Stock Trading Platform

Project Background

The goal of this project will be to create a stock trading web application that supports multiple users. Users will be able to create and 'fund' accounts through the application. Each user of the application will have their own stock portfolio. Users will be able to place buy/sell orders for stocks, create watchlists to track stocks of interest, and receive real-time notifications for events of interest.

To understand how a basic stock market works, some background information is provided at the end of this document. The following sections will outline the minimum requirements of the stock trading platform project. You are encouraged to ask questions to clarify the requirements and constraints of the project. The details for this project may be updated slightly throughout the term if necessary. It is the students' responsibility to ensure they are up to date on any changes to the document. A file summarizing the changes to the document will be maintained and shared with the students.

Provided Data and Assumptions

There is no requirement to tie this web application to any real stock market. You can create your own list of made up stock symbols (1-3 characters, all capitals) or use the symbols provided in the symbols.txt file. It would be beneficial to create several initialization scripts for your project to automatically create some base data to work with. This will speed your testing/development process as you will not need to manually perform setup every time you test a feature. These scripts can also be provided in your final project submission to allow TAs to easily setup specific scenarios to test.

Various parts of this document refer to daily statistics. For the purposes of this project, you do not need to work with real dates. Instead, you can work with simulated time and count the number of days from the server starting. You must provide a way to simulate the ending of a day within your application (e.g., provide an admin control page with an 'End Day' button or a URL that moves time forward one day when requested). When the day is ended, your server should handle any end-of-day processing that is required, and simulated time should proceed to the next day. It would be beneficial from a testing point of view to include the current 'date' on each screen the user sees.

You can assume that integer prices are used for all buy/sell orders.

Technology Constraints

The main server code for your project must use Node.js. All client resources required by your project must be served by your server. Your project's data must be stored using MongoDB. Any additional software/modules/frameworks you use must be able to be installed using your NPM install script. The backend of your project must be able to run successfully on the provided course OpenStack image and your client must work within an up-to-date Chrome browser. An updated list of allowed modules/frameworks/etc. will be included with the project documents. If you are unsure if something is allowed or would like to see if something could be allowed, you should ask for clarification before proceeding.

User Accounts

The application must provide a way for users to create new accounts by specifying a username and password. Usernames must be unique. A user should be able to log in and out of the system using their username and password. Within a single browser instance, only a single user should be able to be logged in at one time (i.e., user A and user B cannot log in to the system within the same browser window). All newly created accounts will have a cash balance of $0 and will not own any shares of any stock.

When a user is logged in, they should be able to view information about their stock portfolio and manage their account. The application must provide a way for the user to:

1. View their current cash balance and total portfolio value.
2. View their current stock portfolio, including the number of shares of each stock they own, the average price they have paid for the shares of that stock, and the current value of the shares of that stock. This information must periodically update in real-time on the page to reflect any changes in the stock prices.
3. View and manage their current orders. The user should be able to see all of their current outstanding orders (i.e., those that have been placed but not completely fulfilled), including the stock symbol for the order, the type of order (buy/sell), the price they entered for the order, and the number of shares of the order that have been fulfilled. The user should be able to cancel an order at any time before it is completely fulfilled, which will void the remaining unfulfilled portions of the order.
4. View and manage their watchlists. More watchlist details are included later in the document.
5. View and manage their event subscriptions. More event subscription details are included later in the document.

6. View and search their account history, which should show details for every transaction that has occurred on the user's account. The user should be able to filter this history by date range and/or by specific action (purchase of stock, sale of stock, deposit to account, withdraw from account).
7. Deposit or withdraw funds from their account. For the purposes of this project, you can simply allow the user to specify how much money to withdraw/deposit and, assuming the values are valid, update the account accordingly.

Viewing, Buying, and Selling Stock

The application must provide a way for logged in users to search for specific stock symbols and view a single stock. You can decide whether a user who is not logged in should be able to search/view basic stock information. When a user selects a stock symbol to view, the application must provide a way for the user to:

1. View daily/current information about the stock, including the current price, the daily low/high prices, the current bid/ask prices, and the number of shares traded so far this day. This information must periodically update in real-time on the page to reflect changes in the market.
2. View historical daily information about the stock including the price of the stock at the end of each day (i.e., the 'closing' price) and the number of shares traded for that stock during each day. The user must be able to specify the date range for this information. You are free to include additional historical information if you want.
3. View the amount of the selected stock they have in their portfolio (if any), including the number of shares, average price paid for those shares, and the current market value of those shares.
4. Place an order for that stock symbol by specifying whether they want to buy or sell the stock, the number of shares of stock they wish to buy/sell, the price they wish to buy/sell the stock for, and whether the order should expire at the end of the day or remain until cancelled. A sell order for X shares should be considered valid if the user owns at least X shares of the specified stock. A buy order for X shares should be considered valid if the user has at least (X * desired price) dollars in their account. A stock order with daily expiration will be removed at the end of the day if it has not been fulfilled. A stock order that is good until it is cancelled will only be removed when it is completely fulfilled, or when the user cancels the order manually. The user must be notified in some way if the order was successfully placed or if the order was not

successfully placed. The user must also be notified in some way when the order is fulfilled successfully.

5. Add the stock symbol to any of their watchlists.
6. Register an event subscription for this stock by specifying the minimum daily price change in the stock required to trigger a notification (e.g., +5%, -10%). You can choose to support other types of event subscriptions as well.

Watchlists

The watchlist functionality will allow a user to maintain named lists of stock symbols so that they can easily view information about those stocks. The user should be able to create a new watchlist by specifying a watchlist name. The user should also be able to delete a watchlist. Watchlists should initially have no stock symbols. A single watchlist should not allow duplicate symbols, but it should be possible to add the same symbol to multiple watchlists. When viewing a watchlist, the user must be able to:

1. See each stock symbol on the watchlist, as well as important information about each stock symbol including current price, current bid/ask, daily number of trades. This information must periodically update in real-time on the page to reflect changes in the market.
2. Navigate to view a single stock through the watchlist (e.g., using a link).
3. Remove a stock from the watchlist.

Event Subscriptions

The event subscription feature will allow a user to receive automated notifications in response to specific events (e.g., stock X changes 5%+ in one day). The user must be able to initialize event subscriptions when viewing a single stock by specifying the minimum percentage change in price that must occur in a day in order for the notification to be triggered. The user must receive a notification if the constraints of the event subscription are ever met. The user should receive only a single notification for each event subscription in a given day. The user must also be able to view and manage their event subscriptions. When viewing their event subscriptions, the user must be able to:

1. See each subscription they have set up, along with important information such as the stock symbol the subscription is for, and the event constraints (e.g., minimum change per day).
2. Remove an event subscription. This will completely remove the subscription from the user's profile.
3. Deactivate or reactivate an event subscription. Deactivating will remove the notification for the event but keep the event subscription in the user's profile

so they can enable it again easily. Reactivating will turn the notification on again.

4. Modify an event subscription to change the properties (e.g., the minimum change per day). You can assume that the stock symbol for an event subscription cannot be changed.

REST API

In addition to the web-based client that most users will use, your stock trading platform must also provide a public JSON REST API that supports the following routes and parameters, at minimum:

1. **GET /stocks** – Allows searching of stock symbols. Returns an array of stock symbols that match the query. Must support at least the following query parameters:
    a. *symbol* – A string that should be considered a match for any stock symbol that contains the given string *symbol*. If no value is given for this parameter, all stock symbols should fulfill the symbol constraint.
    b. *minprice* – A number that should be considered a match if the current price of the stock is greater than or equal to *minprice*. If no value is given for this parameter, the minimum price constraint should be ignored.
    c. *maxprice* – A number that should be considered a match if the current price of the stock is less than or equal to *maxprice*. If no value is given for this parameter, the maximum price constraint should be ignored.
2. **GET /stocks/:*symbol*** – Allows retrieving information about a specific stock with the given *symbol*. Returns an array containing a single object entry for each day in the requested date range. For each day in the given range, this information must include at least the daily high/low prices, the closing/current price, and the number of shares traded during the current day. If no query parameters are specified, then only the current day's information should be sent. This route must support at least the following query parameters:
    a. *startday* – A number representing the starting point for the historical information. If an *endday* parameter is included but the *startday* parameter is not, the *startday* default value should be day 0 (i.e., server start).
    b. *endday* – A number representing the ending point for the historical information. If a *startday* parameter is included but the *endday* parameter is not, the *endday* default value should be the current day.

3. **GET /stocks/:*symbol*/history** – Allows retrieving information about past trades of the stock with the given *symbol*. This information must include the buyer username, seller username, buy price, sell price, and number of shares exchanged for each successful order fulfillment with the given symbol within the given date ranges. If no query parameters are specified, then only the current day's information should be sent. Your API must support at least the following query parameters:
    a. *startday* – A number representing the starting point for the information. If an *endday* parameter is included but the *startday* parameter is not, the *startday* default value should be day 0 (i.e., server start).
    b. *endday* – A number representing the ending point for the information. If a *startday* parameter is included but the *endday* parameter is not, the *endday* default value should be the current day.

Project Report

You will also be required to submit a project report that must explain the overall design of the system, outline any assumptions that were made, provide supporting arguments for the design decisions that were made, discuss technologies used within the application, document your server's API, describe any testing completed as part of the project, highlight key features of the system, and provide documentation on how to use/test the system. You must be able to justify your design and implementation choices from a perspective that considers scalability, robustness, extensibility, and maintainability, as well as the overall user experience. It is important to document the design decisions you make, as well as your reasoning for making those decisions and any alternative approaches you tried or considered as the term progresses. This will make writing a quality report near the end of the term significantly easier. We will discuss potential sections and other outline details for the project reports throughout the term.

Pair Project Extensions

Pair projects will be held to higher standards than individual projects in general. If you are completing a pair project, it is recommended that you include at least 3 of the below extensions into your project, or similar extensions you come up with on your own. For those looking for an A grade on the project, pairs are encouraged to include more than the minimum recommended extensions and individuals are encouraged to include some extensions as well.

1. Use React, Angular, or another supported frontend framework.
2. Implement a GraphQL API in addition to the REST API.

3. Implement your own load balancing solution and simulate a distributed system by instantiating multiple instances of your server. This load balancer should be able to detect any of the servers going offline (e.g., simulating a machine crash). This load balancer could also handle the addition of more servers at runtime.
4. Support a full-sized/desktop interface, as well as a smaller, mobile-friendly interface.
5. Provide in-depth automated testing of your API and system, including documenting specific test cases and their expected results.
6. Implement your own caching mechanism for your server/API. Test your caching system under various circumstances and compare its performance to a version that does not use caching.
7. Integrate your server with an existing online news service and display current news about the real-world stock symbol within your application. Alternatively, have your project provide a news API that randomly generates news stories based on what happens in your stock market (e.g., top stocks of the day, etc.) and provide a way of viewing this news data within your application.
8. Develop a solution to allow algorithmic/automated trading within your system. A simple approach to this could involve modifying the event subscription system to allow automatic ordering to occur when an event is triggered.

Stock Market Background

The goal of a stock market is to facilitate the trading of shares of stock. For this example, we will assume there is a company with the stock symbol X that has issued 100 shares to user A. Adding more companies simply involves tracking orders for different symbols. Each share represents ownership of a small portion of the company. Since there are 100 shares of X, each share represents owning 1% of the company X.

Now, if A wishes to sell some of these shares, A can place an order to sell shares (also called an ‘ask’) by specifying the number of shares they wish to sell and the price at which they want to sell those shares. For example, A could place an order to sell 10 shares at a minimum price of $50 each. A sell/ask order specifies a minimum price, as the person would be willing to accept *more* than this amount if possible.

It is important to realize that these shares are not immediately sold. First, a buyer willing to purchase 10 shares at a price of at least $50 each must be found. Your web application will be responsible for matching buyers and sellers in order to complete the trades. Previously, A placed a sell order to sell 10 shares of X for $50 per share. If another person, B, places a buy order (also called a ‘bid’) to buy 10 shares of X for at most $50 per share, these orders can be matched and fulfilled. After fulfillment, A will own 90 shares of X and also have $500 cash they received from B. B will now own 10 shares of X. The price of a share of X will now be listed at $50, which matches the last price shares were successfully sold for.

The above is a simple example where the orders matched perfectly. In other cases, you may have a number of orders that are awaiting fulfillment. For example, A could place another order to sell 10 more shares of X for $50 per share. B could also place an order to sell 10 shares of X for $55 per share. If we were tracking all of the open orders, we may have recorded something like this:

Sell Orders: A – 10 shares at $50, B – 10 shares at $55
Buy Orders: *empty*

Now, what if user C places a buy order for 5 shares of X at a price of $45? There are no sell orders currently that would fit these constraints, which would leave our record of open orders looking like:

Sell Orders: A – 10 shares at $50, B – 10 shares at $55
Buy Orders: C – 10 shares at $45

Now, imagine another user D places a buy order for 15 shares of X at a maximum price of $55. It would now be possible to fulfill orders in many different ways. We could sell 10 of B's shares to D for $55 and also sell 5 of A's shares to D for anywhere between $50-55. We could also sell 10 of A's shares to D for anywhere between $50-55 and sell 5 of B's shares to D for $55. Assuming we carried out the first option, the order record would then look like (note that A's order has been partially fulfilled):

Sell Orders: A – 5 shares at $50
Buy Orders: C – 10 shares at $45

The stock owner of X would now consist of A (85 shares, with 5 of those still for sale) and D (15 shares). The current bid (highest buy order) for X would be $45 and the current ask (lowest sell offer) for X would be $50.

You are free to implement an order fulfillment algorithm in any way, assuming that it meets the following constraints:

1. If two orders exist simultaneously for the same symbol and with the same price, then the order that was placed first should be fulfilled first.
2. Buy orders never require the user to pay more than the price they specified.
3. Sell orders never result in the user getting less than the price they specified.
4. The fulfillment process will not stop if there are still valid orders to match.

A basic algorithm will keep a queue of orders placed for each price. As long as the current bid (highest-priced buy order) has a higher price than the current ask (lowest-priced sell order), it is possible to match two participants and complete a sale of some shares. Note that while you are free to implement the fulfillment process in any way that meets the constraints, how you choose to implement it may affect the overall quality of your system.