Bundle Size Optimization: A Practical Guide Using Webpack and Statoscope

Overview

Have you ever wondered about the term "bundle size" and its significance in web development? **Bundle size** refers to the combined file size of all JavaScript code, libraries, and dependencies that are bundled together and served to the end-user in their browser. Larger bundle sizes can increase your page load times and ultimately lead to a less responsive and less engaging user experience.

As a developer, there may be times when you come across the need to remove modules that were once part of your project; for example, if they have since been replaced with an alternative, or if they are deprecated and no longer required, but are still in your codebase. Tackling this part of development can be tough. However, as we work through these situations, we often gain useful insights and wisdom that can help others facing similar hurdles down the line.

To that end, I want to share some useful guiding principles and tools we can use to help us with this task. We'll focus on using **Webpack Bundle Analyze**r and **Statoscope** to optimize our app's bundle size. These tools will help us visualize and understand how chunks of code come together to make up our application and in doing so, will help us pinpoint exactly where there may be areas for improvement and optimization in our bundle.

Understanding Bundle Size and Its Importance

You're probably familiar with Webpack, but as a quick refresher - it's a JavaScript module bundler that compiles, transforms, and packages your app's source code, along with its dependencies, into output files that can be loaded into your browser. Webpack is often included by default when using boilerplate setups for React apps or using a framework such as NextJS.

Before we dive into the specifics, let's take a moment to understand why we bundle in web development in the first place. **Bundling** is crucial for optimizing web applications, as it minifies the overall code size, simplifies the management of dependencies, ensuring that all required files are loaded in the correct order, and enables better caching. All this results in a faster, more streamlined user experience. This is important to consider, especially when you don't know what kind of machine your users have or even the speed of their internet connection.

One of the primary benefits of bundling is the reduction of the overall code size. When multiple files are combined into a single file or a few smaller files, they can be minified more effectively, removing unnecessary whitespace, comments, and other redundant information. This leads to a smaller file size, which in turn allows the browser to download and process the code more quickly.

There are often processes in place such as tree shaking, chunking, and code splitting that help keep our bundle sizes small and remove modules that are not being imported in our code. However, these processes don't always catch unused modules, and there are tools and techniques out there that can help us in those situations. When you use barrel files, for example, they serve as an index or entry point, making it easier to import and manage module dependencies. This can sometimes lead to unused modules not being removed from my final bundle. Using `export *` will export everything from the specified module, including exports that you may not be using anywhere in your application. This makes it harder for bundlers and tree-shaking algorithms to determine which exports are actually used and which can be safely removed.

Code splitting is a process used in order to break down your application's code into smaller 'chunks', instead of serving it as one, larger bundle. This is something that NextJS does for you automatically and is a feature that makes it very popular. This allows only certain chunks to be loaded on a page with only the necessary code for that page, and reduces the amount of code that needs to be downloaded and executed for the user to first access your application.

Visualizing Your Bundle Size with Webpack Bundle Analyzer

Webpack Bundle Analyzer is a powerful tool that can generate a visual representation of your JavaScript bundle size, making it easier for you to identify areas for optimization in your project. It can show larger dependencies, duplicated or unused modules that may be able to be removed in order to reduce your bundle size.

First, you need to install it in your project.

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

There is another bundle analyzer specifically for use in NextJS projects - if that's what you're using you can do the following.

```
npm install @next/bundle-analyzer
```

You will need to do some additional configuration after you install it, depending on whether or not your project is in NextJS. More information on the setup for React and the Webpack Bundle Analyzer can be found here. If you're using NextJS, you'll need to create a `` `next.config.js` `` file in the root directory of your project, if you don't already have one, for the inclusion of this plugin.

Below is an example of the setup code you would need in your `` `next.config.js` `` file. You can find more information about the setup here for NextJS as well as here.

```
// next.config.js
module.exports =
   ({ enabled = true, openAnalyzer = true } = {}) =>
       (nextConfig = {}) => {
         return Object.assign({}, nextConfig, {
           webpack(config, options) {
             if (enabled) {
               const { BundleAnalyzerPlugin } =
require('webpack-bundle-analyzer')
               config.plugins.push(
                   new BundleAnalyzerPlugin({
                     analyzerMode: "static",
                     openAnalyzer,
                     generateStatsFile: true,
                     statsFilename: !nextRuntime
                         ? "stats.client.json"
                         : `stats.${nextRuntime}.json`,
                     reportFilename: !options.nextRuntime
                         ? `./analyze/client.html`
                         : `../${options.nextRuntime === "nodejs" ?
"../" : ""}analyze/${
                             options.nextRuntime
                         }.html`,
                   })
               );
             }

             if (typeof nextConfig.webpack === 'function') {
               return nextConfig.webpack(config, options)
             }
             return config
           },
         })
       }
```

Now, when you build your NextJS project using the build command, the plugin will generate the interactive treemap and open it in your browser.

Interpreting the Treemap

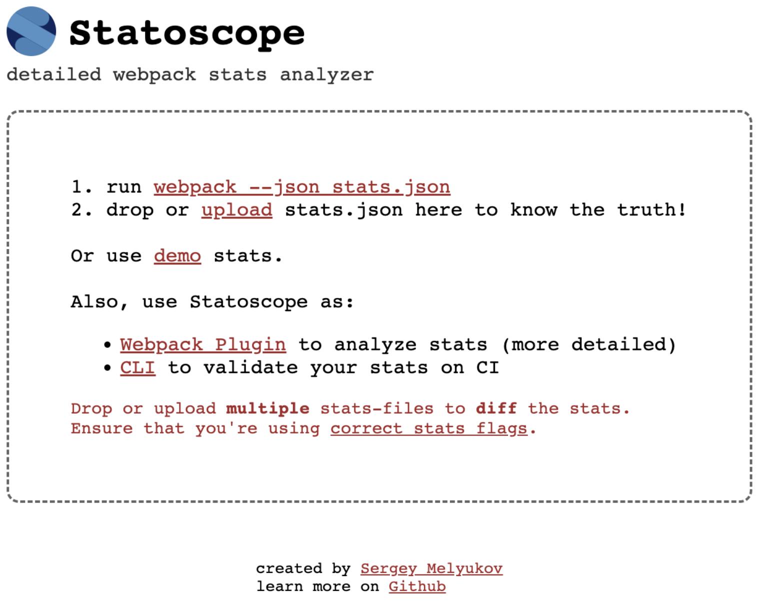
The interactive treemap that's generated provides you with a visual representation of your bundle's size and structure. Each nested rectangle in the treemap represents a module or dependency, with larger rectangles indicating larger modules.
It's important to pay attention to the way the treemap is organized and its hierarchy. It can give you insight into the overall structure and the relationships between modules in your project. The hierarchy in the treemap also enables you to analyze your bundle at different levels of granularity, from broad entry points and packages down to specific modules and files.

By navigating through the treemap and examining the hierarchical structure, you can identify the components of your application that contribute the most to the overall bundle size, and target them for optimization.

By hovering over a module, a tooltip appears, displaying additional information such as the module's size, name, and path. This, along with the search feature in the left panel, is very helpful when looking for specific modules or verifying if a module has been successfully removed from your project. You are also able to change the view of the Treemap output to show either Stat, Parsed or Gzipped sizes, which can be valuable to see as the gzip size is really what we care about most when it comes to file-transfer speed.

(INSERT IMAGE)

Analyzing and Identifying Unused Imports with Statoscope



Statoscope
detailed webpack stats analyzer

1. run webpack --json stats.json
2. drop or upload stats.json here to know the truth!

Or use demo stats.

Also, use Statoscope as:

- Webpack Plugin to analyze stats (more detailed)
- CLI to validate your stats on CI

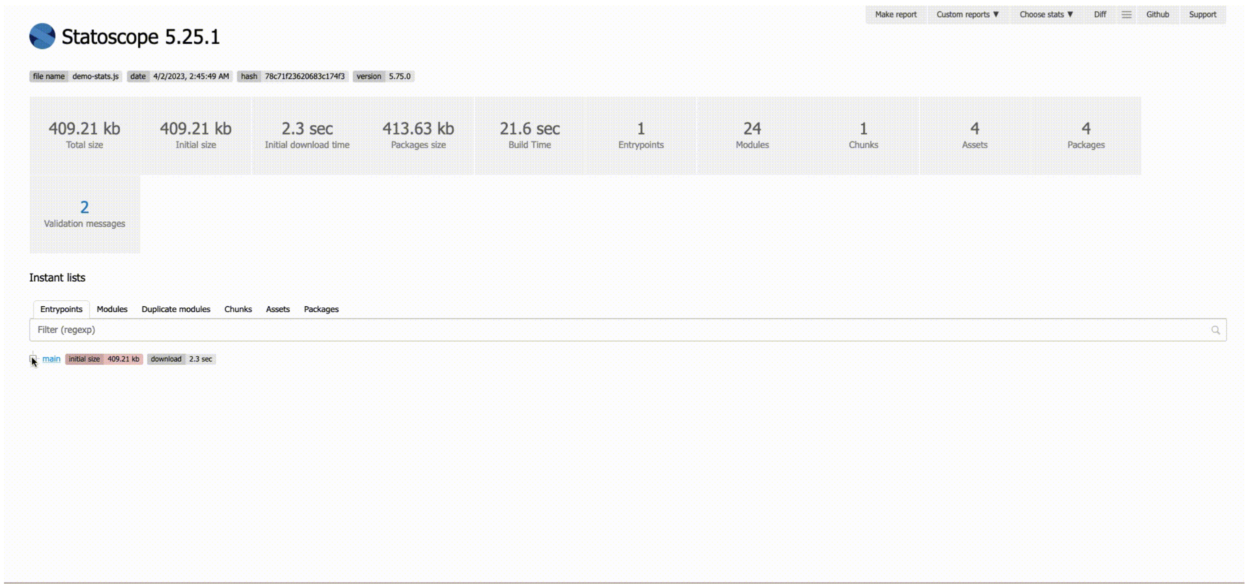
Drop or upload **multiple** stats-files to **diff** the stats.
Ensure that you're using correct stats flags.

created by Sergey Melyukov
learn more on Github

Statoscope is another tool used for analyzing and visualizing your bundle. In order to use this tool, go to statoscope.tech and simply drag and drop your `stats.json` file that gets generated from running the bundle analyzer. It will then produce a very detailed report that you can navigate through to further investigate your bundle.

The first tab you see is **Entrypoints** which shows the starting points of your application from where Webpack begins its bundling process. From there, you can navigate into one of those entry points and then into the specific module you are looking for. You'll then see the **Reasons** and **Issuer Path** sections. Reasons can include various causes for the module being included, such as explicit import statements or dependencies required by another module.

This gif below highlights navigating through the report generated on Statoscope.



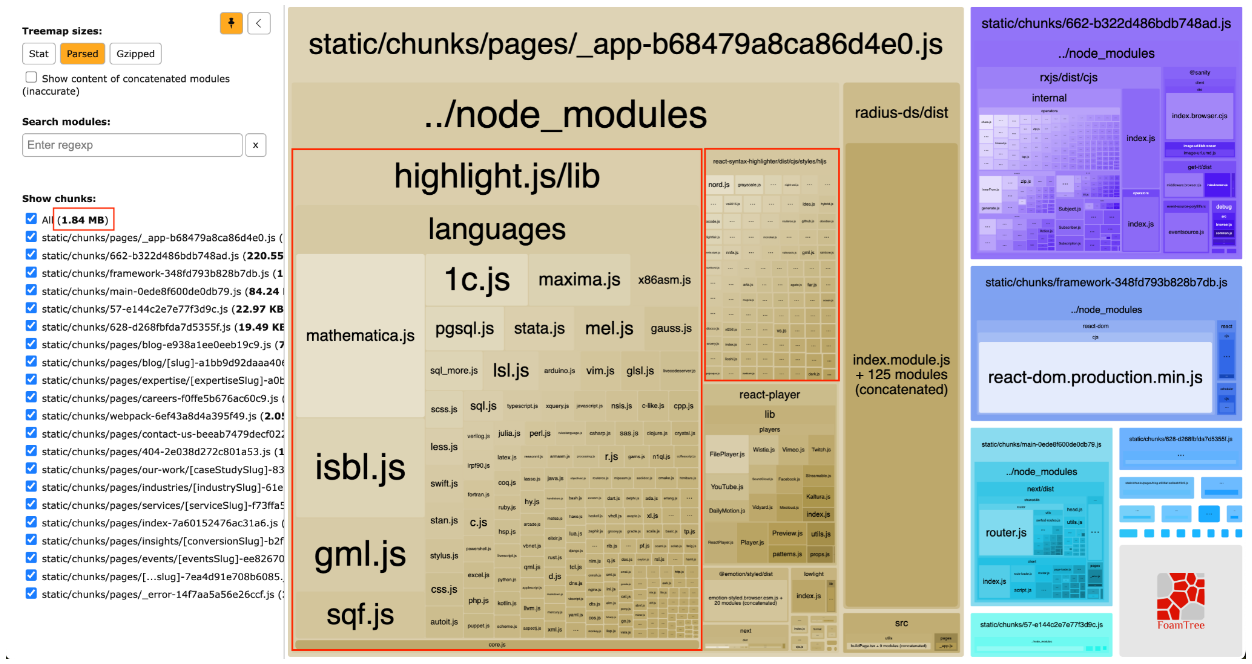
The Issuer Path, that is nested within Reasons on the other hand, shows the chain of modules that led to the inclusion of a specific module, effectively presenting a list of parent modules that required or imported the module in question. This information helps you trace back to the root entry point responsible for the inclusion of the specified module. This will allow you to know exactly where a module is imported so you can go about removing it safely.

While Webpack Bundle Analyzer is excellent at identifying large, duplicated, or unused modules, it doesn't provide a clear picture of where or why these modules are included in your code. This is where Statoscope shines, as it offers detailed information on the path and reasons for module imports within your codebase.

Removing all cases of where that unwanted module has been imported in your project is what you need to do next. After making these changes to your codebase, you can then build your app again, rerunning the bundle analyzer. You'll want to look at the generated Treemap again to verify that your bundle size has the intended outcome you want. If your goal was to remove a specific module for example, use the search feature to ensure that the module longer is included in your bundle. Similarly, if you're replacing one module with another, ensure that you're seeing the changes. This iterative approach allows you to track your progress and verify that you're optimizing your bundle in the way you want.

A Short Case Study

When applying these tools and techniques to our own Rangle website project, we were able to identify an area for optimization that reduced our bundle size by almost half. We noticed that one of the libraries we were importing was quite large and responsible for a large chunk of our bundle. In the first screenshot below, you can see the highlight.js library, outlined in red, and all of the files it's importing for various languages. Additionally, this library is not imported directly into our code anywhere but only used by another library we use - react-syntax-highlighter, also outlined. We use this library to properly show and format code snippets in blogs for example, exactly like the ones you've seen above in this article. However, through our content management system that we use to upload our content, we only support 5 code languages. This is a perfect example of a library bulking up your bundle and is a great area for optimizing.



Treemap sizes:

Stat [✓ Parsed] Gzipped

☐ Show content of concatenated modules (inaccurate)

Search modules:

Enter regexp x

Show chunks:

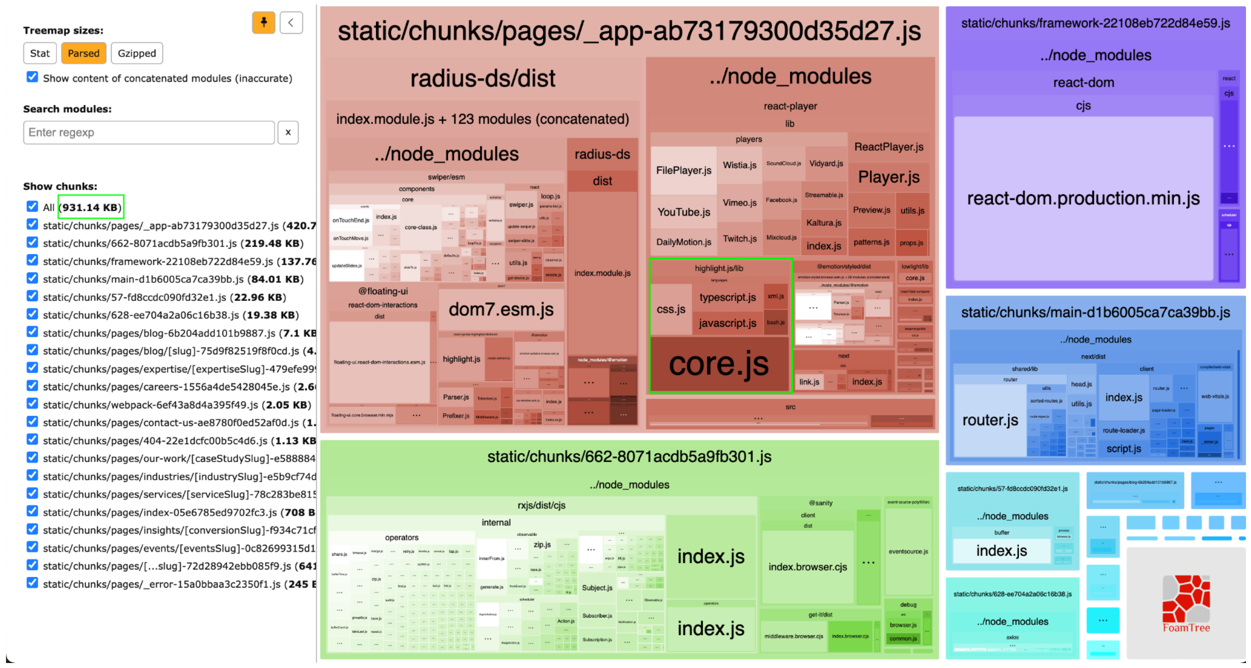
- ✓ All (**1.84 MB**)
- ✓ static/chunks/pages/_app-b68479a8ca86d4e0.js (
- ✓ static/chunks/662-b322d486bdb748ad.js (**220.55**
- ✓ static/chunks/framework-348fd793b828b7db.js (**1**
- ✓ static/chunks/main-0ede8f600de0db79.js (**84.24**
- ✓ static/chunks/57-e144c2e7e77f3d9c.js (**22.97 KB**
- ✓ static/chunks/628-d268fbfda7d5355f.js (**19.49 KB**
- ✓ static/chunks/pages/blog-e938a1ee0eeb19c9.js (**7**
- ✓ static/chunks/pages/blog/[slug]-a1bb9d92daaa40
- ✓ static/chunks/pages/expertise/[expertiseSlug]-a0b
- ✓ static/chunks/pages/careers-f0ffe5b676ac60c9.js (
- ✓ static/chunks/webpack-6ef43a8d4a395f49.js (**2.0**
- ✓ static/chunks/pages/contact-us-beeab7479decf02
- ✓ static/chunks/pages/404-2e038d272c801a53.js (**1**
- ✓ static/chunks/pages/our-work/[caseStudySlug]-83
- ✓ static/chunks/pages/industries/[industrySlug]-61e
- ✓ static/chunks/pages/services/[serviceSlug]-f73ffa5
- ✓ static/chunks/pages/index-7a60152476ac31a6.js (
- ✓ static/chunks/pages/insights/[conversionSlug]-b2f
- ✓ static/chunks/pages/events/[eventsSlug]-ee82670
- ✓ static/chunks/pages/[...slug]-7ea4d91e708b6085.
- ✓ static/chunks/pages/_error-14f7aa5a56e26ccf.js (

static/chunks/pages/_app-b68479a8ca86d4e0.js
../node_modules
highlight.js/lib
languages
mathematica.js
1c.js
maxima.js
x86asm.js
pgsql.js
stata.js
mel.js
gauss.js
isbl.js
gml.js
sqf.js
react-syntax-highlighter/dist/cjs/styles/hljs
react-player
radius-ds/dist
index.module.js + 125 modules (concatenated)
static/chunks/662-b322d486bdb748ad.js
rxjs/dist/cjs
static/chunks/framework-348fd793b828b7db.js
react-dom.production.min.js
static/chunks/main-0ede8f600de0db79.js
router.js
FoamTree

By simply changing the import in our code from `import SyntaxHighlighter from 'react-syntax-highlighter'` which imports everything from that library. By changing this to only import the necessary languages like the following example,

```
import javascript from 'react-syntax-highlighter/dist/cjs/languages/hljs/javascript'
```

we are able to reduce our bundle size by quite a bit.


static/chunks/pages/_app-ab73179300d35d27.js
radius-ds/dist
../node_modules
index.module.js + 123 modules (concatenated)
react-player
core.js
dom7.esm.js
static/chunks/framework-22108eb722d84e59.js
react-dom
react-dom.production.min.js
static/chunks/main-d1b6005ca7ca39bb.js
router.js
static/chunks/662-8071acdb5a9fb301.js
rxjs/dist/cjs
index.js
index.browser.cjs
FoamTree

Treemap sizes:

Stat [✓ Parsed] Gzipped

- [x] Show content of concatenated modules (inaccurate)

Search modules:

Enter regexp ×

Show chunks:

- [x] All (**931.14 KB**)
- [x] static/chunks/pages/_app-ab73179300d35d27.js (**420.7**
- [x] static/chunks/662-8071acdb5a9fb301.js (**219.48 KB**)
- [x] static/chunks/framework-22108eb722d84e59.js (**137.76**
- [x] static/chunks/main-d1b6005ca7ca39bb.js (**84.01 KB**)
- [x] static/chunks/57-fd8ccdc090fd32e1.js (**22.96 KB**)
- [x] static/chunks/628-ee704a2a06c16b38.js (**19.38 KB**)
- [x] static/chunks/pages/blog-6b204add101b9887.js (**7.1 KB**
- [x] static/chunks/pages/blog/[slug]-75d9f82519f8f0cd.js (**4.**
- [x] static/chunks/pages/expertise/[expertiseSlug]-479efe99
- [x] static/chunks/pages/careers-1556a4de5428045e.js (**2.6**
- [x] static/chunks/webpack-6ef43a8d4a395f49.js (**2.05 KB**)
- [x] static/chunks/pages/contact-us-ae8780f0ed52af0d.js (**1.**
- [x] static/chunks/pages/404-22e1dcfc00b5c4d6.js (**1.13 KB**
- [x] static/chunks/pages/our-work/[caseStudySlug]-e588884
- [x] static/chunks/pages/industries/[industrySlug]-e5b9cf74d
- [x] static/chunks/pages/services/[serviceSlug]-78c283be815
- [x] static/chunks/pages/index-05e6785ed9702fc3.js (**708 B**
- [x] static/chunks/pages/insights/[conversionSlug]-f934c71cf
- [x] static/chunks/pages/events/[eventsSlug]-0c82699315d1
- [x] static/chunks/pages/[...slug]-72d28942ebb085f9.js (**641**
- [x] static/chunks/pages/_error-15a0bbaa3c2350f1.js (**245 B**

After making these changes to our code, we then ran the bundle analyzer again and can see the impact that it has had on our bundle size. We went from a total (parsed) size of 1.84mb down to 931.14m - that's a reduction of almost half!

We can see from this short case study that inspecting the treemap to identify large modules, we're able to better understand what we're using in our code and how by simply changing how we're importing what we need from this library, we are able to significantly reduce our bundle size.

Summary

As we've explored together in this article, keeping an eye on your bundle size and continuously optimizing it is important for creating fast and efficient web applications. By using tools like Webpack Bundle Analyzer and Statoscope, we can gain valuable insights into our codebase, which can help us identify areas for improvement. These tools not only help us pinpoint large, duplicated, or unused modules, but also provide information on where and why these modules are included in our code. Being able to visualize how your project is made up of various libraries and modules, was a step that really helped me gain an understanding and appreciation for page load times and how we can optimize that.

Thinking about how we can integrate these tools into our DevOps pipeline can help us with this goal of optimizing. By setting up automated alerts, we can be notified when our bundle size exceeds a certain threshold, much like we do for test coverage. This allows us to take immediate action to optimize our code and maintain high performance standards. Integrating these tools with CI/CD processes helps establish a proactive approach to performance optimization, ensuring that we consistently deliver a fast and efficient user experience.wis