We will see the basics of Webpack for React to get you started, including React Router, Hot Module Replacement (HMR), Code Splitting by Route and Vendor, production configuration and more.

Before we start, here’s the full list of features we are going to set up together:

* React 16
* React Router 5
* Semantic UI as the CSS Framework
* Hot Module Replacement (HMR)
* CSS Autoprefixer
* CSS Modules
* @babel/plugin-proposal-class-properties
* @babel/plugin-syntax-dynamic-import
* Webpack 4
* Code Splitting by Route and Vendor
* Webpack Bundle Analyzer

**Pre-requisites**

Have the following pre-installed:

* Yarn — Package manager, similar to npm
* Node

And you should have at least some basic knowledge of React and React Router.

***Note:****You can use npm if you wish, although the commands will vary slightly.*

**Initial Dependencies**

Let us start by creating our directory and package.json.

In your terminal type the following:

mkdir webpack-for-react && cd $\_

yarn init -y

This first command will create our directory and move into it, then we initialize a package.json accepting defaults.  
If you inspect it you will see the bare bones configuration:

{

"name": "webpack-for-react",

"version": "1.0.0",

"main": "index.js",

"license": "MIT"

}

Now we install our initial (production) dependencies and development dependencies.  
In your terminal type the following:

yarn add react react-dom prop-types react-router-dom semantic-ui-react

yarn add @babel/core babel-loader @babel/preset-env @babel/preset-react @babel/plugin-proposal-class-properties @babel/plugin-syntax-dynamic-import css-loader style-loader html-webpack-plugin webpack webpack-dev-server webpack-cli -D

The development dependencies will only be used, as implied, during the development phase, and the (production) dependencies is what our application needs in production.

{

"name": "webpack-for-react",

"version": "1.0.0",

"main": "index.js",

"license": "MIT",

"dependencies": {

"react": "^16.2.0",

"react-dom": "^16.2.0",

"prop-types": "^15.6.2",

"react-router-dom": "^4.2.2",

"semantic-ui-react": "^0.77.1"

},

"devDependencies": {

"@babel/core": "^7.0.0",

"@babel/plugin-proposal-class-properties": "^7.0.0",

"@babel/plugin-syntax-dynamic-import": "^7.0.0",

"@babel/preset-env": "^7.0.0",

"@babel/preset-react": "^7.0.0",

"babel-loader": "^8.0.1",

"css-loader": "^0.28.10",

"html-webpack-plugin": "^3.0.4",

"style-loader": "^0.19.1",

"webpack": "^4.0.0",

"webpack-cli": "^2.0.14",

"webpack-dev-server": "^3.0.0"

}

}

***Note:****Dependencies versions might be different than yours from the time of this writing.*

* react — I’m sure you know what React is
* react-dom — Provides DOM-specific methods for the browser
* prop-types — Runtime type checking for React props
* react-router-dom — Provides routing capabilities to React for the browser
* semantic-ui-react — CSS Framework
* @babel/core — Core dependencies for Babel
* Babel is a transpiler that compiles JavaScript ES6 to JavaScript ES5 allowing you to write JavaScript “from the future” so that current browsers will understand it. Detailed description in Quora.
* babel-loader — This package allows transpiling JavaScript files using Babel and webpack
* @babel/preset-env — With this you don’t have to specify if you will be writing ES2015, ES2016 or ES2017. Babel will automatically detect and transpile accordingly.
* @babel/preset-react — Tells Babel we will be using React
* @babel/plugin-proposal-class-properties — Use class properties. We don’t use Class Properties in this project, but you will more than likely use them in your project
* @babel/plugin-syntax-dynamic-import — Be able to use dynamic imports
* css-loader — Interprets @import and url() like import/require() and will resolve them
* html-webpack-plugin — Can generate an HTML file for your application, or you can provide a template
* style-loader — Adds CSS to the DOM by injecting a <style> tag
* webpack — Module bundler
* webpack-cli — Command Line Interface, needed for Webpack 4.0.1 and latest
* webpack-dev-server — Provides a development server for your application

**Setting up Babel**

In the root directory (webpack-for-react) we create the Babel configuration file.

touch .babelrc

At this point you can open your, then point the editor to the root of this project and open .babelrc file and copy the following:

{

"presets": ["@babel/preset-env", "@babel/preset-react"],

"plugins": [

"@babel/plugin-syntax-dynamic-import",

"@babel/plugin-proposal-class-properties"

]

}

This tells Babel to use the presets (plugins) we previously installed. Later when we call babel-loader from Webpack, this is where it will look to know what to do.

**Setting up Webpack**

Let’s create the Webpack configuration file.

In your terminal type the following:

touch webpack.config.js

Open webpack.config.js and copy the following:

const webpack = require('webpack');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const port = process.env.PORT || 3000;

module.exports = {

// Webpack configuration goes here

};

This is the basic *shell* for Webpack. We require webpack and html-webpack-plugin. Provide a default port if the environment variable PORT does not exist and export the module.

The following will be additions for webpack.config.js (one after another).

...

module.exports = {

mode: 'development',

};

mode tells Webpack this configuration will be for either development or production. “Development Mode [is] optimized for speed and developer experience… Production defaults will give you a set of defaults useful for deploying your application (webpack 4: mode and optimization)”.

...

module.exports = {

entry: './src/index.js',

output: {

filename: 'bundle.[hash].js'

},

};

To get a running instance of Webpack we need:

* entry — Specifies the entry point of your application; this is where your React app lives and where the bundling process will begin

*Webpack 4 introduced some defaults, so if you don’t include entry in your configuration, then Webpack will assume your entry point is located under the ./src directory, making entry optional as opposed to Webpack 3. leave entry as it makes it obvious where our entry point will be, but you are more than welcome to remove it if you so decide.*

* output — Tells Webpack how to write the compiled files to disk
* filename — This will be the filename of the bundled application. The [hash] portion of the filename will be replaced by a hash generated by Webpack every time your application changes and is recompiled (helps with caching).

...

module.exports = {

...

devtool: 'inline-source-map',

};

devtool will create source maps to help you with debugging of your application. There are several types of source maps and this particular map (inline-source-map) is to be used only in development.

...

module.exports = {

...

module: {

rules: [

// First Rule

{

test: /\.(js)$/,

exclude: /node\_modules/,

use: ['babel-loader']

},

// Second Rule

{

test: /\.css$/,

use: [

{

loader: 'style-loader'

},

{

loader: 'css-loader',

options: {

modules: true,

localsConvention: 'camelCase',

sourceMap: true

}

}

]

}

]

},

};

* module — What types of modules your application will include, in our case we will support ESNext (Babel) and CSS Modules
* rules — How we handle each different type of module

**First Rule**

We test for files with a .js extension excluding the node\_modules directory and use Babel, via babel-loader, to transpile down to vanilla JavaScript (basically, looking for our React files).

Remember our configuration in .babelrc? This is where Babel looks at that file.

**Second Rule**

We test for CSS files with a .css extension. Here we use two loaders, style-loader and css-loader, to handle our CSS files. Then we instruct css-loader to use *CSS Modules*, camel case and create source maps.

**CSS Modules and Camel Case**

This gives us the ability to use import Styles from ‘./styles.css’ syntax (or destructuring like this import { style1, style2 } from ‘./styles.css’).

Then we can use it like this in a React app:

...

<div className={Style.style1}>Hello World</div>

// or with the destructuring syntax

<div className={style1}>Hello World</div>

...

Camel case gives us the ability to write our CSS rules like this:

.home-button {...}

And use it in our React files like this:

...

import { homeButton } from './styles.css'

...

...

module.exports = {

...

plugins: [

new HtmlWebpackPlugin({

template: 'public/index.html',

favicon: 'public/favicon.ico'

})

],

};

This section is where we configure (as the name implies) plugins.

html-webpack-plugin accepts an object with different options. In our case we specify the HTML template we will be using and the favicon.

Later we will be adding other plugins for Bundle Analyzer and HMR.

...

module.exports = {

...

devServer: {

host: 'localhost',

port: port,

historyApiFallback: true,

open: true

}

};

Finally, we configure the development server. We specify localhost as the host and assign the variable port as the port (if you remember, we assigned port 3000 to this variable). We set historyApiFallback to true and open to true. This will open the browser automatically and launch your application in [http://localhost:3000.](http://localhost:3000./)

Now, below is the complete Webpack configuration. (webpack.config.js):

const webpack = require('webpack');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const port = process.env.PORT || 3000;

module.exports = {

mode: 'development',

entry: './src/index.js',

output: {

filename: 'bundle.[hash].js'

},

devtool: 'inline-source-map',

module: {

rules: [

{

test: /\.(js)$/,

exclude: /node\_modules/,

use: ['babel-loader']

},

{

test: /\.css$/,

use: [

{

loader: 'style-loader'

},

{

loader: 'css-loader',

options: {

modules: true,

localsConvention: 'camelCase',

sourceMap: true

}

}

]

}

]

},

plugins: [

new HtmlWebpackPlugin({

template: 'public/index.html',

favicon: 'public/favicon.ico'

})

],

devServer: {

host: 'localhost',

port: port,

historyApiFallback: true,

open: true

}

};

**Creating the React App**

We will be creating a simple Hello World app with three routes: a *home*, a *page not found* and a *dynamic page* that we will be loading asynchronously when we implement code splitting later.

We currently have the following project structure:

|-- node\_modules

|-- .babelrc

|-- package.json

|-- webpack.config.js

|-- yarn.lock

In your terminal type the following:

mkdir public && cd $\_

touch index.html

We create a public directory, move into it and also create an index.html file. Here is where we also have the favicon.

Open the index.html file and copy the following:

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<meta http-equiv="X-UA-Compatible" content="ie=edge">

<link rel="stylesheet" href="//cdnjs.cloudflare.com/ajax/libs/semantic-ui/2.2.13/semantic.min.css"></link>

<title>webpack-for-react</title>

</head>

<body>

<div id="root"></div>

</body>

</html>

Nothing much here (just a standard HTML template) only, we are adding the Semantic UI stylesheet and also creating a div with an ID of root. This is where our React app will render.

Back to your terminal type the following:

cd ..

mkdir src && cd $\_

touch index.js

Open index.js and copy the following:

import React from 'react';

import ReactDOM from 'react-dom';

import App from './components/App';

ReactDOM.render(<App />, document.getElementById('root'));

In your terminal type the following:

mkdir components && cd $\_

touch App.js Layout.js Layout.css Home.js DynamicPage.js NoMatch.js

After creating the React component files, we have the following project structure:

|-- node\_modules

|-- public

|-- index.html

|-- favicon.ico

|-- src

|-- components

|-- App.js

|-- DynamicPage.js

|-- Home.js

|-- Layout.css

|-- Layout.js

|-- NoMatch.js

|-- index.js

|-- .babelrc

|-- package.json

|-- webpack.config.js

|-- yarn.lock

Open App.js and copy the following:

import React from 'react';

import { Switch, BrowserRouter as Router, Route } from 'react-router-dom';

import Home from './Home';

import DynamicPage from './DynamicPage';

import NoMatch from './NoMatch';

const App = () => {

return (

<Router>

<div>

<Switch>

<Route exact path="/" component={Home} />

<Route exact path="/dynamic" component={DynamicPage} />

<Route component={NoMatch} />

</Switch>

</div>

</Router>

);

};

export default App;

We create our basic “shell” with React Router and have a *home*, *dynamic page* and *page not found*route.

Open Layout.css and copy the following:

.pull-right {

display: flex;

justify-content: flex-end;

}

.h1 {

margin-top: 10px !important;

margin-bottom: 20px !important;

}

Open Layout.js and copy the following:

import React from 'react';

import { Link } from 'react-router-dom';

import { Header, Container, Divider, Icon } from 'semantic-ui-react';

import { pullRight, h1 } from './layout.css';

const Layout = ({ children }) => {

return (

<Container>

<Link to="/">

<Header as="h1" className={h1}>

webpack-for-react

</Header>

</Link>

{children}

<Divider />

</Container>

);

};

export default Layout;

This is our container component where we define the layout of the site. Making use of CSS Modules, we are importing two CSS rules from layout.css.

Open Home.js and copy the following:

import React from 'react';

import { Link } from 'react-router-dom';

import Layout from './Layout';

const Home = () => {

return (

<Layout>

<p>Hello World of React and Webpack!</p>

<p>

<Link to="/dynamic">Navigate to Dynamic Page</Link>

</p>

</Layout>

);

};

export default Home;

Open DynamicPage.js and copy the following:

import React from 'react';

import { Header } from 'semantic-ui-react';

import Layout from './Layout';

const DynamicPage = () => {

return (

<Layout>

<Header as="h2">Dynamic Page</Header>

<p>This page was loaded asynchronously!!!</p>

</Layout>

);

};

export default DynamicPage;

Open NoMatch.js and copy the following:

import React from 'react';

import { Icon, Header } from 'semantic-ui-react';

import Layout from './Layout';

const NoMatch = () => {

return (

<Layout>

<Icon name="minus circle" size="big" />

<strong>Page not found!</strong>

</Layout>

);

};

export default NoMatch;

We are done creating the React components. For a final step before running our application, open package.json and add the lines:

{

"name": "webpack-for-react",

"version": "1.0.0",

"main": "index.js",

"license": "MIT",

"scripts": {

"start": "webpack-dev-server"

},

"dependencies": {

"react": "^16.2.0",

"react-dom": "^16.2.0",

"prop-types": "^0.4.0",

"react-router-dom": "^4.2.2",

"semantic-ui-react": "^0.77.1"

},

"devDependencies": {

"@babel/core": "^7.0.0",

"@babel/plugin-proposal-class-properties": "^7.0.0",

"@babel/plugin-syntax-dynamic-import": "^7.0.0",

"@babel/preset-env": "^7.0.0",

"@babel/preset-react": "^7.0.0",

"babel-loader": "^8.0.1",

"css-loader": "^0.28.10",

"html-webpack-plugin": "^3.0.4",

"style-loader": "^0.19.1",

"webpack": "^4.0.0",

"webpack-cli": "^2.0.14",

"webpack-dev-server": "^3.0.0"

}

}

We add the scripts key and also the start key. This will allow us to run React with the Webpack Development Server. If you don’t specify a configuration file, webpack-dev-server will look for webpack.config.js file as the default configuration entry within the root directory.

Type the following in your terminal (remember to be in the root directory) and Yarn will call our start script.

yarn start

Running React

Now we have a working React app powered by our own Webpack configuration. Bundled JavaScript file Webpack generated for us, and as we indicated in the configuration, the filename has a unique hash, bundle.d505bbab002262a9bc07.js.

**Setting up Hot Module Replacement (HMR)**

Back to your terminal, install React Hot Loader as a development dependency.

yarn add react-hot-loader @hot-loader/react-dom -D

Open .babelrc and add lines 3 and 9. Don’t forget to include the comma (,) at the end of line 3:

{

"presets": [

["@babel/preset-env", { "modules": false }],

"@babel/preset-react"

],

"plugins": [

"@babel/plugin-syntax-dynamic-import",

"@babel/plugin-proposal-class-properties",

"react-hot-loader/babel"

]

}

Open webpack.config.js and modify it as below.

...

module.exports = {

entry: './src/index.js',

output: {

...

publicPath: '/'

},

resolve: {

alias: {

"react-dom": "@hot-loader/react-dom",

},

},

...

plugins: [

new webpack.HotModuleReplacementPlugin(),

...

],

devServer: {

...

hot: true

}

};

* publicPath: ‘/’ — Hot reloading won’t work as expected for nested routes without it
* webpack.HotModuleReplacementPlugin — Prints more readable module names in the browser terminal on HMR updates
* hot: true — Enable HMR on the server
* resolve: alias — replaces react-dom with the custom react-dom from hot-loader

Open index.js and change it to the following.

import { hot } from "react-hot-loader/root";

import React from "react";

import ReactDOM from "react-dom";

import App from "./components/App";

import "./index.css";

const render = (Component) =>

ReactDOM.render(<Component />, document.getElementById("root"));

render(hot(App));

Now we are ready to test HMR! Back in the terminal run your app, make a change, and watch as the app updates without a full-page refresh.

yarn start

HMR in action

After updating the file, the page changes without a full refresh. To show this change in the browser I select **Rendering -> Paint flash**ing in Chrome DevTools, which highlights the areas of the page, in green, that changed.

**Code Splitting**

With code splitting, instead of having your application in one big bundle, you can have multiple bundles each loading asynchronously or in parallel. Also you can separate vendor code from you app code which can potentially decrease loading time.

**By Route**

There are several ways we can achieve code splitting by route, however in our case we will be using react-imported-component.

We would also like to show a *loading spinner* when the user navigates to a different route. This is a good practice as we don’t want the user to just stare at a blank screen while he/she waits for the new page to load. So, we will be creating a *Loading* component.

However, if the new page loads really fast, we don’t want the user to see a flashing loading spinner for a couple of milliseconds, so we will delay the *Loading* component by 300 milliseconds. To achieve this, we will be using React-Delay-Render.

Start by installing the two additional dependencies.

In your terminal type the following:

yarn add react-imported-component react-delay-render

Now we are going to create the *Loading* components.

In your terminal type the following:

touch ./src/components/Loading.js

Open Loading.js and copy the following:

import React from 'react';

import { Loader } from 'semantic-ui-react';

import ReactDelayRender from 'react-delay-render';

const Loading = () => <Loader active size="massive" />;

export default ReactDelayRender({ delay: 300 })(Loading);

Now that we have the *Loading* component, open App.js and modify it as follows:

import React from 'react';

import { Switch, BrowserRouter as Router, Route } from 'react-router-dom';

import importedComponent from 'react-imported-component';

import Home from './Home';

import Loading from './Loading';

const AsyncDynamicPAge = importedComponent(

() => import(/\* webpackChunkName:'DynamicPage' \*/ './DynamicPage'),

{

LoadingComponent: Loading

}

);

const AsyncNoMatch = importedComponent(

() => import(/\* webpackChunkName:'NoMatch' \*/ './NoMatch'),

{

LoadingComponent: Loading

}

);

const App = () => {

return (

<Router>

<div>

<Switch>

<Route exact path="/" component={Home} />

<Route exact path="/dynamic" component={AsyncDynamicPAge} />

<Route component={AsyncNoMatch} />

</Switch>

</div>

</Router>

);

};

export default App;

This will create three bundles, or chunks, one for the DynamicPage component, one for the NoMatch component, and one for the main app.

Let’s also change the bundle filename. Open webpack.config.js and change it as follows:

...

module.exports = {

...

output: {

filename: '[name].[hash].js',

...

},

}

It is time to run the app and take a look at code splitting by route in action.

yarn start

Code Splitting by Route

Three different chunks created by Webpack in terminal. Then I highlight that upon the app launching, only the main chuck was loaded. Finally, we see that upon clicking *Navigate to Dynamic Page* the chunk corresponding to this page loaded asynchronously.

We also see that the chunk corresponding to the *page not found* was never loaded, saving the user bandwidth.

**By Vendor**

Now let’s split the application by vendor. Open webpack.config.js and make the following changes:

...

module.exports = {

entry: {

vendor: ['semantic-ui-react'],

app: './src/index.js'

},

...

optimization: {

splitChunks: {

cacheGroups: {

styles: {

name: 'styles',

test: /\.css$/,

chunks: 'all',

enforce: true

},

vendor: {

chunks: 'initial',

test: 'vendor',

name: 'vendor',

enforce: true

}

}

}

},

...

};

* entry.vendor: [‘semantic-ui-react’] — Specifies which library we want to extract from our main app and into the *vendor* chunk
* optimization — if you leave out this entry, Webpack will still split your application by vendor, however I noticed the bundle sizes were big and after adding this entry, the bundle sizes were reduced significantly. (I got this from Webpack 4 migration draft *CommonsChunkPlugin -> Initial vendor ch*unk)

***Note:****Previously Webpack 3 made use of the CommonsChunkPlugin to split the code by vendor and/or commons, however it was deprecated in Webpack 4 and many of its features are now enabled by default. With the removal of CommonsChunkPlugin they have added optimization.splitChunks for those who need fine-grained control over their caching-strategy.*

In terminal, launch the app:

yarn start

Code Splitting by Vendor

In terminal, three previous chunks plus the new *vendor* chunk. Then when we inspect the HTML we see that both vendor and app chunks were loaded.

Since we have made several updates to our Webpack configuration, below you will find the complete webpack.config.js file.

const webpack = require('webpack');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const port = process.env.PORT || 3000;

module.exports = {

mode: 'development',

entry: {

vendor: ['semantic-ui-react'],

app: './src/index.js'

},

output: {

filename: '[name].[hash].js'

},

resolve: {

alias: {

"react-dom": "@hot-loader/react-dom",

},

},

devtool: 'inline-source-map',

module: {

rules: [

{

test: /\.(js)$/,

exclude: /node\_modules/,

use: ['babel-loader']

},

{

test: /\.css$/,

use: [

{

loader: 'style-loader'

},

{

loader: 'css-loader',

options: {

modules: true,

localsConvention: 'camelCase',

sourceMap: true

}

}

]

}

]

},

optimization: {

splitChunks: {

cacheGroups: {

styles: {

name: 'styles',

test: /\.css$/,

chunks: 'all',

enforce: true

},

vendor: {

chunks: 'initial',

test: 'vendor',

name: 'vendor',

enforce: true

}

}

}

},

plugins: [

new webpack.HotModuleReplacementPlugin(),

new HtmlWebpackPlugin({

template: 'public/index.html',

favicon: 'public/favicon.ico'

})

],

devServer: {

host: 'localhost',

port: port,

historyApiFallback: true,

open: true,

hot: true

}

};

**Production Configuration**

Rename the Webpack configuration from webpack.config.js to webpack.config.**development**.js. Then make a copy and name it webpack.config.production.js.

In your terminal type the following:

mv webpack.config.js webpack.config.development.js

cp webpack.config.development.js webpack.config.production.js

We will need a development dependency, Extract Text Plugin. From their docs: “It moves all the required \*.css modules in entry chunks into a separate CSS file. So, your styles are no longer inlined into the JS bundle, but in a separate CSS file (styles.css). If your total stylesheet volume is big, it will be faster because the CSS bundle is loaded in parallel to the JS bundle.”

In your terminal type the following:

yarn add mini-css-extract-plugin -D

Open webpack.config.production.js and make the following bolded changes:

const path = require('path');

const webpack = require('webpack');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {

mode: 'production',

entry: {

vendor: ['semantic-ui-react'],

app: './src/index.js'

},

output: {

// We want to create the JavaScript bundles under a

// 'static' directory

filename: 'static/[name].[hash].js',

// Absolute path to the desired output directory. In our

//case a directory named 'dist'

// '\_\_dirname' is a Node variable that gives us the absolute

// path to our current directory. Then with 'path.resolve' we

// join directories

// Webpack 4 assumes your output path will be './dist' so you

// can just leave this

// entry out.

path: path.resolve(\_\_dirname, 'dist'),

publicPath: '/'

},

// Change to production source maps

devtool: 'source-map',

module: {

rules: [

{

test: /\.(js)$/,

exclude: /node\_modules/,

use: ['babel-loader']

},

{

test: /\.css$/,

use: [

{

// We configure 'MiniCssExtractPlugin'

loader: MiniCssExtractPlugin.loader,

},

{

loader: 'css-loader',

options: {

modules: true,

// Allows to configure how many loaders

// before css-loader should be applied

// to @import(ed) resources

importLoaders: 1,

localsConvention: 'camelCase',

// Create source maps for CSS files

sourceMap: true

}

},

{

// PostCSS will run before css-loader and will

// minify and autoprefix our CSS rules.

loader: 'postcss-loader',

}

]

}

]

},

optimization: {

splitChunks: {

cacheGroups: {

styles: {

name: 'styles',

test: /\.css$/,

chunks: 'all',

enforce: true

},

vendor: {

chunks: 'initial',

test: 'vendor',

name: 'vendor',

enforce: true

}

}

}

},

plugins: [

new HtmlWebpackPlugin({

template: 'public/index.html',

favicon: 'public/favicon.ico'

}),

// Create the stylesheet under 'styles' directory

new MiniCssExtractPlugin({

filename: 'styles/styles.[hash].css'

})

]

};

Notice we removed the port variable, the plugins related to HMR and the devServer entry.

Also since we added PostCSS to the production configuration, we need to install it and create a configuration file for it.

In your terminal type the following:

yarn add postcss-loader autoprefixer cssnano postcss-preset-env -D

touch postcss.config.js

Open postcss.config.js and copy the following:

const postcssPresetEnv = require('postcss-preset-env');

module.exports = {

plugins: [

postcssPresetEnv({

browsers: ['>0.25%', 'not ie 11', 'not op\_mini all']

}),

require('cssnano')

]

};

Here we are specifying what browsers we want autoprefixer to support and minifying the CSS output.

Now for the last step before we create our production build, we need to create a *build* script in package.json.

Open the file and make the following changes to the scripts section:

...

"scripts": {

"dev":"webpack-dev-server --config webpack.config.development.js",

"prebuild": "rimraf dist",

"build": "cross-env NODE\_ENV=production webpack -p --config webpack.config.production.js"

},

...

First thing to notice here is that we changed the start script from start to dev, then we added two additional scripts, prebuild and build.

Finally, we are indicating which configuration to use when in development or production.

* prebuild — Will run before the build script and delete the dist directory created by our last production build. We use the library rimraf for this
* build — First we use cross-env library just in case somebody is using Windows. This way setting up environment variables with NODE\_ENV will work. Then we call Webpack with the -p flag to tell it to optimize this build for production, and finally we specify the production configuration.

In your terminal install the two new dependencies we included in package.json:

yarn add rimraf cross-env -D

Before creating the production build, let us look at our new project structure:

|-- node\_modules

|-- public

|-- index.html

|-- favicon.ico

|-- src

|-- components

|-- App.js

|-- DynamicPage.js

|-- Home.js

|-- Layout.css

|-- Layout.js

|-- Loading.js

|-- NoMatch.js

|-- index.js

|-- .babelrc

|-- package.json

|-- postcss.config.js

|-- webpack.config.development.js

|-- webpack.config.production.js

|-- yarn.lock

At last we can create our production bundle.

In your terminal type the following:

yarn build

Build production bundle

As you noticed, after we ran the build script, Webpack created a dist directory containing our production ready app. Now inspect the files that were created and notice they are minified and each has a corresponding source map. You will also notice PostCSS has added autoprefixing to the CSS file.

Now we take our production files and fire up a Node server to serve our site

Running the production build

At this point we have two working Webpack configurations, one for development and one for production. However, since both configurations are very similar, they share many of the same settings. If we wanted to add something else, we would have to add it to both configurations files.

**Webpack Composition**

Let’s start by installing webpack-merge and Chalk as development dependencies.

In your terminal type the following:

yarn add webpack-merge chalk -D

We will also need a couple of new directories and a few new files.

In your terminal type the following:

mkdir -p build-utils/addons

cd build-utils

touch build-validations.js common-paths.js webpack.common.js webpack.dev.js webpack.prod.js

Now let’s look at our new project structure:

|-- build-utils

|-- addons

|-- build-validations.js

|-- common-paths.js

|-- webpack.common.js

|-- webpack.dev.js

|-- webpack.prod.js

|-- node\_modules

|-- public

|-- index.html

|-- favicon.ico

|-- src

|-- components

|-- App.js

|-- DynamicPage.js

|-- Home.js

|-- Layout.css

|-- Layout.js

|-- Loading.js

|-- NoMatch.js

|-- index.js

|-- .babelrc

|-- package.json

|-- postcss.config.js

|-- webpack.config.development.js

|-- webpack.config.production.js

|-- yarn.lock

Open common-paths.js and copy the following:

const path = require('path');

const PROJECT\_ROOT = path.resolve(\_\_dirname, '../');

module.exports = {

projectRoot: PROJECT\_ROOT,

outputPath: path.join(PROJECT\_ROOT, 'dist'),

appEntry: path.join(PROJECT\_ROOT, 'src')

};

Here we define, as the name implies, the common paths for our Webpack configurations. PROJECT\_ROOT needs to look one directory up as we are working under build-utils directory (one level down from the actual root path in our project).

Open build-validations.js and copy the following:

const chalk = require('chalk');

const ERR\_NO\_ENV\_FLAG = chalk.red(

`You must pass an --env.env flag into your build for webpack to work!`

);

module.exports = {

ERR\_NO\_ENV\_FLAG

};

Later when we modify our package.json we will be requiring –env.env flag in the scripts. These validations are to verify that the flag is present; if not, it will throw an error.

In the next three files, we will be separating the Webpack configurations into configurations that are shared among development and production, configurations that are only for development and configurations only for production.

Open webpack.common.js and copy the following:

const commonPaths = require('./common-paths');

const webpack = require('webpack');

const HtmlWebpackPlugin = require('html-webpack-plugin');

const config = {

entry: {

vendor: ['semantic-ui-react']

},

output: {

path: commonPaths.outputPath,

publicPath: '/'

},

module: {

rules: [

{

test: /\.(js)$/,

exclude: /node\_modules/,

use: ['babel-loader']

}

]

},

optimization: {

splitChunks: {

cacheGroups: {

styles: {

name: 'styles',

test: /\.css$/,

chunks: 'all',

enforce: true

},

vendor: {

chunks: 'initial',

test: 'vendor',

name: 'vendor',

enforce: true

}

}

}

},

plugins: [

new HtmlWebpackPlugin({

template: 'public/index.html',

favicon: 'public/favicon.ico'

})

]

};

module.exports = config;

We basically extracted out what was shared among webpack.config.development.js and webpack.config.production.js and transferred it to this file. At the top we require common-paths.js to set the output.path.

Open webpack.dev.js and copy the following:

const commonPaths = require('./common-paths');

const webpack = require('webpack');

const port = process.env.PORT || 3000;

const config = {

mode: 'development',

entry: {

app: `${commonPaths.appEntry}/index.js`

},

output: {

filename: '[name].[hash].js'

},

resolve: {

alias: {

"react-dom": "@hot-loader/react-dom",

},

},

devtool: 'inline-source-map',

module: {

rules: [

{

test: /\.css$/,

use: [

{

loader: 'style-loader'

},

{

loader: 'css-loader',

options: {

modules: true,

localsConvention: 'camelCase',

sourceMap: true

}

}

]

}

]

},

plugins: [

new webpack.HotModuleReplacementPlugin()

],

devServer: {

host: 'localhost',

port: port,

historyApiFallback: true,

hot: true,

open: true

}

};

module.exports = config;

This is the same concept as with the previous file. Here we extracted out development only configurations.

Open webpack.prod.js and copy the following:

const commonPaths = require('./common-paths');

const webpack = require('webpack');

const MiniCssExtractPlugin = require('mini-css-extract-plugin');

const config = {

mode: 'production',

entry: {

app: [`${commonPaths.appEntry}/index.js`]

},

output: {

filename: 'static/[name].[hash].js'

},

devtool: 'source-map',

module: {

rules: [

{

test: /\.css$/,

use: [

{

// We configure 'MiniCssExtractPlugin'

loader: MiniCssExtractPlugin.loader,

},

{

loader: 'css-loader',

options: {

modules: true,

importLoaders: 1,

localsConvention: 'camelCase',

sourceMap: true

}

},

{

loader: 'postcss-loader'

}

]

}

]

},

plugins: [

new MiniCssExtractPlugin({

filename: 'styles/styles.[hash].css'

})

]

};

module.exports = config;

We extracted out production only configurations.

Now that we have the shared configurations and the ones specific for development and production in separate files, it is time to put everything together.

In terminal, if you are still in build-utils directory, go up one level to the root of the project, then delete the previous Webpack configurations and create a new Webpack configuration. Name it webpack.config.js.

In your terminal type the following:

cd ..

rm webpack.config.development.js webpack.config.production.js

touch webpack.config.js

Before configuring webpack.config.js, let’s open package.json and update the scripts section.

Modify the section as follows:

...

"scripts": {

"dev": "webpack-dev-server --env.env=dev",

"prebuild": "rimraf dist",

"build": "cross-env NODE\_ENV=production webpack -p --env.env=prod"

},

...

Since we removed the –config flag, Webpack will now be looking for the default configuration, which is webpack.config.js. Now we use the –env flag to pass an environment variable to Webpack, env=dev for development and env=prod for production.

Open webpack.config.js and copy the following:

const buildValidations = require('./build-utils/build-validations');

const commonConfig = require('./build-utils/webpack.common');

const webpackMerge = require('webpack-merge');

// We can include Webpack plugins, through addons, that do

// not need to run every time we are developing.

// We will see an example when we set up 'Bundle Analyzer'

const addons = (/\* string | string[] \*/ addonsArg) => {

// Normalize array of addons (flatten)

let addons = [...[addonsArg]]

.filter(Boolean); // If addons is undefined, filter it out

return addons.map(addonName =>

require(`./build-utils/addons/webpack.${addonName}.js`)

);

};

// 'env' will contain the environment variable from 'scripts'

// section in 'package.json'.

// console.log(env); => { env: 'dev' }

module.exports = env => {

// We use 'buildValidations' to check for the 'env' flag

if (!env) {

throw new Error(buildValidations.ERR\_NO\_ENV\_FLAG);

}

// Select which Webpack configuration to use; development

// or production

// console.log(env.env); => dev

const envConfig = require(`./build-utils/webpack.${env.env}.js`);

// 'webpack-merge' will combine our shared configurations, the

// environment specific configurations and any addons we are

// including

const mergedConfig = webpackMerge(

commonConfig,

envConfig,

...addons(env.addons)

);

// Then return the final configuration for Webpack

return mergedConfig;

};

Now, this might seem like a lot of setup, but in the long run, it will come in handy.

At this time, you can launch the application or build the production files, and everything will function as expected (sorry, no GIF this time).

yarn dev

yarn build

**Setting up Webpack Bundle Analyzer**

You don’t necessarily need Webpack Bundle Analyzer, but it does comes in handy when trying to optimize your builds.

Start by installing the dependency and creating the configuration file.

In your terminal type the following:

yarn add webpack-bundle-analyzer -D

touch build-utils/addons/webpack.bundleanalyzer.js

Open webpack.bundleanalyzer.js and copy the following:

const BundleAnalyzerPlugin = require('webpack-bundle-analyzer')

.BundleAnalyzerPlugin;

module.exports = {

plugins: [

new BundleAnalyzerPlugin({

analyzerMode: 'server'

})

]

};

We are just exporting the plugins section, which includes Bundle Analyzer, for Webpack. Then webpack-merge will combine it into the final Webpack configuration. Remember the *addons* in webpack.config.js? Well, this is where it comes into place.

For the final step, let’s open package.json and include the new scripts as follows:

"scripts": {

"dev": "webpack-dev-server --env.env=dev",

"dev:bundleanalyzer": "yarn dev --env.addons=bundleanalyzer",

"prebuild": "rimraf dist",

"build": "cross-env NODE\_ENV=production webpack -p --env.env=prod",

"build:bundleanalyzer": "yarn build --env.addons=bundleanalyzer"

},

* dev:bundleanalyzer — Calls the dev script and passes a new environment variable addons=bundleanalyzer
* build:bundleanalyzer — Calls the build script and passes a new environment variable addons=bundleanalyzer

Time to run the app with the bundle analyzer addon.

In your terminal type the following:

yarn dev:bundleanalyzer

Webpack Bundle Abalyzer

The application launches alongside Webpack Bundle Analyzer.