Software Architecture and Design Report

Mock Market

12/10/2021

Dylan Porter

Table of Contents

1. Context

1.1. Purpose and Scope

Stock market trading has become a popular tool for people interested in their financial future and well-being. With the advent of modern technology, stock market trading has become available to virtually everybody who is willing to trade in the market. Because of this, many people jump into the market with their own money and their own risks. Even though this is expected, many people do not understand the risks associated with the market. Even the most experienced traders are not able to product the winds of industry and when a stock price will go up or down. As a result, *Mock Market* was created.

Mock Market helps to enable those interested in market investing by allowing them to place theoretical trades on the actual market without using real money. The application provides an interface to conduct trades, including buying and selling of securities, using fake (mocked) funds. The interface will also show what the theoretical gain or loss would be given the individual's portfolio. This application will help with learning about market trading without the risk of monetary loses.

Ultimately, the architecture generated to accomplish this functionality will be outlined in detail throughout this document. *Mock Market* is a web-based application that communicates with a remote server to service user interactions within the program. The purpose of the architecture is to allow market trading with real-time stock quotes using mocked funds. In general, the architecture achieves database persistence of user actions so that a user session outlines previous market purchases, sales, and current profit or loss.

1.2. System Overview

Mock market is a web-based application that allows a user to create mocked stock market trades without using their own money. The system keeps track of the theoretical profit or loss of the mocked trades given the current market value of the securities. In addition, the user can also sell securities (stocks) to cash in on their value gained or lost through it. In general, the system does not require any personal information to make an account – allowing an individual to simply start conducting trades right away on account creation.

The *Mock Market* application also allows for people to withdrawal and deposit funds from their account. For instance, to purchase securities on the account, the user would need to deposit enough funds to make that given purchase. As a result, the system keeps track of all deposits and withdrawals that are made throughout the history of the account. In general, all transactions are logged and recorded for the user's reference.

Once a user creates an account, all of their data should be persisted so that when they login, the transactions will be up to date. The application also provides an interface for users to

search for stocks based on ticker name. Once selected, the user can view data related to the stock such as current price, high price, low price, volume, and other market indicators that are relevant. The system also includes information on the amount of stock that is currently owned by the user.

The system is also concerned with maintaining a secure user session throughout the duration of use for the application. As a result, deference was given to OAuth (a third-party application) to handle proper encryption and user account management. There are also database hosting requirements and reliability concerns with the system that was mitigated through the use of *Heroku*. The architecture will be discussed in further detail later in this document.

1.3. Architectural Background

Mock Market is a web-based application that is primarily targeted for desktop computers (mobile device support is still being worked on). At the core of the front-end, the application is using Node.js, HTML, CSS, and JavaScript to render applications. From a higher view, it is using React.js, Redux, and Bootstrap to facilitate the actual application rendering. React.js helps maintain an application state that makes it easier to build larger applications – it is primarily served as an alternative to JQuery.

On the backend, Node.js is primarily being used (in conjunction with Express.js) to formulate API endpoints that the front-end can access. The backend receives an HTTP request from the front-end, interprets the response, and implements application logic to return a response back to the front-end. For the database, postgresql is being used to service and persist the application state. This was setup in this way so that the backend could interact appropriately with the front-end through easy to use REST APIs.

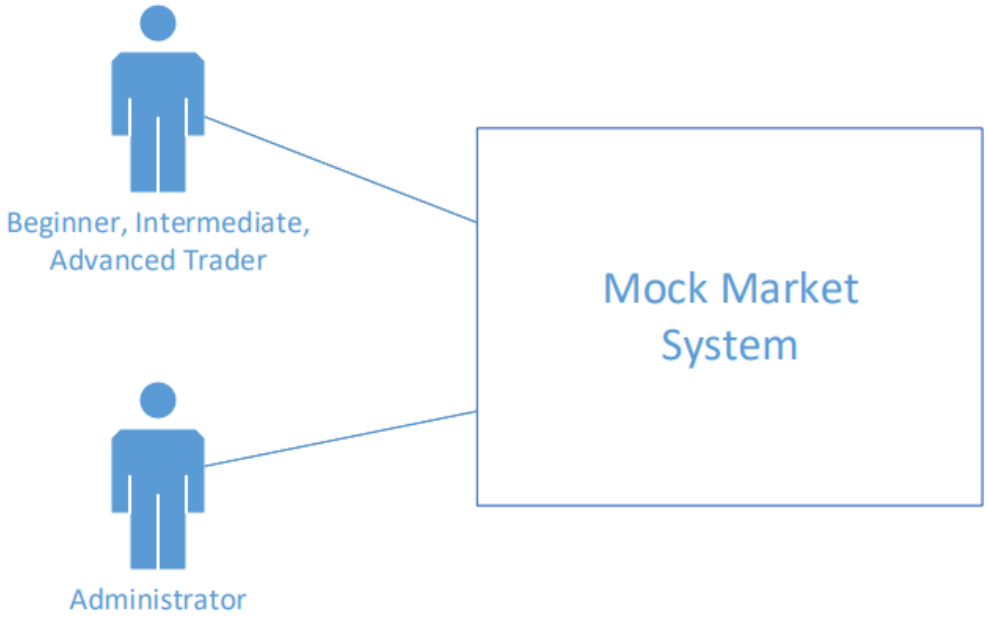
The general application utilizes a front-end and a back-end to separate the logic between the two. As a result, there are two applications that work in conjunction with the other. This decision was made because it is a common technique and would allow for easy application hosting with a separation of concerns. Further, both parts of the application are highly configurable, so migration to a different hosting service would be easier.

2. Definitions and Acronyms

Acronym / Definition	Explanation
API	Application Programming Interface. Used to allow to disparate systems to talk to each other.
CSS	A language used to style webpages.
Express.js	Node.js back end web application framework that helps building web applications and creating APIs.
HTML	Hyper-Text Markup Language. It is the standard language for webpages.
HTTP	Hyper-text transfer protocol. Used to communicate information across the web.
Javascript	A programming language used in conjunction with HTML and CSS.
JSON	A standard format for communicating data between two systems.
Node.js	Backend application that can help render webpages.
Postgres	Database management system used to persist application data.
React.js	Used to maintain application state and build large dynamic-based applications. Concerned an alternative for JQuery.
Redux	Application framework used in conjunction with React.js to create a global application state.

3. Context

Mock Market is a web application that helps beginners, intermediate, and advanced stock market traders trade on the market without the risk of monetary losses. Other applications may have this feature as an add-on, but it requires you to sign up and give a lot of personal information. Additionally, this application is more user friendly and its entire purpose is to place only theoretical trades – so the functionality is not buried behind something else. *Mock market* requires interaction with its back-end API and with APIs to interact and retrieve real-time stock market data in order to process trades.



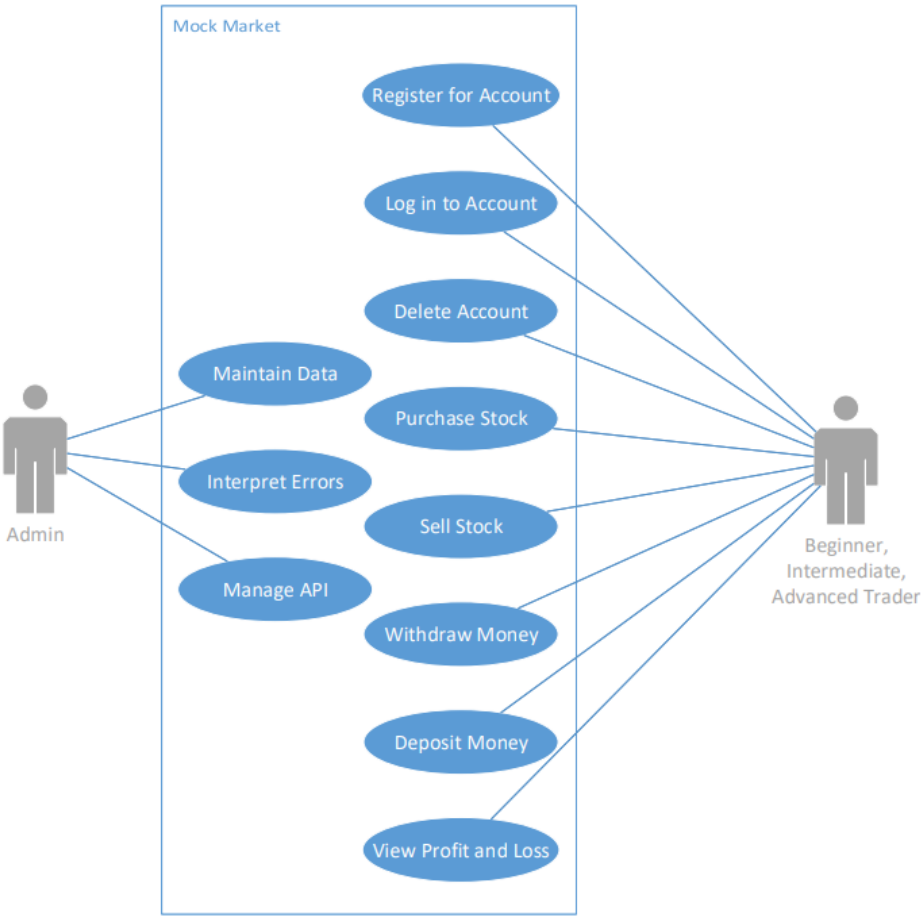
4. Constraints

- Only permits communication through HTTP GET, POST, and DELETE requests
- Relies on third party application APIs to retrieve current stock market prices and other meta information.
- Relies on a postgresql database. This is configurable, but no other databases are supported.
- Quotes are not necessarily in real-time so stock market information may be inaccurate.
- Front-end requires an internet connection to retrieve application state.
- API calls are static and can only be programmatically added.
- Only the back-end application has access to the APIs needed to retrieve stock market information.

5. Functional Overview

Most of the use cases are utilized by the “end-user” of the application. There are some other administrative functions that are not handled by them, however such as system logging. Presumably, the end-user (I.E. the stock market trader) will conduct trades including buying and selling of securities. As a result, they had to register and log in to an account to do so. They will also partake in withdrawing and depositing funds into their account in order to conduct trades within the application. If they desire, they will close their account finalizing their use cases.

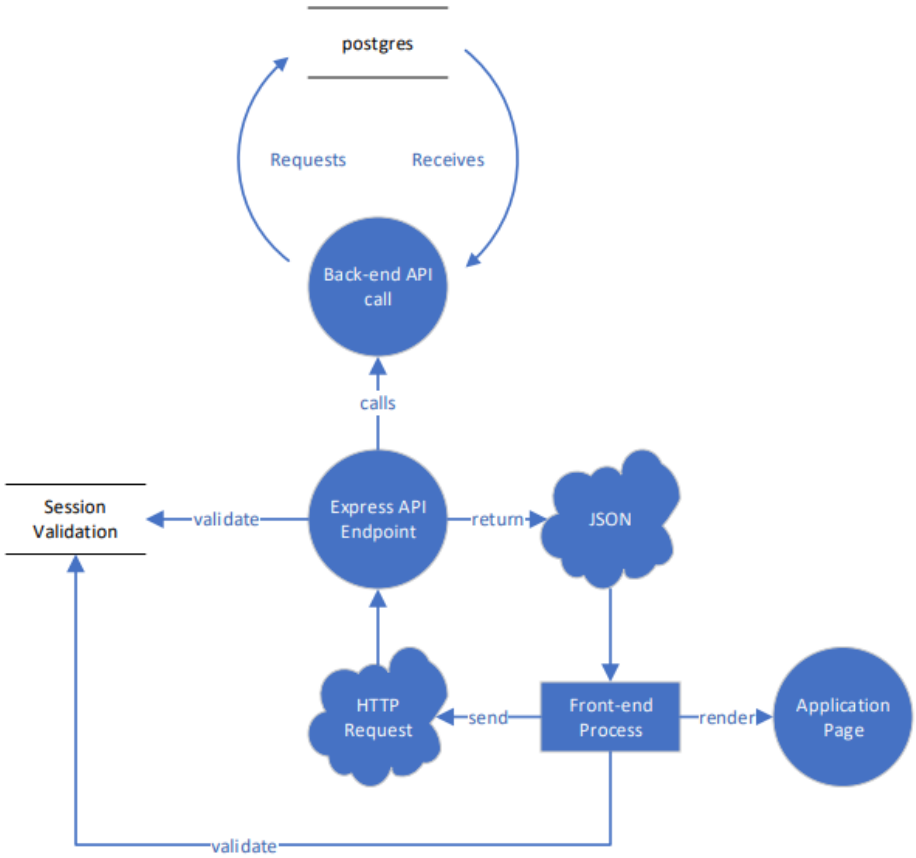
Additionally, the administrator is responsible for maintaining the data (and the associated data model), interpret the errors (as the application outputs them), and manage / adjust the API that the front-end is using. Here is a diagram demonstrating the relationship:



6. Data

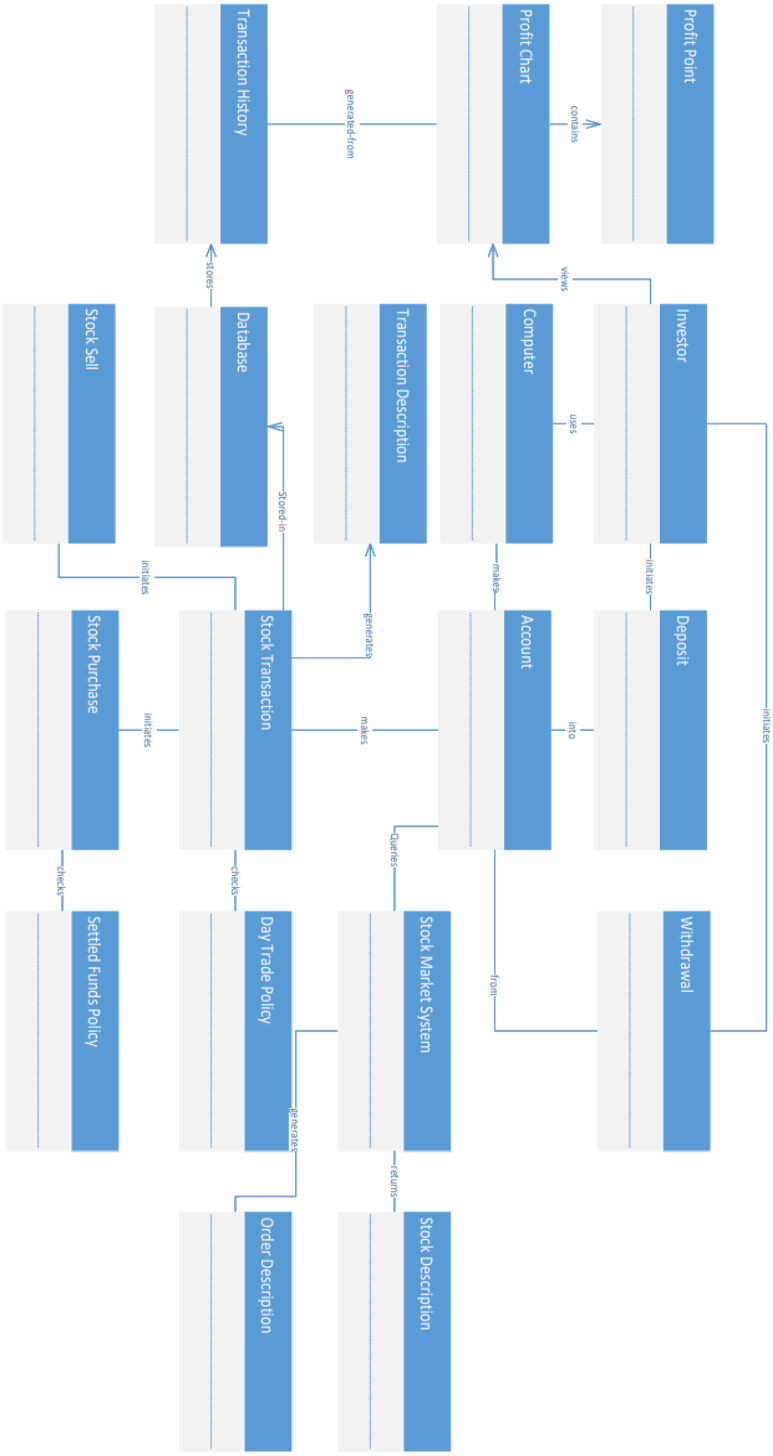
Mock Market has to call a variety of API services in order to retrieve information about the stock market. Not all information was available from a single API source, so two different sources were used. For the first source, I am using the *polygon.io* stock market API to return metadata about the stock market in relation to a particular stock. This API returns a good deal of information about the associated stock ticker, but it does not return the current price (which is obviously needed for the application). As a result, I use the *finnhub.io* stock market api to retrieve the most up-to-date price and other metadata of the stock. Used in conjunction, they both return an aggregate response that is sent to the front-end.

From there, the the back-end returns the response in the form of JSON to the front-end and is then interpreted. For the front-end to receive a response, it must send a request to the backend in the form of an HTTP request such as POST or GET. Additionally, parameters such as username must be provided for the back-end to formulate a response. Throughout all of this, a user session must be established for the back-end and front-end to function. If the user is not logged in, the application will not respond to requests to view webpages. All user-related data is stored in an associated postgresql database.

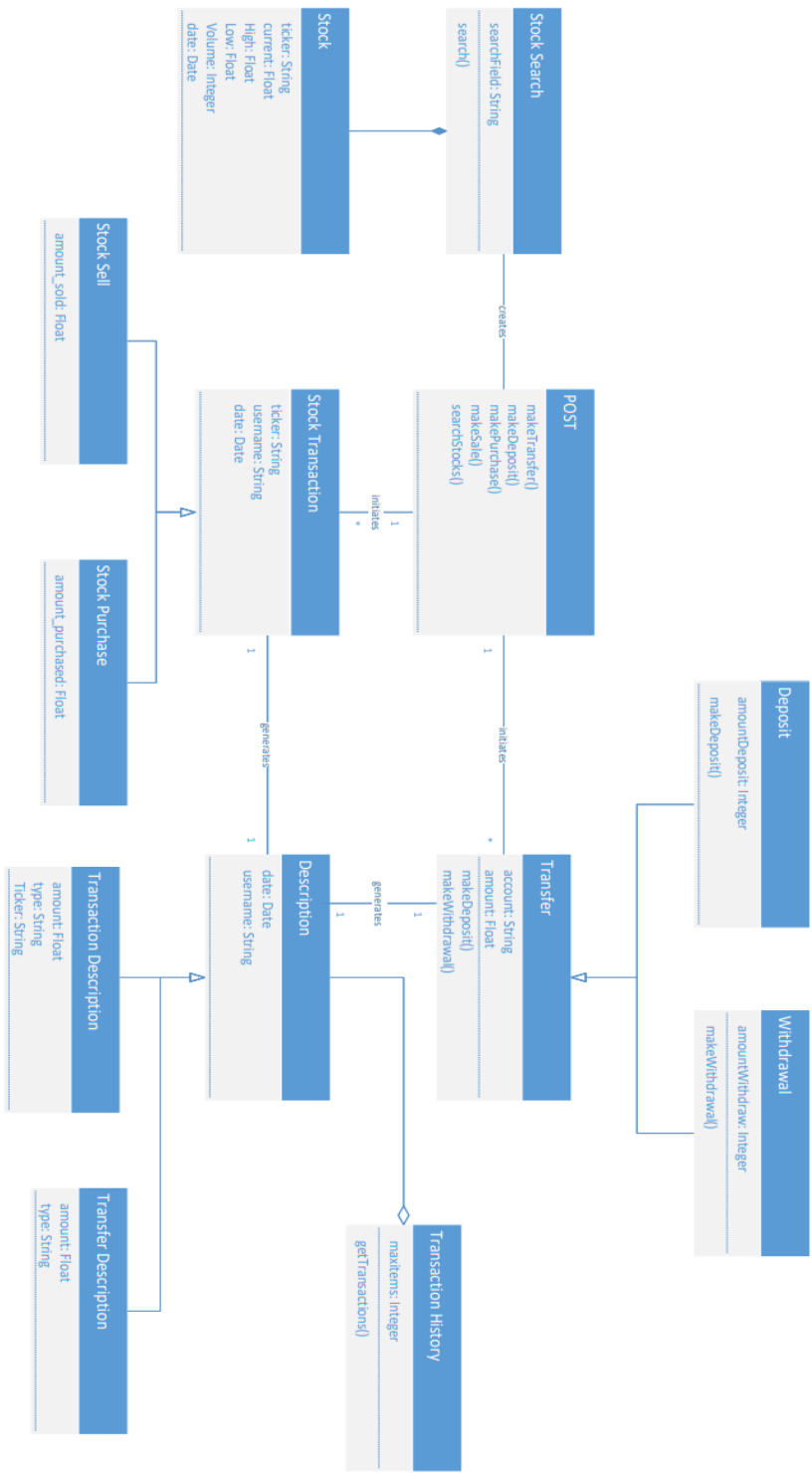


7. System Design and Architecture

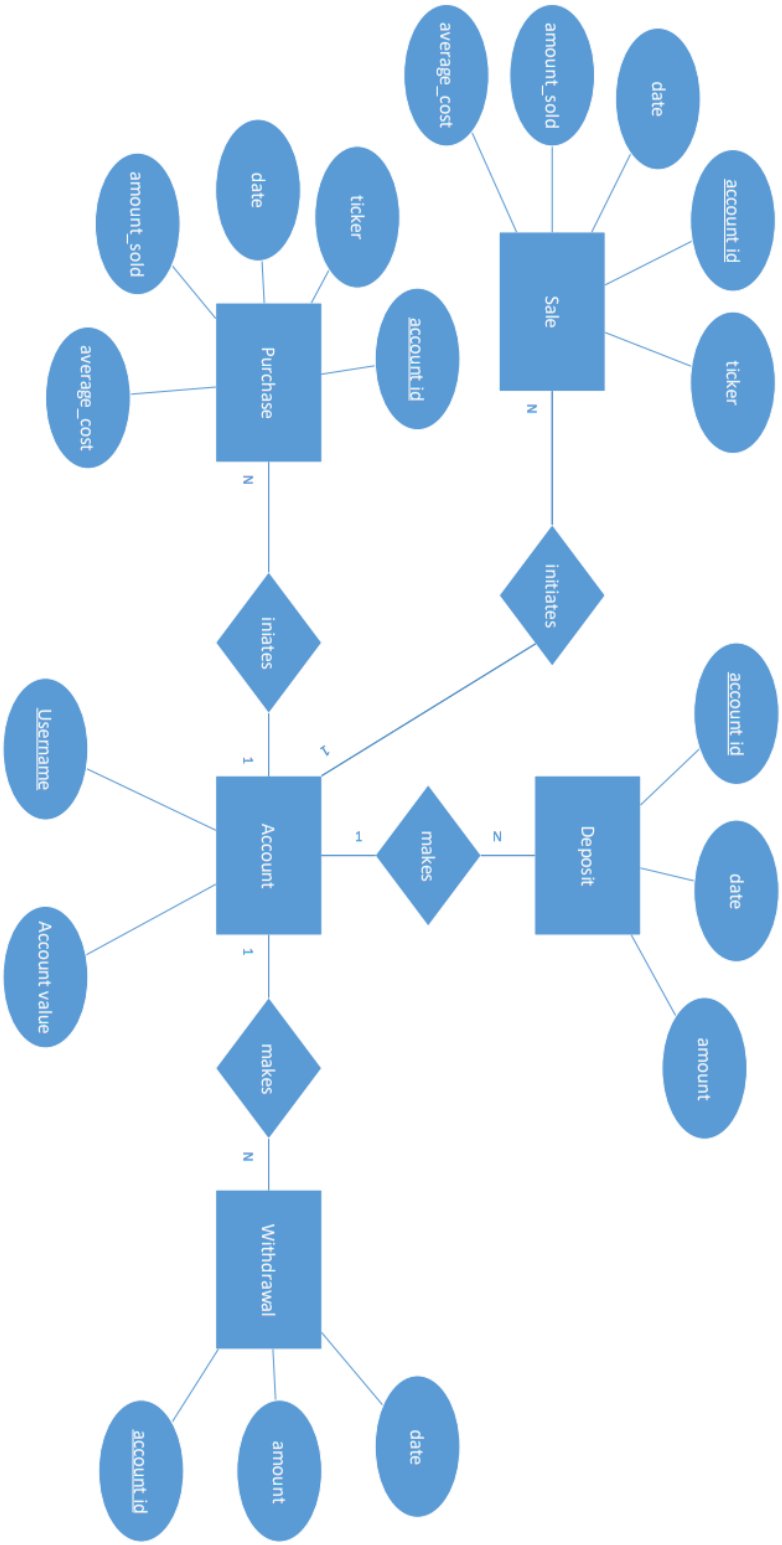
7.1. Domain Model

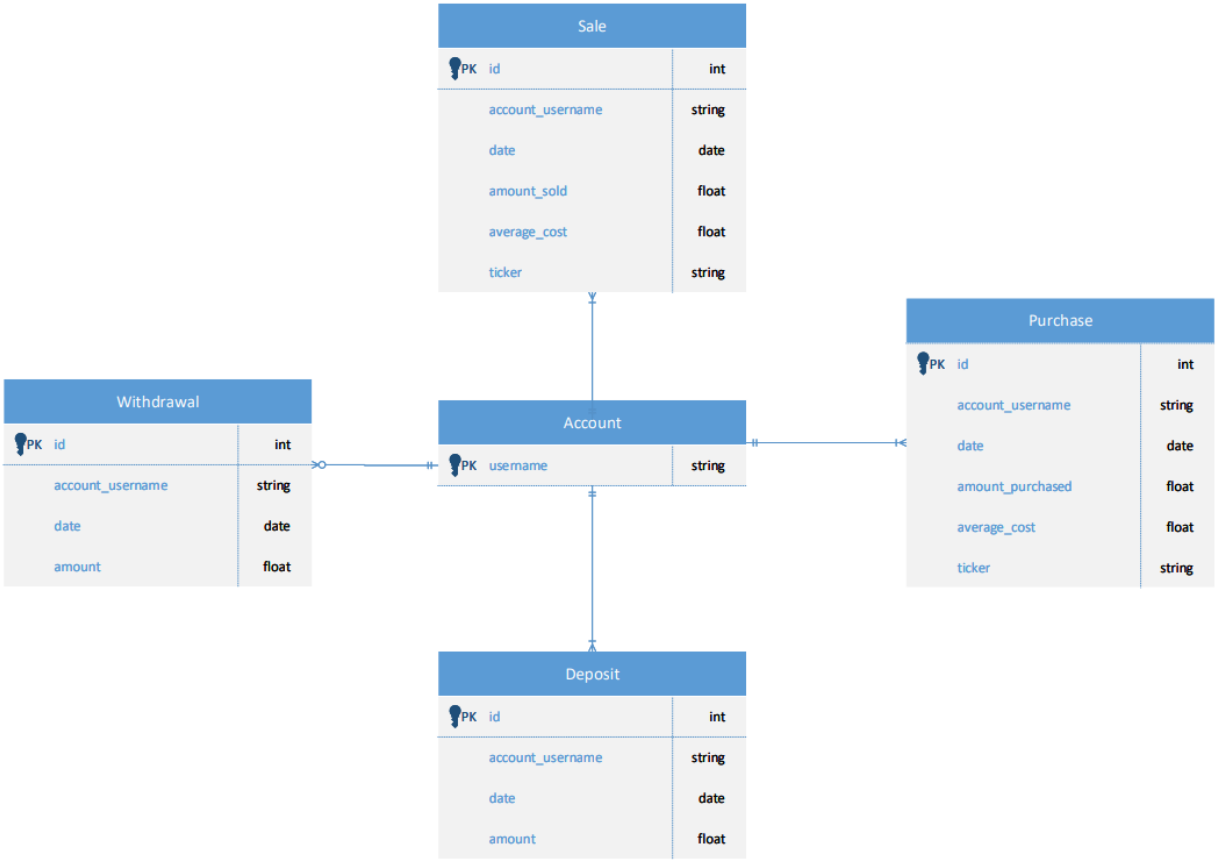


7.2. Class Diagram

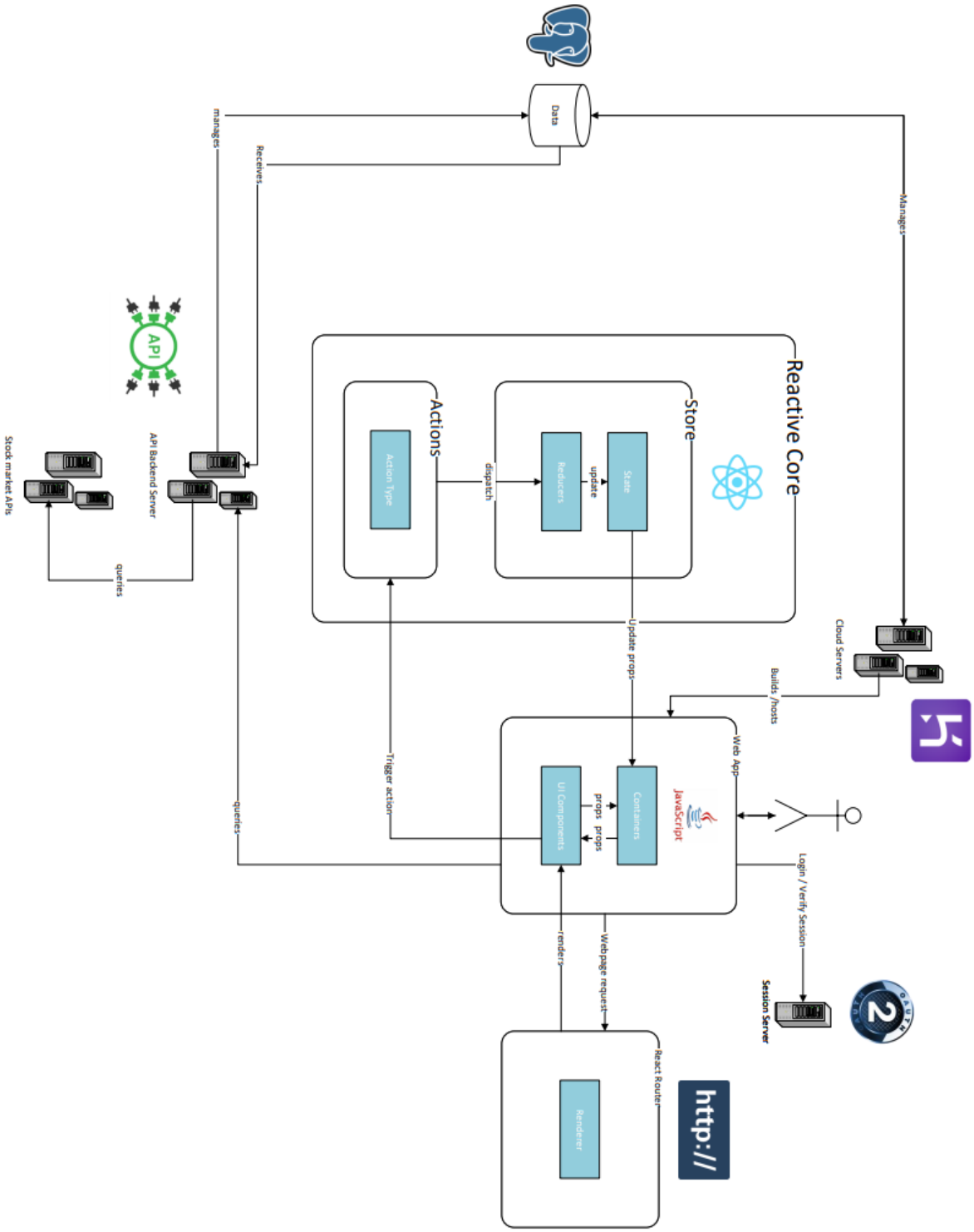


7.3. Data Model




Sale
PK id int
account_username string
date date
amount_sold float
average_cost float
ticker string
Purchase
PK id int
account_username string
date date
amount_purchased float
average_cost float
ticker string
Withdrawal
PK id int
account_username string
date date
amount float
Account
PK username string
Deposit
PK id int
account_username string
date date
amount float

7.4. High-level Architecture



On the front-end:
The architecture points to the fact that we are using React.js in conjunction with Javascript, HTML, and CSS. React.js coupled with Redux has an application state that is maintained and updated. As a controller, React.js updates the application state and creates a new rendering of the web page based on the new application state. To access the React.js application, the user must login and create a verified session. Once logged in, the React Router will direct and generate a webpage for the user based on the application state. The UI Components and Containers of the application are updated accordingly based on the application state. From the front-end, the user can initiate a state change and cause React.js to re-render the webpage.

On the back-end:
The back-end is used as an API server for the front-end. The front-end application queries the back-end server based on its API endpoint availability. The back-end processes the request through Express.js, parses the parameters sent to the server, and generates a JSON response back to the client. The back-end integrates with postgresql database for data persistence and retrieval. The information from the database is aggregated and returned to the front-end on an expected format that both sides can understand. The back-end also requires session validation in order to function and return a response. Finally, the back-end also integrates with various Stock Market APIs to simulate the purchase and sale of securities.

Additional Information:
Both the back-end and front-end are served remotely on the Heroku hosting platform. That way, they can be accessed through a public DNS. Additionally, the Heroku hosting platform has postgres availability and management (which is why it is defined in the architecture). This is important because a lot of the data persistence and reliability problems are handled by this platform. There are latency considerations, but it is reliable.

8. Fulfilling Nonfunctional Requirements

8.1. Usability

The system in general was trying to make the application experience more user-friendly, intuitive, and easy to understand while in development. Most importantly, it was important that it was easy and seamless for a user to accomplish a goal without much interruption from the application. I tried to make it so that the UI actions were clear and straight to the point, minimizing interruption to the user. There were a few factors that the architecture helps address under this nonfunctional requirement:

- React.js framework helps to generate a very responsive UI with algorithmic-intense applications.
- Reduce latency by integrating concurrency in with the application operations. This helps to mitigate wait times when the application is making a variety of API calls.
- The system includes a navbar header with "quick links" to all functions of the application including *Home*, *Search*, and *Transaction.* The user can easily navigate throughout the application without much thought.
- Minimalist design was implemented such that the user's short-term memory is not overburdened with unnecessary UI controls.

8.2. Security

The system accomplishes all security nonfunctional requirements by implementing architecture designed to handle a secure session during the use of the application. Overall, both the front-end and back-end applications use the verification of a secure session to operate. There were a few factors that the architecture helps address under this nonfunctional requirement:

- The use of OAuth for the front-end application to login to the application and receive data.
- The use of OAuth for the back-end application to interact with the front-end application for data responses.
- The hosting of the application on the Heroku platform that provides further security including trusted DNS resolution.
- Database hosting on the Heroku platform that provides security and reliability to the postgres database used by the application.
- All additional security benefits provided by OAuth and Heroku under the system architecture.

8.3. Availability

The system accomplishes all availability nonfunctional requirements through the hosting platforms used that are native to the public application interface. By hosting on Heroku,

there is more reliability for the application's availability. There were a few factors that the architecture helps address under this nonfunctional requirement:

- Heroku hosting provides availability and reliability to the postgres database. Additionally, replication is provided on the platform for postgres.
- The application is being hosted on a well-known platform that generally has better infrastructure and reliability than something in-house.
- The application provides recovery mechanisms for data loss and a bad internet connection – all of which are user-facing and informative.
- All user actions can be sent to the server hosted on Heroku after internet connectivity is re-established.
- The application could likely implement high-availability infrastructure to assist with a server that crashes.
- The application is setup so that it automatically is restarted if it crashes – allowing for a recovery over an unforeseen issue.
- Database issues are logged and documented for further analysis.

9. Algorithmic Component Information

For *Mock Market*, there were a variety of algorithmic components implemented. For instance, there was data aggregation for user profiles (including total withdrawals, deposits) – and their associated database entries. There was logic for account creation, registration, and deletion. On the market API side, there was logic to retrieve metadata about stocks and deliver that information in an expected format to the front-end. However, the bulk of the programming logic came from calculating the profit and loss of the associated account.

For some reason, calculating profit and loss turned out to be quite a difficult issue when implementing the application. There are methods of calculating stock profit or loss including FIFO (First-in-First-Out), LIFO (Last-in-Last-Out), and other averaging techniques that proved to be even more complicated. For this application, I opted to go with the FIFO approach to calculating stock profit and loss. That was, however, not the only component in calculating total account value.

FIFO Pseudo-Algorithm (Calculates previous profit):

```
For(all stocks purchased):
        receive: Aggregate of all purchases in the form of {ticker, bought_price}
        receive: Aggregate of all sales in the form of {ticker, sold_price}
        for(all buys starting from earliest):
                apply earliest sale to earliest stock buy
                if(earliest sale amount covers all of buy amount )
                        calculate profit on buy cost basis
                        continue to next earliest buy;
                        continue to next earliest sale;
                elif(earliest sale amount is less than buy amount)
                        calculate profit on buy cost basis
                        continue to next earliest sale;
                        continue with same buy;
                elif(earliest sale amount is more than buy amount)
                        calculate profit on buy cost basis
                        continue to next earliest buy;
                        continue with same sale;
```

Psuedo-Algorithm (Calculates current profit):

```
For(all stocks purchased):
        receive: current price for stock
        receive: All current buy items in the form of {ticker, bought_price}
        for(all buys starting from earliest)
```

get average cost and amount from buy item
add profit to total by taking (amount purchased) * (current price – average cost)

FIFI Algorithm Sample code:

```
for (const [item, itemActions] of Object.entries(map)) {

  // Seperate the type of actions so that they can be applied in a FIFO order.
  buyActions = [];
  sellActions = [];

  // Iterate through the item actions and seperate them by buy and sell.
  while(itemActions.length =/= 0) {
    let item = itemActions.pop();
    if(item.hasOwnProperty('amount_purchased')) {
      buyActions.push(item);
    } else {
      sellActions.push(item);
    }
  }

  // This calculates profit for already sold stocks
  let soldProfit = 0;
  while(sellActions.length =/= 0) {

    // To calculate profit for a given stock, we need the average price and amount purchased or sold for all actions.
    let buyActionBoughtAt = parseFloat(buyActions.slice(-1)[0].average_cost);
    let buyActionAmount = parseFloat(buyActions.slice(-1)[0].amount_purchased);

    let sellActionBoughtAt = parseFloat(sellActions.slice(-1)[0].average_cost);
    let sellActionAmount = parseFloat(sellActions.slice(-1)[0].amount_sold);

    // In this case, the first buyAction amount is greater than the sell that we are processing. Close out the sale as a result.
    if(buyActionAmount > sellActionAmount) {
      soldProfit = soldProfit + (sellActionAmount * (sellActionBoughtAt - buyActionBoughtAt));
      buyActions[buyActions.length - 1].amount_purchased = (buyActionAmount - sellActionAmount).toString();
      sellActions.pop();

    // In this case, the first buyAction amount is less than the sell amount. Because of this, we close out the buy and move on to the next.
    } else if (buyActionAmount < sellActionAmount) {
      soldProfit = soldProfit + (buyActionAmount * (sellActionBoughtAt - buyActionBoughtAt));
      sellActions[sellActions.length - 1].amount_sold = (sellActionAmount - buyActionAmount).toString();
      buyActions.pop();

    // In this case, the buy amount and sell amount are equal, so we close out both.
    } else {
      soldProfit = soldProfit + (sellActionAmount * (sellActionBoughtAt - buyActionBoughtAt));
      buyActions.pop();
      sellActions.pop();
    }
  }
}
```

Current Profit Sample Code

```
// This calculates the current profit with the actively purchased stocks.
let currentProfit = 0;
let currentPrice;

// Get the current price of the stock
currentPrice = await getCurrentPrice(item);

while(buyActions.length != 0) {
  let entry = buyActions.pop();
  currentProfit = currentProfit + (parseFloat(entry.amount_purchased) * (currentPrice.c - entry.average_cost));
}
```

For more observation of the algorithm, you can view it here. Search for the method named “calculateStockProfitOrLoss”, and you can view the implementation details further. This is the fundamental logic behind the application, and it is the most intense API endpoint call because of all the logic required. Upon further evaluation, caching could be used to mitigate the calculations required. For instance, there is no need to calculate previous profit or loss for an account if no new purchases have been made within that time frame.