4 Ways To Boost Your Vue.js App With Webpack

vuejsdevelopers.com/2017/06/18/vue-js-boost-your-app-with-webpack

Webpack is an essential tool for developing Vue.js single page applications. It makes your development workflow much simpler by managing complex build steps and can optimise your apps size and performance.

In this article I'll explain four ways that Webpack can enhance your Vue app, including:

1. Single file components
2. Optimising the Vue build
3. Browser cache management
4. Code splitting

What about vue-cli?

If you're using a template to build your app from *vue-cli*, a pre-made Webpack config is provided. They're well optimised and there are no improvements I can suggest!

But since they work so well out of the box, you probably don't have much idea of what they're really doing, right? Consider this article an overview of the Webpack config used in the vue-cli templates, as they include the same optimisations I'm discussing here.

1. Single file components

One of Vue's idiosyncratic features is the use of HTML for component templates. These come with an intrinsic problem, though: either your HTML markup needs to be in an awkward JavaScript string, or your template and component definition will need to be in separate files, making them hard to work with.

Vue has an elegant solution called Single File Components (SFCs) that include the template, component definition and CSS all in one neat *.vue* file:

MyComponent.vue

```
<template>
  <div id="my-component">...</div>
</template>
<script>
  export default {...}
</script>
<style>
  #my-component {...}
</style>
```

SFCs are made possible by the *vue-loader* Webpack plugin. This loader splits up the SFCs language blocks and pipes each to an appropriate loader, e.g. the script block goes to *babel-loader*, while the template block goes to Vue's own *vue-template-loader* which transforms the template into a JavaScript `render` function.

The final output of vue-loader is a JavaScript module ready for inclusion in your Webpack bundle.

A typical configuration for `vue-loader` is as follows:

```
module: {
  rules: [
    {
      test: /\.vue$/,
      loader: 'vue-loader',
      options: {
        loaders: {

        }
      }
    },
  ]
}
```

2. Optimising the Vue build

Runtime-only build

If you're only using render functions in your Vue app*, and no HTML templates, you don't need Vue's template compiler. You can reduce your bundle size by omitting the compiler from the Webpack build.

** Remember that single file component templates are pre-compiled in development to render functions!*

There is a *runtime-only* build of the Vue.js library that includes all the features of Vue.js except the template compiler, called *vue.runtime.js*. It's about 20KB smaller than the full build so it's worth using if you can.

The runtime-only build is used by default, so every time you use `import vue from 'vue';` in your project that's what you'll get. You can change to a different build, though, by using the `alias` configuration option:

```
resolve: {
  alias: {
    'vue$': 'vue/dist/vue.esm.js'
  }
},
```

Stripping out warnings and error messages in production

Another way to reduce your Vue.js build size is to remove any error messages and warnings in production. These bloat your output bundle size with unnecessary code and also incur a runtime cost you're best to avoid.

If you inspect the Vue source code you'll see that warning blocks are conditional on the value of an environment variable `process.env.NODE_ENV` e.g.:

```
if (process.env.NODE_ENV !== 'production') {
  warn(("Error in " + info + ": \"" + (err.toString()) + "\""), vm);
}
```

If *process.env.NODE_ENV* is set to `production` then such warning blocks can be automatically stripped out of the code by a minifier during the build process.

You can use the *DefinePlugin* to set the value of *process.env.NODE_ENV*, and the *UglifyJsPlugin* to minify the code and strip out the unused blocks:

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


3. Browser cache management

A user's browser will cache your site's files so that they'll only download if the browser does not already have a local copy, or if the local copy has expired.

If all your code is in one file, then a tiny change would mean the whole file would need to be re-downloaded. Ideally you want your users to download as little as possible, so it'd be smart to separate your app's rarely changing code from its frequently changing code.

Vendor file

The *Common Chunks* plugin can decouple your *vendor* code (e.g. dependencies like the Vue.js library that are unlikely to change very often) from your *application* code (code that may change on every deployment).

You can configure the plugin to check if a dependency is from the `node_modules` folder, and if so, output it into a separate file `vendor.js`:

```
new webpack.optimize.CommonsChunkPlugin({
  name: 'vendor',
  minChunks: function (module) {
    return module.context && module.context.indexOf('node_modules') !== -1;
  }
})
```

If you do this you'll now have two separate files in your build output which will be cached by the browser independently:

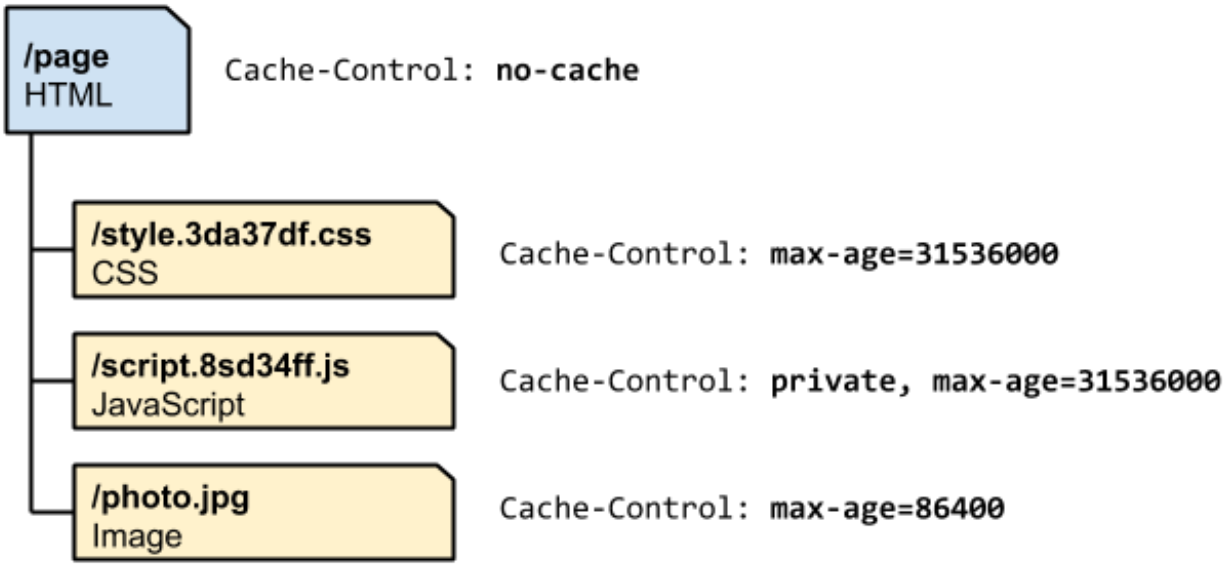
```
<script src="vendor.js" charset="utf-8"></script>
<script src="app.js" charset="utf-8"></script>
```

Fingerprinting

When a build file changes, how do we bust a browser's cache?

By default, only when a cached file expires, or when the user manually clears the cache, will the browser request the file again from the server. The file will be re-downloaded if the server indicates the file has changed (otherwise the server return HTTP 304 Not Modified).

To save an unnecessary server request, we can change a file's name every time its content changes to force the browser to re-download it. A simple system for doing this is to add a "fingerprint" to the file name by appending a hash e.g.:



The Common Chunks plugin emits a "chunkhash" which is updated if the file's content has changed. Webpack can append this hash to the file names when they're outputted:

```
output: {
  filename: '[name].[chunkhash].js'
},
```

When you do this, you'll see that your outputted files will have names like *app.3b80b7c17398c31e4705.js*.

Auto inject build files

Of course if you add a hash you'll have to update the reference to the file in your index file, otherwise the browser won't know about it:

```
<script src="app.3b80b7c17398c31e4705.js"></script>
```

This would be a hugely tedious task to do manually, so use the *HTML Webpack Plugin* to do it for you. This plugin can *auto inject* references to the build files into your HTML file in the bundling process.

Start by removing references to your build file:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>test-6</title>
  </head>
  <body>
    <div id="app"></div>

  </body>
</html>
```

And add the *HTML Webpack Plugin* to your Webpack config:

```
new HtmlWebpackPlugin({
  filename: 'index.html'
  template: 'index.html',
  inject: true,
  chunksSortMode: 'dependency'
}),
```

Now your build files with hashes will automatically be added to your index file. Also, your *index.html* file will now be included in your bundle output so you may need to tell the web server that its location has changed.

4. Code splitting

By default, Webpack will output all your apps code into one large bundle. But if your app has multiple pages it would be more efficient to split the code so each individual pages code is in a separate file, and is only loaded when needed.

Webpack has a feature called "code splitting" that does exactly that. Achieving this in Vue.js also requires *async components*, and is made even easier with *Vue Router*.

Async components

Rather than having a definition object as their second argument, *async components* have a Promise function that resolves the definition object, for example:

```
Vue.component('async-component', function (resolve, reject) {
  setTimeout(() => {
    resolve({

    });
  }, 1000)
})
```

Vue will only call the function when the component actually needs to be rendered. It will also cache the result for future re-renders.

If we architect our app so each "page" is a component, and we store the definition on our server, then we're half-way to achieving code splitting.

require

To load your async component's code from the server, use the Webpack `require` syntax. This will instruct Webpack to bundle `async-component` in a separate bundle when it builds, and better yet, Webpack will handle the loading of this bundle with AJAX, so your code can be as simple as this:

```
Vue.component('async-component', function (resolve) {
  require(['./AsyncComponent.vue'], resolve)
});
```

Lazy loading

In a Vue.js app *vue-router* will typically be the module you use to organise your SPA into multiple pages. *Lazy loading* is a formalised way for achieving code splitting with Vue and Webpack.

```
const HomePage = resolve => require(['./HomePage.vue'], resolve);

const rounter = new VueRouter({
  routes: [
    {
      path: '/',
      name: 'HomePage',
      component: HomePage
    }
  ]
})
```