Project Documentation

EC 327 Project

Tick Stock

Los Perros

Kai Imery, John Culley, Jose Batlle , Ruben Carbajal

****

*The Process & Meaning:*

To start, we created a website using React, which is an open-source front-end javascript library. This was optimal for our project because of its modular structure and numerous packages which provide more customization. Our project’s marketability is focused on its ability to provide quick and meaningful information on a user-inputted stock that would help influence the users choice when making investments. The web application will preload with Google on the page, then the user can input a stock. The web application will then load a live ticker, a YTD candles chart, important details, and relevant news articles. For all of this real-time information to be possible, we used Polygon.io stock APIs, which stands for an application programming interface. This API provides easy access to large-scale data that would otherwise have to be accessed through a querying language such as SQL. One downside to using APIs is the cost to use them on a large scale, our free APIs from Polygon IO are limited to five queries per minute. To operate our live ticker a different API was used called FinnHub which provides thirty queries per second on live stock data and our web application also stored the previous value and changed color between red and green depending on whether the live value is increasing or decreasing.

Because of the modular structure of React applications, we divided our application into six components which are then accessed by App.js to construct the final screen. App.js can be thought of as the ‘main function’ while all of the components are then called inside operating similar to functions. The web application was divided into Header.js, LiveTicker.js, Aggregate.js, Details.js, and News.js. All of these files are then placed accordingly into App.js where there is a divider function that separates each of the components.

Header.js component is constructed using an AppBar provided by the library Materials UI. This AppBar has the title on the left, and then a search bar on the right side. To start with the header, the script first renders a search bar with an *AppBar*. Then it imports the necessary components from the MUI/ Material Library, as well as the Axios, *useEffect,* and *useState* hooks from react. It also uses the styled-components to style the search bar. The Header also has three functions, *inputHandler, handleSubmit,* and *handleKeypress*. The *inputHandler* function operates when there is an *onChange* event within the search bar where it then accepts the user’s input and converts it to lowercase and stores it in another state variable because of *useState*. This new state variable is then used within the other functions. Now that the user’s input is stored in lowercase, the user can either use the submit button or use the enter key to submit their request. When the submit button is pressed, *handleSubmit* is called and checks the user’s input against a string array of stock tickers to verify that they correctly entered a stock. This value is then passed out of the Header.js component to then be used by all of the other components as a parameter for all of their API calls. When the user presses the enter key it first directs to *handleKeypress* where it performs a check to ensure that the enter key was the one pressed, and if correct it will redirect to *handleSubmit*.

The second component we will go into now is the aggregate component, which holds the purpose of displaying a candlestick chart of stock’s data, displaying their logo, a live price, and providing a short description of the company. The aggregate() component takes *query*, which is the property that contains the stock ticker that the component will use to fetch data from the two APIs, as an argument. The first thing that the program does is that it stores the system’s current date, and converts the day variable into the appropriate format for future API calls. Then the component uses the *useState* hooks to define state variables: *series*, *price, pastPrice, priceTime, logourl, mainurl, and descript.* Following the initialization of the useState variables, the program uses three useEffect hooking inorder to define three asynchronous functions: getStocks(), getReal(), and getdetails(), with dependencies on query, query and price, and query respectively. The *getStocks()* function takes in *query*, the variable current (which is the current date), and the corrected formatted day in the variable day. It proceeds to call the API and store the object in its promise variable data. The data variable is then manipulated into an array called historical in order to correctly format it according to the documentation of ApexCharts, and stores the values within series.Following this is the getdetails useEffect which is passed the query string and calls the details section of Polygon.io, and stores the returned object in its promise variable detail. Then the data is partitioned into its needed variables *logourl, mainurl,* and descript which take on their respective values of the url to the company’s logo, url of the company’s home page, and the polygon.io description of the company. Finally, the last *useEffect* has an embedded asynchronous function getlive() in order to control the refresh delay of the *getReal()* function call. The *getlive()* function allows for the *getReal()* function to be called every five seconds which then sets the *price* and *pastPrice* variables to the current price and the previous price of the stock. These two variables are then coupled with a useMemo() function in order to display the price as green, red, or default in relation to whether the price increased, decreased, or stayed the same respectively. Finally, the *Aggergate* component uses the *Chart* feature from react-apexcharts library to render a candlestick chart with the data in the *series* state variable along with *priceTime.*

Moving forward, we then created the News component. The News component had the purpose of displaying recent news of a given company. This is all done by importing the Material UI library to create a grid of new cards. It is a news app that fetches news data from the Polygon API and displays it in a grid of cards with three columns. First, we import Material UI components, such as ‘Card’, ‘CardContent’, ‘CardMedia’, ‘Typography’, ‘CardActionArea’, ‘CardActions’, and ‘Grid’. Also, it imports the *useEffec*t and *useState* hooks from React. Next, the *getNews()* function is declared, which is an asynchronous function that takes query as a parameter and uses it to build the URL for the Polygon API. Then it uses the fetch() function to GET a request to the API for the JSON response. Then the *News()* function is created which also takes *query* as a parameter, and it is passed into the *getNews()* function when it is called inside the *useEffect* hook. The useState hook is used to initialize and manage the *newsdata* state variable, which is an array of *news* data. The *useEffect* hook is used to call the *getNews()* function and update *newsdata* state variable with the SPI response. The *newsdata* array is then mapped over to create a grid of new cards, with three columns. Each card displays the news item’s image, title, and description.

The next component we worked on was the *Details* feature. This had the purpose of fetching data from the API such as the volume, highest, lowest, open, and the close trading price of a respective stock and putting it in a table. It is done by a function defined as *gettablevals()* that takes *query* and uses it to build an API endpoint URL. It then makes an HTTP request to that URL using the fetch() function to get a JSON response. The details component also uses the *useState* and *useEffect* hooks to manage its internal state and perform side effects. The *gettablevals()* function is called in the *useEffect* hook to fetch data from the API when the details component is mounted. The data is stored in the *tablevals* state variable. *Tablevals* is then used to render a UI table to display the information.

*Technical Challenges:*

Now we can get into some of the technical difficulties we had while creating this project. Having all the correct project files and being able to successfully run npm start is an important first step in building our web application. Initially, we struggled to get our first successful API query but eventually, we were successful. Another challenge was figuring out how to add parameters to our queries. Also, learning the ability to pass a variable sibling-sibling component. To continue with the challenges, styling the web application itself and learning how to refresh the graph when new input was added was notable. Another one was formatting the news articles into a three-per-row layout and displaying the details in a table format.

References & Sources:

Polygon.io API: <https://polygon.io/>

Finnhub API: <https://finnhub.io/>

Apex Charts: <https://apexcharts.com/>

Material UI: <https://mui.com/material-ui/getting-started/installation/>

CSS help: <https://www.w3schools.com/css/>