Deployment Introduction

Edit this page on Github (https://github.com/fullstackreact/30-days-of-react/blob/master/day-27/post.md)

Today, we'll explore the different pieces involved in deploying our application so the world can use our application out in the wild.

With our app all tested up through this point, it's time to get it up and live for the world to see. The rest of this course will be dedicated to deploying our application into production.

Production deployment

When talking about deployment, we have a lot of different options:

- Hosting
- Deployment environment configuration
- Continuous Integration (CI, for short)
- Cost cycles, network bandwidth cost
- Bundle size
- and more

We'll look at the different hosting options we have for deploying our react app tomorrow and look at a few different methods we have for deploying our application up. Today we're going to focus on getting our app ready for deployment.

Ejection (from `create-react-app`)

First things first... we're going to need to handle some customization in our web application, so we'll need to run the `npm run eject` command in the root of our directory. This is a permanent action, which just means we'll be responsible for handling customizations to our app structure for now on (without the help of our handy `create-react-app`).

> This is where I *always* say make a backup copy of your application. We cannot go back from `ejecting`, but we can revert to old code.

We can *eject* from the `create-react-app` structure by running the eject command provided by the generator:

```
npm run eject
```

After *ejecting* from the `create-react-app` structure, we'll see we get a lot of new files in our application root in the `config/` and `scripts/` directories. The `npm run eject` command created all of the files it uses internally and wrote them all out for us in our application.

The key method of the `create-react-app` generator is called webpack (https://webpack.github.io), which is a module bundler/builder.

Webpack basics

Webpack is a module bundler with a ginormous community of users, tons of plugins, is under active development, has a clever plugin system, is incredibly fast, supports hot-code reloading, and much much more.

Although we didn't really call it out before, we've been using webpack this entire time (under the guise of `npm start`). Without webpack, we wouldn't have have been able to just write `import` and expect our code to load. It works like that because webpack "sees" the `import` keyword and knows we need to have the code at the path accessible when the app is running.

Webpack takes care of hot-reloading for us, nearly automatically, can load and pack many types of files into bundles, and it can split code in a logical manner so as to support lazy-loading and shrink the initial download size for the user.

This is meaningful for us as our apps grow larger and more complex, it's important to know how to manipulate our build tools.

For example, when we want to deploy to different environments... which we'll get to shortly. First, a tiny introduction to webpack, what it is and how it works.

What it does with `bundle.js`

Looking into the generated files when we ran `npm start` before we ejected the app, we can see that it serves the browser two or more files. The first is the `index.html` and the `bundle.js`. The webpack server takes care of injecting the `bundle.js` into the `index.html`, even if we don't load our app in the `index.html` file.

The `bundle.js` file is a giant file that contains *all* the JavaScript code our app needs to run, including dependencies and our own files alike. Webpack has it's own method of packing files together, so it'll look kinda funny when looking at the raw source.

Webpack has performed some transformation on all the included JavaScript. Notably, it used Babel to transpile our ES6 code to an ES5-compatible format.

If you look at the comment header for `app.js`, it has a number, `254`:

```
/* 254 */
/*!*******************!*\
  !*** ./src/app.js ***!
  \*******************/
```

The module itself is encapsulated inside of a function that looks like this:

```
function(module, exports, __webpack_require__) {
  // The chaotic `app.js` code here
}
```

Each module of our web app is encapsulated inside of a function with this signature. Webpack has given each of our app's modules this function container as well as a module ID (in the case of `app.js`, 254).

But "module" here is not limited to ES6 modules.

Remember how we "imported" the `makeRoutes()` function in `app.js`, like this:

```
import makeRoutes from './routes'
```

Here's what the variable declaration of `makeRoutes` looks like inside the chaos of the `app.js` Webpack module:

```
var _logo = __webpack_require__(/*! ./src/routes.js */ 255);
```

This looks quite strange, mostly due to the in-line comment that Webpack provides for debugging purposes. Removing that comment:

```
var _logo = __webpack_require__(255);
```

Instead of an `import` statement, we have plain old ES5 code.

Now, search for `./src/routes.js` in this file.

```
/* 255 */
/*!**********************!*\
  !*** ./src/routes.js ***!
  \**********************/
```

Note that its module ID is `255`, the same integer passed to `__webpack_require__` above.

Webpack treats *everything* as a module, including image assets like `logo.svg`. We can get an idea of what's going on by picking out a path in the mess of the `logo.svg` module. Your path might be different, but it will look like this:

```
static/media/logo.5d5d9eef.svg
```

If you open a new browser tab and plug in this address (your address will be different... matching the name of the file webpack generated for you):

```
http://localhost:3000/static/media/logo.5d5d9eef.svg
```

You should get the React logo:

So Webpack created a Webpack module for `logo.svg`, one that refers to the path to the SVG on the Webpack development server. Because of this modular paradigm, it was able to intelligently compile a statement like this:

```
import makeRoutes from './routes'
```

Into this ES5 statement:

```
var _makeRoutes = __webpack_require__(255);
```

What about our CSS assets? Yep, *everything* is a module in Webpack. Search for the string `./src/app.css`:

Webpack's `index.html` didn't include any references to CSS. That's because Webpack is including our CSS here via `bundle.js`. When our app loads, this cryptic Webpack module function dumps the contents of `app.css` into `style` tags on the page.

So we know *what* is happening: Webpack has rolled up every conceivable "module" for our app into `bundle.js`. You might be asking: Why?

The first motivation is universal to JavaScript bundlers. Webpack has converted all our ES6 modules into its own bespoke ES5-compatible module syntax. As we briefly touched on, it's wrapped all of our JavaScript modules in special functions. It provides a module ID system to enable one module to reference another.

Webpack, like other bundlers, consolidated all our JavaScript modules into a single file. It *could* keep JavaScript modules in separate files, but this requires some more configuration than `create-react-app` provides out of the box.

Webpack takes this module paradigm further than other bundlers, however. As we saw, it applies the same modular treatment to image assets, CSS, and npm packages (like React and ReactDOM). This modular paradigm unleashes a lot of power. We touch on aspects of that power throughout the rest of this chapter.

Complex, right?

It's okay if you don't understand that out of the box. Building and maintaining webpack is a complex project with lots of moving parts and it often takes even the most experienced developers a while to "get."

We'll walk through the different parts of our webpack configuration that we'll be working with. If it feels overwhelming, just stick with us on the basics here and the rest will follow.

With our newfound knowledge of the inner workings of Webpack, let's turn our attention back to our app. We'll make some modifications to our webpack build tool to support multiple environment configurations.

Environment configuration

When we're ready to deploy a new application, we have to think about a few things that we wouldn't have to focus on when developing our application.

For instance, let's say we are requesting data from an API server... when developing this application, it's likely that we are going to be running a development instance of the API server on our local machine (which would be accessible through `localhost`).

When we deploy our application, we'll want to be requesting data from an off-site host, most likely not in the same location from where the code is being sent, so `localhost` just won't do.

One way we can handle our configuration management is by using `.env` files. These `.env` files will contain different variables for our different environments, yet still provide a way for us to handle configuration in a sane way.

Usually, we'll keep one `.env` file in the root to contain a *global* config that can be overridden by configuration files on a per-environment basis.

Let's install an `npm` package to help us with this configuration setup called `dotenv`:

```
npm install --save-dev dotenv
```

The dotenv (https://github.com/motdotla/dotenv) library helps us load environment variables into the `ENV` of our app in our environments.

> It's usually a good idea to add `.env` to our `.gitignore` file, so we don't check in these settings.
>
> Conventionally, it's a good idea to create an example version of the `.env` file and check that into the repository. For instance, for our application we can create a copy of the `.env` file called `.env.example` with the required variables.
>
> Later, another developer (or us, months from now) can use the `.env.example` file as a template for what the `.env` file should look like.

These `.env` files can include variables as though they are unix-style variables. Let's create our global one with the `APP_NAME` set to 30days:

```
touch .env
echo "APP_NAME=30days" > .env
```

Let's navigate to the exploded `config/` directory where we'll see all of our build tool written out for us. We won't look at all of these files, but to get an understanding of *what* are doing, we'll start looking in `config/webpack.config.dev.js`.

This file shows all of the webpack configuration used to build our app. It includes loaders, plugins, entry points, etc. For our current task, the line to look for is in the `plugins` list where we define the `DefinePlugin()`:

```
module.exports = {
  // ...
  plugins: [
    // ...
    // Makes some environment variables available to
    // the JS code, for example:
    // if (process.env.NODE_ENV === 'development') {
    //  ...
    // }. See `env.js`
    new webpack.DefinePlugin(env),
    // ...
  ]
}
```

The `webpack.DefinePlugin` plugin takes an object with `keys` and `values` and finds all the places in our code where we use the key and it replaces it with the value.

For instance, if the `env` object there looks like:

```
{
  '__NODE_ENV__': 'development'
}
```

We can use the variable `__NODE_ENV__` in our source and it will be replaced with 'development', i.e.:

```
class SomeComponent extends React.Component {
  render() {
    return (
      <div>Hello from {__NODE_ENV__}</div>
    )
  }
}
```

The result of the `render()` function would say "Hello from development".

To add our own variables to our app, we're going to use this `env` object and add our own definitions to it. Scrolling back up to the top of the file, we'll see that it's currently created and exported from the `config/env.js` file.

Looking at the `config/env.js` file, we can see that it takes all the variables in our environment and adds the `NODE_ENV` to the environment as well as any variables prefixed by `REACT_APP_`.

```
// ...
module.exports = Object
  .keys(process.env)
  .filter(key => REACT_APP.test(key))
  .reduce((env, key) => {
    env['process.env.' + key] = JSON.stringify(process.env[key]);
    return env;
  }, {
    'process.env.NODE_ENV': NODE_ENV
  });
```

We can skip all the complex part of that operation as we'll only need to modify the second argument to the reduce function, in other words, we'll update the object:

```
{
  'process.env.NODE_ENV': NODE_ENV
}
```

> This object is the *initial* object of the reduce function. The `reduce` function merges all of the variables prefixed by `REACT_APP_` *into* this object, so we'll always have the `process.env.NODE_ENV` replaced in our source.

Essentially what we'll do is:

1. Load our default `.env` file
2. Load any environment `.env` file
3. Merge these two variables together as well as any default variables (such as the `NODE_ENV`)
4. We'll create a new object with all of our environment variables and sanitize each value.
5. Update the initial object for the existing environment creator.

Let's get busy. In order to load the `.env` file, we'll need to import the `dotenv` package. We'll also import the `path` library from the standard node library and set up a few variables for paths.

Let's update the `config/env.js` file

```
var REACT_APP = /^REACT_APP_/i;
var NODE_ENV = process.env.NODE_ENV || 'development';

const path = require('path'),
      resolve = path.resolve,
      join = path.join;

const currentDir = resolve(__dirname);
const rootDir = join(currentDir, '..');

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

To load the global environment, we'll use the `config()` function exposed by the `dotenv` library and pass it the path of the `.env` file loaded in the root directory. We'll also use the same function to look for a file in the `config/` directory with the name of `NODE_ENV.config.env`. Additionally, we don't want either one of these methods to error out, so we'll add the additional option of `silent: true` so that if the file is not found, no exception will be thrown.

```
// 1. Step one (loading the default .env file)
const globalDotEnv = dotenv.config({
  path: join(rootDir, '.env'),
  silent: true
});
// 2. Load the environment config
const envDotEnv = dotenv.config({
  path: join(currentDir, NODE_ENV + `.config.env`),
  silent: true
});
```

Next, let's concatenate all these variables together as well as include our `NODE_ENV` option in this object. The `Object.assign()` method creates a *new* object and merges each object from right to left. This way, the environment config variable

```
const allVars = Object.assign({}, {
  'NODE_ENV': NODE_ENV
}, globalDotEnv, envDotEnv);
```

With our current setup, the `allVars` variable will look like:

```
{
  'NODE_ENV': 'development',
  'APP_NAME': '30days'
}
```

Finally, let's create an object that puts these variables on `process.env` and ensures they are valid strings (using `JSON.stringify`).

```
const initialVariableObject =
  Object.keys(allVars)
  .reduce((memo, key) => {
    memo['process.env.' + key.toUpperCase()] =
      JSON.stringify(allVars[key]);

    return memo;
  }, {});
```

With our current setup (with the `.env` file in the root), this creates the `initialVariableObject` to be the following object:

```
{
  'process.env.NODE_ENV': '"development"',
  'process.env.APP_NAME': '"30days"'
}
```

Now we can use this `initialVariableObject` as the second argument for the original `module.exports` like. Let's update it to use this object:

```
module.exports = Object
  .keys(process.env)
  .filter(key => REACT_APP.test(key))
  .reduce((env, key) => {
    env['process.env.' + key] = JSON.stringify(process.env[key]);
    return env;
  }, initialVariableObject);
```

Now, anywhere in our code we can use the variables we set in our `.env` files.

Since we are making a request to an off-site server in our app, let's use our new configuration options to update this host.

Let's say by default we want the TIME_SERVER to be set to `http://localhost:3001`, so that if we don't set the `TIME_SERVER` in an environment configuration, it will default to localhost. We can do this by adding the `TIME_SERVER` variable to the global `.env` file.

Let's update the `.env` file so that it includes this time server:

```
APP_NAME=30days
TIME_SERVER='http://localhost:3001'
```

Now, we've been developing in "development" with the server hosted on heroku. We can set our `config/development.config.env` file to set the `TIME_SERVER` variable, which will override the global one:

```
TIME_SERVER='https://fullstacktime.herokuapp.com'
```

Now, when we run `npm start`, any occurrences of `process.env.TIME_SERVER` will be replaced by which ever value takes precedence.

Let's update our `src/redux/modules/currentTime.js` module to use the new server, rather than the hardcoded one we used previously.

```
// ...
export const reducer = (state = initialState, action) => {
  // ...
}

const host = process.env.TIME_SERVER;
export const actions = {
  updateTime: ({timezone = 'pst', str='now'}) => ({
    type: types.FETCH_NEW_TIME,
    meta: {
      type: 'api',
      url: host + '/' + timezone + '/' + str + '.json',

      method: 'GET'
    }
  })
}
```

Now, for our production deployment, we'll use the heroku app, so let's create a copy of the `development.config.env` file as `production.config.env` in the `config/` directory:

```
cp config/development.config.env config/production.config.env
```

Custom middleware per-configuration environment

We used our custom logging redux middleware in our application. This is fantastic for working on our site in development, but we don't really want it to be active while in a production environment.

Let's update our middleware configuration to only use the logging middleware when we are in development, rather than in all environments. In our project's `src/redux/configureStore.js` file, we loaded our middleware by a simple array:

```
let middleware = [
  loggingMiddleware,
  apiMiddleware
];
const store = createStore(reducer, applyMiddleware(...middleware));
```

Now that we have the `process.env.NODE_ENV` available to us in our files, we can update the `middleware` array depending upon the environment we're running in. Let's update it to only add the logging if we are in the development environment:

```
let middleware = [apiMiddleware];
if ("development" === process.env.NODE_ENV) {
  middleware.unshift(loggingMiddleware);
}

const store = createStore(reducer, applyMiddleware(...middleware));
```

Now when we run our application in development, we'll have the `loggingMiddleware` set, while in any other environment we've disabled it.

Today was a long one, but tomorrow is an exciting day as we'll get the app up and running on a remote server.

Great work today and see you tomorrow!

LEARN REACT THE RIGHT WAY

The up-to-date, in-depth, complete guide to React and friends.

DOWNLOAD THE FIRST CHAPTER (/DOWNLOAD/FULLSTACK-REACT/)