Reduce JavaScript payloads with tree shaking

- What is tree shaking?
- Finding opportunities to shake a tree
- Keeping Babel from transpiling ES6 modules to CommonJS modules
- Keeping side effects in mind
- Importing only what's needed
- Go shake some trees!

Today's web applications can get pretty big, especially the JavaScript part of them. As of mid-2018, HTTP Archive puts the median transfer size of JavaScript on mobile devices at approximately 350 KB. And this is just transfer size! JavaScript is often compressed when sent over the network, meaning that the *actual* amount of JavaScript is quite a bit more after the browser decompresses it. That's important to point out, because as far as resource *processing* is concerned, compression is irrelevant. 900 KB of decompressed JavaScript is still 900 KB to the parser and compiler, even though it may be roughly 300 KB when compressed.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/VytWruOnBN3nHJnSm18w.svg", alt="A diagram illustrating the process of downloading, decompressing, parsing, compiling, and executing JavaScript.", width="600", height="106" %}
> The process of downloading and running JavaScript. Note that even though the transfer size of the script is 300 KB compressed, it is still 900 KB worth of JavaScript that must be parsed, compiled, and executed.

JavaScript is an expensive resource to process. Unlike images which only incur relatively trivial decode time once downloaded, JavaScript must be parsed, compiled, and then finally executed. Byte for byte, this makes JavaScript more expensive than other types of resources.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/rEdnN5fuzBC6DOjUqgEL.png", alt="A diagram comparing the processing time of 170 KB of JavaScript versus an equivalently sized JPEG image. The JavaScript resource is far more resource-intensive byte for byte than the JPEG.", width="800", height="504" %}

> The processing cost of parsing/compiling 170 KB of JavaScript vs decode time of an equivalently sized JPEG. (source).

While improvements are continually being made to improve the efficiency of JavaScript engines, improving JavaScript performance is—as always—a task for developers.

To that end, there are techniques to improve JavaScript performance. Code splitting, is one such technique that improves performance by partitioning application JavaScript into chunks, and serving those chunks to only the routes of an application that need them.

While this technique works, it doesn't address a common problem of JavaScript-heavy applications, which is the inclusion of code that's never used. Tree shaking attempts to solve this problem.

What is tree shaking?

Tree shaking is a form of dead code elimination. The term was popularized by Rollup, but the concept of dead code elimination has existed for some time. The concept has also found purchase in webpack, which is demonstrated in this article by way of a sample app.

The term "tree shaking" comes from the mental model of your application and its dependencies as a tree-like structure. Each node in the tree represents a dependency that provides distinct functionality for your app. In modern apps, these dependencies are brought in via static `import` statements like so:

```
// Import all the array utilities!
import arrayUtils from "array-utils";
```

{% Aside %} Note: If you're not sure what ES6 modules are, this explainer at Pony Foo is worth a look. This guide assumes you have working knowledge of how ES6 modules work, so if you don't know anything about them, give that article a read! {% endAside %}

When an app is young—a sapling, if you will—it may have few dependencies. It's also using most—if not all—the dependencies you add. As your app matures, however, more dependencies can get added. To compound matters, older dependencies fall out of use, but may not get pruned from your codebase. The end result is that an app ends up shipping with a lot of unused JavaScript. Tree shaking addresses this by taking advantage of how static `import` statements pull in specific parts of ES6 modules:

```
// Import only some of the utilities!
import { unique, implode, explode } from "array-utils";
```

The difference between this `import` example and the previous one is that rather than importing *everything* from the `"array-utils"` module—which could be a lot of code)—this example imports only specific parts of it. In dev builds, this doesn't change anything, as the entire module gets imported regardless. In production builds, webpack can be configured to "shake" off exports from ES6 modules that weren't explicitly imported, making those production builds smaller. In this guide, you'll learn how to do just that!

Finding opportunities to shake a tree

For illustrative purposes, a sample one-page app is available that demonstrates how tree shaking works. You can clone it and follow along if you like, but we'll cover every step of the way together in this guide, so cloning isn't necessary (unless hands-on learning is your thing).

The sample app is a searchable database of guitar effect pedals. You enter a query and a list of effect pedals will appear.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/bBmp460zNYmTVeAgQhw5.png", alt="A screenshot of a sample one page application for searching a database of guitar effect pedals.", width="800", height="673" %}
> A screenshot of the sample app.

The behavior that drives this app is separated into vendor (i.e., Preact and Emotion) and app-specific code bundles (or "chunks", as webpack calls them):

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/oreVh5rQcXxIDenvQxE7.png", alt="A screenshot of two application code bundles (or chunks) shown in the network panel of Chrome's DevTools.", width="800", height="282" %}
> The app's two JavaScript bundles. These are uncompressed sizes.

The JavaScript bundles shown in the figure above are production builds, meaning they're optimized through uglification. 21.1 KB for an app-specific bundle isn't bad, but it should be noted that no tree shaking is occurring whatsoever. Let's look at the app code and see what can be done to fix that.

{% Aside %} If you don't care for a long-winded explanation, and want to dive into code, go ahead and check out the `tree-shake` branch in the app's GitHub repo. You can also diff this branch to see exactly what was changed to make tree shaking work! {% endAside %}

In any application, finding tree shaking opportunities are going to involve looking for static `import` statements. Near the top of the main component file, you'll see a line like this:

```
import * as utils from "../../utils/utils";
```

You can import ES6 modules in a variety of ways, but ones like this should get your attention. This specific line says "`import` *everything* from the `utils` module, and put it in a namespace called `utils`." The big question to ask here is, "just how much *stuff* is in that module?"

If you look at the `utils` module source code, you'll find there's about 1,300 lines of code.

Do you *need* all that stuff? Let's double check by searching the main component file that imports the `utils` module to see how many instances of that namespace come up.

> {% Img src="image/jL3OLOhcWUQDnR4XjewLBx4e3PC3/Grk6jcj6IiMHIiJPqiEM.png", alt="A screenshot of a search in a text editor for 'utils.', returning only 3 results.", width="800", height="117" %}
> The `utils` namespace we've imported tons of modules from is only invoked three times within the main component file.

As it turns out, the `utils` namespace appears in only three spots in our application—but for what functions? If you take a look at the main component file again, it appears to be only one function, which is `utils.simpleSort`, which is used to sort the search results list by a number of criteria when the sorting dropdowns are changed:

```
if (this.state.sortBy === "model") {
  // `simpleSort` gets used here...
  json = utils.simpleSort(json, "model", this.state.sortOrder);
} else if (this.state.sortBy === "type") {
  // ..and here...
  json = utils.simpleSort(json, "type", this.state.sortOrder);
} else {
  // ..and here.
  json = utils.simpleSort(json, "manufacturer", this.state.sortOrder);
}
```

Out of a 1,300 line file with a bunch of exports, only one of them is used. This results in shipping a lot of unused JavaScript.

{% Aside %} Note: This project is purposefully kept simple, so it's a bit easier in this case to find out where the bloat is coming from. In large projects with many modules, however, things get more complicated. Tools such as Webpack Bundle Analyzer and source-map-explorer may provide further insight. {% endAside %}

While this example app is admittedly a bit contrived, it doesn't change the fact that this synthetic sort of scenario resembles actual optimization opportunities you may encounter in a production web app. Now that you've identified an opportunity for tree shaking to be useful, how is it actually done?

Keeping Babel from transpiling ES6 modules to CommonJS modules

Babel is an indispensable tool, but it may make the effects of tree shaking a bit more difficult to observe. If you're using `@babel/preset-env`, Babel *may* transform ES6 modules into more widely compatible CommonJS modules—that is, modules you `require` instead of `import`.

Because tree shaking is more difficult to do for CommonJS modules, webpack won't know what to prune from bundles if you decide to use them. The solution is to configure `@babel/preset-env` to explicitly leave ES6 modules alone. Wherever you configure Babel—be it in `babel.config.js` or `package.json`—this involves adding a little something extra:

```
// babel.config.js
export default {
  presets: [
    [
      "@babel/preset-env", {
        modules: false
      }
    ]
  ]
}
```

Specifying `modules: false` in your `@babel/preset-env` config gets Babel to behave as desired, which allows webpack to analyze your dependency tree and shake off unused dependencies.

Keeping side effects in mind

Another aspect to consider when shaking dependencies from your app is whether your project's modules have side effects. An example of a side effect is when a function modifies something outside of its own scope, which is a *side effect* of its execution:

```
let fruits = ["apple", "orange", "pear"];

console.log(fruits); // (3) ["apple", "orange", "pear"]

const addFruit = function(fruit) {
  fruits.push(fruit);
};

addFruit("kiwi");

console.log(fruits); // (4) ["apple", "orange", "pear", "kiwi"]
```

In this example, `addFruit` produces a side effect when it modifies the `fruits` array, which is outside its scope.

Side effects also apply to ES6 modules, and that matters in the context of tree shaking. Modules that take predictable inputs and produce equally predictable outputs without modifying anything outside of their own scope are dependencies that can be safely dropped if we're not using them. They're self-contained, *modular* pieces of code. Hence, "modules".

Where webpack is concerned, a hint can be used to specify that a package and its dependencies are free of side effects by specifying `"sideEffects": false` in a project's `package.json` file:

```
{
  "name": "webpack-tree-shaking-example",
  "version": "1.0.0",
  "sideEffects": false
}
```

Alternatively, you can tell webpack which specific files are not side effect-free:

```
{
  "name": "webpack-tree-shaking-example",
  "version": "1.0.0",
  "sideEffects": [
    "./src/utils/utils.js"
  ]
}
```

In the latter example, any file that isn't specified will be assumed to be free of side effects. If you don't want to add this to your `package.json` file, you can also specify this flag in your webpack

config via `module.rules`.

Importing only what's needed

After instructing Babel to leave ES6 modules alone, a slight adjustment to our `import` syntax is required to bring in only the functions needed from the `utils` module. In this guide's example, all that's needed is the `simpleSort` function:

```
import { simpleSort } from "../../utils/utils";
```

Because only `simpleSort` is being imported instead of the entire `utils` module, every instance of `utils.simpleSort` will need to changed to `simpleSort`:

```
if (this.state.sortBy === "model") {
  json = simpleSort(json, "model", this.state.sortOrder);
} else if (this.state.sortBy === "type") {
  json = simpleSort(json, "type", this.state.sortOrder);
} else {
  json = simpleSort(json, "manufacturer", this.state.sortOrder);
}
```

This should be all that's needed for tree shaking to work in this example. This is the webpack output before shaking the dependency tree:

```
                 Asset      Size  Chunks             Chunk Names
js/vendors.16262743.js  37.1 KiB       0  [emitted]  vendors
   js/main.797ebb8b.js  20.8 KiB       1  [emitted]  main
```

This is the output *after* tree shaking is successful:

```
                 Asset      Size  Chunks             Chunk Names
js/vendors.45ce9b64.js  36.9 KiB       0  [emitted]  vendors
   js/main.559652be.js  8.46 KiB       1  [emitted]  main
```

While both bundles shrank, it's really the `main` bundle that benefits most. By shaking off the unused parts of the `utils` module, the `main` bundle shrinks by about 60%. This not only lowers the amount of time the script takes to the download, but processing time as well.

Go shake some trees!

Whatever mileage you get out of tree shaking depends on your app and its dependencies and architecture. Try it! If you know for a fact you haven't set up your module bundler to perform this optimization, there's no harm trying and seeing how it benefits your application.

You may realize a significant performance gain from tree shaking, or not much at all. But by configuring your build system to take advantage of this optimization in production builds and selectively importing only what your application needs, you'll be proactively keeping your application bundles as small as possible.

Special thanks to Kristofer Baxter, Jason Miller, Addy Osmani, Jeff Posnick, Sam Saccone, and Philip Walton for their valuable feedback, which significantly improved the quality of this article.

□□□ #2
□ □□□ □□□ 2026-10-31 22:37:33 UTC
□ □□□ □□□ 2026-10-31 22:37:35 UTC