eCommerce Website Project

After creating your React app (npm create-react-app app-name), you can delete:

* ‘serviceworker’ code from index.js
* Within App.js, remove all content, including the ‘className’ within the div
* Delete the logo import
* Delete everything within App.css

Add SCSS to the project:

* run **npm add node-sass**

SASS/SCSS allows ‘nested’ styling. Using the ‘&; anything sitting between the .main-item element will receive all styling contained within:

.main-item{

width: 80%,

height: 80%

//Nested items

&:first-child {

margin-right: 7px

}

&:last-child {

margin-left: 7px

}

.content {

display: flex;

flex-direction: column;

align-items: center;

padding: 20px 80px;

}

/\* Any menu-item having a property of ‘large’, will attain this larger heightl \*/

&.large {

height: 380px;

}

}

Routing In React

Initially, **single page applications** didn’t have the ‘back button’ functionality. because the navigation within the application’s url didn’t change. This has now been fixed with the browser providing a ‘**History API**’. Libraries like React Router, and other single page application routing libraries, use this API to hijack and mimic urls.

The library we’re going to use is ‘**react-router-dom**’. This is the most popular and sought out library for routing in React.

To install react-router-dom:

* run **npm install react-router-dom**
* Within index.js, add, **import { BrowserRouter } from ‘react-router-dom’**
* Wrap the **<BrowserRouter>** component around the **<App />** component. What this does, is, give our application all of the function of routing:

**<BrowserRouter>**

**<App />**

**</ BrowserRouter>,**

**document.getElementById(“root”)**

Any component that gets rendered by our ‘Route’, gets passed three arguments:

* **history:**
* f push:
* **location:** tells us where we are in our application (ex: ‘pathname’ param)
* **match**:
* IsExact: is only true if the entire url matches the pattern
* params: an object of url parameters **(ex: ‘/:topicId’**. This can dynamically change)
* path: the pattern the router is looking to match
* url: the url of the page

The other ways to navigate is with the ‘**Link**’ component, and the ‘**history**’ and ‘**location**’ prop:

* i**mport { Route, Link } from react-router-dom**
* Syntax:
* **<Link to=”/topics”>TOPICS</ Link>**/\* With ‘Link’, we’re borrowing the url to tell the app what to re-render \*/
* For ‘history’ prop using a button:
  + **<button onClick={ () => props.history.push(‘/topics’) }>TOPICS</ button>**

With Router()

withRouter() is a higher orderl component. A higher orderl component is simply a component, that takes another component as an argument, and returns a modified component containing **match, history,** and **location** props.

* **import { withRouter } from ‘react-router-dom’**
* **export default withRouter(ComponentName);**

To map through an array and pass props to each item in the array:

**render() {**

**return (**

**<div className = 'directory-menu'>**

**{/\* Maps the sections, destructures, and returns the 'id' + the remaining props (...otherSectionProps) into the 'MenuItem' component\*/}**

**{ this.state.sections.map(({ id, ...otherSectionProps }) => (**

**<MenuItem key={ id } { ...otherSectionProps } />**

**))}**

**</div>**

    )

  }

**Also...**

**render() {**

**const { collections } = this.state;**

**return (**

**<div className='shop-page'>**

**{**

**collections.map(({ id, ...otherCollectionsProps}) => (**

**<CollectionPreview key={id} {...otherCollectionsProps} />**

**))**

**}**

**</div>**

**)**

**}**

To filter an array to display a certain amount of items, in this case, 4:

**const CollectionPreview = ({ title, items }) => (**

**<div className='collection-preview'>**

**<h1 className='title'>**

**{ title.toUpperCase() }**

**</h1>**

**<div className='preview'>**

**{**

**/\* Filter the array to display only 4 items \*/**

**items.filter(( item, idx ) => idx < 4).map( item => (**

**<div key={ item.id }>{item.name}</div>**

**))**

**}**

**</div>**

**</div>**

**);**

Adding a Logo as a Link

To add a logo as a link to home page:

* import { Link } from ‘react-router-dom’
* import { ReactComponent as Logo } from ’../../assets/crwn-logo.svg’;
  + ‘ReactComponent’ is a special syntax for importing SVG files
* add link:
  + < Link to=’/ ’ >  
     <Logo className=’logo’ />

</ Link>

### Register and Sign In Components

Each component ( Register and Sign In ) will regulate its own state. This is because these two **do not** need to “talk” to each other. “Sign-In” doesn’t care what’s in the form of “Register”; and vice-versa.  **IF** the state was lifted to the page, containing both the sign-in and register, both components will be re-rendered; which is what we don’t want. Try to keep state to the lowest possible level.

#### The Job of a React Developer

**Remembe**r:

1. Decide On Components
2. Decide the State and Where It Lives
3. What Changes When State Changes

Firebase

Backend Components

* Database
  + Stores the information applicable to the application
* Server
  + Stores to code to most effeciently fetch the information from the database
* Authentication
  + The code to determine if a user’s creditials are valid to access the database’s information

Firebase is a great tool for frontend developers. It’s created by Google, and handles a lot of the backend stuff a backend developer would normally do for us, including:

* Effeciently querying the database
* Effeciently storing our information
* oAuth authentication
* plus more...

Getting Started With Firebase

To get started,

* go to the Firebase website and set up a new project (http://www.firebase.com) and click on ‘Go To Console’.
* Click ‘Create A Project’
* Name the project
* Accept the Terms and Conditions

Configure the project:

* From the ‘**Overview**’ tab, click the ‘**Web**’ icon
* Register the app with a nickname (ex: crwn-db)
* Copy the ‘**firebaseConfig**’ object to be placed within our code
* Go to our terminal and enter ‘**npm add firebase**’

Configure Firebase Within Our App

* Create a new folder **‘firebase**’ within our ‘**src**’ folder
* Create a new file within the ‘firebase’ folder, **‘firebase.utils.js**’
  + type **import firebase from ‘firebase/app’;**
  + **import ‘firebase/firestore’** (for database)
  + **import ‘firebase/auth’** (for authentication)
* Create a const, named ‘config’, using the copied API object
* Type **firebase.InitializeApp(config);**
* For Google database and authentication:
  + **export const auth = firebase.auth();**
  + **export const firestore = firebase.firestore();**

Setup Google Authentication

**/\* For Google Database (store) Authentication \*/**

**export const auth = firebase.auth();**

**export const firestore = firebase.firestore();**

**/\* Setup Google Authentication \*/**

**const provider = new firebase.auth.GoogleAuthProvider();**

**/\* To always trigger the Google popup \*/**

**provider.setCustomParameters({ prompt: 'select\_account' });**

**export const signInWithGoogle = () => auth.signInWithPopup(provider);**

**export default firebase;**

Configure Firebase to Use Google Popup

* Go back to the Firebase project and click on “**Authentication**” (*under the Development tab*)
* Click on “**Sign In Method**”
  + Enable ‘Google’
  + Enter an email address for project support
  + Within our project’s ‘sign-in’ component:
    - **import { signInWith Google } from ’../../firebase/firebase.utils’;**
    - Create ‘Google’ button:
      * **<CustomButton  
         className=’custom-button’  
         onClick={ signInWithGoogle }  
         >  
         {’ ‘}  
         Sign In With Google{’ ‘}  
         </CustomButton>**

To verify users have successfully logged in using the Google Authentication, under ‘**Authentication**’, click the ‘**Users**’ tab. The application itself isn’t aware. To add awareness to the application:

* Inside the application, go to ‘App.js’
* import { auth } from ’./firebase/firebase.utils.js’;

**#Whenever a user logs in with Goggle, or email and password, we want to store it in App.js to pass it to the components:**

* Ensure or convert the **App.js** to a class component; to access state
  + **class App.js extends React.Component{ }**
  + move the ‘**return()**’ method into the ‘**render()**’ method
  + add **‘constructer’, ‘super’ and ‘state**’
  + add **‘componentDidMount()**’ method:
    - **componentDidMount() {**

**auth.onAuthStateChanged(user => {**

**this.setState({ currentUser: user });**

**console.log(user);**

**});**

**#User will remain signed in, until signed out**

oAuth essentially allows users to signin with 3rd party credentials (Google, Facebook, Twitter, GitHub, etc...) **.**

To close the subcription to prevent memory leaks:

* **/\* Setup unsubscribe method \*/**

**unsubscribeFromAuth = null;**

* /\* Close the subscription to prevent memory leaks \*/

**componentWillUnmount() {**

**this.unsubscribeFromAuth();**

**}**

To determine if a user is signed in or out, add the ‘currentUser’ property to header within App.js:

* /\* Adds currentUser property to the header to determine

            if a user is signed in or out (an object or null) \*/  
  **<Header currentUser={ this.state.currentUser } />**

* Within header component, import the “auth” library:
  + **import { auth } from ’./firebase/firebase.utils.js’;**
* /\* Conditionally render a 'div' if currentUser is an object,

            or a 'Link' if it's false \*/

**currentUser ?**

**<div className='option' onClick={() => auth.signOut()}>**

**SIGN OUT**

**</div>**

**:**

**<Link className='option' to='/signin'>SIGN IN</Link>**

* /\* Destructure the currentUser property \*/

**const Header = ({ currentUser })...**