ML-Stock Prediction

Report 2
Software Engineering

The Haystack Group *(Team):*
Alex Danielle Basden
Keita Sakurai
Curtis Hill
Taylor Davis
Lisa Thoms
Brian Luing

Website / GitHub thepressq.com
(Hidden Currently) GitHub
incoming:

All Team Members contributed equally to this Report.

Table of Contents

System Diagrams

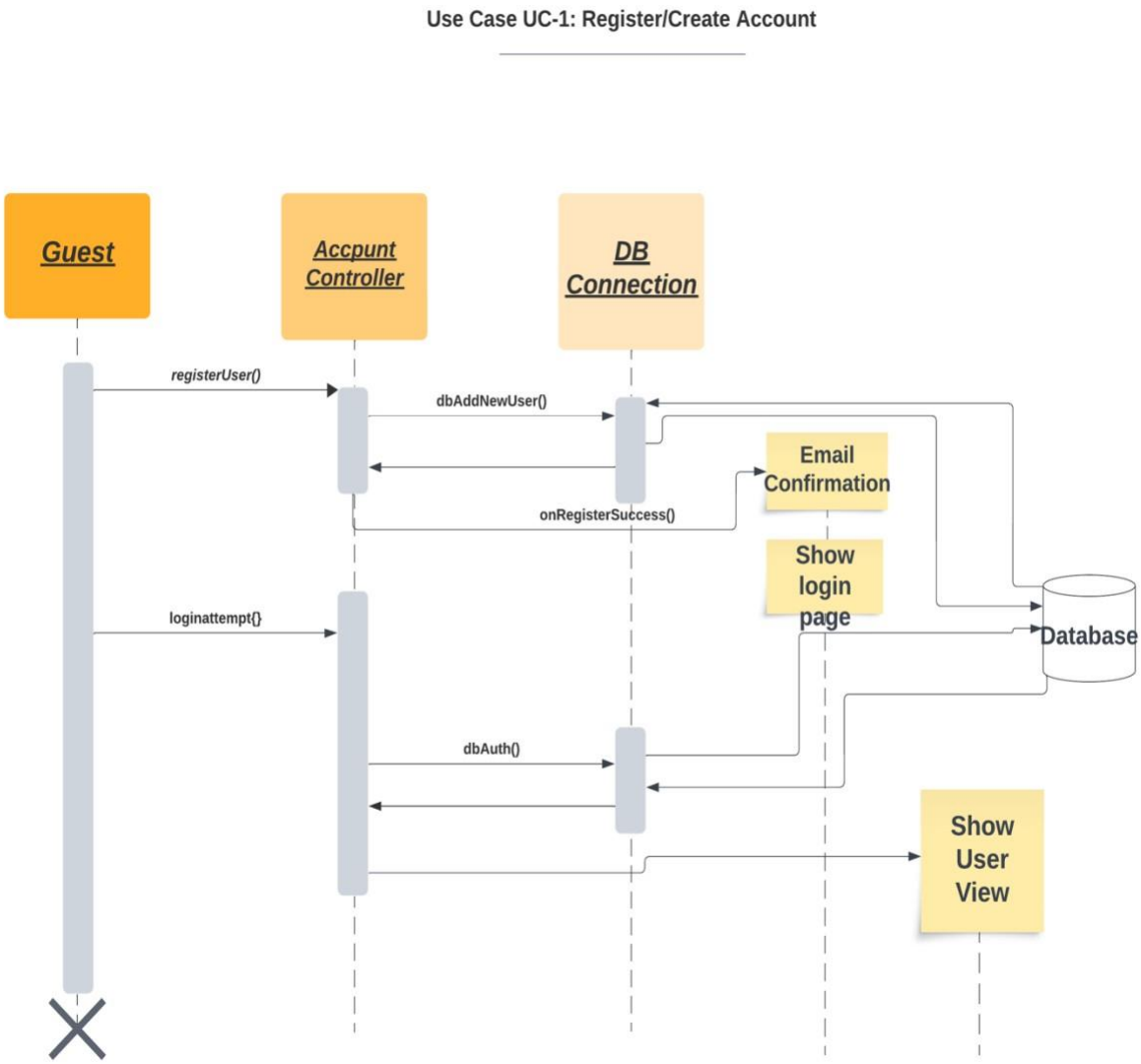
a. Introduction

The interaction diagrams presented in the following section provide a comprehensive overview of system interactions within our software. These diagrams illustrate the vital roles played by the Yahoo Finance API, database systems, and controllers in various scenarios. Given the web-based nature of our application, the database and controller components have become particularly prominent, ensuring the seamless functioning of the software. Users' interactions with the system, including logging in and accessing real-time stock price data from the Yahoo Finance API, are critical for maintaining up-to-date information. This interplay of components is essential for the success of our application.

b. Diagrams

Use Case 1: Register/Create Account

The sequence diagram for Use Case 1 (UC-1) provides two options for a Guest. The Guest can either choose to log in or register a new account. If the Guest opts to register a new account, the information is sent to the Account Controller. The Account Controller will then attempt to verify the absence of duplicate login information in the database using the DB Connection module. If no duplicates are found, the new user's information is stored in the database. Subsequently, a confirmation email is sent to the user, and the Login View is updated.
If a user chooses to log in, the Account Controller endeavors to authenticate the provided login details by cross-referencing them with the database via the DB Connection module. If the details match correctly, the Account Controller transitions the user into investor mode, presenting them with the User View.

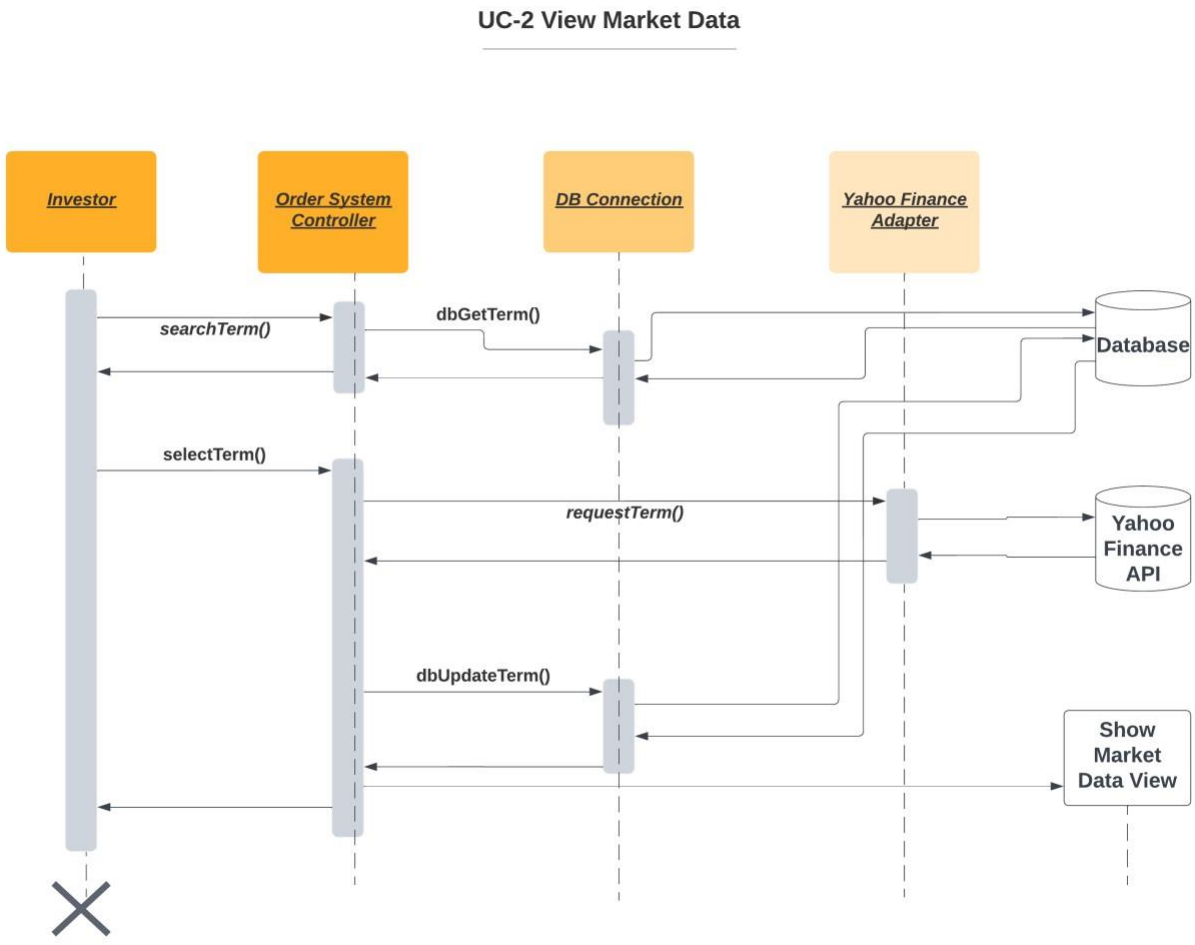


UC-1 Sequence Diagram:

1. **Single Responsibility Principle (SRP)**: Each object in the sequence diagram has a specific responsibility. For example, the Browser handles user input, the System checks the Database for user information and handles registration, and the Database System is responsible for storing and retrieving user data.
2. **Open-Closed Principle (OCP)**: The System is open for extension (can register different types of users) but closed for modification (the registration process remains consistent

Use Case 2: View Market Data

To access market data, an investor begins by conducting a search using a specific term, typically a company name or stock symbol. The Order System Controller is responsible for processing this term. It retrieves potential matches from the database through the DB Connection module and presents them to the user. The investor selects a match from the provided options. The Order System Controller then takes the chosen term and sends a request for its data to the Yahoo! Finance API via the Yahoo! Finance Adapter. Following this, the Order System Controller updates the database using the DB Connection module to store data for this term. Finally, the system proceeds to display the Market Data View, now populated with the selected market data.

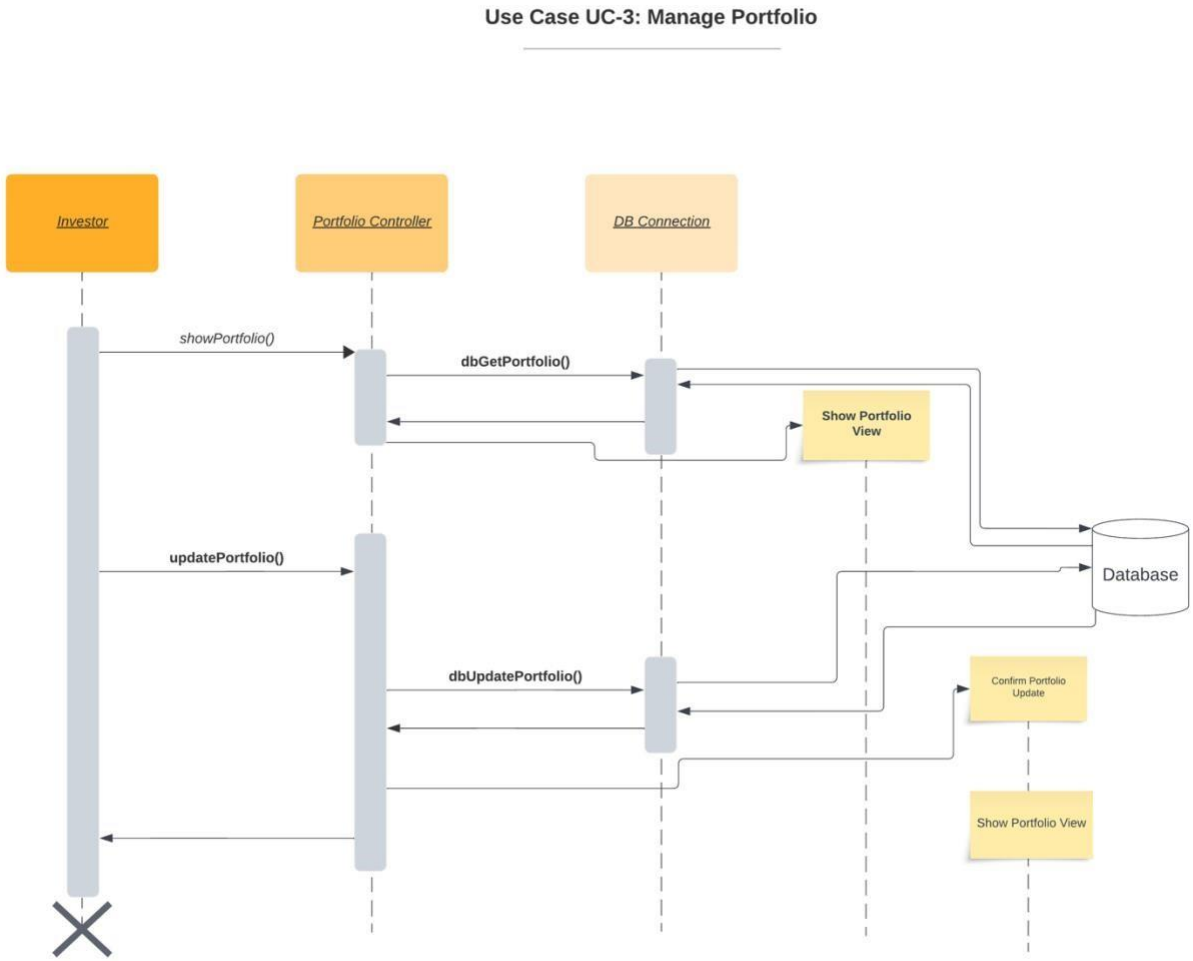


UC-2 Sequence Diagram:

1. **Single Responsibility Principle (SRP)**: Each object has a clear and single responsibility. The Investor searches for a stock, the System interacts with the Database and Yahoo! Finance API, and the Database System and Yahoo! Finance API return relevant information.
2. **Dependency Inversion Principle (DIP)**: The System doesn't depend directly on low-level modules. Instead, it relies on abstractions (Database System and Yahoo! Finance API) to retrieve information, allowing for flexibility and future updates without changing the System.

Use Case 3: Manage Portfolio

Investors should have the capability to access and modify their Portfolio View. When a user clicks to view their portfolio, the Portfolio Controller will retrieve the investor's portfolio stocks from the database using the DB Connection module. Additionally, the investor has the option to update and customize their portfolio view and make changes to various settings.

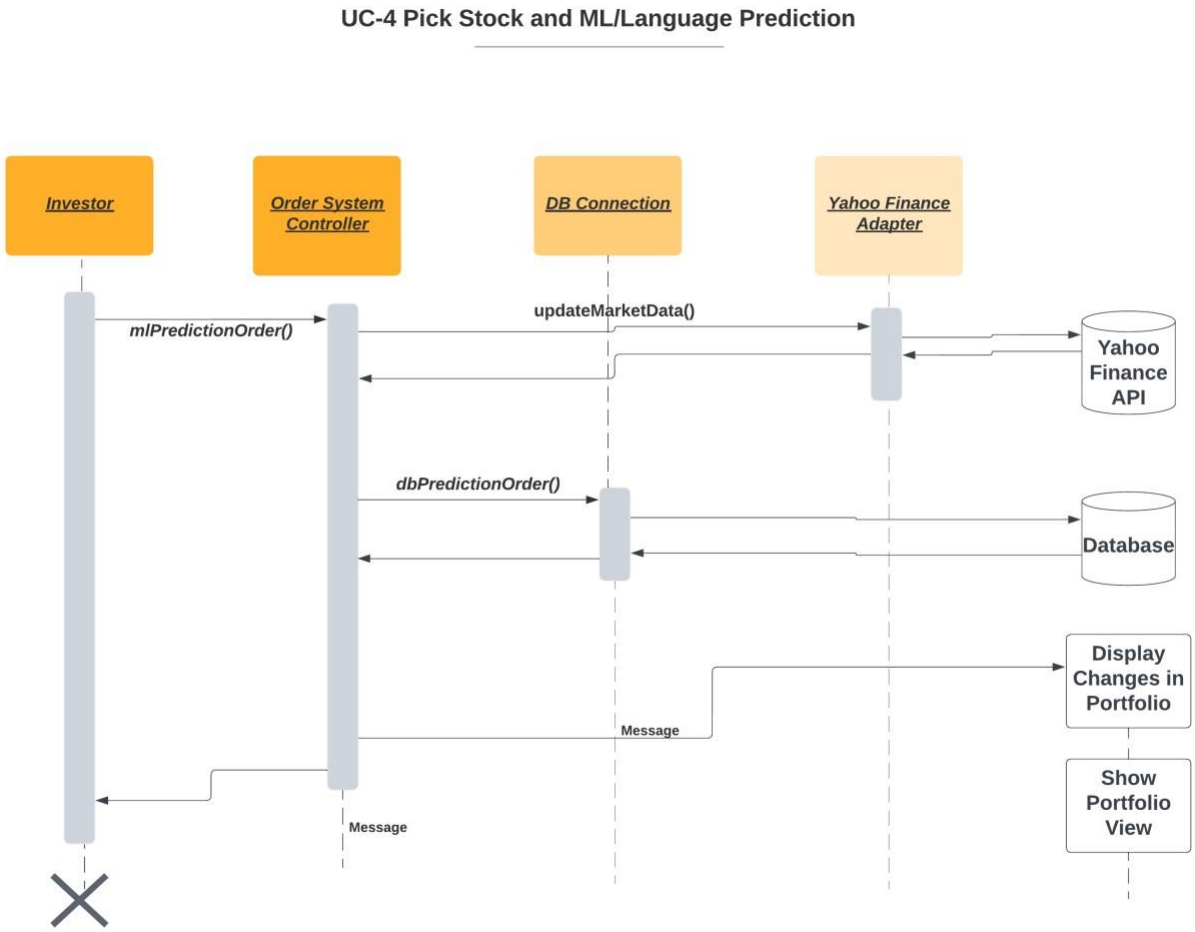


UC-3 Sequence Diagram:

1. **Single Responsibility Principle (SRP)**: Each object has a clear responsibility. The Investor requests to view the portfolio, the System interacts with the Database and Yahoo! Finance, and the Database System and Yahoo! Finance provide the required data.
2. **Separation of Concerns (SoC)**: The responsibilities of each object are clearly separated. The Investor is responsible for viewing the portfolio, the System handles interactions, and the Database System and Yahoo! Finance handle data retrieval.

Use Case 4: Pick Stock and ML/Language Prediction

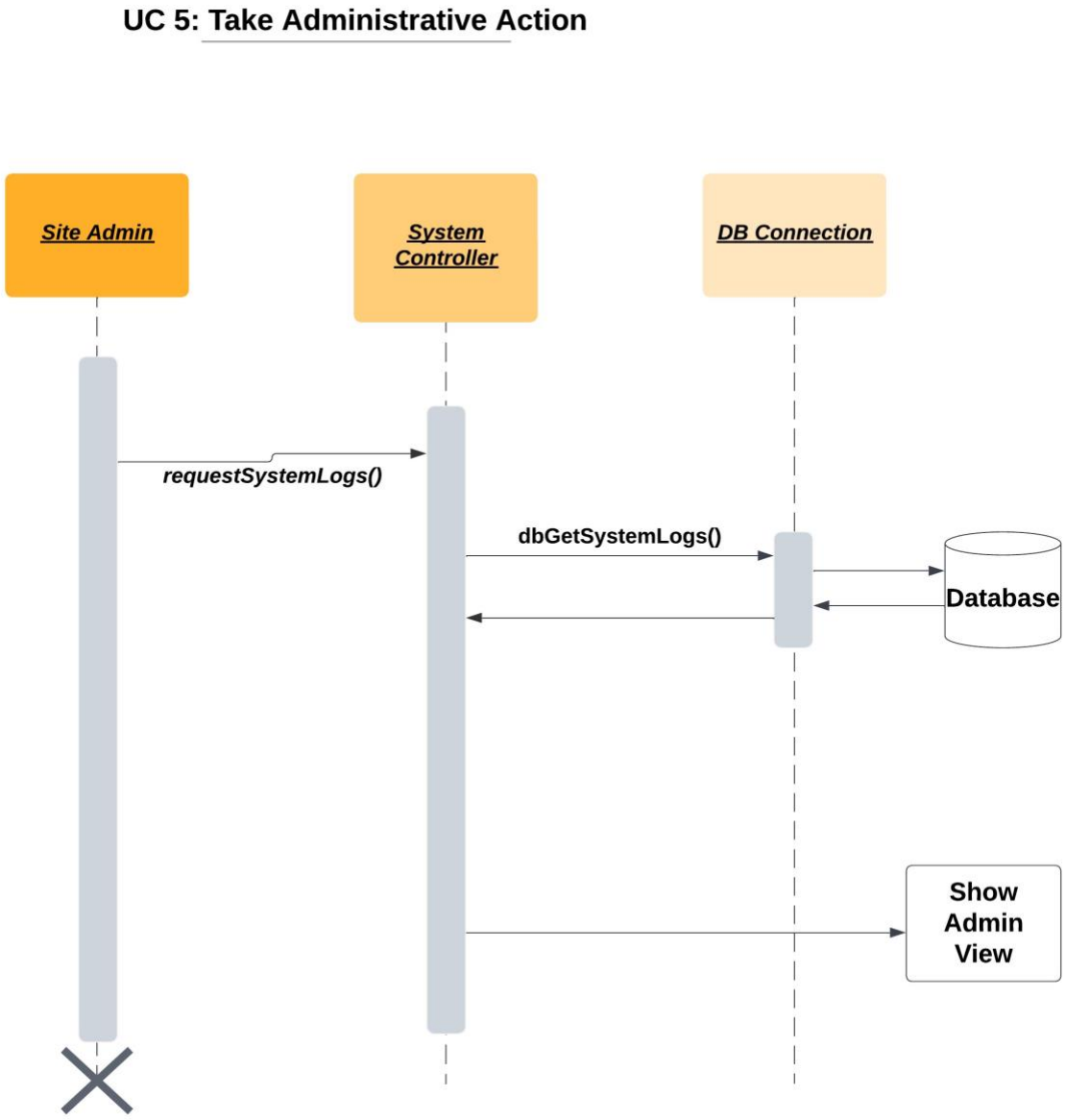
Investors must have the capability to initiate market orders. When an investor places an order, the Order System Controller promptly contacts the Yahoo! Finance API via the Yahoo! Finance Adapter to retrieve the current stock price. Once the current price is obtained, the Order System Controller proceeds to verify with the database through the DB Connection module whether the user has adequate funds for a buy order or enough of the stock for a sell order. Additionally, the Order System Controller can consult the ML Stock Prediction module to consider market predictions and inform the trading decision. After the trade is confirmed, pertinent information is stored in the database via the DB Connection module. Any resulting changes are then reflected in the investor's portfolio to provide an up-to-date view of their holdings.



UC-4 Sequence Diagram:

1. **Single Responsibility Principle (SRP)**: Each object has a specific responsibility. The Investor selects a stock, the System interacts with Yahoo! Finance, records the order in the Database, and displays changes to the Investor.
2. **Liskov Substitution Principle (LSP)**: The Yahoo! Finance API can be substituted with another source of stock data without affecting the behavior of the System.

Use Case 5: Take Administrative Action

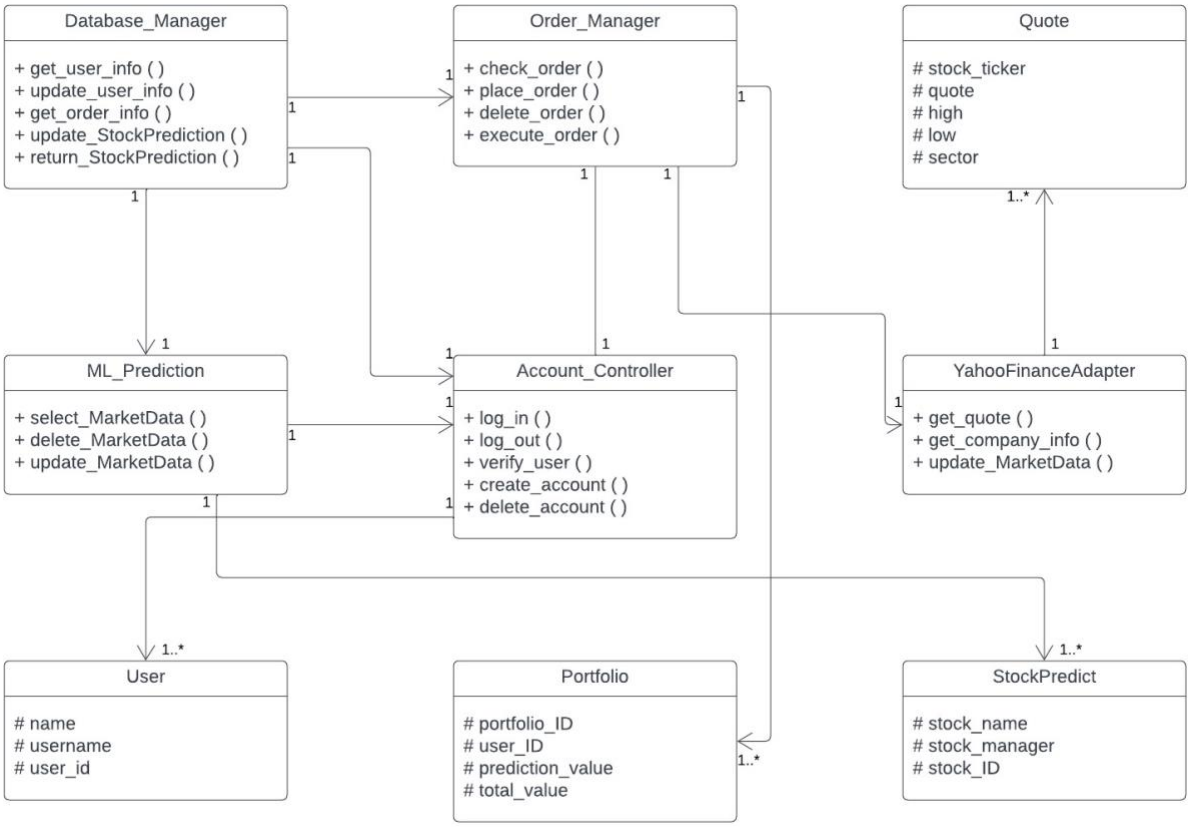


UC-5: Sequence Diagram:

1. **Single Responsibility Principle (SRP)**: The Site Administrator requests logs, and the System saves and returns the logs. Each object has a clear and single responsibility.
2. **Separation of Concerns (SoC)**: The Site Administrator's actions are separate from the System's actions, maintaining a clean separation of concerns.

Class Diagrams and Interface Specifications

a. Class Diagram



b. Class Data Types and Operation Signatures

Here are the class data types and operation signatures for the classes you provided:

Database_Manager
- Class Data Type: `Database_Manager` - Operation Signatures:
- `get_user_info(user_id: int): UserInfo`: Retrieves user information for the specified `user_id`.
- `update_user_info(user_id: int, new_info: UserInfo): void`: Updates user information for the given `user_id` with the provided `new_info`.
- `get_order_info(user_id: int): OrderInfo[]`: Retrieves order information for the specified `user_id`.
- `update_StockPrediction(symbol: string, prediction: StockPrediction): void`: Updates the stock prediction for a specific `symbol` with the provided `prediction`.
- `return_StockPrediction(symbol: string): StockPrediction`: Retrieves the stock prediction for the specified `symbol`.

Order_Manager
- Class Data Type: `Order_Manager` - Operation Signatures:
- `check_order(order_id: int): OrderInfo`: Checks and retrieves order information based on the provided `order_id`.
- `place_order(order: OrderInfo): void`: Places a new order using the provided `order` data.
- `delete_order(order_id: int): void`: Deletes an order with the specified `order_id`.
- `execute_order(order_id: int): void`: Executes an order with the given `order_id`.

ML_Prediction
- Class Data Type: `ML_Prediction` - Operation Signatures:
- `select_MarketData(data: MarketData): void`: Selects and stores market data for prediction.
- `delete_MarketData(data_id: int): void`: Deletes market data with the specified `data_id`.
- `update_MarketData(data_id: int, new_data: MarketData): void`: Updates existing market data with the provided `new_data`.

Account Controller
- Class Data Type: `Account Controller` - Operation Signatures:
- `log_in(username: string, password: string): User`: Logs in a user with the provided `username` and `password`.

- `log_out(user: User): void`: Logs out the specified `user`.
- `create_account(user_info: UserInfo): User`: Creates a new user account with the provided `user_info`.
- `delete_account(user: User): void`: Deletes the user's account.

YahooFinanceAdapter

- Class Data Type: `YahooFinanceAdapter` - Operation Signatures:
- `get_quote(symbol: string): Quote`: Retrieves a quote for a specific stock `symbol`.
- `get_company_info(symbol: string): CompanyInfo`: Retrieves company information for the specified stock `symbol`.
- `update_MarketData(data: MarketData): void`: Updates market data using the provided `data`.

User
- Class Data Type: `User` - Attributes:
- `name: string`
- `username: string`
- `user_id: int`

Portfolio
- Class Data Type: `Portfolio` - Attributes:
- `portfolio_ID: int`
- `user_ID: int`
- `prediction_value: float`
- `total_value: float`

StockPrediction
- Class Data Type: `StockPrediction` - Attributes:
- `stock_name: string`
- `stock_manager: string`
- `stock_ID: int`

Quote
- Class Data Type: `Quote` - Attributes:
- `stock_ticker: string`
- `quote: float`
- `high: float`
- `low: float`
- `sector: string`

It's important to keep in mind that the class data types and operation signatures presented in this diagram are simplified. The precise method signatures and attributes could differ based on your specific needs and the programming language you're using. Furthermore, certain operations and data types have been excluded for the sake of brevity, so you may find it necessary to include additional methods and attributes to suit your application's requirements.

System Architecture and System Design

a. Architectural Styles

To optimize the efficiency of our software, we will integrate several established software tools and principles into our design. We will delve into the details of the following architectural types to not only encompass their general functionalities but also to reflect their role in the overall software. These architectural systems, which are derived from the software's specific requirements, are essential for the software's success. Our architectural approach will include, and may potentially be further elaborated in the future, the following components: Model-View-Controller (MVC), Data-Centric Design, Client-Server Access, and RESTful Design. Each of these architectures will contribute to the overall functionality of the software, playing a crucial role in achieving the desired results.

Model-View Controller

The Model View Controller (MVC) is a user interface implementation approach that divides the software into three distinct components: the model, view, and controller. The view primarily deals with user interface-specific output, such as displaying stock information on a webpage. The model serves as the central component of the MVC approach, housing all the data, functions, and tools. The controller, on the other hand, takes user input and translates it into commands for either the model or the view.

MVC is well-suited for our software because it enables us to break down the design into smaller, manageable sub-problems. This division allows us to separate user interface-related functions from database operations, with the controller acting as the intermediary. Consequently, the design becomes more modular and programming becomes more straightforward, enhancing overall design fluidity.

Data-Centric Design

Data forms the essential foundation of the Haystack Group. Our database will house a wealth of data crucial for all aspects of the software. This database will not only store data retrieved from the Yahoo! Finance API but, more significantly, user-specific information. Each time a user logs in, they should have seamless access to their personalized dataset, which encompasses their complete portfolio, predictions, and settings, among other elements. Furthermore, it's imperative that

this data is organized in a manner that allows multiple subsystems to access it as needed. This approach keeps the data-specific aspects of the software abstract and readily accessible, promoting efficient interaction across various components.

Client-Server Access

Users will engage with the software interface regularly, and these interactions are fundamentally structured on a client-server basis. The user, being the primary client, consistently interacts with other subsystems. The user's interaction extends to accessing all the infrastructure offered by the Haystack Group, facilitating seamless communication between the various components of the MVC architecture and between the client and infrastructure.

Moreover, the infrastructure provided by the Haystack Group will have the capability to access infrastructure from non-associated systems. This ensures that our software can interact with external systems effectively.

Representational State Transfer

As a software implementing a client-server access system, the use of a REST system is inherently implied. RESTful design principles dictate that in addition to a client-server access system, the software should have components that can scale, maintain a uniform interface, be stateless, and support caching. This approach allows for the development of a smooth, modular codebase. Following the interface specifications outlined by REST ensures streamlined interactions for both users and designers. Users will clearly understand their actions when, for example, they click on a link within a web page. The request is then converted and sent to the controller.

It's important to note that the RESTful implementation can be applied at multiple levels. This system will be designed to work seamlessly on Android and iOS platforms in addition to standard web interfaces. This ensures a consistent user experience across all mediums, whether a user places an order on their mobile device or online. Utilizing the RESTful system will greatly facilitate this transition between different platforms.

b. Identify Subsystems

The Haystack Group is committed to deploying its platform across multiple interfaces. Consequently, the identification of subsystems is a crucial part of the initial software analysis. At a high level, our platform consists of a front-end system and a back-end system. However, when delving deeper, it becomes evident that each of these subsystems can be further broken down into more granular components.

Front-end systems primarily encompass user interfaces and object interactions with users. On this layer, we find the user interface responsible for displaying views and specific data to users on various platforms. This includes different implementations and mappings for iOS, Android, and native web applications. The front-end system is required to maintain continuous communication with the back-end system to ensure data consistency and regular data retrieval. It must effectively transmit information from user commands to the back end.

On the other hand, the back-end system is responsible for retrieving necessary data and information, and then returning this data to the user interface for the display of requested pages or information. This system also handles the database schema, implementation, and interactions with relevant hardware. It encompasses nonassociated elements that are vital to the overall success of our system.

The back-end system holds paramount importance within our infrastructure. As we adhere to the MVC framework, we further dissect the back end into distinct controller and database subsystems, in addition to the financial retrieval and queuing systems as previously outlined. This division allocates the bulk of command processing to our back-end subsystem. Not only does the back-end system need to foster internal communication among its own subsystems, but it must also maintain seamless interaction with the front-end UI system to respond to user commands and extend its communication to non-associated systems.

Upon closer examination, we emphasize the significance of the financial retrieval system and the queuing system. The financial retrieval system will interface with Yahoo! Finance to obtain relevant information as directed by the controller, triggered by user input from the front end. The queuing system will manage various processes, facilitating communication with non-associated systems. It plays a pivotal role in queuing and overseeing all back-end processes, ensuring that the

correct commands are processed at the appropriate times. The success of these modules, the overall efficacy of the back-end system, and the efficiency of communication among the subsystems collectively underpin the software's overall success.

c. Mapping Hardware to Subsystems

The Haystack Group platform utilizes a MySQL database server, which is hosted on a single machine. However, the system spans across multiple machines. The system is compartmentalized into two distinct sections: a front-end that operates within the client's preferred web browser and a back-end component that resides on the server side, interacting with the database.

The front-end serves as the primary graphical user interface (GUI) connecting the system with the client. It manages communication between the GUI and the database, facilitating tasks such as confirming market orders and updating investors' portfolios. Any modifications made in the front-end are seamlessly reflected in the back end of the server. The back end is responsible for executing market orders correctly and keeping users informed about their transactions.

d. Persistent Data Storage

Data storage lies at the heart of the Haystack Group's operational plan. Given the heavy reliance on well-maintained and up-to-date data in our software, it is paramount that our database schema accurately represents all relevant objects consistently. This means that the data must perpetually and precisely reflect various aspects, including user data, stock information, ticker variables, settings, achievements, leaderboards, and other pertinent objects.

The Haystack Group will heavily rely on the MySQL relational database. Relational databases align well with the specific requirements of this software. They comprise multiple indexed tables populated with diverse object attributes, as depicted in the class diagrams on the preceding page. This structure is essential due to the large volume of objects within the software. Tables will not only cater to user data and settings, such as login and profiles, but also house information related to stocks and portfolios. Furthermore, these databases must be regularly updated to maintain accurate and up-to-date information. Components like MLStock Prediction and data visible in each user's portfolio need to consistently reflect precise data.

Data retrieval involves querying the respective database tables in response to user commands. When a user initiates a request to access data, a query is executed, and the table is searched to retrieve the appropriate data value(s). For example, if a user wishes to access their settings, a query is used to extract the currently selected settings, which are then presented to the user via the user interface.

If a user decides to make changes to their settings, these modifications are sent back to the database, updated, and saved for future access. This same process is replicated and applied to all aspects of the software, utilizing multiple tables, as outlined in the diagrams above. The software's effectiveness relies on the accurate and up-to-date values always returned from the database. Consequently, the database must regularly receive data updates from the Yahoo! Finance API to ensure that data is constantly refreshed and reflective of the most recent information when queries are executed.

This continuous update process ensures that users have access to consistently accurate information, including their portfolio performance, ML-prediction statistics, stock tickers, and recent selections throughout the site. This approach distinguishes the Haystack Group's software from others and ensures the fulfillment of its core objectives and requirements with efficiency.

e. Network Protocol

As is customary for software of this kind, the Haystack Group will employ the standard Hypertext Transfer Protocol (HTTP). HTTP operates by structuring text and using hyperlinks to facilitate text-based communication between nodes. While this protocol is not unique to our situation, it is worth emphasizing that it will be the primary method of communication between users and the software interface. Furthermore, the HTTP protocol will be utilized not only on web-based devices but also on Android and iOS devices. Users, from any of these platforms, will have access to various webpages and links on the Haystack Group's website. They can retrieve all pertinent stock, portfolio, and related information through these webpages using the HTTP protocol.

f. Global Control Flow

Execution Order:

The Haystack Group primarily employs an event-driven approach in its system implementation. Almost all system features require activation by some entity, whether it's the user themselves or another component of the system. Primarily, the user end of the system is responsible for driving most of the event-driven characteristics. Many functions, such as stock trading, portfolio management, and ML-based stock prediction, can only be initiated by the user. Nonetheless, there are some event-driven functions initiated by the system itself. After a user submits an order, it undergoes processing and is added to the database. Subsequently, the system triggers the order verification process and leverages the Yahoo Finance API to retrieve stock market information, obtain a quote, and ultimately execute the order.

Certain functionalities must be executed in a specific sequence. Prior to commencing their investment journey, users must complete the following steps:

1. Registration/Account Creation: Every user is required to register on our website before you can get results from ML-based stock predictions.
2. Users must be able to view each stock's price history on the User Interface and be ready to start investing.

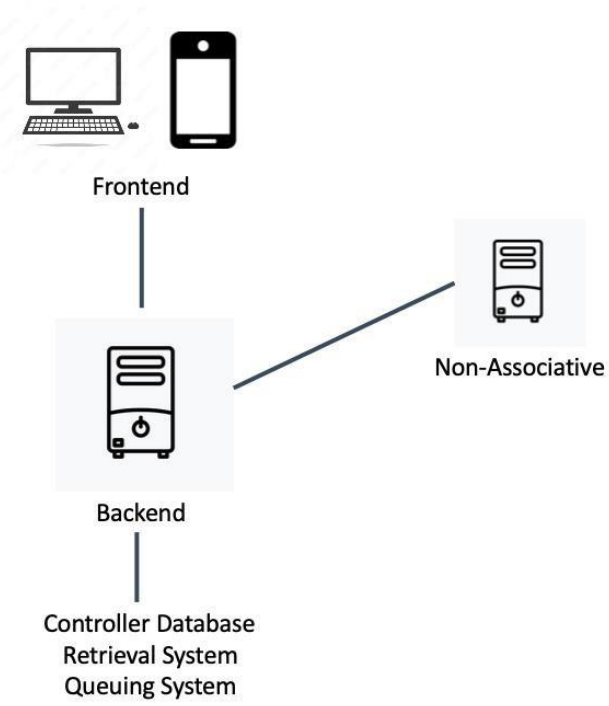


Figure 3.1: Diagram of the network protocol.

Time Dependency

Generally, at the Haystack Group, the system primarily operates in real-time, with certain components unaffected by time. The real-time functionality is closely tied to the stock market, which itself adheres to specific operating hours. While users are browsing the website, real-time timers are utilized to facilitate the processing of incoming information.

– Stock Market Opening and Closing: The stock market follows specific time intervals, differentiating between its opening and closing times.

– Queuing System: User-placed orders are queued, and depending on market conditions, this may exert substantial server loads. To maintain balanced server performance, efficient order splitting is essential. The timer in this system monitors unexecuted or outstanding orders and manages their processing.

Concurrency

Within our main system, there are subsystems that require meticulous consideration concerning concurrency. The most significant among these is the queuing subsystem, which presents a concurrency challenge. Our primary concern is to ensure that only one order is inserted into the queue at any given time. Additionally, we must guarantee that only one order is dequeued from a specific queue. Other than these aspects, our system currently does not demand synchronization. It's important to note that this requirement may evolve as we continue implementing our system.

g. Hardware Requirements

The hardware requirements for ML-Stock Prediction are heavily focused on the server side and are the main contributions to the operation of the application. This leaves the client-side with minimal hardware requirements to operate the application. With the hardware requirements focused primarily on the server side, the only requirement on the client side is an internet connection with the capability to run a modern web browser.

Internet Connection

For ML-Stock Prediction to run and use any of its core functions (such as updating user portfolios, retrieving stock information, tracking administrative actions, etc.), an internet connection will be required. As most of the data being transferred is text-based (executable instructions), a low band of frequency is required. At the time of writing, a complete scalable analysis has not been performed on the system, so a low band of frequency is based off of the current needs of the website. For the most ideal performance, higher bandwidths of frequency may be required in order to reduce any overload.

As well, a network connection between the server and the Yahoo! Finance API is necessary when called to retrieve the most up-to-date information from the API to ensure accurate data for the clients' needs.

Disk Space

The server must have adequate hard drive space to be able to store all of the database information. All of the data being stored is the sum of all program instructions for the system. At this time, 10 GB of storage space should be sufficient for the system.

System Memory

As the ML-Stock Prediction system is in active development, at this time there is limited concrete evidence that supports the overall performance of the system. The system will load copies of database-stored information in order to operate over it. For the most efficient throughput, the system memory should be managed using a Least Recently Used (LRU) scheme, in order to keep the system memory populated with the most useful information. An LRU scheme will release

any bits of memory and information that have not been accessed in a long time, and will replace that information with that which is used more often.

As well, any operations used on loaded information will also use up system memory. With this in mind, a minimum of 512MB of memory should be used for testing the ML-Stock Prediction system. As user-based interactions with the system expand, it can be predicted that the system memory requirements will have to expand along with it.

Client-Side Hardware Requirements

As noted previously, the core hardware requirements on the client-side of the system will be an internet connection with access to a modern web browser. This requirement is essential for the client to be able to remotely connect to the server in order to access our product and database. Without an internet connection, the client will not be able to use the web browser to access the ML-Stock Prediction website and application.

In addition to an internet connection, if the client is accessing the website via personal computer, then a functioning mouse and keyboard will also be required for the best experience, as well as a graphical display to see their portfolio. To display the ML-Stock Prediction website, a screen resolution with a minimum of 800x600 pixels will be adequate.

User Interface Design and Implementation

a. Updated Progress

As of now, there has been no significant development from the initial UI mockups presented in report 1. However, we anticipate making numerous minor yet substantial adjustments to the final website as we initiate user interaction studies. Additionally, we expect some minor changes for the initial demonstration, although these have not been executed or finalized as of yet.

b. Enhance Site Efficiency

A critical focus for us is to ensure that the website performs swiftly for all users, regardless of the device or connection they are using. To achieve this goal, we will strategically break down the website to enable client-side caching of elements that tend to remain constant. This involves separating the header and content of each page. Since the header remains consistent throughout the site, it only needs to be loaded once on the client side and can be cached for the duration of the visit or longer.

To further alleviate the load on clients, we will heavily rely on HTML and CSS for our User Interface, minimizing the use of images. Any images displayed will be resized to the maximum allowable size and converted to an appropriate web format.

Furthermore, as emphasized throughout our reports, our objective is to make our application accessible on a wide range of devices, including mobile, tablet, and desktop. To achieve this, we may utilize the Twitter Bootstrap CSS framework to facilitate the creation of a responsive website.

Of course, there is a trade-off involved. We will not provide support for older browsers that cannot handle HTML5, CSS3, or modern web standards. However, this is expected to have minimal impact, as most devices and users already have modern web browsers, and those that do not typically do not fall within our target audience.

Design of Tests

While no application can ever be considered truly complete, a crucial aspect of ensuring a project's viability is rigorous testing. Testing serves a multitude of purposes, such as validating expected functionality, identifying potential security vulnerabilities, and guarding against regression as the project evolves. Attempting to launch a product without thorough unit and integration testing, along with the practice of "dogfooding" an alpha version, is a surefire recipe for releasing a buggy and subpar product. Nonetheless, even with comprehensive testing efforts, it's important to acknowledge that it's impossible to uncover and address every flaw before shipment. To address this limitation, developers employ testing suites to streamline integration and unit testing, making the process efficient and effective.

A contemporary approach to strike a balance in this trade-off is to construct an application's feature set based on quantifiable, predefined tests. This methodology, known as Test-driven Development (TDD), involves developers iteratively crafting tests for planned future features, verifying that these features are not yet implemented (by running those tests), and subsequently implementing the solutions. Although this approach doesn't account for all potential interactions between components, it is typically adopted in fast-paced development environments like ours, where the test coverage provided is usually sufficient to preempt most issues.

As a result, our first step involves defining the features and associated tests that we intend to develop. We then assess the coverage afforded by these tests and briefly outline our strategy for testing the integration of various components.

a. Test Cases

The Haystack Group application is currently undergoing active development, and as a result, each test case outlined is relevant only to the existing functions at this developmental stage. To ensure comprehensive testing, we will conduct unit tests for each component that presently exists within the system.

Haystack Group develops web applications using Python. Depending on the development situation, the Django framework may be used. We may also use scikit learn, a library for implementing stock prediction using Machine Learning, as it is the most common library for stock price prediction. All team members are familiar with Python, so there will be no problem in choosing the right technology for the development. While the Haystack Group application necessitates communication between Yahoo! Finance, our MySQL database, and our server, conducting unit tests on these individual components is not the most efficient approach. Instead, we will carry out integration tests on these units to evaluate their interactions with one another.

b. Unit Testing

Database Manager

The following tests are related to our MySQL database but are independent of the actual database implementation.

Test Case 1: Retrieving User Information

- **Operation:** get_user_info(user_id: int): UserInfo
- **Test Input:** Provide a valid user_id.
- **Success Criteria:** The function should return a valid UserInfo object for the specified user_id.
- **Fail Criteria:** The function returns an error or raises an exception.

Test Case 2: Updating User Information

- **Operation:** update_user_info(user_id: int, new_info: UserInfo): void
- **Test Input:** Provide a valid user_id and a valid new_info object.
- **Success Criteria:** The function should update the user information for the specified user_id with the provided data. Subsequent retrieval of the user information should reflect the changes.
- **Fail Criteria:** The function doesn't update the user information, raises an error, or does not persist the changes.

Test Case 3: Retrieving Order Information

- **Operation:** get_order_info(user_id: int): OrderInfo[]
- **Test Input:** Provide a valid user_id.
- **Success Criteria:** The function should return a list of valid OrderInfo objects for the specified user_id.
- **Fail Criteria:** The function returns an error or raises an exception.

Test Case 4: Updating Stock Prediction

- **Operation:** update_StockPrediction(symbol: string, prediction: StockPrediction): void

- **Test Input:** Provide a valid symbol and a valid StockPrediction object.
- **Success Criteria:** The function should update the stock prediction for the specified symbol with the provided prediction data. Subsequent retrieval of the stock prediction should reflect the changes.
- **Fail Criteria:** The function doesn't update the stock prediction, raises an error, or does not persist the changes.

Test Case 5: Retrieving Stock Prediction

- **Operation:** return_StockPrediction(symbol: string): StockPrediction
- **Test Input:** Provide a valid symbol.
- **Success Criteria:** The function should return a valid StockPrediction object for the specified symbol.
- **Fail Criteria:** The function returns an error or raises an exception.

Order_Manager

The following tests are related to our design but is independent on our own implementation.

Test Case 1: Checking and Retrieving Order Information

- **Operation:** check_order(order_id: int): OrderInfo
- **Test Input:** Provide a valid order_id that exists in the system.
- **Success Criteria:** The function should successfully retrieve and return valid OrderInfo for the specified order_id.
- **Fail Criteria:** The function returns an error, raises an exception, or does not find the order for the given order_id.

Test Case 2: Placing a New Order

- **Operation:** place_order(order: OrderInfo): void
- **Test Input:** Provide a valid OrderInfo object for placing an order.

- **Success Criteria:** The function should successfully place the order using the provided order data. Subsequent checks should confirm that the order exists in the system.
- **Fail Criteria:** The function doesn't place the order, raises an error, or does not persist the order data.

Test Case 3: Deleting an Order

- **Operation:** delete_order(order_id: int): void
- **Test Input:** Provide a valid order_id of an existing order in the system.
- **Success Criteria:** The function should successfully delete the order with the specified order_id. Subsequent checks should confirm that the order no longer exists.
- **Fail Criteria:** The function doesn't delete the order, raises an error, or the order still exists after deletion.

Test Case 4: Executing an Order

- **Operation:** execute_order(order_id: int): void
- **Test Input:** Provide a valid order_id of an existing order in the system.
- **Success Criteria:** The function should successfully execute the order with the specified order_id. The order status should reflect that it has been executed.
- **Fail Criteria:** The function doesn't execute the order, raises an error, or the order status remains unchanged after execution.

ML_Prediction

The following tests are related to our design but is independent on our own implementation.

Test Case 1: Selecting and Storing Market Data

- **Operation:** select_MarketData(data: MarketData): void
- **Test Input:** Provide a valid MarketData object for selection and storage.
- **Success Criteria:** The function should successfully store the provided market data for prediction. Subsequent operations should confirm that the data has been selected and stored.
- **Fail Criteria:** The function doesn't store the market data, raises an error, or the data cannot be retrieved after selection.

Test Case 2: Deleting Market Data

- **Operation:** delete_MarketData(data_id: int): void
- **Test Input:** Provide a valid data_id of existing market data in the system.
- **Success Criteria:** The function should successfully delete the market data with the specified data_id. Subsequent checks should confirm that the data no longer exists.
- **Fail Criteria:** The function doesn't delete the data, raises an error, or the data still exists after deletion.

Test Case 3: Updating Existing Market Data

- **Operation:** update_MarketData(data_id: int, new_data: MarketData): void
- **Test Input:** Provide a valid data_id of existing market data and a valid new_data object for updating.
- **Success Criteria:** The function should successfully update the existing market data with the provided new_data. Subsequent checks should confirm that the data has been updated.
- **Fail Criteria:** The function doesn't update the data, raises an error, or the data remains unchanged after the update.

Account_Controller

The following tests are related to our design but is independent on our own implementation.

Test Case 1: Logging In a User

- **Operation:** log_in(username: string, password: string): User
- **Test Input:** Provide a valid username and password that correspond to an existing user in the system.
- **Success Criteria:** The function should successfully log in the user with the provided username and password and return a valid User object.
- **Fail Criteria:** The function fails to log in the user, raises an error, or returns an incorrect user object.

Test Case 2: Logging Out a User

- **Operation:** log_out(user: User): void
- **Test Input:** Provide a valid User object to log out.

- **Success Criteria:** The function should successfully log out the specified user, and the user's session is ended.
- **Fail Criteria:** The function doesn't log out the user, raises an error, or the user's session remains active.

Test Case 3: Creating a New User Account

- **Operation:** create_account(user_info: UserInfo): User
- **Test Input:** Provide valid UserInfo data to create a new user account.
- **Success Criteria:** The function should successfully create a new user account with the provided user information and return a valid User object.
- **Fail Criteria:** The function fails to create the account, raises an error, or returns an incorrect user object.

Test Case 4: Deleting a User Account

- **Operation:** delete_account(user: User): void
- **Test Input:** Provide a valid User object to delete the user's account.
- **Success Criteria:** The function should successfully delete the user's account, and the user's data is removed from the system.
- **Fail Criteria:** The function doesn't delete the account, raises an error, or leaves the user's data intact in the system.

YahooFinanceAdapter

The following tests are related to our design but is independent on our own implementation.

Test Case 1: Retrieving a Quote

- **Operation:** get_quote(symbol: string): Quote
- **Test Input:** Provide a valid stock symbol that corresponds to an existing stock.
- **Success Criteria:** The function should successfully retrieve a valid Quote for the specified stock symbol.
- **Fail Criteria:** The function fails to retrieve the quote, raises an error, or returns an incorrect Quote object.

Test Case 2: Retrieving Company Information

- **Operation:** get_company_info(symbol: string): CompanyInfo

- **Test Input:** Provide a valid stock symbol that corresponds to an existing company.
- **Success Criteria:** The function should successfully retrieve valid CompanyInfo for the specified stock symbol.
- **Fail Criteria:** The function fails to retrieve the company information, raises an error, or returns incorrect CompanyInfo.

Test Case 3: Updating Market Data

- **Operation:** update_MarketData(data: MarketData): void
- **Test Input:** Provide a valid MarketData object for updating.
- **Success Criteria:** The function should successfully update the market data using the provided data.
- **Fail Criteria:** The function fails to update the market data, raises an error, or does not persist the changes to the data.

c. Test Coverage

The perfect test coverage would involve having tests that encompass every possible edge case for every method. However, this is not just unattainable; it is outright impossible, as it's impossible to know all conceivable edge cases. To address this, our strategy is to focus on testing the core functionality to ensure a fundamental level of testing. Subsequently, by engaging with end users through alpha and beta builds, we will identify unforeseen ways in which users interact with the system. This will enable us to incorporate additional testing to address these novel edge and usage scenarios, which will also aid in debugging and preventing regressions in the future.

d. Integration Testing

Integration testing will be conducted on a local developer machine, simulating the server environment. The system may not go live until the current system functions seamlessly in the integration environment. This is achieved by maintaining two branches of source code, namely "master" and "dev." All new work will be performed on the "dev" branch, which will then be pulled into the local integration machine for testing and debugging. After the system has been thoroughly debugged, with detailed records kept for any necessary system configuration changes, the source code will be merged into the "master" branch. Following this, any system configuration adjustments will be implemented on the production server to accommodate the new branch. Once these changes are in place, the "master" branch will be incorporated into the production machine, and a second round of integration testing will commence by launching the service on a developer port. If the system passes all the tests, the developer port will be closed, and the system will relaunch the website on the standard HTTP port.

Project Management and References

a. Plan of Work and Product Ownership

Team members from all three groups will support one another based on the complexity of the task, and knowledge sharing is crucial throughout this process.

Team Pair 1: Lisa & Brian

- Functionality: User registration, account management, and authentication.
- Qualitative Property: Ensuring data security and user privacy.
- Ownership: Lisa will lead the registration and authentication development, while Brian will focus on account management.

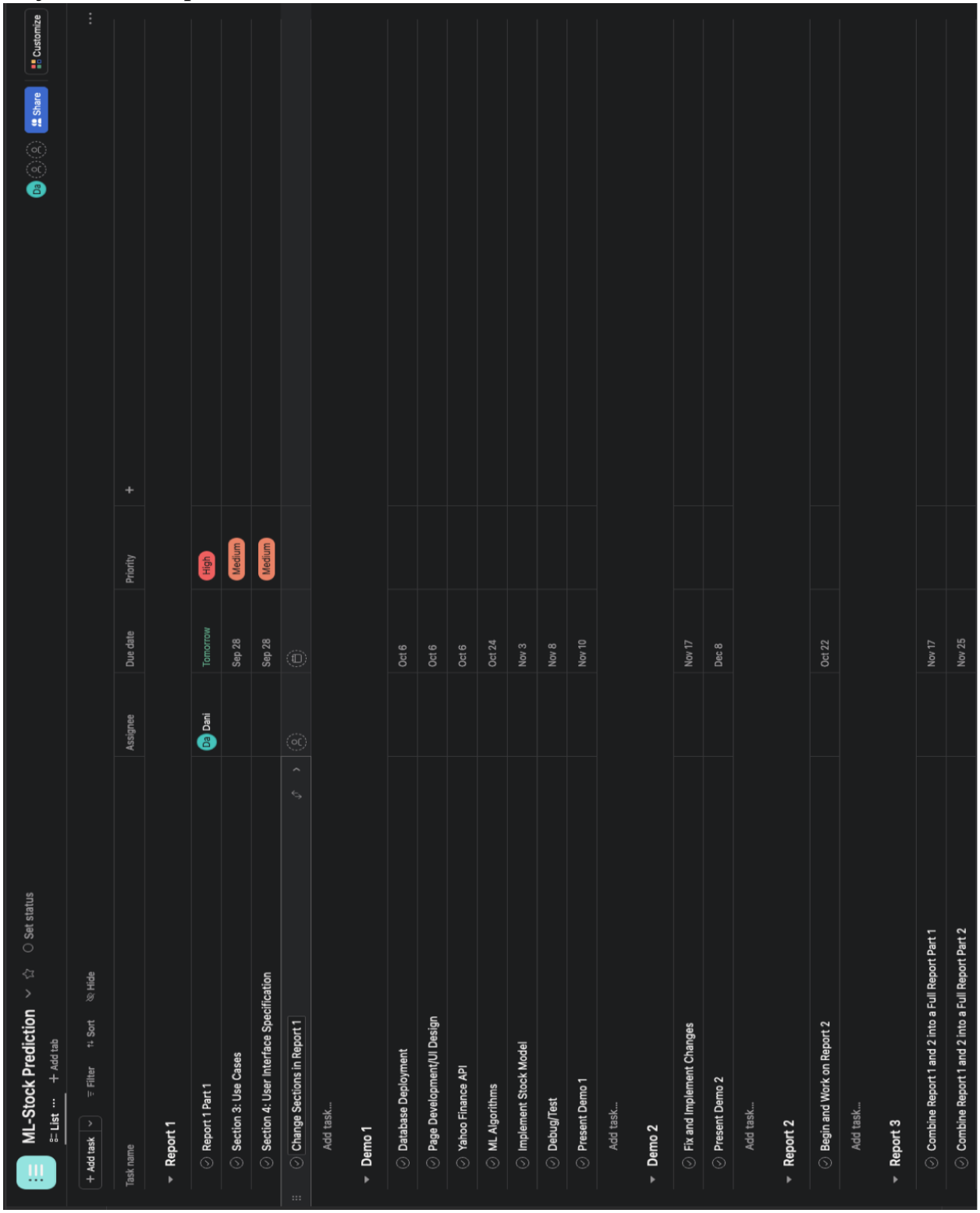
Team Pair 2: Keita & Curtis

- Functionality: Data capture, storage, and integration with financial APIs.
- Qualitative Property: Optimize API response time for improved speed.
- Ownership: Keita will work on data capture and integration, while Curtis will focus on optimizing API responses.

Team Pair 3: Alex & Taylor

- Functionality: ML-based stock price predictions, data visualization, and portfolio management.
- Qualitative Property: Ensure the accuracy of ML predictions.
- Ownership: Alex and Taylor will lead the development of the ML prediction model and data visualization, with the backup from team assisting as needed.

b. Project Roadmap



For the remainder of this project, the development of the Haystack Group will be divided into two major deliverables. The first deliverable will be an alpha release that includes a set of core functionalities essential for delivering a minimum viable product. It is anticipated that this initial deliverable will be finalized by the end of October.

Following the completion of this milestone, the project will go live in an alpha state at an undisclosed IP address, with no search engine indexing. We may extend invitations to a group of users to utilize the product and provide feedback, while also conducting in-house user observations. The objective is to understand how users engage with the system and gather insights into what additional features they desire to enhance the product. This approach aligns closely with the Agile development methodology.

c. References

1. https://bitflyer.com/en-jp/s/regulations/security
2. https://getbootstrap.com/2.0.2/