Team Alex, Jennifer, Vadim

1) **An issue with the stock page loading price and information**
    a) After initially searching up a stock price, or selecting a stock that the user owns, any subsequent searches for a new stock would continuously display the previous stock's information. The solution required the stock quote information to return the response and await on a variable waiting for the response.
2) **Displaying Financial Information**
    a) The FinnHub API also allowed for us to query financial information at any time frame, so we decided to query the previous quarter. However, the APIs response at times didn't always match the format we were expecting. We required data from the report object, an example of this issue would be:

```
{
    "cik": "320193",
    "data": [
        {
            "accessNumber": "0000320193-24-000006",
            "symbol": "AAPL",
            "cik": "320193",
            "year": 2024,
            "quarter": 1,
            "form": "10-Q",
            "startDate": "2023-10-01 00:00:00",
            "endDate": "2023-12-30 00:00:00",
            "filedDate": "2024-02-02 00:00:00",
            "acceptedDate": "2024-02-01 18:03:38",
            "report": {
```

```
{
    "cik": "320193",
    "data": [
       {
             “Misc”: ...
       }
        {
            "accessNumber": "0000320193-24-000006",
            "symbol": "AAPL",
            "cik": "320193",
            "year": 2024,
            "quarter": 1,
            "form": "10-Q",
            "startDate": "2023-10-01 00:00:00",
        "endDate": "2023-12-30 00:00:00",
        "filedDate": "2024-02-02 00:00:00",
        "acceptedDate": "2024-02-01 18:03:38",
        "report": {
```

- We tried to place as many edge cases as we could but not all stocks were covered, so if the response was not in the expected format we would display, “Could not retrieve financial information”

3) **The Rotating Navigation Bar is not rotating far enough**
    a) Based on the Udemy course for a rotating nav bar, our implementation of the navigation worked properly until more HTML was placed on top of the nav bar, it would rotate to the point where both the close and open buttons were displayed. By analyzing the code and performing guesses and checks we fixed this by placing the close and open buttons farther apart and increasing the rotation amount and decreasing rotation speed.
4) **Rotating Navigation Jumping**
    a) After opening or closing the rotating nav bar, the circle that had the buttons to open and close would jump up and down before completely closing. After some debugging, we realized that the rotating navigation and search bar (two features implemented from the Udemy course) were messing each other up.The reasoning is that whenever the rotating navigation would be clicked, it would change its position to be relative to right underneath the search bar. However, setting the position of the search bar to fixed or absolute led to problems with it displaying. We fixed that problem by setting the width to 100vw, but the search bar would disappear when the rotating navigation is rotated. Perhaps add show-nav class (used by rotating navigation) to the search bar to make it rotate as well? But that doesn't work, in particular because the CSS does everything based on it being in the container class for rotating navigation. So why not put the search bar within container? That doesn't quite work either with positioning. Then we changed the z-index of the search bar to go above the page, which helped see the search bar at all times, but it covered the menu button. Changing the z-index of the menu button to be higher than the search bar worked, but not when it was in the rotated state for some reason. Apparently z-index has troubles with rotated content. The solution we thought of combined everything, plus adding a little animation to slide the search bar off screen.
5) **Environment Variables**
    a) Our server.js contained all of our backend code and pipelines from and to our database and API calls to FinnHub. Due to this, we also had to reference our API key and our database password as plain text in the code. We understood this to be a major security hazard, we also knew about the existence of env variables, but we believed them to be similar to PATH variables and placed our API keys and passwords in this method. This was not the case and instead after further research,

we found the dotenv module which can use a .env file and reference any key/value pairs within that file to fill in any information you need without requiring the plain text.

6) **Issues with CORS**
    a) In our previous project we had webpages that were statically linked to one another and had scripts attached to each html page that were querying our database, this required a CORS exception on the database side which would allow traffic from a specific URL. However, this project required node to host HTML, CSS and JS, and since node also has CORS restrictions it took a while before we realize that exceptions should be placed within our node server, server.js as well.

7) **Chat Bot Initial Implementation**
    a) In the early stages of development, we created a direct copy of the LinkedIn Learning chatbot course which would communicate to a Slack chatroom. After getting familiar with it we decided to make a test page that the chat bot will communicate with, this caused a lot of issues like require not being defined, modules must be in a mjs file, and const not being defined. What we did to fix it was create a basic server.js file and bring in the required classes and methods from the LinkedIn Learning course along with the endpoint of the queries to the NLP/LLM API, called Wit.AI, used by the course. Moreover, rather than sending and receiving messages over Slack, which is unrelated to our website, we implemented HTML and JS to have the chat history display directly on the website. This fixed all of the errors described previously and we were able to communicate with the chatbot.

8) **Chat Bot Wit.AI**
    a) We used Wit.AI, an NLP API, to help us with our chat bot. This way we could ask questions and make requests in different ways, while still reaching this same conclusions/responses. One problem we faced is that after fetching the API, we weren't obtaining enough information (ie. we only received entities but not intents or traits). The problem was really simple. In the LinkedIn Learning, on the server side we set it so that we would only return the entities received from the bot. Instead, we set it so that all information would be sent back to the client side.

9) **Chat Bot Close Button**
    a) For the chatbot, whenever we would type a message but click the X button rather than the Send button, the message would still be sent. We realized this is because the code sends the message whenever a "submit" button/action is recognized. To fix this, in my code to open/close the chatbot, I changed the value of the text input box to an empty string. In the code, if the message is empty, it isn't sent, so this worked out.

```
// To open and close the chatbox.
function toggleChatBox() {
 textInput.value = '';
 const chatBox = document.getElementById('chat-area');
 chatBox.style.display = (chatBox.style.display === 'block') ?
'none' : 'block';
}
```

10) **Chat Bot Stock Transactions**

   a) The chat bot has several features, including greeting the user, providing information about the website, and providing information about stocks and trading in general. This is all made possible thanks to Wit.AI. However, the struggle came from permitting users to buy and sell stocks through the chat bot, by asking it directly (eg. "Buy 2 shares of AAPL"). Even though we reimplemented the code from buying and selling the normal way, through the Stock page, we kept running into problems, specifically when we would use the chat bot on the Stock page. On the Home and Account pages, it worked correctly. The problems included content not displaying, errors being thrown, and the transactions just not working. Eventually, we figured out that it was due to a problem with the way JS complies code. We were using two JS files on the Stock page, stockScript.js which handled the page's JS, and chatbot.js which handled the chatbot JS functionality. Since we reused code from stockScript.js, we ran into problems with global variables and identical function names. The code would use the wrong functions and variables. To fix this, we made sure to not use global variables in chatbot.js, and to change all the function names. This fixed the problems.

11) **Hashing**

   a) When we first began the project we were storing passwords as plaintext for testing purposes, we then moved on to hashing the passwords with the same salt, after some research we learned we needed to use unique salts during every hash, but we were confused as to how and where we would store these unique salts so that we can check user passwords. After reading through the documentation for the module bcrypt we found that the salt is stored within the hash itself, and bcrypt can just get the salt from the hash apply to the user input compare it to the hash stored in the database and confirm whether it is a valid password

Hash format for bcrypt:

$algo$cost$salt+hash

Where algo is the algorithm identifier (e.g., 2a for bcrypt), cost is the number of rounds, and salt+hash is the concatenated salt and hash.

12) **Storing user variables**

   a) Our project hosts all of our webpages on a node.js webserver, because of this we needed to store data such as the user id, stock information, and the balances of the user as they move from page to page. Of course, we could continuously query our databases and API for the information again, but it would be computationally and monetarily inefficient since we were using the free version of FinnHub. We found a module that is built in javascript called SessinoStorage allowing you to store information locally on the user's browser allowing pages to access variables that was archived from another page

13) **Receiving Number values from API or DB queries**

  a) The project had to manipulate alot of numbers such as a user's account balance, stock price, and financial data, we believed it would be an easy task since sometimes you need to perform arithmetic operations on them and other times you just need to display them. It was not as simple as we originally thought, some scenarios would throw errors such as 'Cannot parse value' or 'Cannot perform expression on int'; from our research, it seemed that many if not all values that come in from the html or queries are not properly parsed and may be a type such that operations cannot be performed on them. To solve it we explicitly stated what type of value should be used by using parseInt() or parseFloat() and also force all decimal values to 2 decimal places to avoid confusion or better understanding.