**Deploying React + Redux to Heroku**

In my ongoing quest to prove to myself that React is awesome, and not just a Facebook conspiracy, in this post we'll be deploying our CatBook application to Heroku!

You can checkout the code for this post [here](https://github.com/sophiedebenedetto/catbook-redux/tree/blog-post), and view a deployed version of the application [here](https://catbook-react.herokuapp.com/) (feel free to log in as the test user: sophie@email.com, with a password of password).

We'll cover the following:

* Configuring webpack for production.
* Configuring environment variables to hold our API host name for both development and production.
* Writing a production build task.
* Configuring our Express server for production.
* Pushing up to Heroku!

Let's get started.

**Setting Up The Production Environment with Webpack**

First things first, we'll configure our production environment with the help of webpack.

In development, webpack compiles our application from our src/ directory, and stores the bundled version in memory.

It also loads up some development-specific plugins, like the HotModuleReplacement plugin which enables the hot-reloading feature that we so enjoy in development.

// webpack.config.dev.js

import webpack from 'webpack';

import path from 'path';

export default {

debug: true,

devtool: 'cheap-module-eval-source-map',

noInfo: false,

entry: [

'eventsource-polyfill', // necessary for hot reloading with IE

'webpack-hot-middleware/client?reload=true', //note that it reloads the page if hot module reloading fails.

'./src/index'

],

target: 'web',

output: {

path: \_\_dirname + '/dist',

publicPath: '/',

filename: 'bundle.js'

},

devServer: {

contentBase: './src'

},

plugins: [

new webpack.HotModuleReplacementPlugin(),

new webpack.NoErrorsPlugin()

],

module: {

loaders: [

{test: /\.js$/, include: path.join(\_\_dirname, 'src'), loaders: ['babel']},

{test: /(\.css)$/, loaders: ['style', 'css']},

{test: /\.eot(\?v=\d+\.\d+\.\d+)?$/, loader: 'file'},

{test: /\.(woff|woff2)$/, loader: 'url?prefix=font/&limit=5000'},

{test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/, loader: 'url?limit=10000&mimetype=application/octet-stream'},

{test: /\.svg(\?v=\d+\.\d+\.\d+)?$/, loader: 'url?limit=10000&mimetype=image/svg+xml'}

]

}

};

In production, however, we need to compile and output a real, physical file, and serve *that* to the browser.

So, our webpack production configuration will have to be instructed on how and where to output that file.

We'll store our production build in a root level directory, public/. Make sure you create that empty directory at the root of your app!

Let's take a look at our production webpack configuration and break it down:

// webpack.config.prod.js

const path = require('path')

const webpack = require('webpack')

export default {

devtool: 'source-map',

entry: [

'./src/index'

],

output: {

path: path.join(\_\_dirname, 'public'),

filename: 'bundle.js',

publicPath: '/public/'

},

plugins: [

new webpack.optimize.DedupePlugin(),

new webpack.optimize.UglifyJsPlugin({

minimize: true,

compress: {

warnings: false

}

}),

new webpack.DefinePlugin({

'process.env': {

'NODE\_ENV': JSON.stringify('production')

}

})

],

module: {

loaders: [

{ test: /\.js?$/,

loader: 'babel',

exclude: /node\_modules/ },

{ test: /\.scss?$/,

loader: 'style!css!sass',

include: path.join(\_\_dirname, 'src', 'styles') },

{ test: /\.png$/,

loader: 'file' },

{ test: /\.(ttf|eot|svg|woff(2)?)(\?[a-z0-9]+)?$/,

loader: 'file'}

]

}

}

First, we set the dev tool to source-map:

devtool: 'source-map'

[Source map](https://developer.mozilla.org/en-US/docs/Tools/Debugger/How_to/Use_a_source_map) is a debugging tool that will map errors to the original, un-minified, source file that is throwing the error.

Next up, we define our app's entry point, which in our case is src/index.js.

entry: [

'./src/index'

]

In React, our "entry point" is the place where we actually use React DOM to insert our component tree and run our React app.

Then, we tell webpack where to store, or output, the compiled version of app to be served to the browser. In this case, we specify that is should compile our app into a file called bundle.js, and store that file in the public/ directory.

output: {

path: path.join(\_\_dirname, 'public'),

filename: 'bundle.js',

publicPath: '/public/'

}

Next we set up a few plugins for webpack to use:

* The [Dedupe Plugin](https://webpack.github.io/docs/list-of-plugins.html#dedupeplugin) identifies any duplicated files and de-duplicates them in output.
* The [UglifyJS Plugin](https://webpack.github.io/docs/list-of-plugins.html" \l "uglifyjsplugin) minimizes the output of JS chunks.
* The [Define Plugin](https://webpack.github.io/docs/list-of-plugins.html#defineplugin) allows you to create global constants at compile time--making constants that are defined server-side available client-side after compilation.

Lastly, we set up a series of [loaders](https://webpack.github.io/docs/loaders.html), telling webpack how to load different types of files/run different types of tasks.

**Configuring Environment Variables**

One of the first things I set out to do before deploying was un-hardcode the API URL from my API modules.

In a previous post, we built out a few classes--SessionApi, CatApi and HobbyApi--that use Fetch to make requests to our Rails 5 API. All three of these classes had the API URL host hardcoded in. For example, the CatApi's getAllCats function looked like this:

class CatsApi {

static requestHeaders() {

return {'AUTHORIZATION': `Bearer ${localStorage.jwt}`}

}

static getAllCats() {

const headers = this.requestHeaders();

const request = new Request('<http://localhost:5000/api/v1/cats>', {

method: 'GET',

headers: headers

});

return fetch(request).then(response => {

return response.json();

}).catch(error => {

return error;

});

}

...

You can see that our API host is hardcoded in as http://localhost:5000. This won't work in production, obviously. And, we don't want to have to manually switch back and forth between having our function hit http://localhost:5000 and having it hit https://catbook-api.herokuapp.com. That would be time consuming and impracticable and very very un-DRY of us.

So, we'll set an environment variable, API\_HOST, in both our development and production environments, with the help of webpack and the DefinePlugin.

In our webpack.config.dev.js, we'll set API\_HOST to the localhost port that our API is running on:

...

plugins: [

new webpack.optimize.DedupePlugin(),

new webpack.optimize.UglifyJsPlugin({

minimize: true,

compress: {

warnings: false

}

}),

new webpack.DefinePlugin({

'process.env': {

'NODE\_ENV': JSON.stringify('development'),

'API\_HOST': 'http://localhost:5000'

}

})

...

In webpack.config.prod.js we'll set our API\_HOST to the production URL of our API:

...

plugins: [

new webpack.optimize.DedupePlugin(),

new webpack.optimize.UglifyJsPlugin({

minimize: true,

compress: {

warnings: false

}

}),

new webpack.DefinePlugin({

'process.env': {

'NODE\_ENV': JSON.stringify('production'),

'API\_HOST': 'https://catbook-api.herokuapp.com'

}

})

...

Lastly, we'll update all of our API class functions to use API\_HOST when defining their Fetch requests. For example,

// src/api/catApi.js

class CatsApi {

static requestHeaders() {

return {'AUTHORIZATION': `Bearer ${localStorage.jwt}`}

}

static getAllCats() {

const headers = this.requestHeaders();

const request = new Request(`${API\_HOST}/api/v1/cats`, {

method: 'GET',

headers: headers

});

return fetch(request).then(response => {

return response.json();

}).catch(error => {

return error;

});

}

...

Now we're ready to define the build task that we'll run to compile our application for production.

**The Production Build Tasks**

In order to serve our app in production, we have to teach npm to do the following:

* Clean out the previous build from public/
* Build the production index.html file that serves as the location of our DOM and the location at which React attaches to the DOM
* Compile and output our app to the public/ directory.
* Run our production server

We'll define a series of scripts in our package.json to accomplish these tasks. We'll also build out the code to back the running of these scripts.

Let's take a look at the scripts we'll need to add to our package.json first.

// package.json

...

scripts: [

...

"clean-public": "npm run remove-public && mkdir public",

"remove-public": "node\_modules/.bin/rimraf ./public",

"build:html": "babel-node tools/buildHtml.js",

"prebuild": "npm-run-all clean-public lint build:html",

"build": "babel-node tools/build.js",

"postbuild": "babel-node tools/publicServer.js"

]

Let's break down these tasks:

* The clean-public task, which in turn relies on the remove-public task, is pretty self-explanatory. We simple delete and re-create the public/ directory to ensure we are getting rid of past builds.
* The build:html task, which we'll define in tools/buildHtmls.js, has a very important job. We told webpack to output our compiled app to the public/ directory, and serve it from there. But, our original index.html file is in the src/ directory. Oh no! So, we need to generate a copy of that src/index.html file, in the public/ directory, for our app to use in production.
* The prebuild task runs our previously defined clean-public, build:html and lint tasks.
* The build task will run the code in tools/build.js, which we will define soon to actually do the work of compiling and outputting our app to public/.
* The postbuild tasks will start up our production server, which we still need to define.

It's worth pointing out the we'll need the following dependencies defined in our package.json, in order to run our app in production mode:

...

"dependencies": {

"babel-polyfill": "6.8.0",

"babel-cli": "6.8.0",

"bootstrap": "^3.3.7",

"cheerio": "^0.20.0",

"colors": "1.1.2",

"compression": "^1.6.1",

"open": "0.0.5",

"react": "^15.3.1",

"react-redux": "^4.4.5",

"react-router": "^2.7.0",

"redux": "^3.5.2",

"redux-thunk": "^2.1.0",

"serve-favicon": "^2.3.0",

"babel-preset-es2015": "6.6.0",

"babel-preset-react": "6.5.0",

"express": "4.13.4"

}

...

You can check out the full package.json file [here](https://github.com/SophieDeBenedetto/catbook-redux/blob/blog-post/package.json).

Now, we'll go ahead and actually build out the code required to execute these tasks.

**The build:html Task**

As we stated earlier, our app's index.html file lives in the src/ directory. But, we are in the process of compiling our app and outputting it to the public/ directory, from where it will be served to the browser. So, we need a version of index.html in the public/ directory.

Our build:html script will help us out with this. We'll use the Cheerio package to accomplish this task. [Cheerio](https://github.com/cheeriojs/cheerio) can parse HTML, returning a jQuery object to you so that you can parse the DOM.

The code for your buildHtml script will go in tools/buildHtml.js. The tools/ directory should be at the root level of our application.

Let's take a look:

import fs from 'fs';

import cheerio from 'cheerio';

import colors from 'colors';

/\*eslint-disable no-console \*/

fs.readFile('src/index.html', 'utf8', (err, markup) => {

if (err) {

return console.log(err);

}

const $ = cheerio.load(markup);

$('head').prepend('');

fs.writeFile('public/index.html', $.html(), 'utf8', function (err) {

if (err) {

return console.log(err);

}

console.log('index.html written to /public'.green);

});

});

We're using the fs, or File System, module to read the contents of the src/index.html file. We're feeding those contents to a call to cheerio, which will load the HTML and allow us to use jQuery to then traverse the newly created DOM. Once we've loaded the HTML with cheerio, we'll write it to public/index.html using fs.

Now that we have our code in place, let's take a quick look back at the build:html script from our package.json

"build:html": "babel-node tools/buildHtml.js"

Our script is actually pretty simple--we're using babel-node to execute the code that we just wrote in tools/buildHtml.js.

Let's move on to the next script, the build task.

**The build Task**

The build task is pretty critical. This is where we'll actually write the code to compile our app and output it to public/.

We'll define our build code in tools/build.js

/\*eslint-disable no-console \*/

import webpack from 'webpack';

import webpackConfig from '../webpack.config.prod';

import colors from 'colors';

process.env.NODE\_ENV = 'production';

console.log('Generating minified bundle for production via Webpack...'.blue);

webpack(webpackConfig).run((err, stats) => {

if (err) { // so a fatal error occurred. Stop here.

console.log(err.bold.red);

return 1;

}

const jsonStats = stats.toJson();

if (jsonStats.hasErrors) {

return jsonStats.errors.map(error => console.log(error.red));

}

if (jsonStats.hasWarnings) {

console.log('Webpack generated the following warnings: '.bold.yellow);

jsonStats.warnings.map(warning => console.log(warning.yellow));

}

console.log(`Webpack stats: ${stats}`);

console.log('Your app has been compiled in production mode and written to /public.'.green);

return 0;

});

The magic really happens here:

import wepbackConfig from '../webpack.config.prod';

...

webpack(webpackConfig).run..

Here, we're creating our compilier with the webpack(webpackConfig) portion, passing it our production webpack configuration from the webpack.config.prod.js file.

Then, we call .run() on the compiler instance that webpack(webpackConfig) returns, thus creating our application bundle and outputting it to the public/ directory, as per the instructions in our wepback config.

**The postbuild Task and our Production Server**

Our last script is the postBuild script. This is the script that runs out production server. We'll define out production server in tools/publicServer.js. It mainly differentiates from our dev server in that is routes web requests to the app being served from the public/ directory.

Let's take a look:

import express from 'express';

import path from 'path';

import open from 'open';

import compression from 'compression';

import favicon from 'serve-favicon';

/\*eslint-disable no-console \*/

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

const app = express();

app.use(compression());

app.use(express.static('public'));

app.use(favicon(path.join(\_\_dirname,'assets','public','favicon.ico')));

app.get('\*', function(req, res) {

res.sendFile(path.join(\_\_dirname, '../public/index.html'));

});

app.listen(port, function(err) {

if (err) {

console.log(err);

} else {

open(`[http://localhost:${port}`](http://localhost:$%7Bport%7D%60));

}

});

There's just one last piece of configuration we need to do...

**The Procfile and Why We Need It**

We need to tell Heroku to handle the web requests that our app receives by running the publicServer.js file. Without this last bit of instruction, Heroku will try to run our start script, defined in the package.json. Our start script is strictly for our dev environment.

So, we'll create a Procfile in the root of our app:

web: babel-node tools/publicServer.js

**Build and push to Heroku!**

Now we're ready to deploy! We just need to create our Heroku app, build for production, and push.

$ heroku create my-amazing-react-app

$ npm run build

$ git add .

$ git commit -m "built production"

$ git push heroku master

And that's it!