScript on a page.



Split the JavaScript bundle to only send the code needed for the initial route when the user loads an application. This minimizes the amount of script that needs to be parsed and compiled, which results in faster page load times.

Popular module bundlers like [webpack](https://webpack.js.org/guides/code-splitting/), [Parcel](https://parceljs.org/code_splitting.html), and [Rollup](https://rollupjs.org/guide/en#dynamic-import) allow you to split your bundles using [dynamic imports](https://developers.google.com/web/updates/2017/11/dynamic-import). For example, consider the following code snippet that shows an example of a someFunction method that gets fired when a form is submitted.

import moduleA from "library";  
  
form.addEventListener("submit", e => {  
 e.preventDefault();  
 someFunction();  
});  
  
const someFunction = () => {  
 // uses moduleA  
}

In here, someFunction uses a module imported from a particular library. If this module is not being used elsewhere, the code block can be modified to use a dynamic import to fetch it only when the form is submitted by the user.

form.addEventListener("submit", e => {  
 e.preventDefault();  
 import('library.moduleA')  
 .then(module => module.default) // using the default export  
 .then(someFunction())  
 .catch(handleError());  
});  
  
const someFunction = () => {  
 // uses moduleA  
}

The code that makes up the module does not get included into the initial bundle and is now **lazy loaded**, or provided to the user only when it is needed after the form submission. To further improve page performance, [preload critical chunks to prioritize and fetch them sooner](https://web.dev/preload-critical-assets).

Although the previous code snippet is a simple example, lazy loading third party dependencies is not a common pattern in larger applications. Usually, third party dependencies are split into a separate vendor bundle that can be cached since they don't update as often. You can read more about how the [**SplitChunksPlugin**](https://webpack.js.org/plugins/split-chunks-plugin/) can help you do this.

Splitting on the route or component level when using a client-side framework is a simpler approach to lazy loading different parts of your application. Many popular frameworks that use webpack provide abstractions to make lazy loading easier than diving into the configurations yourself.

**Minify and compress network payloads**

Nov 5, 2018

Appears in: [Fast load times](https://web.dev/fast)

[](https://web.dev/authors/houssein/)

[*Houssein Djirdeh*](https://web.dev/authors/houssein/)

* [Twitter](https://twitter.com/hdjirdeh)
* [GitHub](https://github.com/housseindjirdeh)
* [Glitch](https://glitch.com/@housseindjirdeh)
* [Blog](https://houssein.me/)

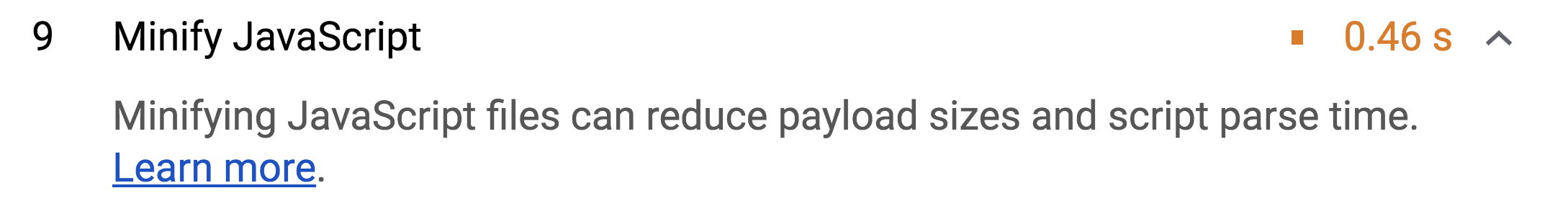
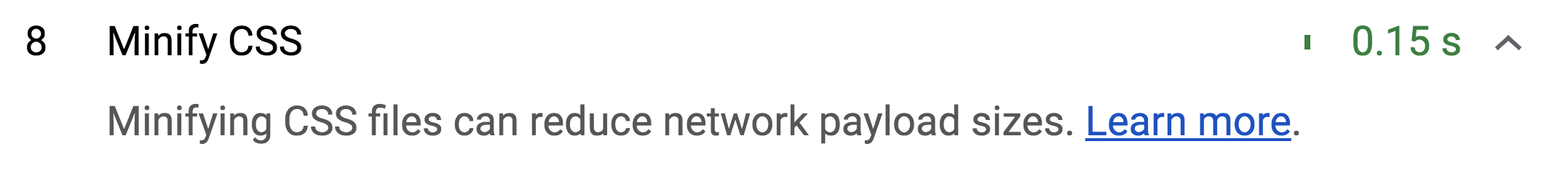
There are two useful techniques that can be used to improve the performance of your web page:

* Minification
* Data compression

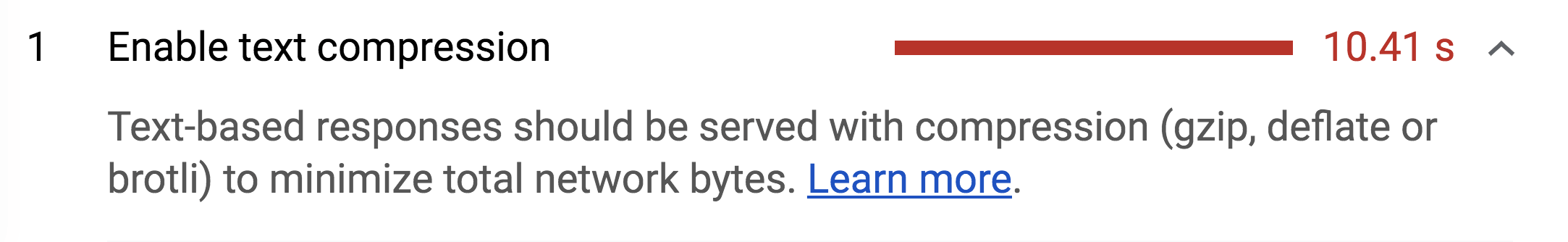
Incorporating both of these techniques reduces payload sizes and in turn improves page load times.

**Measure** [**#**](https://web.dev/reduce-network-payloads-using-text-compression/#measure)

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



It also audits for any uncompressed assets.



**Minification** [**#**](https://web.dev/reduce-network-payloads-using-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](https://github.com/terser-js/terser) is a popular JavaScript compression tool and [webpack](https://webpack.js.org/) 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](https://webpack.js.org/plugins/terser-webpack-plugin/) 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](https://github.com/terser-js/terser) provides instructions.

**Data compression** [**#**](https://web.dev/reduce-network-payloads-using-text-compression/#data-compression)

**Compression** is the process of modifying data using a compression algorithm. [Gzip](https://www.youtube.com/watch?v=whGwm0Lky2s&feature=youtu.be&t=14m11s) is the most widely used compression format for server and client interactions. [Brotli](https://opensource.googleblog.com/2015/09/introducing-brotli-new-compression.html) 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** [**#**](https://web.dev/reduce-network-payloads-using-text-compression/#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](https://expressjs.com/) is a popular web framework for Node and provides a [compression](https://github.com/expressjs/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](https://github.com/aickin/shrink-ray#readme) to compress via Brotli to achieve better compression ratios.

**Try it**! Use express.js to compress assets with [gzip](https://web.dev/codelab-text-compression) and [Brotli](https://web.dev/codelab-text-compression-brotli).

**Static compression** [**#**](https://web.dev/reduce-network-payloads-using-text-compression/#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](https://github.com/mynameiswhm/brotli-webpack-plugin) with webpack to compress your assets as part of your build step. Otherwise, use [CompressionPlugin](https://github.com/webpack-contrib/compression-webpack-plugin) 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'));

**Remove unused code**

npm makes adding code to your project a breeze. But are you really using all  
those extra bytes?

Nov 5, 2018

Appears in: [Fast load times](https://web.dev/fast)

[](https://web.dev/authors/houssein/)

[*Houssein Djirdeh*](https://web.dev/authors/houssein/)

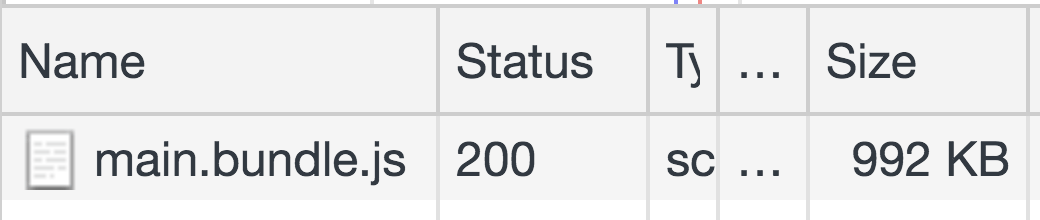
* [Twitter](https://twitter.com/hdjirdeh)
* [GitHub](https://github.com/housseindjirdeh)
* [Glitch](https://glitch.com/@housseindjirdeh)
* [Blog](https://houssein.me/)

Registries like [npm](https://docs.npmjs.com/getting-started/what-is-npm) have transformed the JavaScript world for the better by allowing anyone to easily download and use over *half a million* public packages. But we often include libraries we're not fully utilizing. To fix this issue, **analyze your bundle** to detect unused code. Then remove **unused** and **unneeded** libraries.

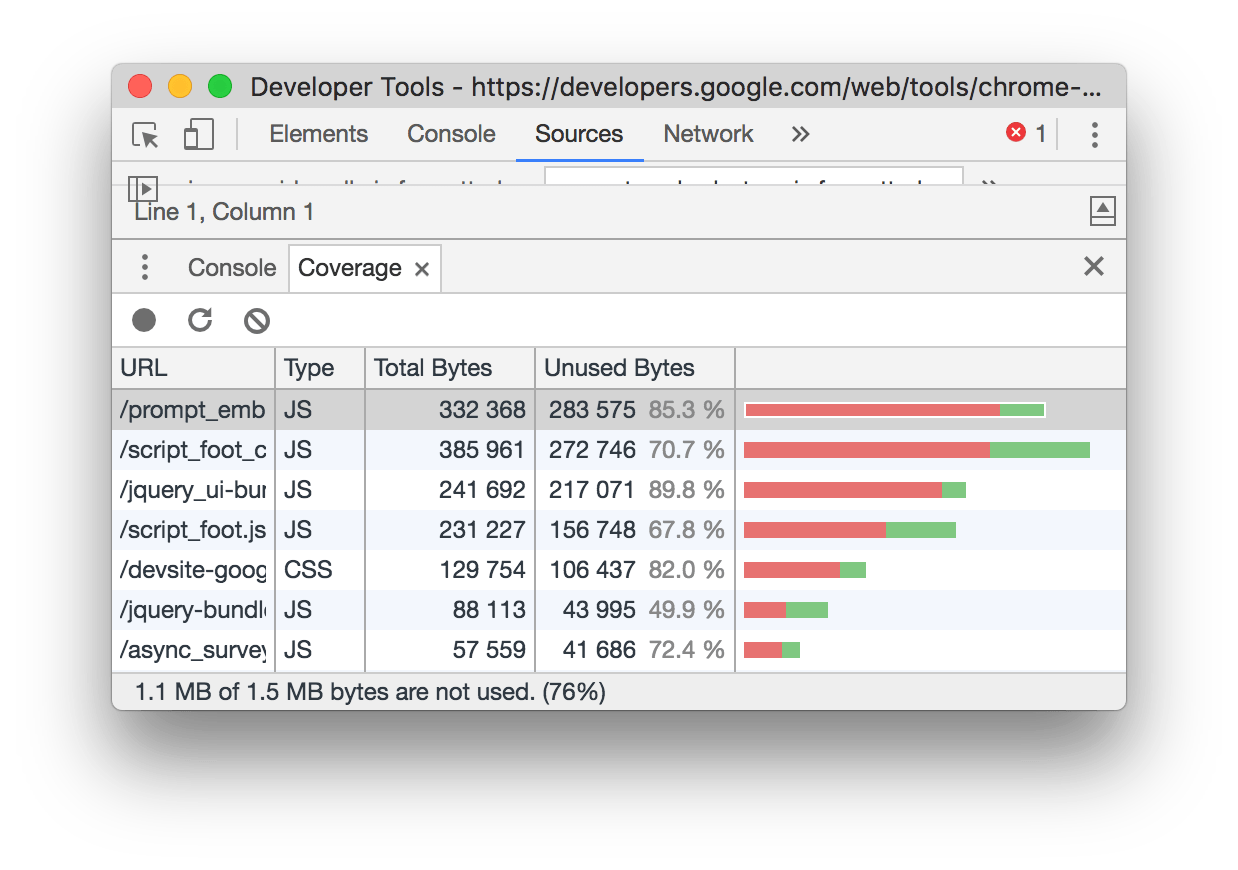
**Analyze your bundle** [**#**](https://web.dev/remove-unused-code/#analyze-your-bundle)

DevTools makes it easy to see the size of all network requests:

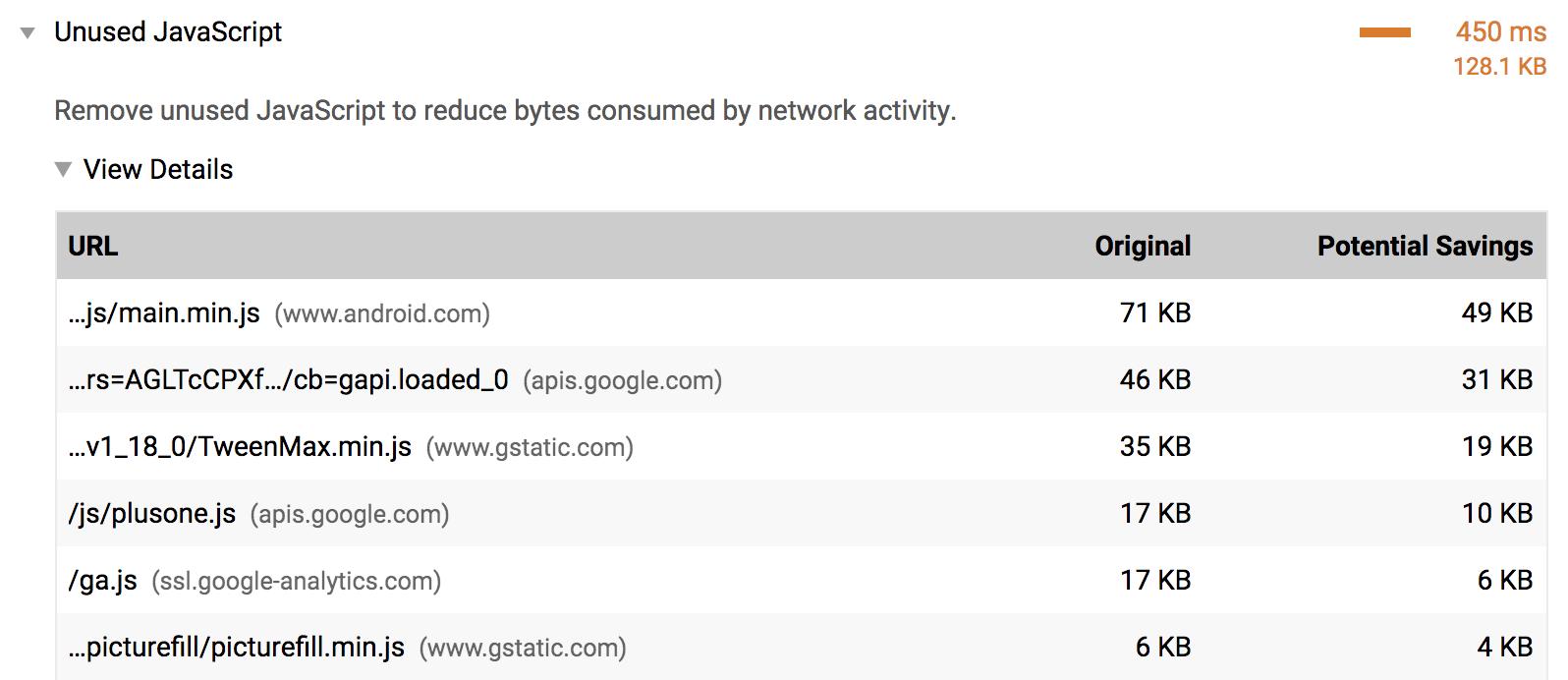
1. Press `Control+Shift+J` (or `Command+Option+J` on Mac) to open DevTools.
2. Click the **Network** tab.
3. Select the **Disable cache** checkbox.
4. Reload the page.



The [Coverage](https://developer.chrome.com/docs/devtools/coverage/) tab in DevTools will also tell you how much CSS and JS code in your application is unused.



By specifying a full Lighthouse configuration through its Node CLI, an "Unused JavaScript" audit can also be used to trace how much unused code is being shipped with your application.

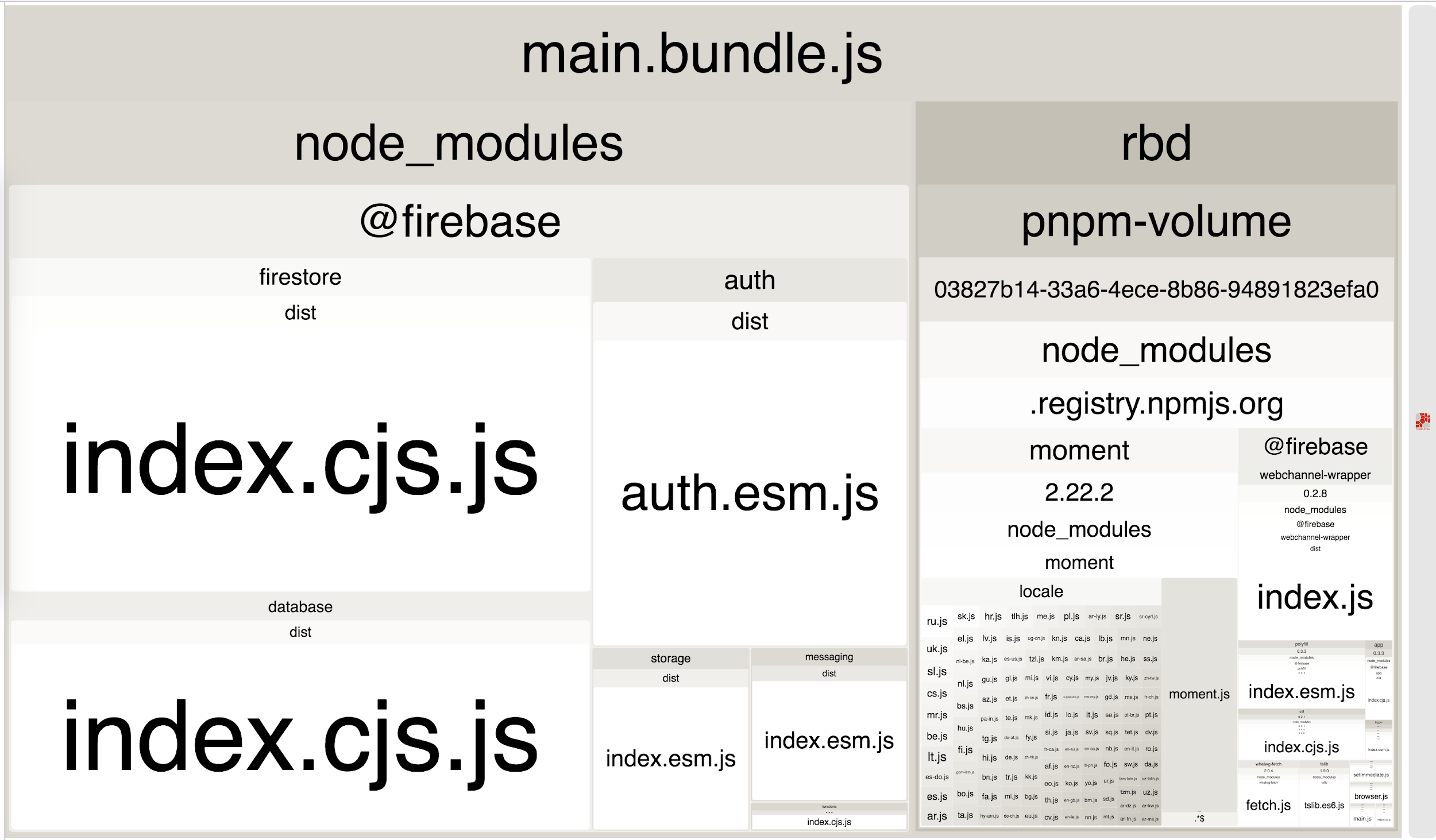


If you happen to be using [webpack](https://webpack.js.org/) as your bundler, [Webpack Bundle Analyzer](https://github.com/webpack-contrib/webpack-bundle-analyzer) will help you investigate what makes up the bundle. Include the plugin in your webpack configurations file like any other plugin:

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

Although webpack is commonly used to build single-page applications, other bundlers, such as [Parcel](https://parceljs.org/) and [Rollup](https://rollupjs.org/guide/en), also have visualization tools that you can use to analyze your bundle.

Reloading the application with this plugin included shows a zoomable treemap of your entire bundle.



Using this visualization allows you to inspect which parts of your bundle are larger than others, as well as get a better idea of all the libraries that you're importing. This can help identify if you are using any unused or unnecessary libraries.

**Remove unused libraries** [**#**](https://web.dev/remove-unused-code/#remove-unused-libraries)

In the previous treemap image, there are quite a few packages within a single @firebase domain. If your website only needs the firebase database component, update the imports to fetch that library:

i  
import firebase from 'firebase/app';  
import 'firebase/database';

It is important to emphasize that this process is significantly more complex for larger applications.

For the mysterious looking package that you're quite sure is not being used anywhere, take a step back and see which of your top-level dependencies are using it. Try to find a way to only import the components that you need from it. If you aren't using a library, remove it. If the library isn't required for the initial page load, consider if it can be [lazy loaded](https://web.dev/reduce-javascript-payloads-with-code-splitting).

And in case you're using webpack, check out [the list of plugins that automatically remove unused code from popular libraries](https://github.com/GoogleChromeLabs/webpack-libs-optimizations).

**Try it**! [Remove unused code.](https://web.dev/codelab-remove-unused-code)

**Remove unneeded libraries** [**#**](https://web.dev/remove-unused-code/#remove-unneeded-libraries)

Not all libraries can be easily broken down into parts and selectively imported. In these scenarios, consider if the library could be removed entirely. Building a custom solution or leveraging a lighter alternative should always be options worth considering. However, it is important to weigh the complexity and effort required for either of these efforts before removing a library entirely from an application.

--

**Webpack Bundle Analyzer**

Visualize size of webpack output files with an interactive zoomable treemap.

**Install**

# NPM

npm install --save-dev webpack-bundle-analyzer

# Yarn

yarn add -D webpack-bundle-analyzer

**Usage (as a plugin)**

const BundleAnalyzerPlugin = require('webpack-bundle-analyzer').BundleAnalyzerPlugin;

module.exports = {

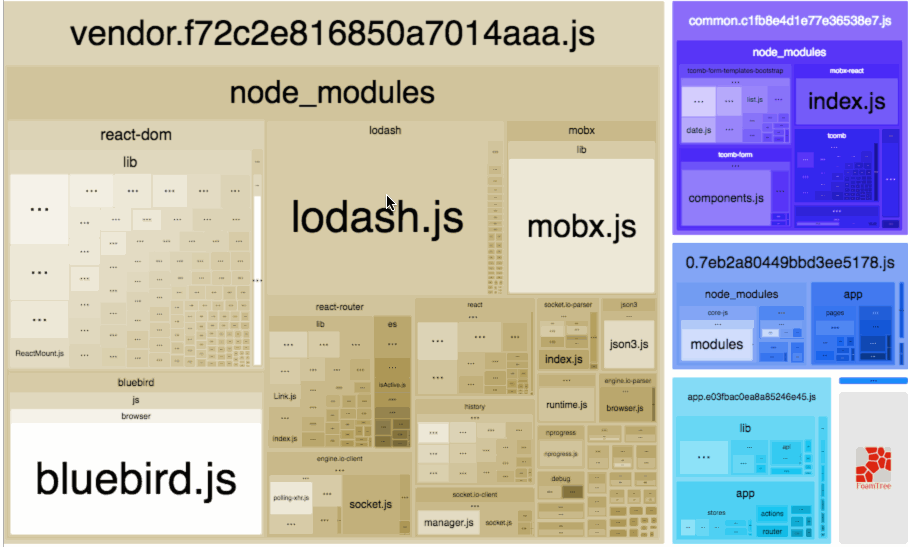
plugins: [

new BundleAnalyzerPlugin()

]

}

It will create an interactive treemap visualization of the contents of all your bundles.

[](https://cloud.githubusercontent.com/assets/302213/20628702/93f72404-b338-11e6-92d4-9a365550a701.gif)

This module will help you:

1. Realize what's *really* inside your bundle
2. Find out what modules make up the most of its size
3. Find modules that got there by mistake
4. Optimize it!

And the best thing is it supports minified bundles! It parses them to get real size of bundled modules. And it also shows their gzipped sizes!