CSC301 - Assignment 2 Report

Frontend

React

React was the first frontend framework we explored as it is immensely popular, and something we have used before in projects previously. React is very well documented and almost any issue that comes up with this framework is easily resolvable by one simple search on Google. The many benefits include the ability to reuse components, a state management library (react-redux), component libraries (MUI), very simple to setup (npx), fast rendering and test friendly (easy to write unit/integration tests). Some of the less appealing factors about React is that it is generally considered a bit JavaScript intensive and the code can be a bit too verbose (it can get messy). We picked React as the frontend framework for our application for the reasons mentioned above.

Vue

The benefits for using Vue is that it is well documented and also comes with its own state management library (Vuex) and material design framework Vuetify. Compared to React, it is conceptually very similar and many believe it to be less JavaScript intensive. There was very little to pick between React and Vue, but since we have used React more recently, it would be easier for us to get started.

No Framework (Plain HTML/CSS and Vanilla JS)

We very briefly considered not using a frontend framework at all and working with plain HTML and CSS. The pros was that this is quite simple to deploy, allows a lot of freedom for customization and it is well supported by pretty much any backend framework (in particular Django). However, the effort required to build individual components, make the application reactive and write CSS manually to make the UI look visually appealing was simply too much compared to frameworks that provide all those features of the box.

Backend

Express

Since we decided to use React for our frontend framework, it was natural to lean towards a JavaScript backend. The benefits of using Express include the extremely simple setup and deployment process, the ease of adding middleware (including libraries for serving static files and CORS configuration), and the ability to integrate with an easy to use database like

MongoDB. However, Express doesn't come with a extensive set of prebuilt functionalities like Django (for example user authentication and built in database). In the end, we did eventually decide to go with Express due to the familiarity we have with JavaScript and its deployment process.

Django

Django was considered for its built-in database and its user authentication system. It also allows us to use Python, which is a very familiar language to us. There are also many resources online on the Django framework as well as Django's own tutorials and guides if we ever ran into trouble or did now know how to do something with Django. The downside was that we either had to use no framework for the frontend or we would have to convert it to an API for the frontend to use. This would mean that the built in user authentication system couldn't be used, removing one of the benefits for us. As mentioned before, no framework for the frontend would make it difficult to create a responsive and reactive UI.

No Backend

We considered having no database and backend and instead doing all the logic and storing all the data in the frontend of the application. We would not have to spend time connecting a database and making API calls but could instead just get information from functions and reading files. This could save time on deployment and allow us to focus on other areas. However, setting up and deploying a database did not prove to be too difficult and not having a database limited the extensibility of the application. So a backend with database was decided instead of doing everything on the frontend.

Database

MongoDB

MongoDB was almost the obvious choice given that we went with React for our frontend and Express for our backend. It is very easy to integrate (simply have to add the mongoose library to create schemas and query data). In addition, we felt that since in this assignment the data we were going to be dealing with is pretty simple and we don't need too many complex queries it would be easier to go with a NoSQL database like MongoDB. There are no real downsides to using MongoDB for a small assignment like this in particular. However, if in the future we decide to expand functionality that would require making complex queries and more restrictions on memory usage, we might prefer switching to a SQL database. It was a simple choice to go with MongoDB since we could deploy it easily to production and it integrates very easily with our JavaScript frontend and backend frameworks.

PostgreSQL

PostgreSQL was the SQL database that we considered, it is widely popular and well documented. There are also ORMs available (for example, sequelize) that would make querying

and defining schemas easier. The disadvantage with going with PostgreSQL was that we didn't have much experience with using it and it required more effort to integrate than the database we eventually decided to use (MongoDB).

SQLite

SQLite was considered because it is bundled together with Python, is lightweight and is reliable. It is also what Django naturally uses, if we wanted to use Django or Python for the backend. Since we were familiar with Python this was naturally one of the choices to consider. However, we had no experience connecting SQLite to a frontend, and we had experience connecting other databases to create a full stack application so we decided to go with other choices that we were already used to.

Summary

We have decided to use the MERN Stack (MongoDB, Express.js, React, Node.js) to develop a full stack Price Calculator application. Even though the functionality of the application is very simple, our goal was to make the application as extensible and configurable as possible so that if in the future a developer decided to add functionality it would be very easy. We went with proven frameworks that give both high performance and a good development experience. We developed our frontend with React and backend with Express and Node.js. We bundled our React code into static files and served it on the same server as our Express backend. We added API routes that retrieved item data from a production MongoDB database. These API routes are called by our frontend React application where required. In addition to MongoDB, we also used local storage in the browser to save a user's shopping cart. If a user decides to exit the tab, their data will still be persisted. Unit tests can be run both on the CI/CD pipeline and locally. The code is neatly organized in a monorepo with client and server folders. The CI/CD pipeline automatically runs unit tests and deploys on Heroku (using Github Actions) when a pull request is merged on the master branch. It will also log errors and notify the developer of any failures. This whole setup allows a developer to push their code onto production in less than 5 minutes.

References

References are added as comments in the specific code files they were used. They are also listed here and in the README in the repository.

- This application is largely based on the following tutorials (repository structure, package.json for client, and backend):
    - https://www.youtube.com/watch?v=lR1gR9WhY10
    - https://www.youtube.com/watch?v=swgH9MGM9nM
    - The code from the tutorials is: https://github.com/SparkDevTeams/ds2020_mauricio.
- Dockerfile is based on this guide:
    - https://nodejs.org/en/docs/guides/nodejs-docker-webapp/
- Mongoose Schemas is based on the documentation:
    - https://mongoosejs.com/docs/guide.html
- For setting up state management:
    - https://redux.js.org/tutorials/fundamentals/part-3-state-actions-reducers
- For setting up Heroku Deployment using Github Actions:
    - https://github.com/marketplace/actions/deploy-to-heroku
- For setting up Nodejs building and testing using Github Actions:
    - https://github.com/actions/setup-node
- The react components were taken from MUI:
    - https://mui.com
- The layout of the cards was taken from:
    - https://javascript.plainenglish.io/render-react-cards-and-images-dynamically-2387434e809d