Member only

5.6k 119 Save

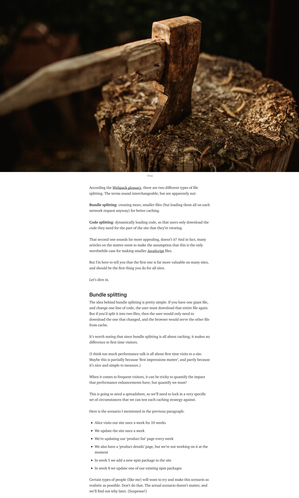
The 100% correct way to split your chunks with Webpack

Jack Herrington Follow
Jan 4, 2018 · 10 min read

Working out the best way to serve up files to your users can be a tricky business. There's so many different scenarios, different technologies, different terminology.

In this post I hope to give you everything you need so that you can:

1. know what the splitting strategy will work best for your site and your users
2. know how to do it



Image

According the Webpack glossary, there are two different types of file splitting. The terms sound interchangeable, but are apparently not:

Bundle splitting: creating more, smaller files (but loading them all on each network request anyway) for better caching.

Code splitting: dynamically loading code, so that users only download the code they need for the part of the site that they're viewing.

That second one sounds far more appealing, doesn't it? And in fact, many articles on the matter seem to make the assumption that this is the only worthwhile case for nothing smaller than bundling files.

But I'm here to tell you that the first one is far more valuable on many sites, and should be the first thing you do for all sites.

Let's dive in.

Bundle splitting

The idea behind bundle splitting is pretty simple. If you have one giant file, then any change to any part of the app means everyone has to redownload the app. But if you'd split it into two files, then the user would only need to download the one that changed, and the browser would serve the other file from cache.

It's worth noting that since bundle splitting is all about caching, it makes no difference to first time visitors.

(I think too much performance talk is all about first time visits to a site. Maybe this is partially because "first impressions matter", and partly because it's sexy and simple to measure.)

When it comes to frequent visitors, it can be tricky to quantify the impact that performance enhancements have, but quantify we must!

This is going to need a spreadsheet, so we'll need to lock in a very specific set of circumstances that we can test each caching strategy against.

Here is the scenario I mentioned in the previous paragraph:

- Alice visits our site once a week for 10 weeks
- We update the site once a week
- We're updating our 'product list' page every week
- We also have a 'product details' page, but we're not working on it at the moment
- In week 5 we add a new open package to the site
- In week 8 we update one of our existing open packages

Certain types of people (like me) will want to try and make this scenario as realistic as possible. Don't do that. The actual scenario doesn't matter, and we'll find out why later. (Surprises!)

The baseline

Let's say our vendor.js/react package is a heavy 400 kB, and we're currently loading this as a single file called app.js.

We have a Webpack config looking something like this (I'm leaving our non-relevant cruft out):

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

module.exports = {
  entry: {
    app: path.resolve(__dirname, 'src/index.js'),
    vendor: [
      'react',
      'react-dom',
    ],
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.[contenthash].js',
  },
}
```

webpack.config.js hosted with ❤ by GitHub view raw

Webpack config for a single app and vendor file

(For those new to cache busting: any time I say app.js, what I actually mean is something like app.e8e7f8e8a.js, where the crazy string of letters is a hash of the contents of the file. This means a different file name when the code in your application changes, thus forcing the browser to download the new file.)

Each week when we push some new changes to the site, the .contenthash of this package changes. So each week, Alice visits our site and has to download a new 400 kB file.

If we were to make a very table of these events, it would look like this.

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10
main.js	400	400	400	400	400	400	400	400	400	400
Total	**400**	**400**	**400**	**400**	**400**	**400**	**400**	**400**	**400**	**400**

Total kB that Alice will need to download if we're using a single app.js file

That's 4.1 MB, over 10 weeks.

We can do better.

Splitting out vendor packages

Let's split our packages into app.js and vendor.js file.

This is easy, sort of:

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

module.exports = {
  entry: {
    app: path.resolve(__dirname, 'src/index.js'),
    vendor: [
      'react',
      'react-dom',
    ],
  },
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: '[name].[contenthash].js',
  },
  optimization: {
    splitChunks: {
      chunks: 'all',
    },
    runtimeChunk: true
  },
  plugins: [
    new HtmlWebpackPlugin({
      inject: false,
      hash: true,
      template: './src/index.ejs',
      filename: 'index.html'
    })
  ],
}
```

webpack.config.js hosted with ❤ by GitHub view raw

Webpack 4 makes an effort to do the best thing for you, without you having to tell it exactly how you want to split your bundles.

This back is several instances of "key that's new. Nice one, Webpack!"

And many instances of "WHAT THE HELL ARE YOU DOING TO MY BUNDLES?"

Anyhow, setting optimization.splitChunks.chunks: 'all' is a way of saying "put everything in node_modules into a file called vendor~main.js".

With this basic bundle splitting in place, Alice still downloads a new 200 kB app.js on each visit, but only downloads the 200 kB vendor.js in week one, eight and five (not in that order).

	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10
main.js	200	200	200	200	200	200	200	200	200	200
vendor~main.js	200				200				200	
Total	**400**	**200**	**200**	**200**	**400**	**200**	**200**	**200**	**400**	**200**

Total kB that Alice will need to download if we're using app.js and vendor.js files

That's 2.04 MB.

A 50% reduction. Not bad for adding five lines of code to our config. Go do that now before reading any further. If you need to upgrade from Webpack 3 to 4, don't fret, it's pretty painless (and still fast).

I think this performance improvement seems somewhat more obscure because it's spread out over 10 weeks, but it's a real 50% reduction in bytes shipped to a hard user and we should be proud of ourselves.

But we can do better.

Splitting out each npm package

Our vendor.js solves the same problem that our original app.js file did — a change to one part of it means re-downloading all parts of it.

So why not have a separate file for each npm package? It's easy enough to do.

So, let's split out our react, lodash, redux, moment, etc., into different files:

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

module.exports = {
  entry: {
    app: path.resolve(__dirname, 'src/index.js'),
    vendor: [
      'react',
      'react-dom',
      'lodash',
      'redux',
      'moment',
      'react-router',
      'react-router-dom',
      'react-redux',
      'react-bootstrap',
      'react-bootstrap-table',
      'react-select',
      'react-table',
      'react-tooltip',
      'react-toastify',
      'react-tree-table',
      'react-virtualized',
      'react-vis',
      'react-vis-force',
      'react-vis-network-graph',
      'react-vis-sankey',
      'react-vis-tree',
      'react-vis-treemap',
      'react-vis-zoom',
      'react-vis-zoom-and-drag',
      'react-vis-zoom-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag',
      'react-vis-zoom-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select-and-drag-and-select
```

Windows 8/IE11	HTTP 1.1	Fibre	1	342	312	337	358	307		337	6%
Windows 8/IE11	HTTP 1.1	Fibre	19	346	425	356	350	385		356	

Loading two versions of the same 190 KB site (from Firebase static hosting)

On 3G and 4G this site loaded in 30% less time when there were 19 files.

Or did it?

This is pretty noisy data. For example, on 4G on run 2, the site loaded in 646ms, then two runs later it took 1,116ms — 73% longer, with no change. So it seems a bit sneaky to claim that HTTP/2 is '30% faster'.

(Coming soon: a custom chart type designed to visualise the difference in page load times.)

I created this table to try and quantify what difference HTTP/2 made, but really the only thing I can say is "it probably makes no significant difference".

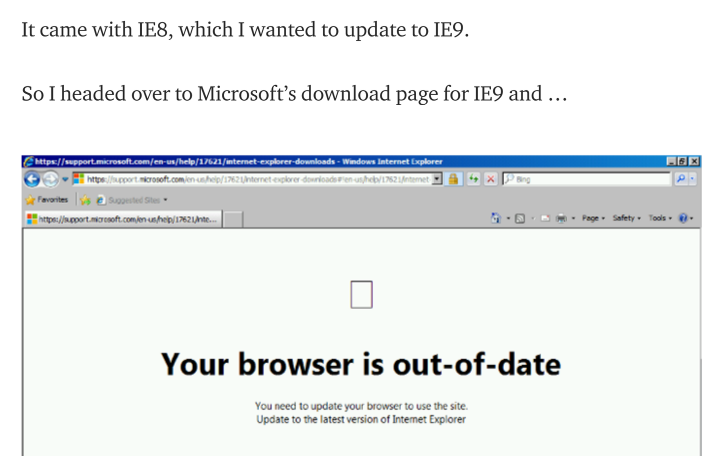
The real surprise was those last two rows. That's old Windows and HTTP/1.1 which I would have bet would be much slower. I guess I need slower internet.

. . .

Story time! I downloaded a Windows 7 virtual machine from Microsoft's site to test these things.

It came with IE8, which I wanted to update to IE9.

So I headed over to Microsoft's download page for IE9 and …



OMG. SMH. FML.

One final word on HTTP/2, did you know it's built into Node now? If you want to have a play, I wrote a little 100-line HTTP/2 server with gzip, brotli and response caching for your testing pleasure.

. . .

That's everything I have to say about bundle splitting. I think the only downside to this approach is constantly having to convince people that loading lots of small files is OK.

. . .

Now, to talk about the other type of file-chopping…

Code splitting (don't load code you don't need to)

This particular approach only makes sense on some sites, says me.

I like to apply the 20/20 rule that I just made up: if there's a part of your site that only 20% of users go to, and it's bigger than 20% of your site's JavaScript, then you should only load that code on demand.

Adjust those numbers to taste, obviously, and there's more complex scenarios than that, obviously. The point is, there's a balance, and it's OK to decide that code splitting simply doesn't make sense for your site.

How to decide?

Let's say you've got a shopping site and you're wondering if you should split out the code for the 'checkout', because only 30% of your visitors make it there.

The first thing to do is sell better stuff.

Top highlight

The second thing is to work out how much code is completely unique to the checkout. Since you should always do 'bundle splitting' before you do 'code splitting', you might already know how big this part of your code is.

(It might be smaller than you think, so do the sums before you get too excited. If you have a React site, for example, then your store, reducers, routing, actions, etc. will all be shared across the whole site. The unique parts will mostly be components and their helpers.)

So, you note that the code that is completely unique to your checkout page is 7 KB. The rest of the site is 300 KB. I would look at this and say meh, I'm not going to bother code splitting this, for a few reasons:

- Loading it up front is no slower. Remember that you're loading all these files in parallel. See if you can record a difference in the load times between 300 KB and 307 KB.
- If you load this code later, the user will have to wait for that file after clicking on 'TAKE MY MONEY' — the very time at which you want the least friction.
- Code splitting requires changes to your application code. It introduces asynchronous logic where previously there was only synchronous logic. It's not rocket science, but it's complexity that I think should be justified by a perceivable improvement to the user experience.

OK, that's all the party pooper "this exciting tech might not apply to you" stuff.

Let's look at two examples of code-splitting…

Polyfills

I'll start with this because it applies to most sites, and is a nice simple introduction.

I'm using a bunch of fancy features in my site, so I've got a file that imports all the polyfills I need. It includes these eight lines:

```
require('whatwg-fetch');
require('intl');
require('url-polyfill');
require('core-js/web/dom-collections');
require('core-js/es6/map');
require('core-js/es6/string');
require('core-js/es6/array');
require('core-js/es6/object');
```

polyfills.js hosted with ❤ by GitHub view raw

I import this file right at the top of my entry point, `index.js`.

```
import './polyfills';
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App/App';
import './index.css';

const render = () => {
  ReactDOM.render(<App />, document.getElementById('root'));
}

render(); // yes I am pointless, for now
```

index-always-poly.js hosted with ❤ by GitHub view raw

With the Webpack config from the bundle splitting section, my polyfills will be automatically split into four different files since there's four npm packages here. They're about 25 KB all up, and 90% of browsers don't need them, so it's worth loading these dynamically.

With Webpack 4 and the `import()` syntax (not to be confused with the `import` syntax), conditionally loading the polyfills is pretty easy.

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './App/App';
import './index.css';

const render = () => {
  ReactDOM.render(<App />, document.getElementById('root'));
}

if (
  'fetch' in window &&
  'Intl' in window &&
  'URL' in window &&
  'Map' in window &&
  'forEach' in NodeList.prototype &&
  'startsWith' in String.prototype &&
  'endsWith' in String.prototype &&
  'includes' in String.prototype &&
  'includes' in Array.prototype &&
  'assign' in Object &&
  'entries' in Object &&
  'keys' in Object
) {
  render();
} else {
  import('./polyfills').then(render);
}
```

index-sometimes-poly.js hosted with ❤ by GitHub view raw

Make sense? If all that stuff is supported, then render the page. Otherwise, import polyfills then render the page. When this code runs in the browser, Webpack's runtime will handle the loading of those four npm packages, and when they've been downloaded and parsed, will call `render()` and things will carry on.

(BTW, to use `import()`, you'll need Babel's dynamic-import plugin. Also, as the Webpack docs explain, `import()` uses promises, so you'll need to polyfill that separate to the other polyfills.)

Easy, right?

Here's something a bit trickier…

Route-based dynamic loading (React specific)

Going back to the Alice example, let's say that the site now has an 'administration' section, where *sellers* of products can log in and administer the crap that they have for sale.

This section has many wonderful features, plenty of charts, and a big fat charting library from npm. Because I was already doing bundle splitting, I could see that these were a shade over 100 KB all up.

Currently, I have a routing setup that will render `<AdminPage>` when the user is viewing an `/admin` URL. When Webpack bundles everything up, it's going to find `import AdminPage from './AdminPage.js'` and say "hey, I need to include this in the initial payload".

But we don't want it to. We need to put that reference to the admin page inside a dynamic import, like `import('./AdminPage.js')` so that Webpack knows to load it dynamically.

It's pretty cool, no config required.

So instead of referring to `AdminPage` directly, I could create another component that will be rendered when the user goes the the `/admin` URL. It might look something like this:

```
import React from 'react';

class AdminPageLoader extends React.PureComponent {
  constructor(props) {
    super(props);

    this.state = {
      AdminPage: null,
    }
  }

  componentDidMount() {
    import('./AdminPage').then(module => {
      this.setState({ AdminPage: module.default });
    });
  }

  render() {
    const { AdminPage } = this.state;

    return AdminPage
      ? <AdminPage {...this.props} />
      : <div>Loading...</div>;
  }
}

export default AdminPageLoader;
```

AdminPageLoader.js hosted with ❤ by GitHub view raw

The concept is pretty straightforward, right? When this component mounts (meaning that the user is at the `/admin` URL), we'll dynamically load `./AdminPage.js` and then save a reference to that component in state.

In the render method, we simply render `<div>Loading...</div>` while we're waiting for the `<AdminPage>` to load, or the `<AdminPage>` once it's been loaded and stored in state.

I wanted to do this myself for fun, but in the real world, you'd just use `react-loadable`, as described in the React docs on code-splitting.

. . .

Righto, I think that's everything. Is there any point in saying what I already said above, but in fewer words?

- If people ever visit your site more than one time, split your code up into many little files.
- If you have large parts of your site that most users don't visit, load that code dynamically.

Thanks for reading, have a tops day!

Dammit I forgot to mention CSS.

Hackernoon Newsletter curates great stories by real tech professionals

Get solid gold sent to your inbox. Every week!

Email

First Name Last Name

Sign Up

If you are ok with us sending you updates via email, please tick the box. Unsubscribe whenever you want.

Terms of Service

I agree to leave medium.com and submit this information, which will be collected and used according to Upscribe's privacy policy.

Powered by Upscribe

Sign up for Get Better Tech Emails via HackerNoon.com

By HackerNoon.com

how hackers start their afternoons. the real shit is on hackernoon.com. Take a look.

Your email

Get this newsletter

By signing up, you will create a Medium account if you don't already have one. Review our Privacy Policy for more information about our privacy practices.

JavaScript Webpack Web Web Development Website

11.3K 51

WRITTEN BY

David Gilbertson

I like web stuff.

Follow

HackerNoon.com

Elijah McClain, George Floyd, Eric Garner, Breonna Taylor, Ahmaud Arbery, Michael Brown, Oscar Grant, Atatiana Jefferson, Tamir Rice, Bettie Jones, Botham Jean

Follow

More From Medium

10 Basic Fundamental Concepts of JavaScript
Md Shahadat Hossain

When To Subscribe
Michael Lorton in Angular In Depth

How to Build Your Own Codepen-Style Editor App
Moshe Kerbel in Frontend Weekly

Is Angular dying because of React?
Aphinya Dechalert in hashmap

Passing ID vs. Entity as props in React
Marco in Weekly Webtips

Getting Started With React.Js
Okonu Deborah in Weekly Webtips

Get started on React Native with Expo
Karolina Porcioncula

Developing Content Layouts for Content and Experience Cloud
Sivakumar Balagopalan in Oracle Developers

Learn more.

Medium is an open platform where 170 million readers come to find insightful and dynamic thinking. Here, expert and undiscovered voices alike dive into the heart of any topic and bring new ideas to the surface. Learn more

Make Medium yours.

Follow the writers, publications, and topics that matter to you, and you'll see them on your homepage and in your inbox. Explore

Share your thinking.

If you have a story to tell, knowledge to share, or a perspective to offer — welcome home. It's easy and free to post your thinking on any topic. Write on Medium

Medium

About Help Legal