Lesson Sunday

React (Part-Time React track) (/react-part-time-react-track) / React with NoSQL (Part 1) (/react-part-time-react-track/react-with-nosql-part-1) / Adding Firebase to React

Text

We're ready to connect our help queue application to Firebase. Either clone down your existing copy of the help queue application or use this repo:

Example GitHub Repo for Help Queue (https://github.com/epicodus-lessons/week-4-updated-react-nosql-starter-project)

Step 1: Install Firebase

First, we'll need to install Firebase in our project:

```
npm install firebase@7.8.0
```

Note that it's important to use the version pinned in this lesson. Because Firebase changes frequently, using a different version may mean different steps to setting up your application's configuration.

Step 2: Add `.env` File

Next, we need to add the key-value pairs from the `firebaseConfig` object in our React application. However, we want to conceal this information using a `.env` file. Otherwise, our

Firebase database configuration will be exposed to everyone, including potentially malicious users.

Fortunately, `create-react-app` automatically comes with `dotenv`, the npm package we used in Intermediate JavaScript to store sensitive API keys in an `.env` file.

First, we need to add `.env` to our `.gitignore` file. Note that `create-react-app` automatically adds a number of these kinds of files to our `.gitignore` including `.env.local`, `.env.development.local`, and so on. `create-react-app` does this because in larger projects, it can be helpful to have multiple files for environment variables. They can be split up for testing, production, and development. For more information on different environment variable file types in `create-react-app`, see Adding Custom Environment Variables (https://create-react-app.dev/docs/adding-custom-environment-variables/).

Since our application is small, we will just create a basic `.env` file. Add `.env` to the `.gitignore` and then commit and push the updated `.gitignore` file to Github. Don't create the `.env` file just yet. As you may recall from Intermediate JavaScript, if we push an updated `.gitignore` file at the same time as we push the file that should be ignored, Github won't know it's supposed to ignore it - meaning it will be added to the repo.

Next, create a `.env` file in the root directory of the project. Environment variables can only be set up for strings, not objects. For that reason, each key-value pair in the `firebaseConfig` object needs to be broken down into its own constant like this:

.env

```
REACT_APP_FIREBASE_API_KEY = "YOUR-UNIQUE-CREDENTIALS"
REACT_APP_FIREBASE_AUTH_DOMAIN = "YOUR-PROJECT-NAME.firebase
app.com"
REACT_APP_FIREBASE_DATABASE_URL = "https://YOUR-PROJECT-NAM
E.firebaseio.com"
REACT_APP_FIREBASE_PROJECT_ID = "YOUR-PROJECT-FIREBASE-PROJE
CT-ID"
REACT_APP_FIREBASE_STORAGE_BUCKET = "YOUR-PROJECT-NAME.appsp
ot.com"
REACT_APP_FIREBASE_MESSAGING_SENDER_ID = "YOUR-PROJECT-SENDE
R-ID"
REACT_APP_FIREBASE_APP_ID = "YOUR-PROJECT-APP-ID"
```

Replace the placeholders in the values above with the value of each key from your own Firebase application. (If you've misplaced this info, click on the gear in the upper left of the page, click `project settings`, and scroll to the bottom of the page.)

Note: It is very important that every environment variable in your application is prefaced by `REACT_APP`. Otherwise, the environment variable **won't work**. This is a safeguard put in place by `create-react-app` to ensure that sensitive environment variables aren't accidentally exposed in our applications.

Step 3: Create Configuration File with Firebase Reference

Next, we'll create a file in our `src` directory called `firebase.js`. This is where we'll initialize Firebase in our application and create a database reference.

Add the following code to the file:

src/firebase.js

```
import * as firebase from 'firebase';
import 'firebase/firestore';

const firebaseConfig = {
  apiKey: process.env.REACT_APP_FIREBASE_API_KEY,
  authDomain: process.env.REACT_APP_FIREBASE_AUTH_DOMAIN,
  databaseURL: process.env.REACT_APP_FIREBASE_DATABASE_URL,
  projectId: process.env.REACT_APP_FIREBASE_PROJECT_ID,
  storageBucket: process.env.REACT_APP_FIREBASE_STORAGE_BUCK
ET,
  messagingSenderId: process.env.REACT_APP_FIREBASE_SENDER_I
D,
  appId: process.env.REACT_APP_FIREBASE_APP_ID
}

firebase.initializeApp(firebaseConfig);
firebase.firestore();

export default firebase;
```

We start by importing `firebase` and `firebase/firestore`. Next, we have the same configuration object that we copied from the Firebase UI. There are a few small tweaks: first, we save the

configuration in a `const` called `firebaseConfig`. (No more `var` for us!) Next, all the values are environment variables. We aren't exposing our sensitive data.

Next, we call the `initializeApp` method. This creates and initializes an instance of our Firebase application. We pass in our `firebaseConfig` as an argument. That way, Firebase knows exactly which Firebase project should be accessed.

Then, because we are using Firestore as our database, we call `firebase.firestore()`. Finally, we `export default firebase` to make our configuration available where it's needed.

At this point, we've successfully added Firebase and Firestore to our application. However, it's considerably more involved to actually start communicating with our database. We could do this without external libraries, but this is more challenging. Fortunately, there are several bindings (just as we used the React Redux bindings in the last course section) that we can use to make it easier to integrate React with Firebase and Firestore. We'll add those next.

Step 4: Add and Configure Bindings

We will add two external libraries for bindings. Make sure you use the versions indicated below:

```
npm install react-redux-firebase@3.1.1 redux-firestore@0.12.
0
```

React Redux Firebase offers a higher-order component (HOC) so our React application has access to Firebase. It works somewhat similarly to React Redux, which also offers a HOC with the `connect` function. We'll go over the similarities more as we add React Redux Firebase bindings to our application. If we were just using Firebase as our database, we wouldn't need to add any other bindings.

Because we are using Firestore as our database, though, we are also adding Redux Firestore to provide extra functionality. This library provides a `firestoreReducer` (while the Firebase library provides a `firebaseReducer`). All of our communication with Firestore will go through our `firestoreReducer` - this means we

don't need to create additional reducers, use async actions, or use middleware like Redux-Thunk. (We haven't learned about async actions and middleware yet, but we will in future lessons.)

Next, we need to update our entry point file.

src/index.js

```
...
import { ReactReduxFirebaseProvider } from 'react-redux-fire
base';
import { createFirestoreInstance } from 'redux-firestore';
import firebase from "./firebase";

const store = createStore(rootReducer);

const rrfProps = {
  firebase,
  config: {
        userProfile: "users"
    },
  dispatch: store.dispatch,
  createFirestoreInstance
}

ReactDOM.render(
  <Provider store={store}>
    <ReactReduxFirebaseProvider {...rrfProps}>
      <App />
    </ReactReduxFirebaseProvider>
  </Provider>,
  document.getElementById('root')
)
...
```

We have three new import statements:

```
import { ReactReduxFirebaseProvider } from 'react-redux-fire
base';
import { createFirestoreInstance } from 'redux-firestore';
import firebase from "./firebase";
```

`ReactReduxFirebaseProvider` is a component much like the `Provider` component that Redux provides. We can wrap our root component in the `ReactReduxFirebaseProvider` component to

make additional functionality available throughout our application, including the `withFirestore()` function, which allows us to make Firestore available via a component's props.

We also need to import `createFirestoreInstance`, which does exactly what it sounds like.

Finally, we import `firebase` from the `firebase.js` config file we created earlier in this lesson. Remember these lines?

src/firebase.js

```
firebase.initializeApp(firebaseConfig);
firebase.firestore();

export default firebase;
```

We initialized Firebase with our unique Firebase config, set it up to use Firestore, and then exported it to make it available elsewhere in our application - specifically, our entry point file.

This gets passed into `rrfProps`:

src/index.js

```
const rrfProps = {
  firebase,
  config: {
        userProfile: "users"
    },
  dispatch: store.dispatch,
  createFirestoreInstance
}
```

The React Redux Firebase bindings require these props to be passed into the `<ReactReduxFirebaseProvider>` component. We can add different key-value pairs to `config`. `userProfile: "users"` simply states that any data on users will be stored in a collection called `"users"`.

Our `App` component is now wrapped in two different `Provider` components:

src/index.js

```
...
<Provider store={store}>
  <ReactReduxFirebaseProvider {...rrfProps}>
    <App />
  </ReactReduxFirebaseProvider>
</Provider>
...
```

Both of these provide different **context** to the rest of our application. Redux's `Provider` component provides our Redux store's context while `ReactReduxFirebaseProvider` provides Firebase and Firestore context. With both, we'll need to use higher order components in order to actually provide functionality from that context to components where it's needed.

Step 5: Add Firestore Reducer

We have just one more configuration step. We'll add a `firestoreReducer` to the `index.js` file where our root reducer lives. This will take care of our communication with Firestore:

src/reducers/index.js

```
...

import { firestoreReducer } from 'redux-firestore';

const rootReducer = combineReducers({
  formVisibleOnPage: formVisibleReducer,
  mainTicketList: ticketListReducer,
  // new line of code below
  firestore: firestoreReducer
});

...
```

We import the `firestoreReducer` from Redux Firestore and then we specify the `firestoreReducer` will handle the `firestore` state slice. Remember that this root reducer gets imported into `src/index.js` and is used when we first create and initialize our store. This means that the store we pass down into our application via provider components will now be able to use Firestore.

At this point, we've completed all necessary setup and configuration and we are ready to start communicating with our database. Over the next several lessons, we'll add full CRUD functionality to our help queue application - this time with Firestore providing the data!

Lesson 6 of 15
Last updated more than 3 months ago.

Epicodus (http://www.epicodus.com)