Decrease Front-end Size

By Ivan Akulov

Contracting software engineer · Blogging, open source, performance, UX

One of the first things to do when you're optimizing an application is to make it as small as possible. Here's how to do this with webpack.

Use the production mode (webpack 4 only)

Webpack 4 introduced the new `mode` flag. You could set this flag to `'development'` or `'production'` to hint webpack that you're building the application for a specific environment:

```
// webpack.config.js
module.exports = {
  mode: 'production',
};
```

Make sure to enable the `production` mode when you're building your app for production. This will make webpack apply optimizations like minification, removal of development-only code in libraries, and more.

Further reading

- What specific things the `mode` flag configures

Enable minification

Note: most of this is webpack 3 only. If you use webpack 4 with the production mode, the bundle-level minification is already enabled – you'll only need to enable loader-specific options.

Minification is when you compress the code by removing extra spaces, shortening variable names and so on. Like this:

```
// Original code
function map(array, iteratee) {
  let index = -1;
  const length = array == null ? 0 : array.length;
  const result = new Array(length);

  while (++index < length) {
    result[index] = iteratee(array[index], index, array);
  }
  return result;
}

↓

// Minified code
function map(n,r){let t=-1;for(const a=null==n?0:n.length,l=Array(a);++t<a;)l[t]=
```

Webpack supports two ways to minify the code: *the bundle-level minification* and *loader-specific options*. They should be used simultaneously.

Bundle-level minification

The bundle-level minification compresses the whole bundle after compilation. Here's how it works:

1. You write code like this:

   ```
   // comments.js
   import './comments.css';
   export function render(data, target) {
     console.log('Rendered!');
   }
   ```

2. Webpack compiles it into approximately the following:

   ```
   // bundle.js (part of)
   "use strict";
   Object.defineProperty(__webpack_exports__, "__esModule", { value: true });
   /* harmony export (immutable) */ __webpack_exports__["render"] = render;
   /* harmony import */ var __WEBPACK_IMPORTED_MODULE_0__comments_css__ = __web
   /* harmony import */ var __WEBPACK_IMPORTED_MODULE_0__comments_css_js___defa
   __webpack_require__.n(__WEBPACK_IMPORTED_MODULE_0__comments_css__);

   function render(data, target) {
   ```

```
  console.log('Rendered!');
}
```

3. A minifier compresses it into approximately the following:

```
// minified bundle.js (part of)
"use strict";function t(e,n){console.log("Rendered!")}
Object.defineProperty(n,"__esModule",{value:!0}),n.render=t;var o=r(1);r.n(o
```

In webpack 4, the bundle-level minification is enabled automatically – both in the production mode and without one. It uses the UglifyJS minifier under the hood. (If you ever need to disable minification, just use the development mode or pass `false` to the `optimization.minimize` option.)

In webpack 3, you need to use the UglifyJS plugin directly. The plugin comes bundled with webpack; to enable it, add it to the `plugins` section of the config:

```
// webpack.config.js
const webpack = require('webpack');

module.exports = {
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
  ],
};
```

Note: In webpack 3, the UglifyJS plugin can't compile the ES2015+ (ES6+) code. This means that if your code uses classes, arrow functions or other new language features, and you don't compile them into ES5, the plugin will throw an error.

If you need to compile the new syntax, use the uglifyjs-webpack-plugin package. This is the same plugin that's bundled with webpack, but newer, and it's able to compile the ES2015+ code.

Loader-specific options

The second way to minify the code is loader-specific options (what a loader is). With loader options, you can compress things that the minifier can't minify. For example, when you import a CSS file with `css-loader`, the file is compiled into a string:

```
/* comments.css */
.comment {
```

```
  color: black;
}
```

↓

```
// minified bundle.js (part of)
exports=module.exports=__webpack_require__(1)(),
exports.push([module.i,".comment {\r\n  color: black;\r\n}",""]);
```

The minifier can't compress this code because it's a string. To minify the file content, we need to configure the loader to do this:

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          'style-loader',
          { loader: 'css-loader', options: { minimize: true } },
        ],
      },
    ],
  },
};
```

Further reading

- The UglifyJsPlugin docs
- Other popular minifiers: Babel Minify, Google Closure Compiler

Specify `NODE_ENV=production`

Note: this is webpack 3 only. If you use webpack 4 with the production mode, the `NODE_ENV=production` optimization is already enabled – feel free to skip the section.

Another way to decrease the front-end size is to set the `NODE_ENV` environmental variable in your code to the value `production`.

Libraries read the `NODE_ENV` variable to detect in which mode they should work – in the development or the production one. Some libraries behave differently based on this variable. For example, when `NODE_ENV` is not set to `production`, Vue.js does additional checks and prints warnings:

```
// vue/dist/vue.runtime.esm.js
// …
if (process.env.NODE_ENV !== 'production') {
  warn('props must be strings when using array syntax.');
}
// …
```

React works similarly – it loads a development build that includes the warnings:

```
// react/index.js
if (process.env.NODE_ENV === 'production') {
  module.exports = require('./cjs/react.production.min.js');
} else {
  module.exports = require('./cjs/react.development.js');
}

// react/cjs/react.development.js
// …
warning$3(
  componentClass.getDefaultProps.isReactClassApproved,
  'getDefaultProps is only used on classic React.createClass ' +
  'definitions. Use a static property named `defaultProps` instead.'
);
// …
```

Such checks and warnings are usually unnecessary in production, but they remain in the code and increase the library size. **In webpack 4,** remove them by adding the `optimization.nodeEnv: 'production'` option:

```
// webpack.config.js (for webpack 4)
module.exports = {
  optimization: {
    nodeEnv: 'production',
    minimize: true,
  },
};
```

In webpack 3, use the `DefinePlugin` instead:

```
// webpack.config.js (for webpack 3)
const webpack = require('webpack');

module.exports = {
  plugins: [
    new webpack.DefinePlugin({
      'process.env.NODE_ENV': '"production"',
    }),
    new webpack.optimize.UglifyJsPlugin(),
  ],
};
```

Both the **`optimization.nodeEnv`** option and the **`DefinePlugin`** work the same way – they replace all occurrences of **`process.env.NODE_ENV`** with the specified value. With the config from above:

1. Webpack will replace all occurrences of **`process.env.NODE_ENV`** with "**`production`**":

```
 // vue/dist/vue.runtime.esm.js
if (typeof val === 'string') {
  name = camelize(val);
  res[name] = { type: null };
} else if (process.env.NODE_ENV !== 'production') {
  warn('props must be strings when using array syntax.');
}
```

↓

```
// vue/dist/vue.runtime.esm.js
if (typeof val === 'string') {
  name = camelize(val);
  res[name] = { type: null };
} else if ("production" !== 'production') {
  warn('props must be strings when using array syntax.');
}
```

2. And then the minifier will remove all such **`if`** branches – because "**`production`**" **`!==`** '**`production`**' is always false, and the plugin understands that the code inside these branches will never execute:

```
// vue/dist/vue.runtime.esm.js
if (typeof val === 'string') {
  name = camelize(val);
  res[name] = { type: null };
} else if ("production" !== 'production') {
```

```
  warn('props must be strings when using array syntax.');
}
```

↓

```
// vue/dist/vue.runtime.esm.js (without minification)
if (typeof val === 'string') {
  name = camelize(val);
  res[name] = { type: null };
}
```

Further reading

- What “environment variables” are
- Webpack docs about: `DefinePlugin`, `EnvironmentPlugin`

Use ES modules

The next way to decrease the front-end size is to use ES modules.

When you use ES modules, webpack becomes able to do tree-shaking. Tree-shaking is when a bundler traverses the whole dependency tree, checks what dependencies are used, and removes unused ones. So, if you use the ES module syntax, webpack can eliminate the unused code:

1. You write a file with multiple exports, but the app uses only one of them:

```
// comments.js
export const render = () => { return 'Rendered!'; };
export const commentRestEndpoint = '/rest/comments';

// index.js
import { render } from './comments.js';
render();
```

2. Webpack understands that `commentRestEndpoint` is not used and doesn’t generate a separate export point in the bundle:

```
// bundle.js (part that corresponds to comments.js)
(function(module, __webpack_exports__, __webpack_require__) {
  "use strict";
  const render = () => { return 'Rendered!'; };
```

```
  /* harmony export (immutable) */ __webpack_exports__["a"] = render;

  const commentRestEndpoint = '/rest/comments';
  /* unused harmony export commentRestEndpoint */
})
```

3. The minifier removes the unused variable:

```
// bundle.js (part that corresponds to comments.js)
(function(n,e){"use strict";var r=function(){return"Rendered!"};e.b=r})
```

This works even with libraries if they are written with ES modules.

Note: In webpack, tree-shaking doesn't work without a minifier. Webpack just removes export statements for exports that aren't used; it's the minifier that removes unused code. Therefore, if you compile the bundle without the minifier, it won't get smaller.

You aren't required to use precisely webpack's built-in minifier (`UglifyJsPlugin`) though. Any minifier that supports dead code removal (e.g. Babel Minify plugin or Google Closure Compiler plugin) will do the trick.

Warning: Don't accidentally compile ES modules into CommonJS ones.

If you use Babel with `babel-preset-env` or `babel-preset-es2015`, check the settings of these presets. By default, they transpile ES' `import` and `export` to CommonJS' `require` and `module.exports`. Pass the `{ modules: false }` option to disable this.

The same with TypeScript: remember to set `{ "compilerOptions": { "module": "es2015" } }` in your `tsconfig.json`.

Further reading

- "ES6 Modules in depth"
- Webpack docs about tree shaking

Optimize images

Images account for more than a half of the page size. While they are not as critical as JavaScript (e.g., they don't block rendering), they still eat a large part of the bandwidth. Use `url-loader`, `svg-url-loader` and `image-webpack-loader` to optimize them in webpack.

`url-loader` inlines small static files into the app. Without configuration, it takes a passed file, puts it next to the compiled bundle and returns an url of that file. However, if we specify the `limit` option, it will encode files smaller than this limit as a Base64 data url and return this url. This inlines the image into the JavaScript code and saves an HTTP request:

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.(jpe?g|png|gif)$/,
        loader: 'url-loader',
        options: {
          // Inline files smaller than 10 kB (10240 bytes)
          limit: 10 * 1024,
        },
      },
    ],
  }
};
```

```
// index.js
import imageUrl from './image.png';
// → If image.png is smaller than 10 kB, `imageUrl` will include
// the encoded image: '…'
// → If image.png is larger than 10 kB, the loader will create a new file,
// and `imageUrl` will include its url: `/2fcd56a1920be.png`
```

Note: Inlined images reduce the number of separate requests, which is good (even with HTTP/2), but increase the download/parse time of your bundle and memory consumption. Make sure to not embed large images or a lot of them – or increased bundle time would outweigh the benefit of inlining.

`svg-url-loader` works just like `url-loader` – except that it encodes files with the URL encoding instead of the Base64 one. This is useful for SVG images – because SVG files are just a plain text, this encoding is more size-effective:

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
```

```
      {
        test: /\.svg$/,
        loader: 'svg-url-loader',
        options: {
          // Inline files smaller than 10 kB (10240 bytes)
          limit: 10 * 1024,
          // Remove the quotes from the url
          // (they're unnecessary in most cases)
          noquotes: true,
        },
      },
    ],
  },
};
```

Note: svg-url-loader has options that improve Internet Explorer support, but worsen inlining for other browsers. If you need to support this browser, apply the `iesafe: true` option.

`image-webpack-loader` compresses images that go through it. It supports JPG, PNG, GIF and SVG images, so we're going to use it for all these types.

This loader doesn't embed images into the app, so it must work in pair with `url-loader` and `svg-url-loader`. To avoid copy-pasting it into both rules (one for JPG/PNG/GIF images, and another one for SVG ones), we'll include this loader as a separate rule with `enforce: 'pre'`:

```
// webpack.config.js
module.exports = {
  module: {
    rules: [
      {
        test: /\.(jpe?g|png|gif|svg)$/,
        loader: 'image-webpack-loader',
        // This will apply the loader before the other ones
        enforce: 'pre',
      },
    ],
  },
};
```

The default settings of the loader are already good to go – but if you want to configure it further, see the plugin options. To choose what options to specify, check out Addy Osmani's excellent guide on image optimization.

Further reading

- "What is base64 encoding used for?"
- Addy Osmani's guide on image optimization

Optimize dependencies

More than a half of average JavaScript size comes from dependencies, and a part of that size might be just unnecessary.

For example, Lodash (as of v4.17.4) adds 72 KB of minified code to the bundle. But if you use only, like, 20 of its methods, then approximately 65 KB of minified code does just nothing.

Another example is Moment.js. Its 2.19.1 version takes 223 KB of minified code, which is huge – the average size of JavaScript on a page was 452 KB in October 2017. However, 170 KB of that size is localization files. If you don't use Moment.js with multiple languages, these files will bloat the bundle without a purpose.

All these dependencies can be easily optimized. We've collected optimization approaches in a GitHub repo – check it out!

Enable module concatenation for ES modules (aka scope hoisting)

Note: if you're using webpack 4 with the production mode, module concatenation is already enabled. Feel free to skip this section.

When you are building a bundle, webpack is wrapping each module into a function:

```
// index.js
import {render} from './comments.js';
render();

// comments.js
export function render(data, target) {
  console.log('Rendered!');
}

↓
```

```
// bundle.js (part  of)
/* 0 */
(function(module, __webpack_exports__, __webpack_require__) {

  "use strict";
  Object.defineProperty(__webpack_exports__, "__esModule", { value: true });
  var __WEBPACK_IMPORTED_MODULE_0__comments_js__ = __webpack_require__(1);
  Object(__WEBPACK_IMPORTED_MODULE_0__comments_js__["a" /* render */])();

}),
/* 1 */
(function(module, __webpack_exports__, __webpack_require__) {

  "use strict";
  __webpack_exports__["a"] = render;
  function render(data, target) {
    console.log('Rendered!');
  }

})
```

In the past, this was required to isolate CommonJS/AMD modules from each other. However, this added a size and performance overhead for each module.

Webpack 2 introduced support for ES modules which, unlike CommonJS and AMD modules, can be bundled without wrapping each with a function. And webpack 3 made such bundling possible – with module concatenation. Here's what module concatenation does:

```
// index.js
import {render} from './comments.js';
render();

// comments.js
export function render(data, target) {
  console.log('Rendered!');
}


↓


// Unlike the previous snippet, this bundle has only one module
// which includes the code from both files

// bundle.js (part of; compiled with ModuleConcatenationPlugin)
/* 0 */
(function(module, __webpack_exports__, __webpack_require__) {
```

```
  "use strict";
  Object.defineProperty(__webpack_exports__, "__esModule", { value: true });

  // CONCATENATED MODULE: ./comments.js
  function render(data, target) {
    console.log('Rendered!');
  }

  // CONCATENATED MODULE: ./index.js
  render();

})
```

See the difference? In the plain bundle, module 0 was requiring `render` from module 1. With module concatenation, `require` is simply replaced with required function, and module 1 is removed. The bundle has fewer modules – and less module overhead!

To turn on this behavior, **in webpack 4**, enable the `optimization.concatenateModules` option:

```
// webpack.config.js (for webpack 4)
module.exports = {
  optimization: {
    concatenateModules: true,
  },
};
```

In webpack 3, use the `ModuleConcatenationPlugin`:

```
// webpack.config.js (for webpack 3)
const webpack = require('webpack');

module.exports = {
  plugins: [
    new webpack.optimize.ModuleConcatenationPlugin(),
  ],
};
```

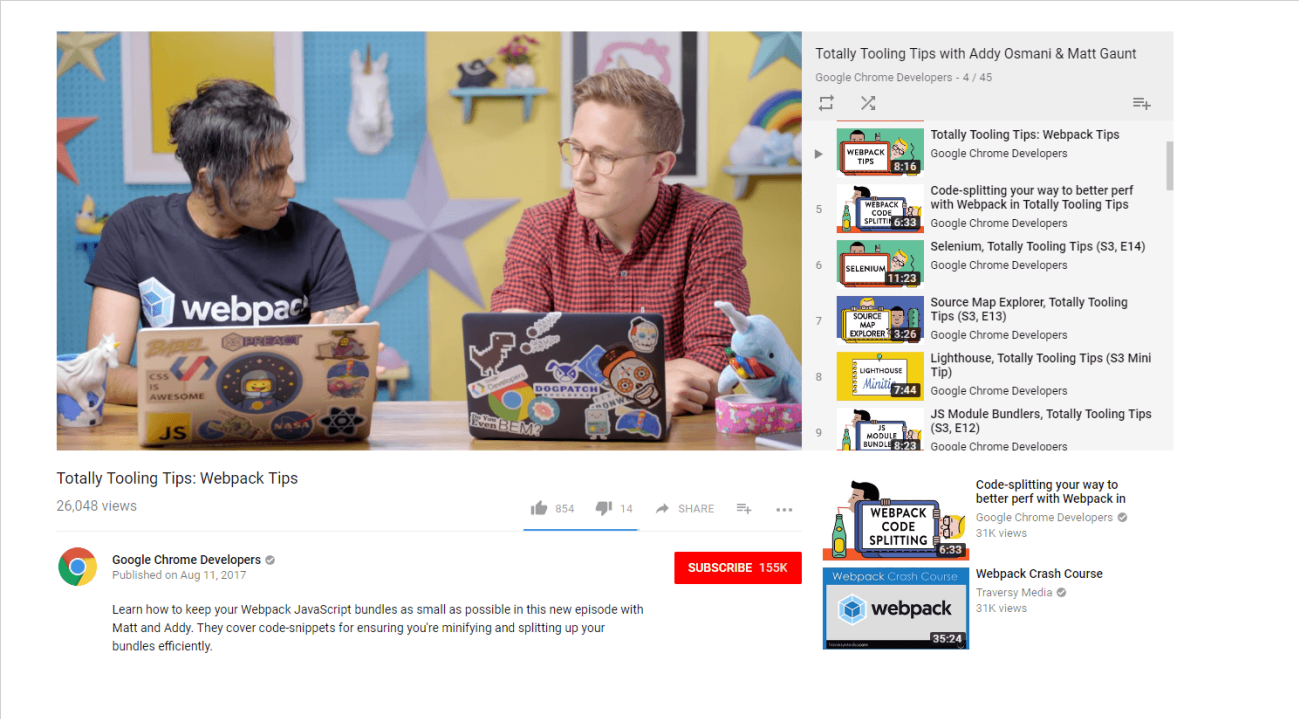
Note: Wonder why this behavior is not enabled by default? Concatenating modules is cool, [but it comes with increased build time and breaks hot module replacement](). That's why it should only be enabled in production.

Further reading

- Webpack docs for the ModuleConcatenationPlugin
- "Brief introduction to scope hoisting"
- Detailed description of what this plugin does

Use `externals` if you have both webpack and non-webpack code

You might have a large project where some code is compiled with webpack, and some code is not. Like a video hosting site, where the player widget might be built with webpack, and the surrounding page might be not:



(A completely random video hosting site)

If both pieces of code have common dependencies, you can share them to avoid downloading their code multiple times. This is done with the webpack's `externals` option – it replaces modules with variables or other external imports.

If dependencies are available in `window`

If your non-webpack code relies on dependencies that are available as variables in `window`, alias dependency names to variable names:

```
// webpack.config.js
module.exports = {
  externals: {
```

```
    'react': 'React',
    'react-dom': 'ReactDOM',
  },
};
```

With this config, webpack won't bundle `react` and `react-dom` packages. Instead, they will be replaced with something like this:

```
// bundle.js (part of)
(function(module, exports) {
  // A module that exports `window.React`. Without `externals`,
  // this module would include the whole React bundle
  module.exports = React;
}),
(function(module, exports) {
  // A module that exports `window.ReactDOM`. Without `externals`,
  // this module would include the whole ReactDOM bundle
  module.exports = ReactDOM;
})
```

If dependencies are loaded as AMD packages

If your non-webpack code doesn't expose dependencies into `window`, things are more complicated. However, you can still avoid loading the same code twice if the non-webpack code consumes these dependencies as AMD packages.

To do this, compile the webpack code as an AMD bundle and alias modules to library URLs:

```
// webpack.config.js
module.exports = {
  output: { libraryTarget: 'amd' },

  externals: {
    'react': { amd: '/libraries/react.min.js' },
    'react-dom': { amd: '/libraries/react-dom.min.js' },
  },
};
```

Webpack will wrap the bundle into `define()` and make it depend on these URLs:

```
// bundle.js (beginning)
define(["/libraries/react.min.js", "/libraries/react-dom.min.js"], function () {
```

If non-webpack code uses the same URLs to load its dependencies, then these files will be loaded only once – additional requests will use the loader cache.

Note: Webpack replaces only those imports that exactly match keys of the `externals` object. This means that if you write `import React from 'react/umd/react.production.min.js'`, this library won't be excluded from the bundle. This is reasonable – webpack doesn't know if `import 'react'` and `import 'react/umd/react.production.min.js'` are the same things – so stay careful.

Further reading

- Webpack docs on `externals`

Summing up

- Enable the production mode if you use webpack 4
- Minimize your code with the bundle-level minifier and loader options
- Remove the development-only code by replacing `NODE_ENV` with `production`
- Use ES modules to enable tree shaking
- Compress images
- Apply dependency-specific optimizations
- Enable module concatenation
- Use `externals` if this makes sense for you

Previous
← Introduction

Next
Make Use of Long-term Caching →