Case Study 1: Real-Time Stock Market Dashboard

1. How would you implement a WebSocket connection in a React component for realtime data fetching?

A. To get real-time data, we can use WebSockets in React. we can create a WebSocket connection inside a useEffect hook, so it opens when the component loads and closes when it unmounts. This way, the connection is always up while the component is live, and we can use the data it receives to update state and display it on the dashboard.

2.Describe how you would create a responsive table to display stock prices.

A. For the table, I'd use CSS or a library like react-bootstrap to ensure it looks good on all screen sizes. Flexbox or Grid would be helpful for structuring the table. It would display the stock data in rows and columns and should adjust to different devices. On smaller screens, the columns could collapse into a more compact view for readability.

3. How can you implement a search bar to filter stocks based on the user's input?

A. I'd add an input field for the search bar and use React's state to store the user's input. As the user types, the table would automatically filter the stock list based on the input (e.g., filtering by stock symbol or company name). I'd do this by creating a filtered version of the stock data array and displaying that filtered list in the table.

4. Explain the steps for handling connection errors and displaying appropriate messages to the user.

A. If the WebSocket connection fails, I'd show a clear error message to the user. To do this, I'd add error handling in the WebSocket setup and catch any issues with connection or data. When an error occurs, I'd update the UI to show a message like "Connection lost. Trying to reconnect..." and possibly try to reconnect automatically after a delay.

5. How would you ensure the efficient updating of stock prices in the UI without performance degradation?

A. To avoid performance problems, especially with a large number of stocks, I'd only update the parts of the UI that need it. React's diffing algorithm already helps with this, but I'd also use tools like React.memo to prevent unnecessary re-renders. Additionally, I could use request animation frames to ensure the UI updates are smooth and avoid bogging down the browser by batching updates together.