Creating a Fullstack App

Hunter Jorgensen

Next Steps: Getting Frontend and Backend Together

There are different ways to build a React app to work with Node, but we will be doing the following:

1. Run only Node and serve React on the server
    - Will need to “build” React and serve it from base (‘/’)
    - (Optional) Can make it a monorepo with something like “lerna”
    - Let’s learn this now!

Single Server App

Before we build out an app that can run on a single server, we need to investigate briefly some of the technology we've been using. This can be complicated, so if you're not following ask questions!

Also, since we've moving everything into a single NPM package (i.e., our frontend code is extremely close to our backend code), I want to be extremely explicit about the difference between frontend and backend:

Frontend -> /src

Backend -> server.js and /routes

react-scripts

When we deploy using Heroku, Heroku runs 2 scripts for us when we deploy:

npm run build

npm run start (this is identical to ‘npm start’)

When we run *npm run X* in an NPM directory (that is, in a directory with a package.json) it will try to run the script with the same name in package.json

react-scripts

npm run start (npm start) - this creates something known as WebpackDevServer (which is just a simple server on your computer) and restarts everytime you make a code change

npm run build - this uses some tools called Babel and Webpack to make your React code readable to a browser. We can send this to the browser when we start our Node server so that it behaves similar to the ***npm start*** script above (except without things like the hot swapping). Let's see the output!

For those feeling brave, the source code to these scripts is here.

New code!

Let’s start building a new app.

- (create new React app) npx create-react-app {app-name}
- (go to root of React directory) cd {app-name}
- (create new Express server) touch server.js
- (install dependencies) npm i -s express concurrently axios react-router-dom
- Copy this code into your server.js

What is this server.js doing?

(concurrently is a new one, but it will allow us to run several scripts at once)

Update package.json

Open your package.json and replace the scripts and add the proxy attribute like here.

Now when you code locally, you’ll want to use: ***npm run devstart***

Lots of changes here:

build and ***start*** are called in that order when your code is deployed to Heroku.

proxy lets us make local API calls a little more easily (no need to write localhost)

concurrently is letting run the nodemon and the react-scripts simultaneously

Deploy to Heroku!

That’s it! When you send this code to Heroku, your Node app will be supplying your built React app.

Unique Identifier

We have some great libraries to generate ID’s for us to help us minimize collision, but we’ll be using UUID.

Demo

(Remember to add ‘Content-Type: application/json’ to your POST/PUT requests!)

What are the benefits of using a random identifier?

Note: notice how in POST we only return the ID? This is considered standard REST behavior to not return the entire object, but a way to reference the item, but you are welcome to return what you think is best for your app.

PUT/DELETE

A PUT request is one that updates an existing resource. You typically want to provide the new object in its entirety, including an previously set values.

A DELETE request is one that removes an endpoint. You typically only need an ID to make this request.

Demo

Note that in standard REST, a PUT and DELETE do not contain a response body, only a success or failure (or only success in the case of delete.) You are welcome to respond with whatever you think is helpful, but should be ready to defend your decision.

Async/Await

Async/Await is an attempt to simplify the Promise API. Instead of writing callback functions, users can try to write use more common syntax. In order to work with async/await, you declare the function you’re calling as async, and declare any logic that would be asynchronous inside of it with an await.

await can ONLY be called within **async** functions. **awaits** essentially act as **then** blocks in a Promise chain. I’m not expecting you to use this, but are welcome to try it. If I want to check for errors, then I would put this into a try/catch block.

Making A Call From the Frontend

Now that we have set up our backend and can make API requests to it, let’s look at how the frontend can query the backend. While there is a built in functionality called “fetch”, there is a library called Axios that is easier to use.

Demo

Any request from Axios returns a promise. Do you remember those?

Async/Await

- **async**: applied to the outer function; if it returns anything, it will return a promise; this indicates to JavaScript that the logic within the function will happen simultaneously
- **await**: is used within an async function; it takes a promise and allows you to handle it as a simple variable.

Note: Redux/API

For this assignment, you are not required to use redux or context. However, if you elect to choose these libraries, they work well with a library called “thunk” to handle asynchronous logic.

Lab: Adding Delete and Edit

As before, add functionality to our Redux/React app so that it can also delete and update (put) food items.

Things to consider:

- Will you need to update the state?
- What kind of actions will you need?
- What is the behavior on the website when a use clicks add/delete?
- Are you able to refactor the create container so that duplicate code is minimized between it and the update container/component?