Sponsor webpack and get apparel from the official shop or get stickers here! All proceeds go to our open collective!

Tree Shaking

Tree shaking is a term commonly used in the JavaScript context for dead-code elimination. It relies on the static structure of ES2015 module syntax, i.e. `import` and `export`. The name and concept have been popularized by the ES2015 module bundler rollup.

The webpack 2 release came with built-in support for ES2015 modules (alias *harmony modules*) as well as unused module export detection. The new webpack 4 release expands on this capability with a way to provide hints to the compiler via the `"sideEffects"` `package.json` property to denote which files in your project are "pure" and therefore safe to prune if unused.

The remainder of this guide will stem from Getting Started. If you haven't read through that guide already, please do so now.

Add a Utility

Let's add a new utility file to our project, `src/math.js`, that exports two functions:

project

```
webpack-demo
|- package.json
|- webpack.config.js
|- /dist
  |- bundle.js
  |- index.html
|- /src
  |- index.js
+ |- math.js
|- /node_modules
```

src/math.js

```
export function square(x) {
  return x * x;
}

export function cube(x) {
  return x * x * x;
}
```

You could need to set the development mode to be sure that bundle is not minified:

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
- }
+ },
+ mode: "development"
};
```

With that in place, let's update our entry script to utilize one of these new methods and remove `lodash` for simplicity:

src/index.js

```diff
- import _ from 'lodash';
+ import { cube } from './math.js';

  function component() {
-   var element = document.createElement('div');
+   var element = document.createElement('pre');

-   // Lodash, now imported by this script
-   element.innerHTML = _.join(['Hello', 'webpack'], ' ');
+   element.innerHTML = [
+     'Hello webpack!',
+     '5 cubed is equal to ' + cube(5)
+   ].join('\n\n');

    return element;
  }

  document.body.appendChild(component());
```

Note that we **did not `import` the `square` method** from the `src/math.js` module. That function is what's known as "dead code", meaning an unused `export` that should be dropped. Now let's run our npm script, `npm run build`, and inspect the output bundle:

dist/bundle.js (around lines 90 - 100)

```js
/* 1 */
/***/ (function(module, __webpack_exports__, __webpack_require__) {
  'use strict';
  /* unused harmony export square */
  /* harmony export (immutable) */ __webpack_exports__['a'] = cube;
  function square(x) {
    return x * x;
  }

  function cube(x) {
    return x * x * x;
  }
});
```

Note the `unused harmony export square` comment above. If you look at the code below it, you'll notice that `square` is not being imported, however, it is still included in the bundle. We'll fix that in the next section.

Mark the file as side-effect-free

In a 100% ESM module world, identifying side effects is straightforward. However, we aren't there just yet, so in the mean time it's necessary to provide hints to webpack's compiler on the "pureness" of your code.

The way this is accomplished is the `"sideEffects"` package.json property.

```json
{
  "name": "your-project",
  "sideEffects": false
}
```

All the code noted above does not contain side effects, so we can simply mark the property as `false` to inform webpack that it can safely prune unused exports.

> *A "side effect" is defined as code that performs a special behavior when imported, other than exposing one or more exports. An example of this are polyfills, which affect the global scope and usually do not provide an export.*

If your code did have some side effects though, an array can be provided instead:

```json
{
  "name": "your-project",
  "sideEffects": [
    "./src/some-side-effectful-file.js"
  ]
}
```

The array accepts relative, absolute, and glob patterns to the relevant files. It uses micromatch under the hood.

Note that any imported file is subject to tree shaking. This means if you use something like `css-loader` in your project and import a CSS file, it needs to be added to the side effect list so it will not be unintentionally dropped in production mode:

```
{
  "name": "your-project",
  "sideEffects": [
    "./src/some-side-effectful-file.js",
    "*.css"
  ]
}
```

Finally, `"sideEffects"` can also be set from the `module.rules` configuration option.

Minify the Output

So we've cued up our "dead code" to be dropped by using the `import` and `export` syntax, but we still need to drop it from the bundle. To do that, we'll use the `-p` (production) webpack compilation flag to enable `UglifyJSPlugin`.

As of webpack 4, this is also easily toggled via the `"mode"` configuration option, set to `"production"`.

webpack.config.js

```
const path = require('path');

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist')
  },
- mode: "development"
+ mode: "production"
};
```

Note that the `--optimize-minimize` flag can be used to enable `UglifyJSPlugin` as well.

With that squared away, we can run another `npm run build` and see if anything has changed.

Notice anything different about `dist/bundle.js`? Clearly the whole bundle is now minified and mangled, but, if you look carefully, you won't see the `square` function included but will see a mangled version of the `cube` function (`function r(e){return e*e*e}n.a=r`). With minification and tree shaking our bundle is now a few bytes smaller! While that may not seem like much in this contrived example, tree shaking can yield a significant decrease in bundle size when working on larger applications with complex dependency trees.

Conclusion

So, what we've learned is that in order to take advantage of *tree shaking*, you must...

- Use ES2015 module syntax (i.e. `import` and `export`).
- Add a "sideEffects" property to your project's `package.json` file.
- Include a minifier that supports dead code removal (e.g. the `UglifyJSPlugin`).

You can imagine your application as a tree. The source code and libraries you actually use represent the green, living leaves of the tree. Dead code represents the brown, dead leaves of the tree that are consumed by autumn. In order to get rid of the dead leaves, you have to shake the tree, causing them to fall.

If you are interested in more ways to optimize your output, please jump to the next guide for details on building forproduction.

Further Reading

- webpack 4 beta — try it today!
- Debugging Optimization Bailouts
- Issue 6074 - Add support for more complex selectors for sideEffects

Contributors

MijaelWatts alexjoverm avant1 byzyk dmitriid gish

lumo10 pnevares probablyup simon04 zacanger