## Learning the Basics

Before doing this assignment, I was completely new to web development, to the extent I had no idea what VSCode/IDE was! I was introduced to various overarching concepts such as the MVC model as well as RESTful APIs, as well as various terminologies used in REACT such as components, props, state and so on.

One of the difficulties I faced was that many tutorials were using outdated versions of react libraries. For example, the most recent version of react-router is V6, while most tutorials still use v5. I was initially quite confused as to why some of the route props such as match, location, history were not working in react-router library. Apparently v6 route components were unable to receive route props such as match, location, history; hooks replaced them instead. This meant that to access these props, I had to modify my route components to be function components instead of class components in order to use hooks such as useLocation, useNavigate, and useParams.

Another concept I was introduced to was network security. When I was testing my app locally (frontend on port 3000 and backend on port 3001) it could not connect to my API; it gave a CORS policy error. This was because browsers block cross-origin requests: meaning that a request between code executed on different protocols, ports or domains have different origins. As my frontend and backend ran on different ports, it will experience communication problems unless CORS is configured to allow for such cross-origin requests. Adding the gem "rack-cors" and editing the config files correctly solved this problem.

## React-Redux

Although redux was not needed for the successful deployment of the app, I still decided to explore it as the concept of a "global" state (vis a vis "local" state) sounded very useful. This would especially be so if there are hundreds or thousands of components to be rendered and it would be a hassle to pass state props from one component to another. Just accessing the "global" store would save all the hassle.

At first, I intended my redux actions to be synchronous in nature when accessing the API endpoint. But this could pose a huge problem. A synchronous action dispatch (for API call) would result in a delay between the initial sync action and when the store receives the action to process. This is because we have to initiate the request, wait for server to respond, and so on – and while all these are happening, all other user interaction and code execution is stopped until the call returns. We have to find a way to execute code in the background without disrupting the flow of the programme.

This is where asynchronous actions come in, where other code continues to execute and the user can continue to interact with the page even if we're waiting for a response from the server. Redux thunk it is a middleware which allows for allows for side effects to be run without blocking state updates. It allows us to call action creators that return functions rather than just an action object. The function receives a dispatch method as an argument, which is used to dispatch the synchronous actions inside the function's body once the asynchronous operation (API call) has been completed. This allows for delayed actions, and resolving each promise that gets returned(calling fetch() returns a promise).

Another concept that I learnt was the immutability of the redux store. The state of the app can only be changed by a category of pure functions called reducers. I initially wrote a reducer to sort the redux store alphabetically, and this reducer directly mutates the state. But it did not work out as plan and the component was not re-rendering even though I thought I had changed the state. This was because I had directly mutated the state object, and while this would change the values of the root state object, the object itself will not change. Redux performs shallow equality check (vis a vis deep equality check). When comparing objects, it does not compare their attributes. Rather, it only checks whether 2 variables reference to the same object. So even though I had changed the attribute of the object, the pointer still references to the same object as no new object is created. Hence redux did not detect the state mutation and will not re-render. I eventually used the state.slice() method to create a copy of the state array before sorting.

**AWS**

My initial plan was to deploy my app to Heroku, but I decided to go for AWS instead for 2 main reasons. The first was to broaden my learning exposure: It is only a matter of time I have to learn AWS, be it whether I choose to specialize in software engineering or head my own startup. I also found out that many of the apps are deployed on AWS in CVWO (please pardon me if I'm wrong haha) and learning AWS now would definitely be useful should I get selected to join CVWO. The second reason is more practical in nature. Heroku is more user-friendly to new developers, automatically configuring the required infrastructure/database, and is more suited for small tasks like this, AWS provides powerful and flexible infrastructure capabilities. The ease of configuration and scaling up that AWS brings is crucial for any infrastructure that is expecting huge volumes of traffic in the long run.

I had so much trouble understanding the various services that AWS offers, but I eventually chose to host my front end on a S3 bucket while the backend on an EC2 instance. The frontend deployment was pretty straightforward. I spent some time learning how to configure my bucket, such as editing the permissions bucket policy to give users read access to my bucket Easy command line interface also helps: I just had to add a script "deploy" to package.json which will use AWS CLI to sync the working directory to the S3 bucket. And just run the command yarn build and yarn deploy would deploy to your S3 bucket!

The real pain was configuring the backend. My first option was to do everything manually (configuring everything on the EC2 instance) but it became too tedious. I then proceeded to try using Amazon Elastic Beanstalk with RDS (relational database), but a problem left me stuck for days and I was forced to try another route.

Eventually I chose to deploy my app using Capistrano, which is a tool for deploying applications using automated deployment scripts. After SSH-ing into my EC2 instance, I installed a whole list of dependencies such as nodejs and yarn repositories, as well as Ruby version manager rbenv as well as Bundler, which would provide a consistent environment for Ruby Projects. I then proceeded to install Nginx (web server) as well as Passenger (application server) and edited some configurations. This was followed by setting up a Postgresql database. Back on my local machine, I then installed

Capistrano, modified some config files and then deployed my backend using Capistrano.

## Future Plans

I really wanted to do a user authentication system but I didn't have time for that as I spent quite abit of time on getting AWS to work. Hopefully I can explore more of that in future.