Why Webpack 2's Tree Shaking is not as effective as you think

A rundown on how tree shaking works and why it's not delivering the results people expect

Tamás Sallai

An Aha! moment, delivered to your inbox every week. Check out the JS Tips & Tricks Newsletter!

As WebPack 2 barrels forward, Tree Shaking — or more technically, the removal of unused exports using static analysis — is finding its way to the mainstream. Developers are putting their hopes high, as it promises to solve the pressing problem of bloated packages. Usually, only a fraction of code is actually needed from each dependency but their entire codebase is bundled, increasing the size.

The promise of Tree Shaking is to remove this clutter during the build, allowing developers to add dependencies without worrying about the user experience.

But even a quick search indicates that it might not work that well in practice.

What causes this discrepancy? Let's find out!

Tree Shaking basics

Let's consider an example on how it works. As it optimizes exports, add a **lib.js** which exposes two variables:

```
export const a = "A_VAL_ES6";
export const b = "B_VAL_ES6";
```

Then add an **entry.js**, which imports them but then uses only *a*:

```
import {a, b} from "./lib.js";

console.log(a);
```

Since *b* is not used in *entry.js*, it can be removed, but it's nothing new; UglifyJS already does that.

But then the export *b* in *lib.js* is also no longer used, therefore it can be removed too. Inspecting the bundle, *A_VAL_ES6* is present, but *B_VAL_ES6* is not.

Rollup

When talking about Tree Shaking, I have to mention rollup.js. As far as I know, it was the first full-fledged bundler that supported the concept of exports removal. It is available for more than a year now, but I'm yet to hear the success stories of massive amounts of saved bytes.

Test setup

The code is available if you want to reproduce the results.

Lodash-es is the ES6-compatible version of Lodash. It has the same functionality, but instead of exporting in the UMD module format, it uses ES6 modules. Its main simply re-exports every module, and as Lodash is a "utility belt" collection, it should not matter if an individual part or the whole bundle is included.

Therefore, `import {map} from "lodash-es";` and `import map from "lodash-es/map";` should be equivalent.

To set up a testbed, add a **package.json** with dependencies to WebPack, Babel, and lodash-es:

```
{
 ...
 "devDependencies": {
  "webpack": "2.2.0",
  "babel-core": "6.16.0",
  "babel-loader": "6.2.7",
  "babel-preset-es2015": "6.22.0",
  "lodash-es": "4.17.4"
 },
 ...
}
```

Then add a **webpack.config.js** with some minimal WebPack bootstrapping:

```
...
 loader: 'babel-loader',
 options: {
  presets: [['es2015', {modules: false}]]
 }
 ...
 plugins: [
  new webpack.LoaderOptionsPlugin({
   minimize: true
  }),
  new webpack.optimize.UglifyJsPlugin({
  })
 ]
...
```

Note the `{modules: false}` part. Babel converts everything to CommonJS by default, and while it is a great way to achieve the widest compatibility, it prevents exports analysis. This config turns it off, as WebPack supports native ES6 modules from version 2.

After the boilerplate is in place, add an **entry.js** with the import:

```
import {map} from "lodash-es";

console.log(map([1, 2], function(i) {return i + 1}));
```

Running `npm run build` and inspecting the bundle, it weights **139.224** bytes.

Then change the import to the individual module:

```
import map from "lodash-es/map";
```

The result is a mere **25.531** bytes.

This indicates that Tree Shaking is less effective in a real life situation than optimizing by hand.

Modules

The code is available to test drive both module formats.

CommonJS

To understand the limitations of static analysis, we need to look into the differences between CommonJS and ES6 modules.

Note: Libraries mostly use UMD for better compatibility, which is usually interpreted as AMD and not CommonJS. But since they behave the same to the extent of this rundown, and CommonJS is, well, more common, I'll use that as an illustration.

In a CommonJS environment, the `exports` object is all that matters. After running the code, the contents of that object will be exported.

For simple cases, it's common to simply set the properties:

```
exports.a = "A_VAL_COMMONJS";
exports.b = "B_VAL_COMMONJS";
```

This will export both *a* and *b*.

But CommonJS can set the *exports* dynamically:

```
for(let i = 0; i < 5; i++) {
 exports["i" + i] = i;
}
```

This will export *i0*, *i1*, *i2*, *i3*, and *i4*.

It even does not require exports to be deterministic:

```
if (Math.random() < 0.5) {
 exports.rand = "RANDOM";
}
```

This will export *rand* based on luck.

This dynamic nature of CommonJS nicely fits the dynamic nature of JavaScript itself but completely defeats static analysis.

As an illustration, add a **lib_commonjs.js** that exports two values:

```
exports.a = "A_VAL_COMMONJS";
exports.b = "B_VAL_COMMONJS";
```

Then modify the **entry.js** to import both, but only use one:

```
import {a as a_commonjs, b as b_commonjs} from "./lib_commonjs.js";

console.log("Hello world:" + a_commonjs);
```

After bundling and inspecting the results, both `A_VAL_COMMONJS` and `B_VAL_COMMONJS` are present in the file; nothing was removed.

This indicates that Tree Shaking is not working for any CommonJS module.

Since many libraries exports primarily in AMD/CommonJS, no big gains can be expected until that changes.

ES6

ES6 modules are static in nature. They must be deterministic and no dynamic exports are allowed. This opens the way for static analysis.

To see it in action, we already have a **lib.js** that exports two values:

```
export const a = "A_VAL_ES6";
export const b = "B_VAL_ES6";
```

Then modify the **entry.js** to import both, but use only one:

```
import {a as a_es6, b as b_es6} from "./lib.js";

console.log("Hello world:" + a_es6);
```

Inspecting the bundle, only `A_VAL_ES6` is there, *b*'s value is removed.

The problem

The root of the problem is **side effects**. In many use-cases importing a library does not necessarily result in a bounded piece of code that is completely separated from the rest of the app. For example, using the css-loader to `import css from 'file.css';`, the contents of the variable is not important; but the style is already applied to the document.

If WebPack would remove all unused dependencies regardless of side effects, imports like this would break. As a consequence, side effects must be retained. They provide expected functionality when they write the console, add a style tag or modify the HTML in other ways, assign global variables, and so on.

But there is another category of code, improperly identified as side effects. An *Object.freeze* does not modify anything, so as all function calls that are pure. These should be also be removed.

As an illustration, modify the **lib.js** to export a frozen version of the value:

```
export const b = Object.freeze("B_VAL_ES6");
```

This minute change results in the inclusion of `B_VAL_ES6` in the bundle.

Similarly, a simple function call also triggers this behavior:

```
Object.freeze(b);
```

This results in the inclusion of almost the whole library, even though there are no side effects.

Side effects are hard to identify properly. There is some ongoing work, though.

Conclusion

Bundlers choose the safe path and rather include unneeded code than break the app. This results in bigger bundles and less effective code.

But Tree Shaking might still save a few bytes, and it's a good thing; everything that lowers the bandwidth requirements of a webapp is a change in the positive direction. But in practice, its efficacy is a lot less than the expectations.

People are working on solutions to better identify side effects. But their dynamic nature prevents a solution that works for every codebase. Instead, there are certain heuristics that can be used. There is also a proposal to annotate pure functions, but community-wide support is unlikely to happen anytime soon.

Tree Shaking might help a bit, but small bundles are still far away.

Update

As jdalton pointed out, babel-plugin-lodash and lodash-webpack-plugin cherry-pick what you actually use from Lodash. To save some bytes, they are definitely worth checking out.

Webpack 3 Webdev 9

07 February 2017

You won't miss out future posts by subscribing to our Newsletter and RSS feed!

« Next Previous »

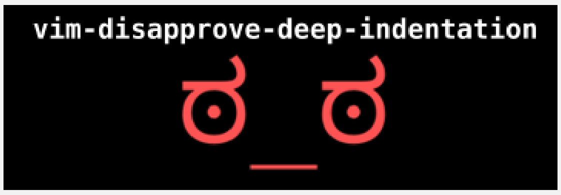
Related posts

- **What to expect when you decide to migrate from Javascript to Typescript** 27 Feb 2018
- **Getting started with Browserify** 31 Jan 2017

- **Best practices on how to work with collections in Javascript** 13 Mar 2018
- **Profile-based optimization techniques in the JVM** 01 Mar 2017
- **JSPM basics and review** 17 Jan 2017
- **More readable Javascript without variables** 17 May 2016
- **Walkthrough for a TDD Kata in Eclipse** 27 Oct 2015
- **Why Bower is still relevant** 21 Dec 2016
- **JVM JIT optimization techniques** 27 May 2016
- **Parallel Processing in JS** 09 Aug 2016

All posts »

Some of our other projects



Newsletter

Get early access, prompt notifications and future exclusive content by signing up to our newsletter!

Email Address

Subscribe

Recent posts

- **Let's Encrypt hooks use cases**
  26 Jun 2018
- **Working with the system clipboard in Vim**
  12 Jun 2018
-

- **When to use Let's Encrypt's webroot and standalone authorization**
  05 Jun 2018
- **Let's Encrypt tips**
  29 May 2018
- **Getting started with Let's Encrypt**
  22 May 2018

All posts »

Tags

Javascript 30 | Java 21 | Test 12 | Vim 11 | Webdev 9 | Maven 9 | Integration-test 7 | Eclipse 6 | Linux 6 | Docker 6

All tags »

Archive

- **2018 (16)**
- **2017 (25)**
- **2016 (24)**
- **2015 (28)**
- **2014 (7)**

All posts »

Statistics

To make sure we don't lose counts:

100	Posts
2	Authors

Feeds