Webpack From Zero to Hero

Chapter 5: Route Based Code Splitting with React

Rubens Pinheiro Gonçalves Cavalcante Follow
Jul 22 · 5 min read ★

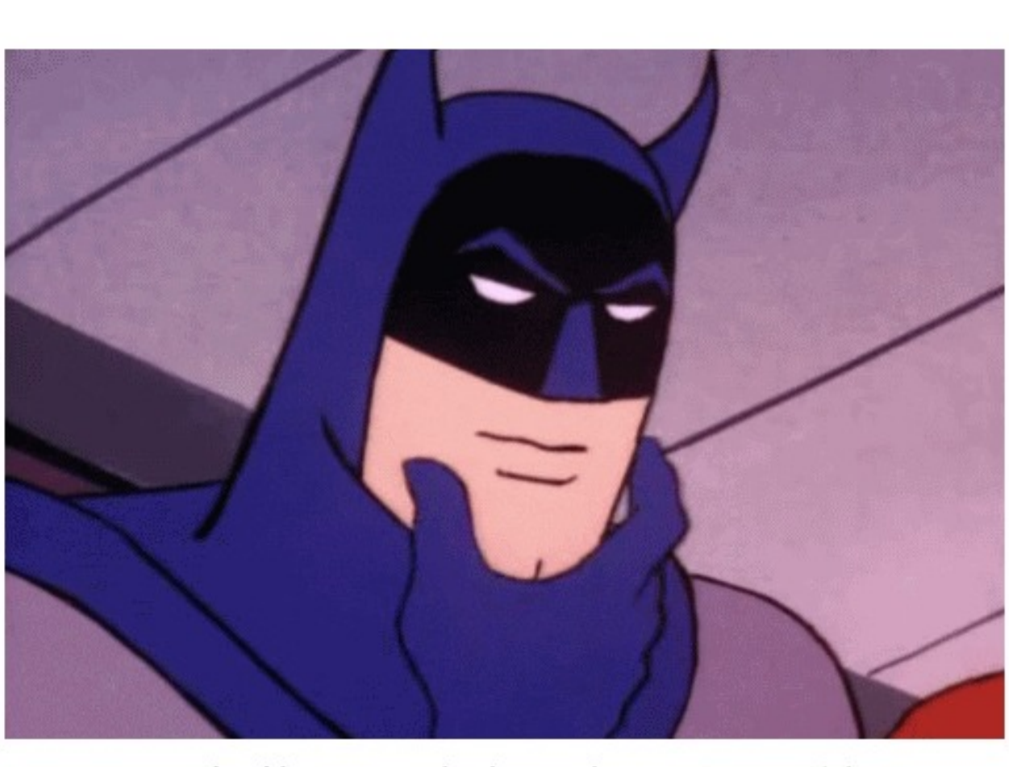
Fork in the Road — Picture under Creative Commons License (source: Flickr)

This article is part of the **Webpack from Zero to Hero series**, for more background or for the index you can check the “Chapter 0: History”.

Previous — Chapter 4: Dynamic Imports and Code Splitting

Introduction

In the last chapter you learned how to add dynamic imports to your app and play around how Webpack does code-splitting. In this last chapter, we’ll add React and react-router to the app and do **routing based code splitting**. Let’s start!



Now the things are getting interesting… — Source: Giphy

Adding React to the App

Adding JSX support on Babel

As the job to transpile the code isn’t in Webpack but in BabelJS’ hands, let’s make it read and transpile JSX.

```
yarn add @babel/preset-react --dev
```

And add it to `.babelrc`:

```
{
  "presets": ["@babel/preset-env", "@babel/preset-react"],
  "plugins": ["@babel/plugin-syntax-dynamic-import"]
}
```

.babelrc hosted with ❤ by GitHub view raw

Now that all is set, let’s install React as a **dependency**:

```
yarn add react react-dom
```

Webpack tweaks 📦

Babel loader

But wait! Webpack rule must be changed. Remember it only sends files having names ending with `.js` to Babel? Let’s change the regex to accept `.jsx` too (`.jsx` are files which will render our React components):

```
{
  test: /\.jsx?$/,
  use: "babel-loader"
},
```

babel-rule.js hosted with ❤ by GitHub view raw

The `?x` means that it is optionally present in a match. This means that it matches both `.js` and `.jsx`.

To finish, let’s make Webpack be able to resolve all `.jsx` without the necessity of explicitly specifying this extension on imports.

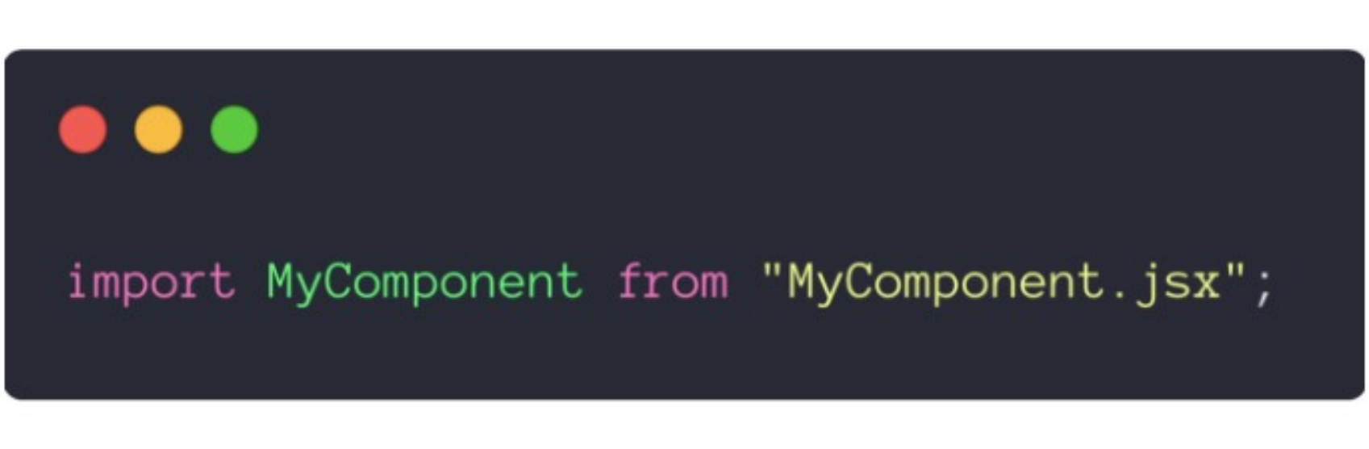
Resolving JSX

On webpack config we add to the default extension values the `.jsx`:

```
resolve: {
  extensions: [".wasm", ".mjs", ".js", ".json", ".jsx"];
},
```

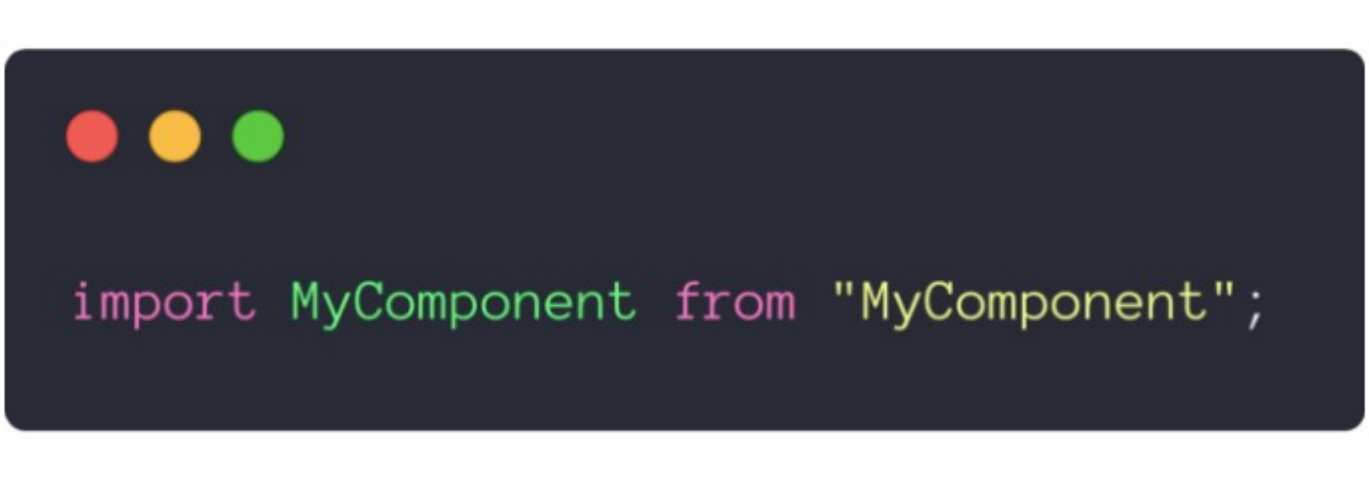
webpack-resolve.js hosted with ❤ by GitHub view raw

Now instead of doing:



```
import MyComponent from "MyComponent.jsx";
```

You can import without explicitly saying the **extension**:



```
import MyComponent from "MyComponent";
```

> 🤓 — *“Why don’t you make images, styles and media extensions to be automatically resolved too?”*

My personal view is, if it’s an asset it should be explicitly stated that **it’s an asset**, and we can easily determine it from extensions. Another problem to avoid here is **naming collision**, like a style file with the same name as the component, or having the same image but with different extensions like `.png` and `.webp`, etc.

React Development Mode

React Development bundle is **way bigger** than the production one. To tell React which one we want to use, we need to provide it to Node.js using the `NODE_ENV` variable.

Since Webpack doesn’t use `NODE_ENV` anymore though, we can use the `mode` parameter to provide this value to React. Let’s include the Define Plugin on `webpack.config.js`:

```
const { DefinePlugin } = require(“webpack”);
```

And add it to the plugins section:

```
new DefinePlugin({
  "process.env": {
    NODE_ENV: JSON.stringify(argv.mode)
  }
}),
```

define-plugin.js hosted with ❤ by GitHub view raw

Now we can use React development/production builds accordingly.

HTML template to Component

Throughout the chapters, we were using a string template to create our HTML, but it’s time to convert it into a React Component. First let’s rename our `index.js` to `index.jsx`, and replace the content to use a React component:

```
import React, { useState } from "react";
import { render } from "react-dom";

import andHisNameIs from "./assets/and-his-name-is.mp3";
import johnCena from "./assets/unexpected.jpg";
import "./style.scss";

const audio = new Audio(andHisNameIs);

const App = () => {
  const [personState, setPersonState] = useState("😴");
  const wakeUp = () =>
    import(/* webpackChunkName: "myAwesomeLazyModule", webpackPreload: true */ "./lazy-c
      mod => setPersonState(mod.default)
    );
  const lazyBtnStyle = {
    margin: "10px auto",
    display: "flex",
    fontSize: "4rem"
  };

  return (
    <div id="myMemes">
      <h1>You can't expect...</h1>
      <img src={johnCena} role="button" onClick={() => audio.play()} />
      <button style={lazyBtnStyle} onClick={() => wakeUp()}>
        {personState}
      </button>
    </div>
  );
};

const wrapper = document.createElement("div");
wrapper.setAttribute("id", "app");
document.body.appendChild(wrapper);

render(<App />, wrapper);
```

index.jsx hosted with ❤ by GitHub view raw

Now we have exactly the same app from the other chapters, but with React. 🎉

The Bundle became HUGE! 📦 💥

Now that we have `vendors` code, we need to worry about another thing, the size of the main bundle and possible duplications.

If you run `yarn analyse` you’ll see that a big slice of your app is basically **React**. Luckily for us, Webpack opens some **optimization** setup to us.

Normally I split my apps into two bundles, first and third party code. Let’s add a configuration for this on `webpack.config.js`:

```
optimization: {
    splitChunks: {
      cacheGroups: {
        commons: {
          test: /[\\/]node_modules[\\/]/,
          name: 'vendors',
          chunks: 'all'
        }
      }
    }
  },
```

webpack-optimization.js hosted with ❤ by GitHub view raw

Running `yarn analyse` again, you’ll see the `vendors.js` bundle, with anything that comes from `node_modules` in it.

Setting Up React Router

After the tweak to keep our bundle split between source and vendors, let’s setup **react-router** and start to create some routing in our app.

First let’s install it:

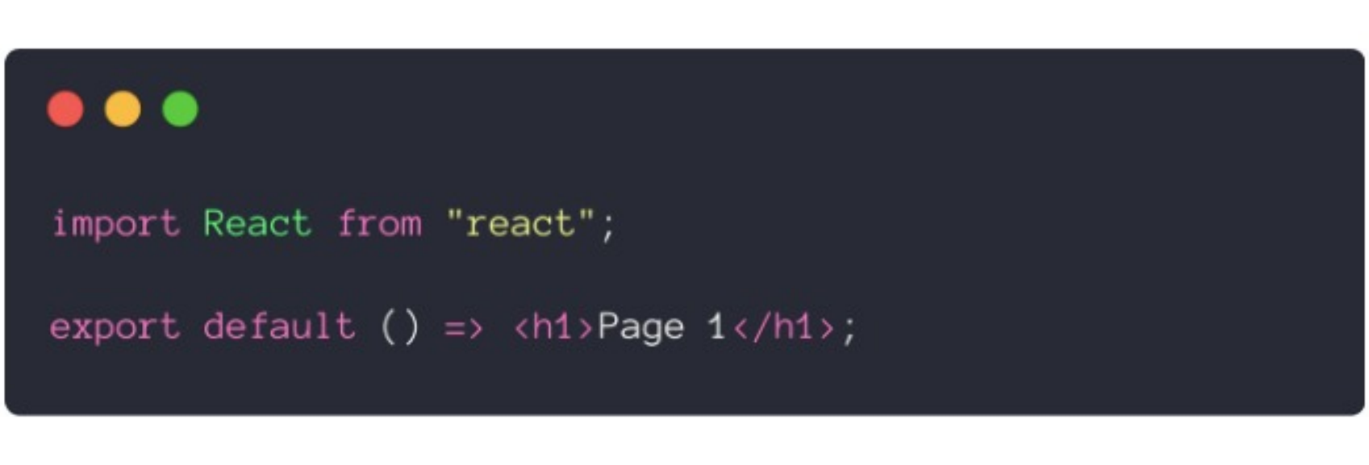
```
yarn add react-router-dom
```

Then let’s enable the history API on *webpack-dev-server*, in the `package.json` change the script “**start:dev**” to:

```
webpack-dev-server --mode=development --history-api-fallback
```

Creating the Routes

Let’s create 4 modules with some dummy components inside the `modules/` directory:



```
import React from "react";

export default () => <h1>Page 1</h1>;
```

Each page only changes the `Page ${number}` and is named after it - `Page-${number}.jsx`.

Now let’s get rid of the content in `index.jsx` and replace it with our routes:

```
import React, { Fragment } from "react";
import { BrowserRouter, Route, Link } from "react-router-dom";
import { render } from "react-dom";

import Page1 from "./modules/Page-1";
import Page2 from "./modules/Page-2";
import Page3 from "./modules/Page-3";
import Page4 from "./modules/Page-4";

const App = () => (
  <BrowserRouter>
    <Fragment>
      <Route path="/" exact component={() => <h1>Home Page</h1>} />
      <Route path="/page-1" component={Page1} />
      <Route path="/page-2" component={Page2} />
      <Route path="/page-3" component={Page3} />
      <Route path="/page-4" component={Page4} />

      <ul>
        {[1, 2, 3, 4].map(number => (
          <li key={number}>
            <Link to={`/page-${number}`}>Page {number}</Link>
          </li>
        ))}
      </ul>
    </Fragment>
  </BrowserRouter>
);

const wrapper = document.createElement("div");
wrapper.setAttribute("id", "app");
document.body.appendChild(wrapper);

render(<App />, wrapper);
```

index.jsx hosted with ❤ by GitHub view raw

Lazy Loading the Components on the Routes

Ok, nothing new in the code from above. Just some static defined routes, right?

But now, with `React.lazy` and `<Suspense>`, we’re capable to defer the loading of a component. For that we just need to use the previously seen **dynamic import** and convert the **static routes** to **lazy routes**. Let’s change the module imports code for this:

```
import React, { Suspense, lazy } from "react";

// ... other imports

const lazyRoute = lazyModule => {
  const LazyComponent = lazy(lazyModule);
  return (
    <Suspense fallback={<div>Loading ...</div>}>
      <LazyComponent />
    </Suspense>
  );
};

const Page1 = () => lazyRoute(() => import("./modules/Page-1"));
const Page2 = () => lazyRoute(() => import("./modules/Page-2"));
const Page3 = () => lazyRoute(() => import("./modules/Page-3"));
const Page4 = () => lazyRoute(() => import("./modules/Page-4"));

// ... the App component goes after that :)
```

lazy-routes-def.jsx hosted with ❤ by GitHub view raw

If you open the **network** tab in your browser, you can see the page loading **only after clicking the link**. Other thing to notice is, Webpack caches any dynamic imported module already requested, so if you click on a previous visited link, no network requests are made again.

Prefetching Important Page

Let’s say the `Page-4` is very important to your business, and you don’t want any delay if a user clicks on it. Let’s use what we already learned in the previous chapter **to prefetch this page**:

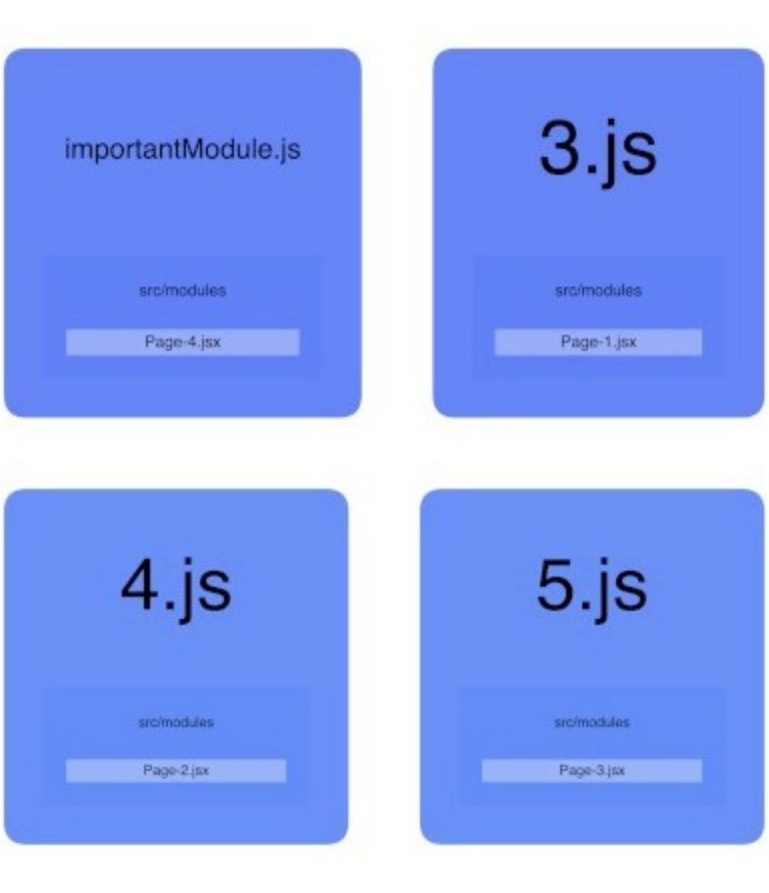
```
const Page4 = () =>
  lazyRoute(() =>
    import(
      /* webpackPrefetch: true, webpackChunkName: "importantModule" */
      "./modules/Page-4"
    )
  );
```

page-4-prefetch.js hosted with ❤ by GitHub view raw

And in the `<head>` of the page you will find something like:

```
<link rel="prefetch" as="script" href="importantModule.js" />
```

Analysing the Results



As you can see with `yarn analyse`, the pages were split into 4 small chunks, one of them being our prefetched module `importantModule.js`

You can keep up this strategy not only for routes, but also for sub-routes — depending on their sizes, it can be a smart choice!

. . .

Finally we finished a production-ready web app setup. It was quite a long journey of whole five chapters, but I hope that it was totally worth it for you!

If you liked this article, please don’t mind giving it some claps 👏, subscribe to OLX Tech Blog for more updates and share it to your friends! Your support is very important to us! See you!

React Webpack Learning Reactjs JavaScript

38 claps