ML-Stock Prediction

Report 3
Software Engineering

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

Website: thepressq.com(Hidden)
GitHub: https://github.com/Stagnantmoon15/MLPrediction

All Team Members contributed equally to this Report.

Table of Contents

1. Customer Problem Statement

a. Problem Statement

The global financial landscape is significantly influenced by major stock exchanges such as the New York Stock Exchange (NYSE), Nasdaq, Shanghai Stock Exchange, Euronext, and Japan Exchange Group. Notably, the NYSE and Nasdaq, primarily functioning within the United States, are key barometers of the private sector's vitality and public economic confidence. This pivotal role has naturally attracted a growing number of individuals eager to invest in these markets, aiming to augment their personal financial wealth.

However, many aspiring investors, irrespective of age, find themselves hindered by substantial entry barriers to these markets. Recognizing this challenge, The Haystack Group is dedicated to developing an innovative ML-Stock Prediction application. This application is designed to ease access to American markets and is equipped with educational resources to break down these barriers. It promises a smooth and efficient registration process, allowing users to quickly begin their investment journey. The platform offers the unique ability to track and predict the future values of a diverse range of stocks from various global markets, aiming for predictions that closely mirror future market realities.

To further empower users with a deep understanding of financial markets, the application will provide vital market metrics. These include timely news updates concerning companies in users' portfolios, interactive charts for enhanced visual analysis, and comprehensive data on market opening and closing activities. This information is vital for informed investment decisions.

A key focus is on delivering a user experience that is uniformly excellent across different platforms, including mobile devices, tablets, and desktop computers. The amalgamation of these features is intended to create a rich, immersive experience, enabling users to gain comprehensive insights into stock market dynamics and predictions.

Currently, the platform faces challenges such as the inability to display real-time stock prices through APIs, hindering effective market analysis due to the absence of various key indicators. Additionally, the lack of automated trading functionalities has been identified as a significant gap. To address this, we plan to integrate an AI-driven model and APIs that facilitate automated trading, enriching the platform's capabilities.

User interface simplicity and intuitiveness are also primary goals. We aim to design an interface that is straightforward and accessible, even for beginners, encompassing a broad spectrum of user-friendly trading tools. This includes standard trading features as well as advanced options like limit orders and pre-orders.

Security is a paramount concern. We intend to implement robust, periodically updated security measures to ensure safe trading. This includes a two-pronged approach: establishing a secure system for asset management and implementing proactive measures against cyber threats, including viruses, hacking, phishing emails, and malicious websites. A multi-signature system is being considered for enhanced security infrastructure.

The cornerstone of our project is the development of a sophisticated Machine Learning algorithm for stock price prediction. Utilizing historical stock data and market indicators, this ML-based predictor is designed to provide precise stock price forecasts. It will serve a broad spectrum of users, from novice investors to experienced traders, offering valuable, data-driven insights to inform their investment strategies.

b. Glossary of Terms

1. Stock:

- **Definition**: A stock represents a share of ownership in a company. Ownership of stocks entitles the holder to a claim on the company's earnings and assets, proportional to the number of shares owned.
- **Dynamics**: Stock values fluctuate throughout the day based on the company's performance, influencing buying and selling decisions within its fiscal cycle.

2. Stock Ticker:

- **Definition**: A tool used to monitor the real-time rise and fall of a stock's value throughout the trading day.
- **Functionality**: Provides users with current stock prices and tracks changes over time.

3. Stock Graph:

- **Definition**: A graphical representation comprising data/values collected over the course of a day, showing the stock prices' fluctuations.
- **Purpose**: Aids in visualizing stock price trends and predicting future movements based on the day's data.

4. scikit-learn (sklearn):

- **Description**: A widely-used Python library for machine learning.
- **Features**: Offers tools and algorithms for classification, regression, clustering, dimensionality reduction, etc.
- **User Experience**: Known for its user-friendly and consistent API, suitable for both novices and experts.
- **Utilities**: Provides data preprocessing, model selection, and model evaluation tools.

5. Linear Regression:

- **Definition**: A fundamental machine learning algorithm used for predicting prices, estimating trends, and analyzing variable relationships.
- **Applications**: Commonly applied in economics, finance, and scientific fields.

6. Tick Indicators:

- **Use**: Identifies stocks based on their last trade being an uptick or a downtick.
- **Importance**: Serves as an indicator for assessing market sentiment and trend direction.

7. Simple Moving Average (SMA):

- **Function**: Calculates the average asset price over a specific time period.
- **Role**: Helps in smoothing price fluctuations and identifying price movement trends.

8. Exponential Moving Average (EMA):

- **Comparison with SMA**: Similar to SMA but places more emphasis on recent prices.

- **Utility**: Useful for identifying short-term trends and responding quickly to market changes.

9. Bollinger Bands:

- **Composition**: Consists of a middle band (SMA) and two outer bands set at standard deviations.
- **Purpose**: Assists traders in identifying market volatility and potential price reversals.

10. Ichimoku Kinko Hyo (IKH):

- **Nature**: A comprehensive Japanese charting system.
- **Function**: Offers insights into support/resistance levels, trend directions, and potential trading signals.

11. Volume:

- **Definition**: A measure of the number of shares or contracts of a financial asset traded within a specific period.
- **Relevance**: Indicative of the trading activity and liquidity of an asset.

12. Relative Strength Index (RSI):

- **Type**: A momentum oscillator that measures the speed and change of price movements.
- **Range**: Varies from 0 to 100, used to identify overbought (above 70) or oversold (below 30) conditions.

13. Moving Average Convergence Divergence (MACD):

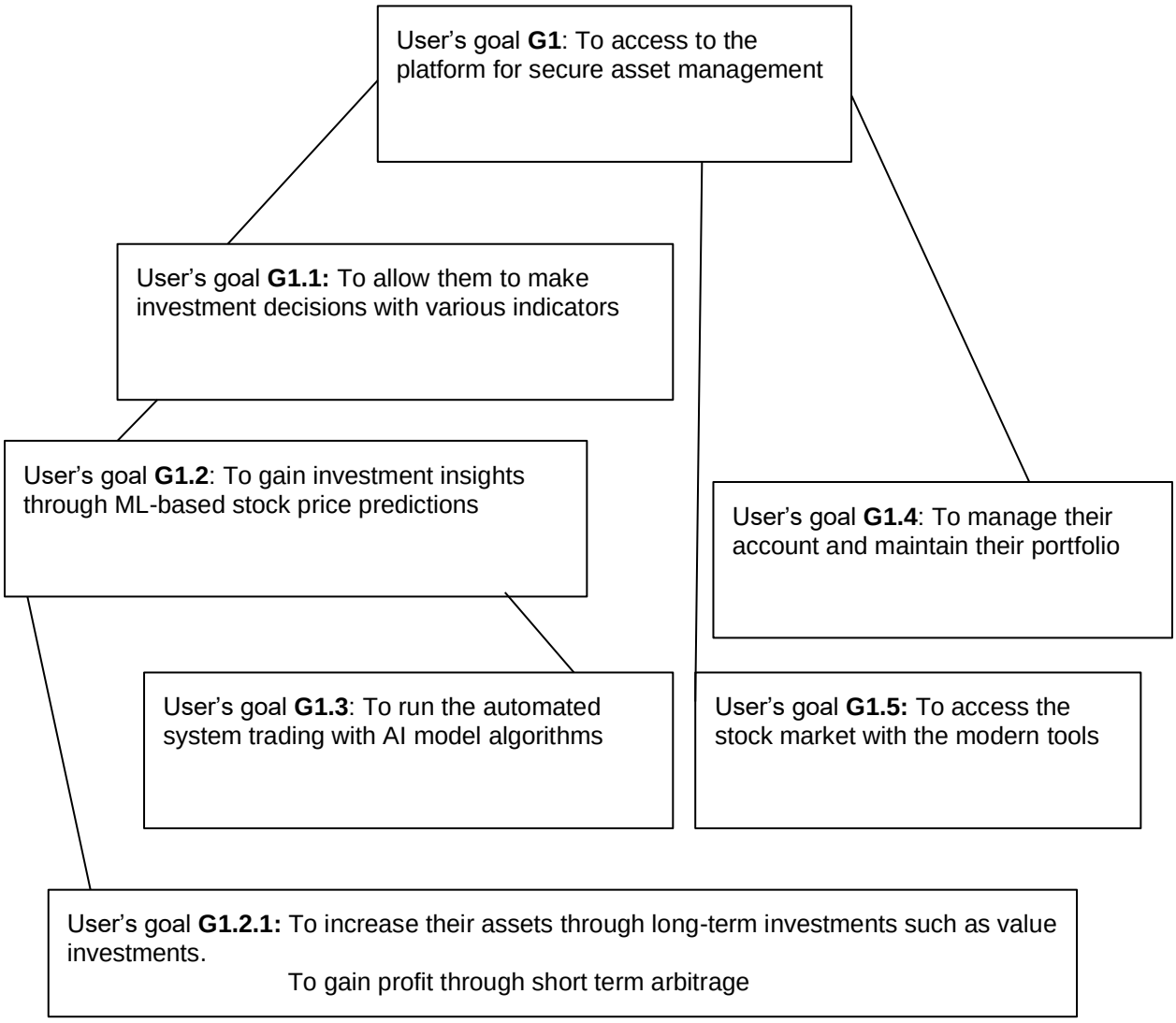
- **Description**: A trend-following momentum indicator.
- **Components**: Comprises the MACD line and a signal line, showing the relationship between two moving averages of an asset's price.

14. Historical Volatility:

- **Purpose**: Measures an asset's past price fluctuations over a specified time frame.
- **Application**: Helps traders in assessing risk levels, aiding in decision-making regarding position sizing and risk management

2. System Requirements

a. User and Business Goals



3. Functional Requirement Specifications

a. Stakeholders

The described software is primarily aimed at two key demographics: first-time investors and existing investors. However, it is designed to evolve into a comprehensive tool for both short and long-term trading decision planning. The intention is for the software to eventually expand its user base beyond comparable platforms by offering enhanced features, such as portfolio importing, which can entice investors to spend more time using the platform.

Initially, the software will be offered as a free service, with the long-term plan of transitioning to a subtle advertising model that does not disrupt the user experience. Advertisements will only be introduced once a significant user base has been established, providing a source of revenue for the company. This approach, as a free service with future ads, is expected to attract many users and, due to its increased functionality, keep them engaged for extended periods.

The software's target audience is not limited to potential investors; it is intended to cater to anyone seeking a deeper understanding of the financial industry and those interested in practicing trading as a learning experience before venturing into the actual market.

b. Actors and Goals:

Guest:

1. Description: A visitor to the website who has either not logged in or is just a simple visitor.
2. Capabilities:
    1. Register and create an account using (Future OpenID or OAuth2)
    2. View a random stock.

Investor:

3. Description: A user who has an account on the server and is logged into their account.
4. Capabilities:
    1. Research the latest updates in the market.
    2. View their portfolio.
    3. Execute orders of any kind.

Database System:

5. Description: Stores information for the accounts of all users.
6. Responsibilities:
    1. Insert information as accounts are created.
    2. Push data back to views about users/events.
    3. Store new data about users and events.

Financial API:

7. Description: Provides the stocks in the database with up-to-date prices.
8. Responsibilities:
    1. Fetch real-world information and update the database accordingly.

Site Administrator:

9. Description: Manages the overall website.
10. Responsibilities:
    1. Ensure Site is running smoothly.

Browser:

11. Description: The middleman between the user and the system.
12. Responsibilities:
    1. Present data to the user.
    2. Retrieve data from the user.

Yahoo! Finance:

13. Description: The unit that knows about current financial statistics.
14. Responsibilities:
    1. Retrieve data about stocks.

c. Use Cases

UC-1 Register/Create Account - A user can register, create, and authenticate their account to fully access all the features of the application by using OpenID or OAuth2. Derived from ST-1 and ST-2.

UC-2 View Market Data - A user can search for stocks and view real time market data from Yahoo! Finance API. Derived from ST-3, ST-6, ST-7, and ST-8.

UC-3 Manage Portfolio - A user can manage their stock portfolio and access up to date information regarding company and market activity that use interactive charts. Derived from ST-9 and ST-10.

UC-4 Pick Stock and ML-Language Prediction - A user can execute market orders and view ML-based stock price predictions to aid in investment decisions. Derived from ST-3 and ST-5.

UC-5 Take Administrative Actions - The site administrator can capture and store historical stock data, as well as manage and maintain the application site to run smoothly. Derived from ST-4.

Use Case Diagram:

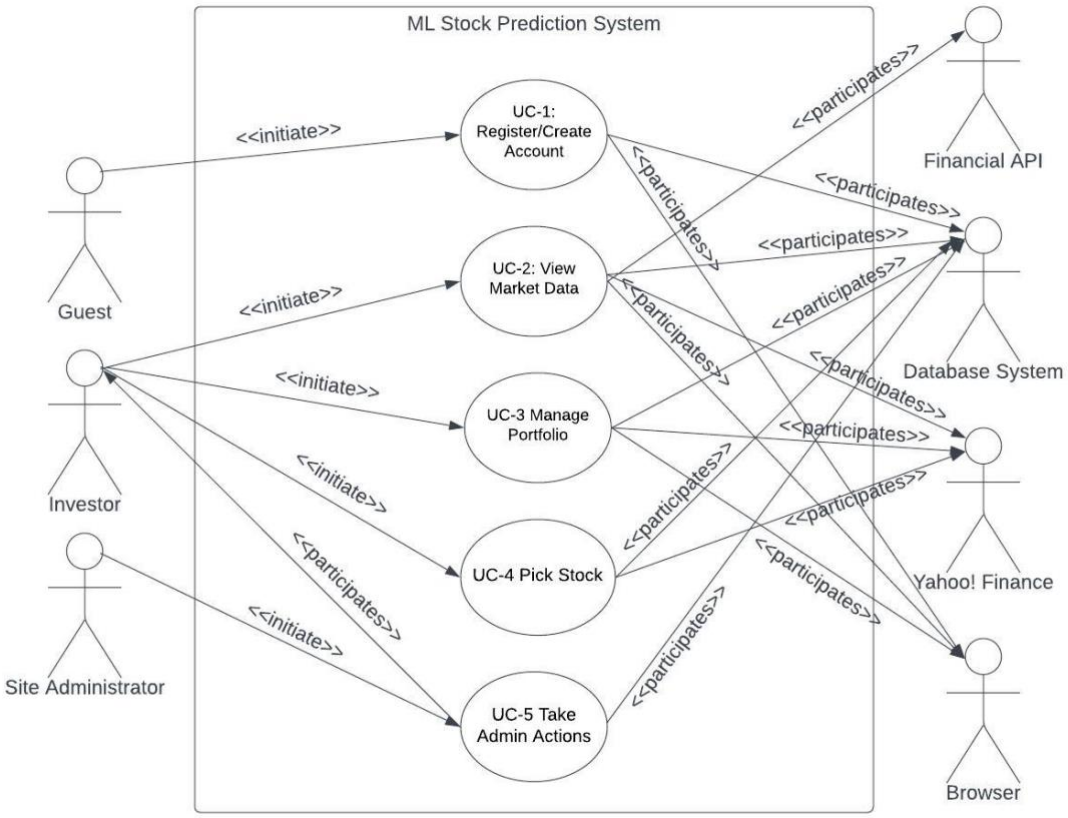


Figure 3.1: This graphic illustrates the relationships between the core actors of our platform.

Use Case UC-1:	**Register/Create Account**
Related Requirements:	ST-1, ST-2
Initiating Actor:	Guest
Actor's Goal:	Register and create account with website servers
Participating Actors:	Guest, Database System, Browser

Preconditions:	Guest is not a registered user
Postconditions:	New user account information and data is stored by the Database System.
	Flow of Events for Main Success Scenario:
→	1. Guest navigates to application website and attempts to log in on Browser
→	2. System checks Database for user information of Investor to see if found or not
←	3. System receives OpenID/OAuth2 data and registers user information in the Database as Investor
←	4. System sends registration confirmation to Investor and displays portfolio in Browser
	Flow of Events for Alternative Scenario:
→	1. Investor navigates to application website and attempts to log in
→	2. System checks Database for user information of Investor to see if found or not
←	3. Systems retrieves Investor user information from Database
←	4. System displays Investor’s profile on Browser

Status for Final Demo

- **Current Implementation Status**: The 'Register/Create Account' use case, while planned, will not be included in the final demo.
- **Reason for Exclusion**: Resource limitations and prioritization of other features have led to the postponement of this use case.
- **Future Work Consideration**: As a critical entry point for new users, this use case is a priority for future development phases. Implementing a smooth and user-friendly registration process is vital for expanding the user base and

enhancing the overall user experience. It will be a significant feature to integrate in the subsequent updates following the final demo.

Use Case UC-2: View Market Data

Overview

This use case encapsulates the process through which an Investor searches for specific stocks and accesses real-time market data, using the system integrated with Yahoo! Finance API and the Database System.

Actors

- **Initiating Actor**: Investor.
- **Participating Actors**: Database System, Yahoo! Finance API, Browser.

Preconditions

- The Investor is logged into the system. (NOT IN DEMO, GUEST ONLY)
- Yahoo! Finance stock data requests are operational.

Postconditions

- Accurate stock and market information is displayed to the Investor.

Flow of Events for Main Success Scenario

1. **Initiating Stock Search**: The Investor searches for a stock using its symbol via the Browser.
2. **Database Query**: The System sends this search request to the Database.
3. **Suggested Symbols Retrieval**: The Database returns a list of suggested stock symbols to the System.
4. **Display of Suggestions**: The System displays these suggested stock symbols to the Investor.
5. **Symbol Selection**: The Investor selects a preferred stock symbol from the suggestions and sends a request back to the System.
6. **Data Request**: The System forwards the request to both the Database and Yahoo! Finance API.

7. **Data Update and Integration**: The Database is updated with the latest stock information from Yahoo! Finance, along with ML-based price predictions.
8. **Investor Presentation**: The final aggregated data is presented to the Investor in the Browser, offering real-time market insights.

Implementation Status for Final Demo

- **UC-2 Implementation**: 'View Market Data' is a key use case that will be fully implemented for the final demo.
- **Importance**: This use case is crucial as it allows Investors to access real-time, relevant market data, essential for making informed investment decisions.
- **Impact**: The successful execution of UC-2 in the final demo represents a significant feature of the platform, showcasing its capability to provide timely and accurate financial data, which is fundamental to the user experience and the core functionality of the platform.

Use Case UC-3:	**Manage Portfolio**
Related Requirements:	ST-9, ST-10
Initiating Actor:	Investor
Actor's Goal:	Manage investments by viewing portfolio
Participating Actors:	Database System, Yahoo! Finance, Browser
Preconditions:	Investor is logged in and Yahoo! Finance stock data requests are working
Postconditions:	Investor can view and make changes to their portfolio
	Flow of Events for Main Success Scenario:

→	1. Investor requests to view their portfolio and navigates to area on Browser
←	2. System send request for Investor portfolio from Database
→	3. System displays investment portfolio to Investor in Browser
→	4. Investor makes applicable changes needed to portfolio
←	5. System updates Database portfolio information of user
→	6. System shows confirmation to Investor of updated portfolio in Browser

Status for Use Case UC-3: Final Demo

- **Current Implementation Status**: 'Manage Portfolio' is a planned use case that will not be included in the final demo.
- **Reason for Exclusion**: Due to constraints in resources and prioritization of other core functionalities, this use case is postponed for future implementation.
- **Future Consideration**: As an essential feature for user investment management, 'Manage Portfolio' will be a priority in the next phase of development. Its implementation will empower users with direct control over their investment choices, making the platform more dynamic and user-centric.

Use Case UC-4: Pick Stock and ML-Language Prediction

Overview

This use case details the process by which an Investor views ML-based stock price predictions and executes market orders. It is essential for enabling informed trading decisions.

Actors

- **Initiating Actor**: Investor.
- **Participating Actors**: Database System, Yahoo! Finance.

Preconditions

- The Investor must be logged in, and the integration with Yahoo! Finance for stock data must be functional.

Postconditions

- The Investor's positions and actions are accurately updated in the Database.

Flow of Events for Main Success Scenario

1. **Stock Selection and Order Placement**: The Investor chooses a stock and places a market order through the Browser.
2. **Data Request to Yahoo! Finance**: The System sends a request to Yahoo! Finance to get the latest stock data.
3. **Data Reception**: The System receives updated stock data from Yahoo! Finance.
4. **Record Order**: The System records the Investor's order in the Database.
5. **Display Portfolio Changes**: The System updates and displays the changes in the Investor's portfolio in the Browser. (NOT IN FINAL DEMO)

Implementation Status for Final Demo

This use case, 'Pick Stock and ML-Language Prediction,' will be fully implemented and functional for the final demo. It is critical for providing Investors with the capability to make informed decisions based on ML-based stock price predictions and real-time market data from Yahoo! Finance. The successful

integration of this feature into the final demo represents a significant milestone in offering a comprehensive and intelligent trading platform.

Significance

The implementation of UC-4 is pivotal in achieving the platform's goal of blending traditional stock trading functionalities with advanced machine learning predictions. It not only enhances user experience by providing insightful stock forecasts but also empowers users to execute trades based on a blend of AI-driven insights and real-time market data. This fusion of technology and finance is a key differentiator for the platform, positioning it as an innovative solution in the fintech domain.

Use Case UC-5:	**Take Administrative Actions**
Related Requirements:	ST-4
Initiating Actor:	Site Administrator
Actor's Goal:	Manage and maintain the application site to run smoothly. Derived from ST-4.
Participating Actors:	Database System, Investors
Preconditions:	User is logged in and has Site Administrator permissions
Postconditions:	Maintain application site to capture and store historical stock data
	Flow of Events for Main Success Scenario:
→	1. Site Administrator requests System logs
←	2. System saves, closes and returns logs

Status for Final Demo

- **Current Implementation Status**: This use case, 'Take Administrative Actions,' is planned but will not be implemented for the final demo.
- **Reason for Exclusion**: Due to prioritization and resource allocation, this use case has been deferred and is considered for future development phases of the project.
- **Future Consideration**: The implementation of this use case is crucial for the long-term maintenance and management of the application. It will be a focus in subsequent updates post the final demo, enhancing administrative capabilities and overall system robustness

d. Traceability Matrix

Requirements	Priority Weight	UC-1	UC-2	UC-3	UC-4	UC-5
ST-1	10	X				
ST-2	8	X				
ST-3	8		X		X	
ST-4	10					X
ST-5	10				X	
ST-6	4		X			
ST-7	6		X			
ST-8	4		X			
ST-9	10			X		
ST-10	4			X		
Total Priority		18	22	14	18	10

Figure 3.2: The traceability matrix presented here is based on only the full dressed use cases above.

e. System Sequence Diagrams

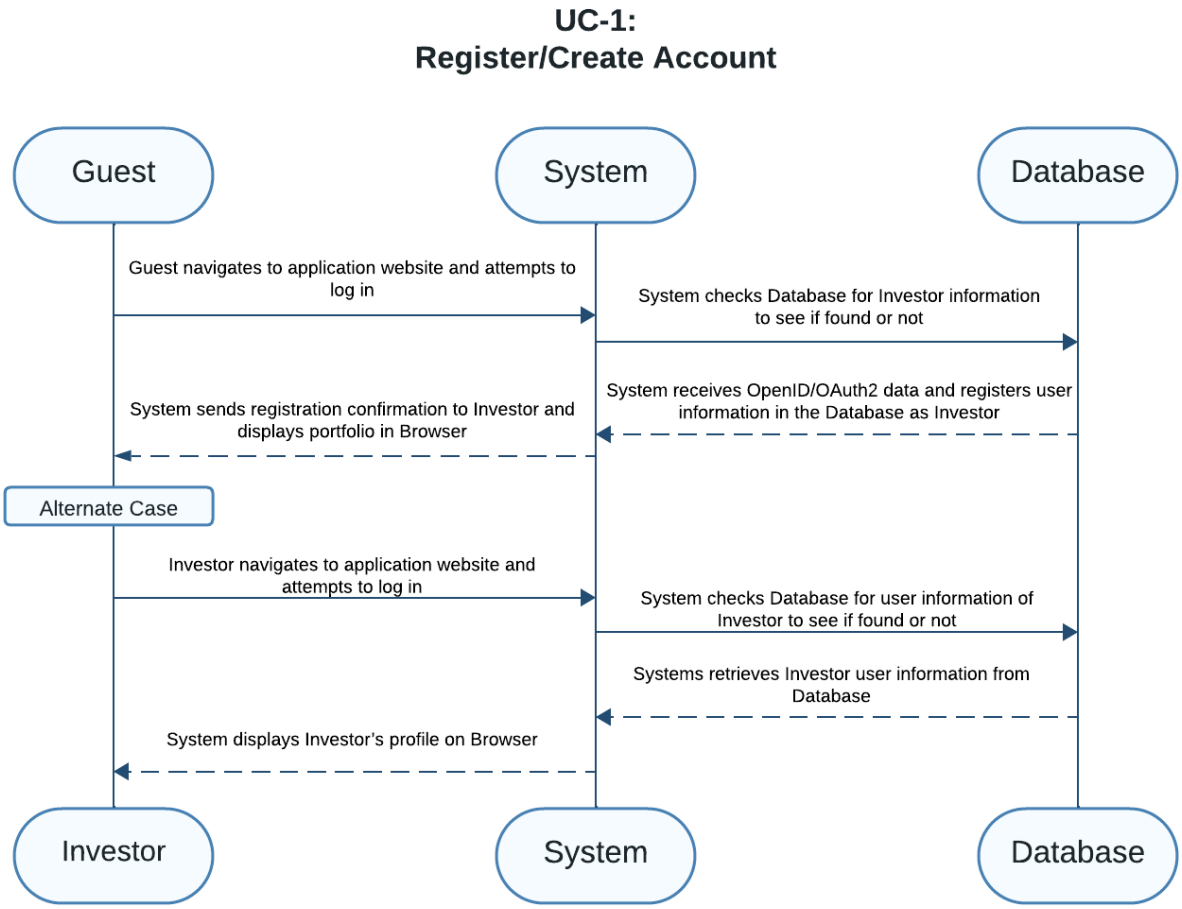


Figure 3.3: See UC-1 on page 14. When the guest navigates to the application website, this use case is triggered. The system checks to see if there is investor information in the database, and if not, it receives guest OpenID/OAuth2 data. The guest information is then registered as an investor. The system sends the registration confirmation to the investor and displays the portfolio in the browser. If the system confirms that the guest information is already registered in the database, it receives the investor information and displays the profile in the browser.

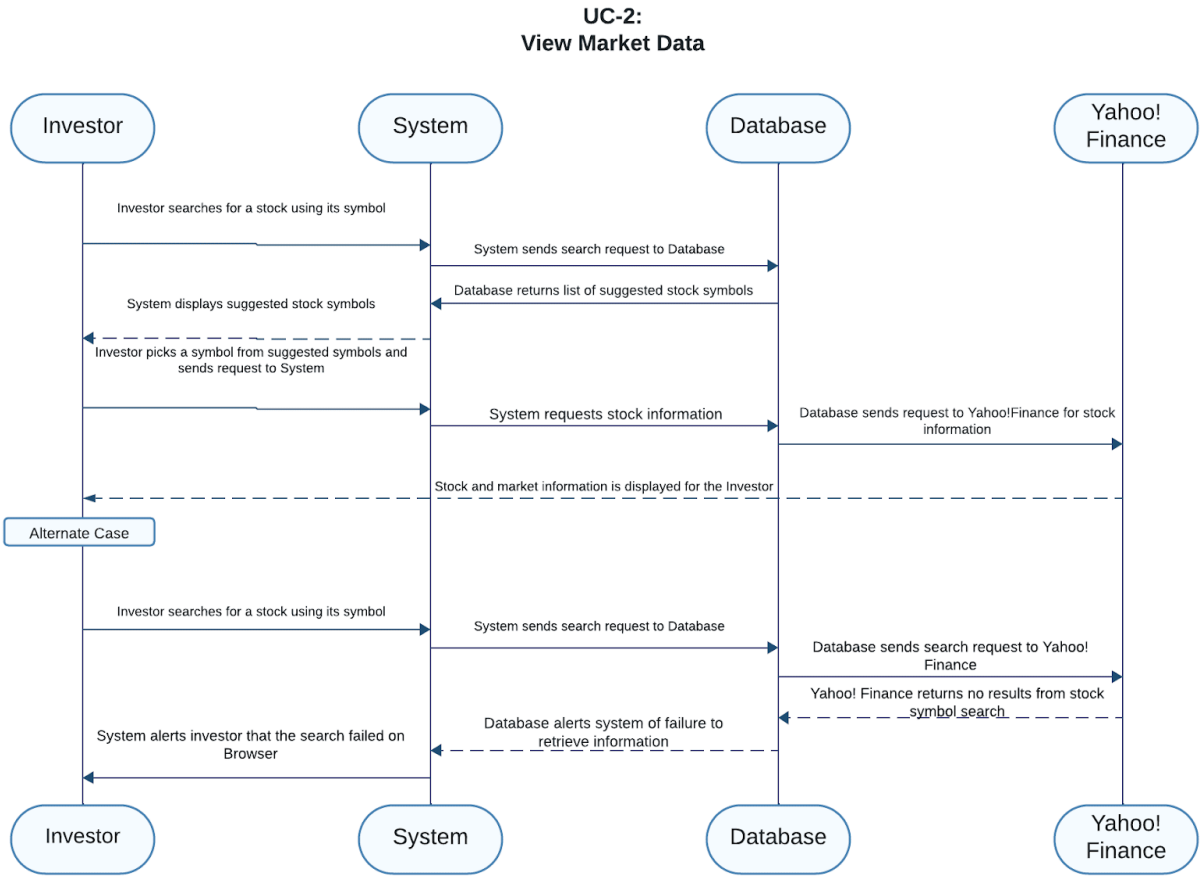


Figure 3.4: See UC-2 on page 15. This use case is triggered when an investor searches a stock with symbols. The system sends a search request to the database, which returns a list of corresponding stock symbols to the system, which displays the stock symbols. And the investor selects the symbol of the stock he/she wants to see from the list. The system requests the stock information corresponding to the symbol from the database, which requests the stock information from Yahoo! Finance. Finally, Yahoo! Finance provides the relevant stock and market information to the investor. If Yahoo! Finance cannot find the information requested by the database, it returns a no result to the database, which returns a failure to retrieve to the system. If Yahoo! Finance cannot find the information, it returns "no result" to database. Database returns "failure to retrieve" to system and system returns "failed" to investor.

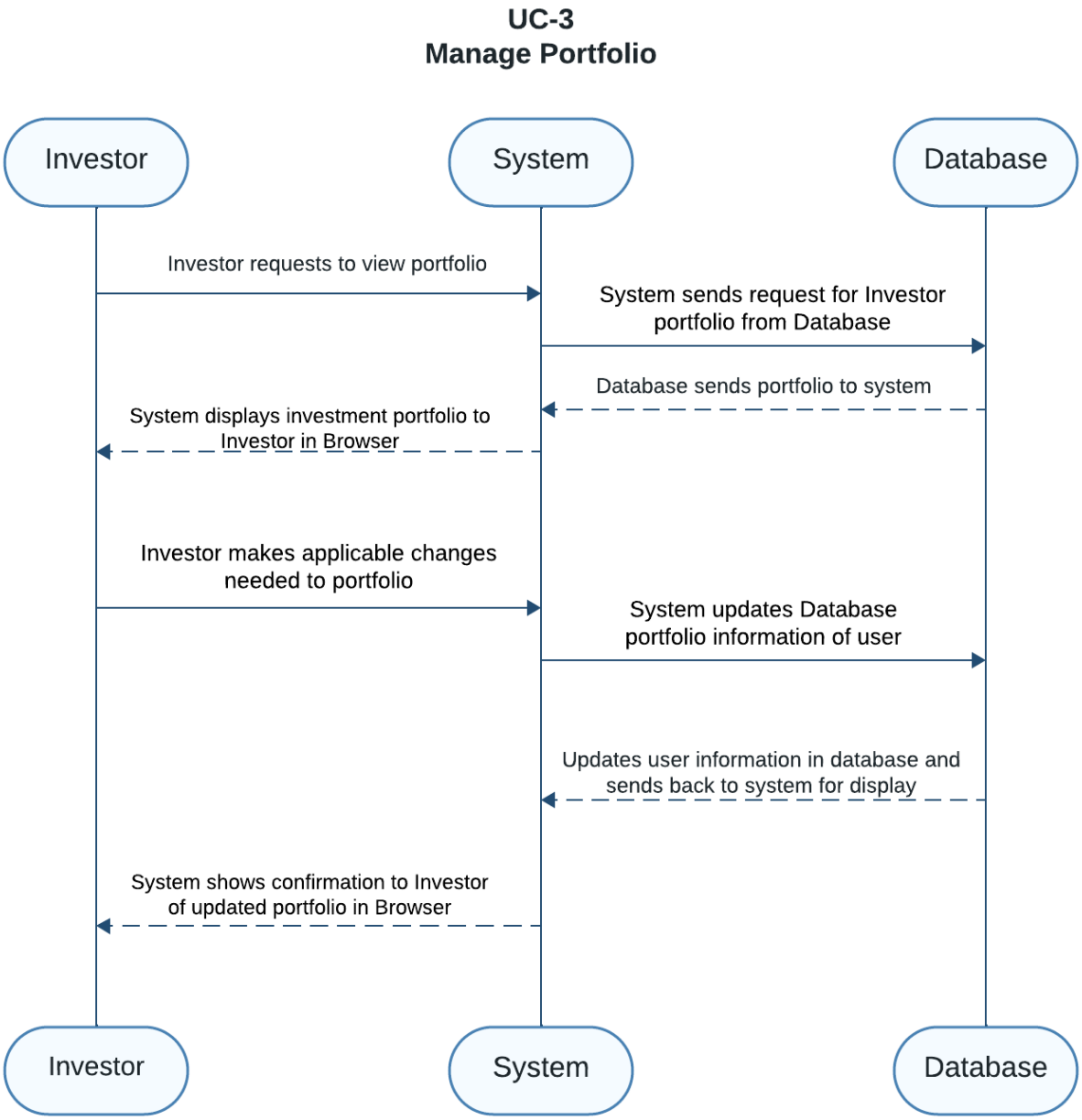


Figure 3.5: See UC-3 on page 15. This shows the investor to system to database back to investor in regards for the user to manage the portfolio.

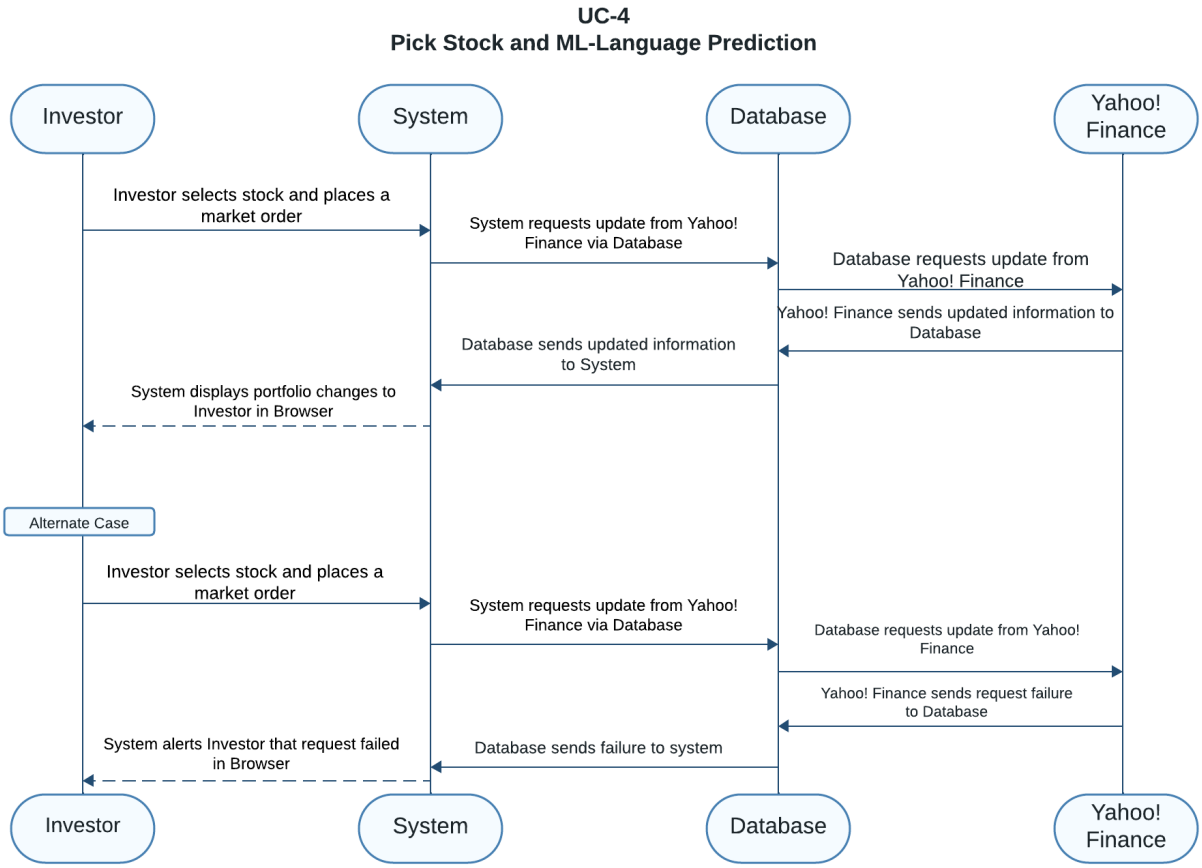


Figure 3.6: See UC-4 on page 16. This shows the Investor to our system to the database to Yahoo Finance API and back to pick a stock and use it for ML-Language prediction.

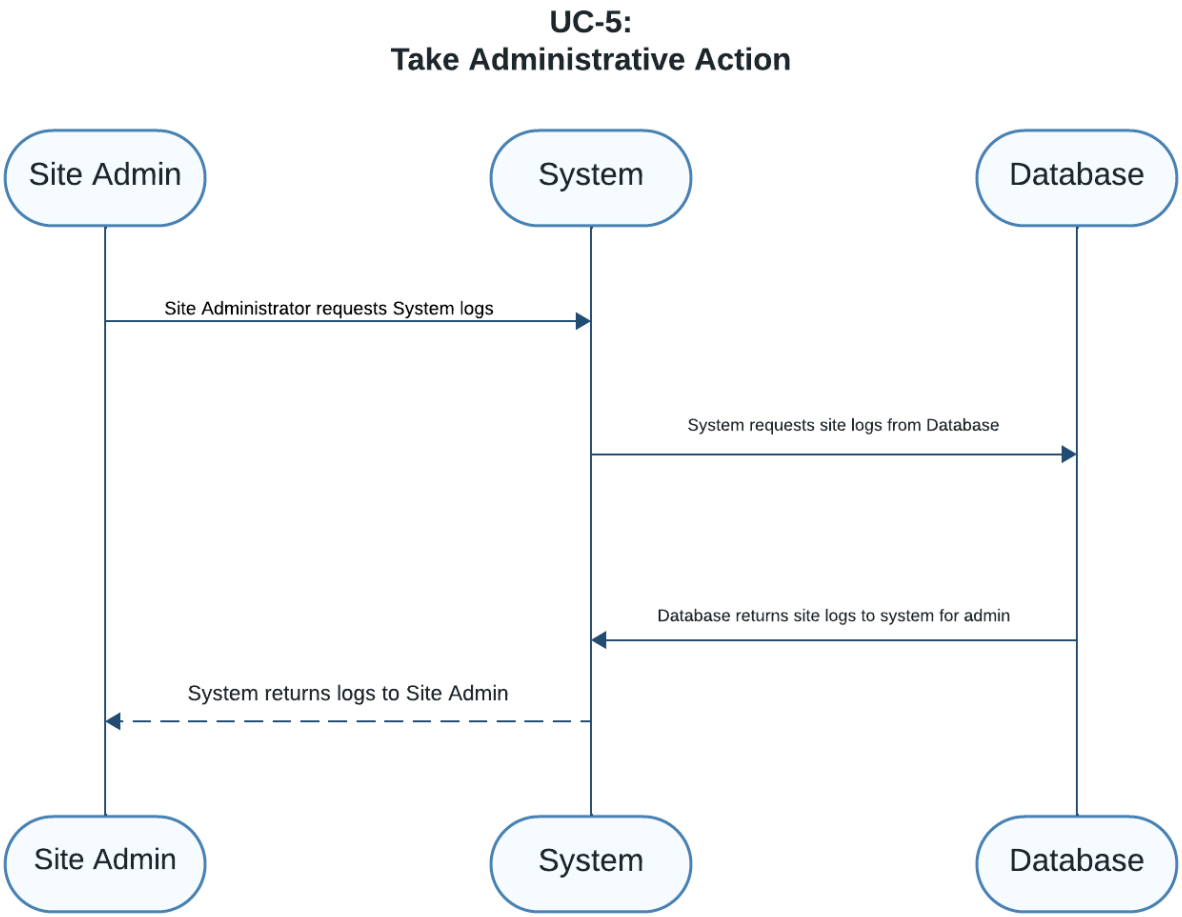


Figure 3.7: See UC-5 on page 16. This figure shows how the site admin will verify and look into various issues within the system and database via logs.

4. Effort Estimation using Use Case Points

a. Background

Effort estimation using Use Case Points (UCP) is a method to predict how much time and effort will be required to complete a software project based on its use cases. Let's go through the process step by step, assuming the productivity factor (PF) is 28 hours per use case point.

Steps for Effort Estimation Using Use Case Points

1. Calculate Unadjusted Use Case Weight (UUCW)

- Identify and classify each use case as Simple, Average, or Complex.
- Assign weight to each use case: Simple (5), Average (10), Complex (15).
- Calculate UUCW by summing the weights:
  UUCW=∑(Number of Use Cases×Weight)UUCW=∑(Number of Use Cases ×Weight)

2. Calculate Unadjusted Actor Weight (UAW)

- Classify each actor as Simple, Average, or Complex.
- Assign weight to each actor: Simple (1), Average (2), Complex (3).
- Calculate UAW by summing the weights:
  UAW=∑(Number of Actors×Weight)UAW=∑(Number of Actors×Weight)

3. Calculate Use Case Points (UCP)

- UCP is the sum of UUCW and UAW, adjusted for technical and environmental factors.
- Technical Complexity Factor (TCF) and Environmental Complexity Factor (ECF) need to be calculated.
- UCP=(UUCW+UAW)×TCF×ECFUCP=(UUCW+UAW)×TCF×ECF

4. Calculate Technical Complexity Factor (TCF)

- Identify and rate technical factors (e.g., distributed system, performance, end-user efficiency).
- Use a scale from 0 to 5 for each factor and sum the total.

- TCF=0.6+(0.01×Sum of Technical Factor ratings)TCF=0.6+(0.01×Sum of T echnical Factor ratings)

5. Calculate Environmental Complexity Factor (ECF)

- Identify and rate environmental factors (e.g., team experience, tools, development schedule).
- Use a scale from 0 to 5 for each factor and sum the total.
- ECF=1.4+(−0.03×Sum of Environmental Factor ratings)ECF=1.4+(−0.03×S um of Environmental Factor ratings)

6. Calculate Effort

- Finally, calculate the total effort using the productivity factor (PF).
- Effort=UCP×PFEffort=UCP×PF
- With PF = 28 hours per use case point, the formula becomes: Effort=UCP×28Effort=UCP×28

b. Unadjusted Use Case Points

Actor	Description	Complexity	Weight
Investor/Guest	A normal user interacting with the site through a graphical interface (GUI)	Complex	3
Site Administrator	Administrator requires a private GUI	Complex	3
Database	System interacts with a database layer through a predefined framework	Average	2
Web Browser	Browser interfaces with the application through an API over HTTP to navigate and submit forms	Simple	1
Finance Adaptor	System interacts with Yahoo Finance through an API	Simple	1

Use Case	Description	Complexity	Weight
Register UC-1	Simple User Interface, 3-4 steps for success, 2 participating actors: Database and Site Admin	Average	10

Use Case	Description	Complexity	Weight
View Data UC-2	Simple user interface, 4-5 steps for a successful scenario, 2 participating actors: Database, Investor/Guest	Complex	15
Manage Portfolio UC-3	Simple User Interface, 3-5 steps for main success, 2 actors: Database, Finance	Complex	10
ML-Language UC-4	Average User Interface, 3-5 steps for main success, 3 actors: Database, Finance, and Database	Complex	15
Admin Action UC-5	Average user interface, 3 steps for main success, participating actors: Database, Site Admin	Simple	5

c. Technical Complexity Factors

Technical Factor	Description	Weight	Perceived Complexity
Distributed System	The system is distributed, with end users having access through the web and main servers.	2	3
System Performance	Users expect good performance but nothing exceptional.	1	4
User Efficiency	End users expect efficiency, but there are no exceptional demands.	1	3
Complex Internal Processing	The system needs to read data from an API and then create a very complex model to predict values.	3	6
Reusability	There are no requirements for the system to be reusable, but adding dates and future data can help.	1	2
Ease of Installation	Installation ease is high because it uses an ML language and might not work well with a database unless the user fills it.	2	3
Ease of Use	Ease of use for users is imperative.	0.5	4
Portability	Portability is sufficient to allow for ease of development on various platforms.	2	5
Ease of Change	The system will only change marginally, so ease of change is a low priority.	1	2
Concurrent Use	Concurrency is an issue because users have access to activity and history feeds, and the system needs to poll finance data in near real-time.	2	5
Security	Security for users is important, but extensive measures are not deemed necessary.	1	2
Third Party Access	Due to the web-interface, third-party support is possible but not currently supported.	1	2

Technical Factor	Description	Weight	Perceived Complexity
Training Requirements	The system is relatively easy to use and includes a ReadMe for guidance.	1	2

d. Environmental Complexity Factors

Environmental Factor	Description	Weight	Perceived Impact
Development Experience	Beginners with UML-based development and the Construction process.	1.5	2
Application Experience	Novices or intermediates in the field of finance.	0.5	0
Paradigm Experience	Beginners or intermediates in the use of databases and web frameworks.	1	2
Lead Capabilities	Leads have some prior experience, but for this kind of project, none.	0.5	0
Motivation	Motivation was high but fluctuates over the semester due to difficulty.	1	3
Stable Requirements	Requirements are somewhat well-known but only approximate (expertise is needed for complex issues).	2	3
Part-Time	All developers are working only a few hours a week.	-1	5
Language	Developers are using a collection of modern languages. Some are more expert than others.	-1	2.5

e. Calculations

1. **Unadjusted Use Case Weight (UUCW):** 55
2. **Revised Unadjusted Use Case Points (UUCP):** 79
3. **Revised Total Use Case Points (UCP):** Approximately 122.80
4. **Revised Estimated Effort:** Approximately 3,438.44 hours

Calculation

- **UUCW** (Unadjusted Use Case Weight) is the sum of the weights of all use cases, calculated as 55.
- **UUCP** is recalculated as the sum of UAW (Unadjusted Actor Weight, 24) and UUCW, resulting in 79.
- **UCP** is then recalculated using the revised UUCP, along with the previously calculated TCF (1.29) and ECF (1.205), yielding approximately 122.80.
- The **Effort** in hours is recalculated by multiplying the revised UCP with the Productivity Factor (PF) of 28 hours per use case point. The resulting effort estimation is approximately 3,438.44 hours.

5. Domain Model

a. Concept Definitions

R1: Allow new user to create an account to partake in stock trading game as an Investor. And Login existing user.
Type: D
Concept: Account Controller

R2: Initialize accounts with fixed amount of capital.
Type: D
Concept: Account Controller

R3: Retrieve information about Stocks, Trades, and Companies.
Type: K
Concept: Yahoo! Finance Adapter

R4: Display information about user's Stocks and Trades.
Type: K
Concept: Investor View

R5: Record and Execute Buy/Sell/Short Stock trades.
Type: D
Concept: System Controller

R6: Display Welcome screen to create an account or log-in.
Type: K
Concept: Login View

R6: Display Welcome screen to create an account or log-in.
Attribute: WelcomeScreen
Concept: Login View

R7: Know if user login failed
Attribute: LoginFailed
Concept: Login View

R8: Investor’s name and OpenID
Attribute: Name/OpenID
Concept: Investor View

R9: Investor Account (Saved Stocks, Predicted Stocks, Forecast)
Attribute: Investor Account
Concept: Investor View

R10: Stocks Investor added to Saved List.
Attribute: Saved Stocks
Concept: Investor View

R11: Know if user is logged in
Attribute: isLoggedIn
Concept: Account Controller

b. Domain Model

The figure (our domain model) depicts a projection of our future final product, showcasing how the system is segmented into various subsystems and their interactions. Each box symbolizes an entity within our system, and each arrow indicates a function enabling communication between these entities. This representation is not of the final demo but rather an envisioned final product.

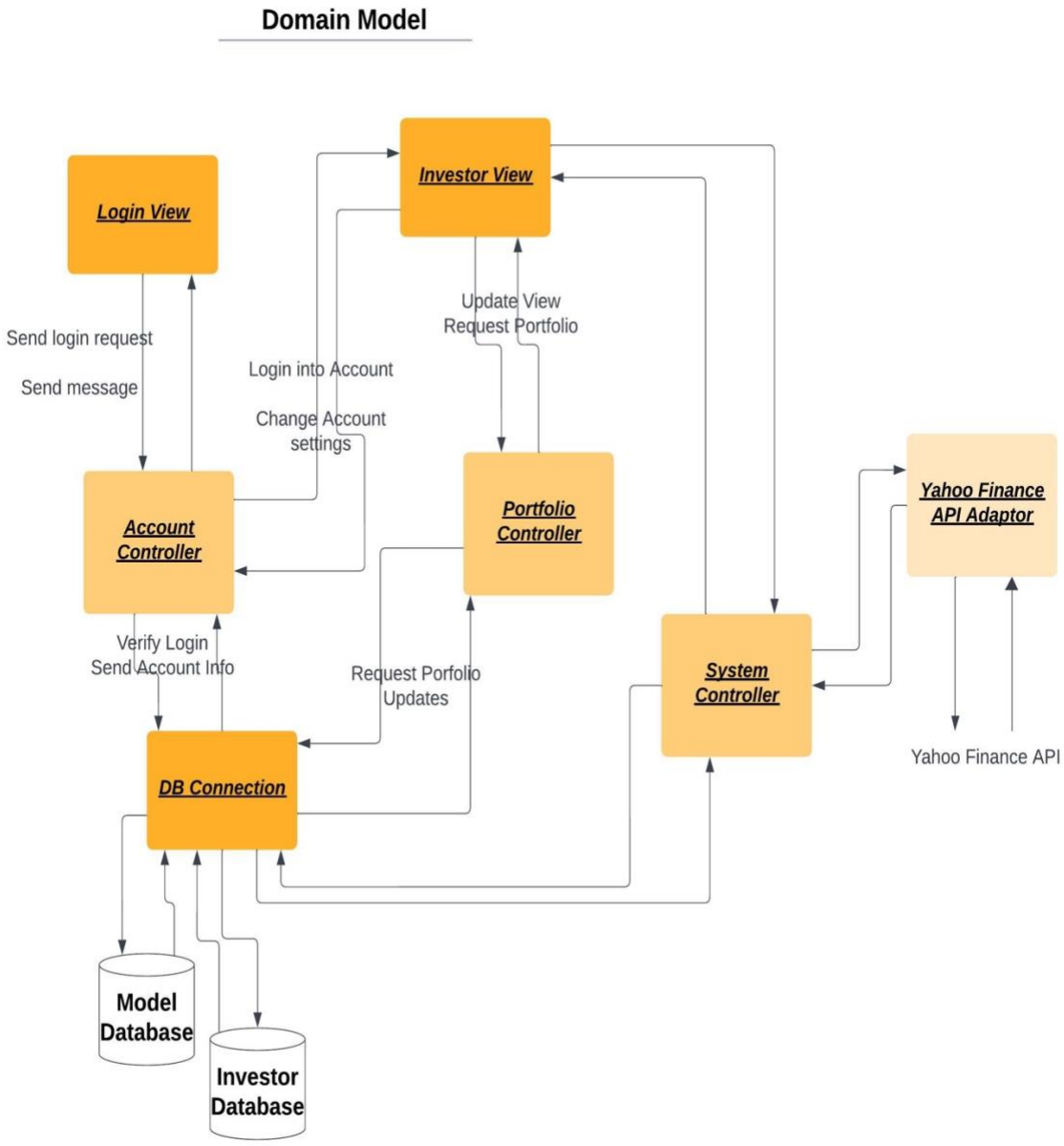


Figure 6.1: Domain Model

Account Controller:

Not Implemented in Final Demo, but the initial action individuals should take when using this system is to establish access by registering and creating an account. The Account Creator serves as an interface enabling users to generate a fresh Investor account. It will first verify whether an account with identical information exists in the database. If no matching account is found, it will then proceed to create the new account and save the provided details in the database.

Investor View:

Not Implemented in Final Demo, but will be for Guests, but The Haystack Group's Investor View feature presents user statistics and stored preferences when a user logs in. The Investor View feature interacts with the Yahoo! Finance Adaptor API to retrieve information regarding the stocks in the user's Saved List and any searched Target stock. The Haystack Group's Investor View feature presents user statistics and stored preferences when a user logs in. The Investor View feature interacts with the Yahoo! Finance Adaptor API to retrieve information regarding the stocks in the user's saved list and any searched Target stock.

Login View:

Not Implemented in Final Demo, but the Login View displays a UI to allow the user to login in. This will send a request to the account controller. If this fails the Login View will be updated to reflect this, if succeeds the user will see the Investor View.

Database Connection:

We aim to implement a unified subsystem that controls access to the database, making the process of retrieving information modular. This approach enables us to expand the application's functionality in the future, accommodating additional features that may also require database access. Importantly, it adds a layer of security by preventing direct database access by individual subsystems.

Furthermore, this unified subsystem provides a standardized interface for interacting with the database, abstracting away the underlying database implementation details. Over time, this design will allow us to seamlessly switch to a different type of database, such as NoSQL, without the need for extensive code refactoring.

Employing a database connection also enhances our system's security by mitigating potential threats, such as SQL injection attacks.

Portfolio Controller:

The Portfolio Controller will update and request updates of the stock the investor/guest picked. This will update the Investor View.

Yahoo Finance Adapter (API):

Our application relies heavily on the nearly real-time stock data provided by the Yahoo! Finance API. Any interruptions or downtime experienced by Yahoo! Finance will directly impact the functionality of our application.

To bridge the gap between the CSV files generated by Yahoo! Finance's API and our application, we have developed the Yahoo! Finance API Adaptor. This Adaptor acts as a translator, converting the data from Yahoo! Finance's spreadsheet format into a syntax that our application can comprehend.

Notably, our Adaptor is designed with modularity in mind, enabling various subsystems within our application to request live stock updates through it.

System Controller:

Any stock selected by an investor will be processed through the Prediction System. This system is responsible for creating a model and predicting the future price of the chosen stock. To achieve this, the Prediction System will communicate with the Yahoo! Finance API Adaptor to obtain the current price of the selected stock, and then make predictions based on the stock's historical data and other relevant factors.

c. Traceability Matrix

Use Case	PW	Account Controller	Portfolio Controller	System Controller	Login View	Investor View	Yahoo Finance API	DB Connection
UC1	18	X			X			
UC2	22	X	X	X		X	X	X
UC3	14	X		X		X		X
UC4	18	X	X	X		X	X	X
UC5	10	X		X		X		X
Max PW		22	22	22	18	22	22	22
Total PW		82	40	64	18	64	40	64

6. Interaction Diagrams

a. Introduction

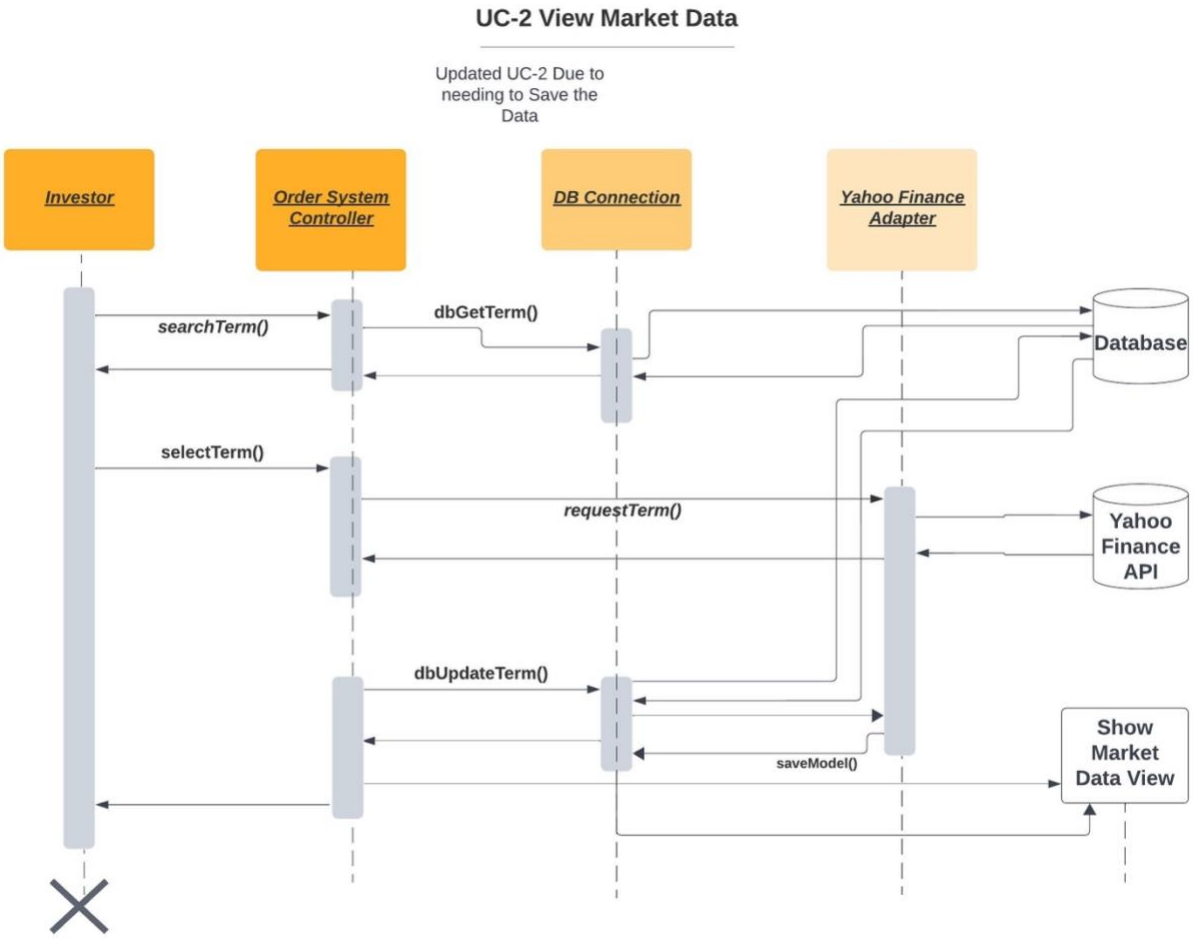
The subsequent section showcases detailed interaction diagrams that map out the essential exchanges within our software architecture. These visuals highlight the critical functions of the Yahoo Finance API, our database systems, and various controllers across different use cases. In the context of our web-based application, the database and controller modules stand out for their pivotal roles in enabling smooth operations. User engagements, such as signing in and retrieving current stock price data from the Yahoo Finance API, are crucial for ensuring the information within the system remains current. The synergy between these elements is foundational to our application's effectiveness.

b. Updated Diagram

UC-2 and UC-4 will be changed due to increase in complexity.
UC-2: View Market Data
UC-4: Pick Stock and ML/Language Prediction

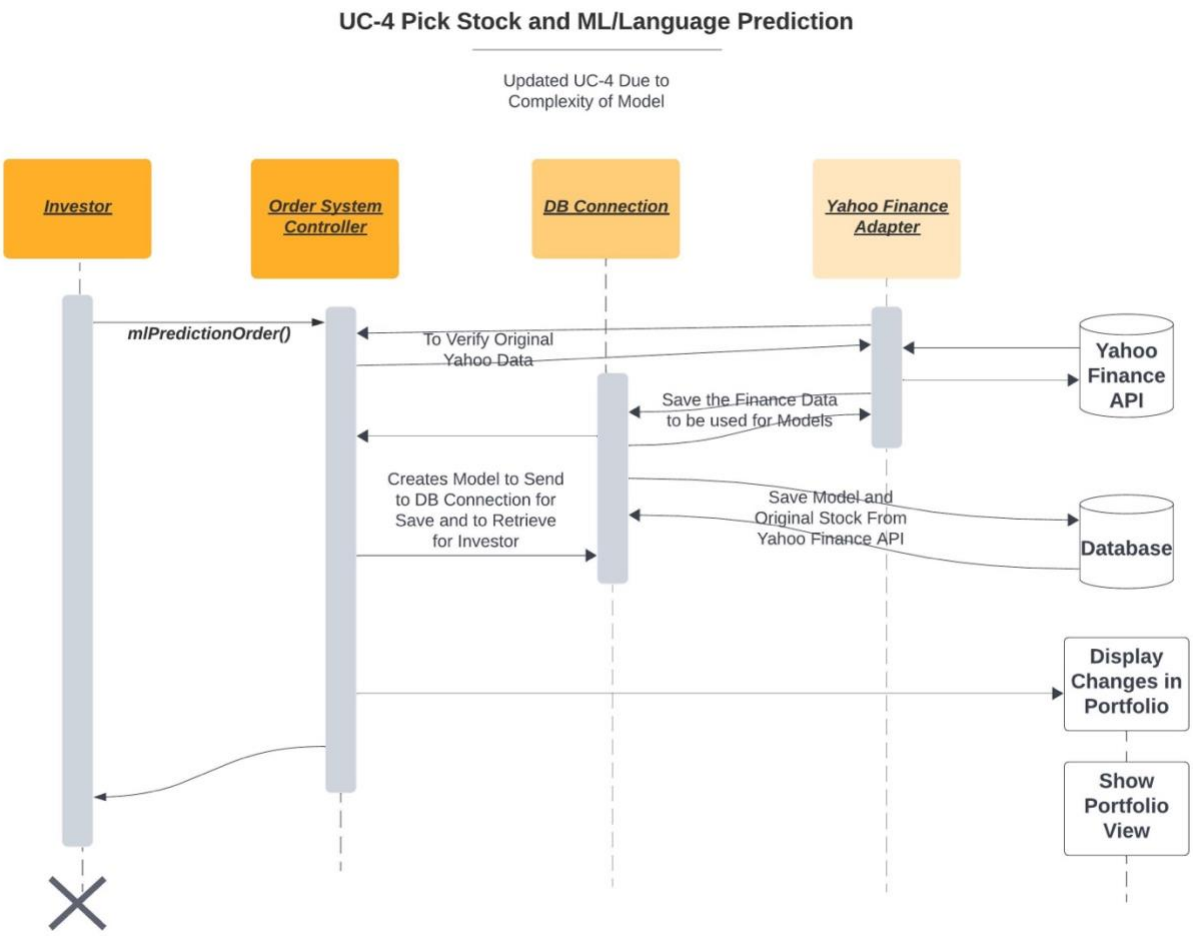
These two use cases are very difficult to implement due to the cost of servers and creating models on instantly. On a local machine it works well due to the ability to use the users' own CPU/RAM/GUI vs a cost on a server which requires higher end due to RAM limitations.

Use Case 2: View Market Data



The updated View Market Data is used to Save the Yahoo Finance API data for the user for future use. This is needed due to the complexity of the model and the cost of running it in real time for a server.

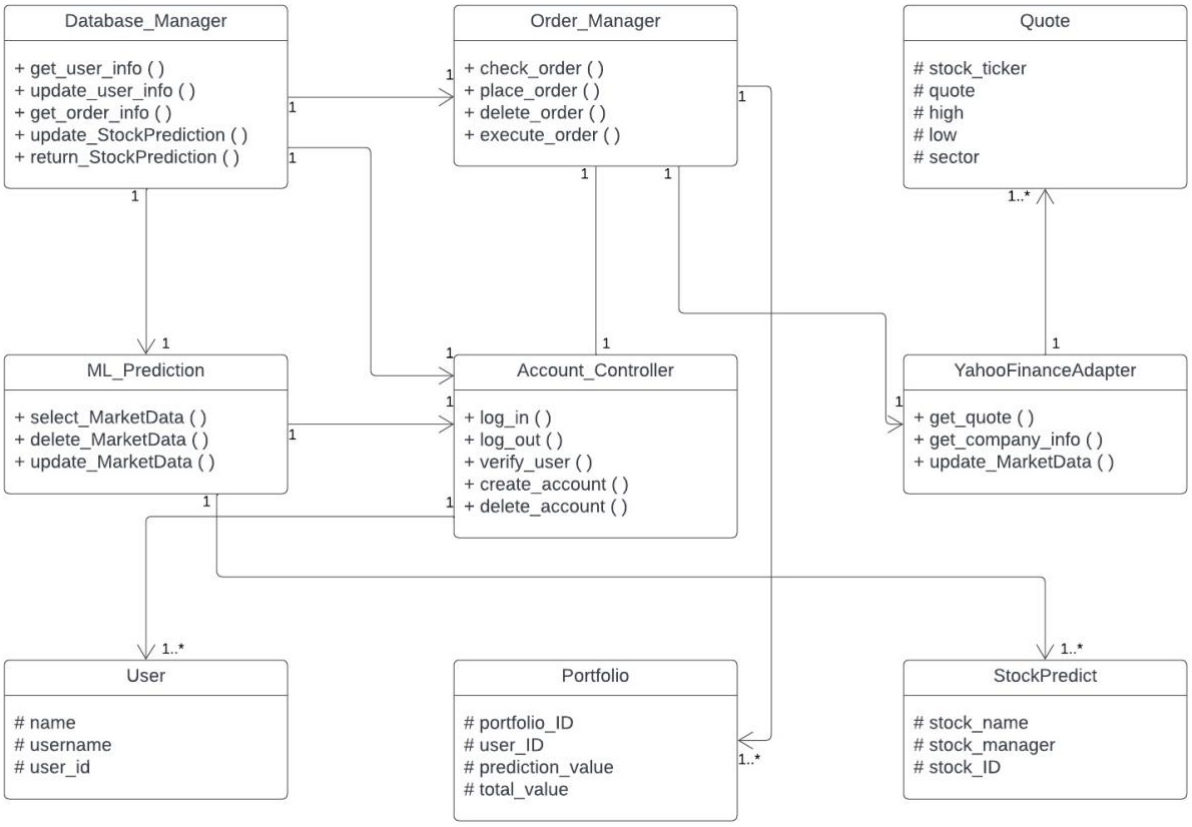
Use Case 4:



UC-4: Pick Stock and ML/Language Prediction needed to be updated due to the need of saving a model vs running the model in real time on a server. This change was necessary to save on costs of running the software as a web application. If have plenty of resources it is possible to run the forecast(prediction of stock prices) in real time.

7. Class Diagram and Interface Specification

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.

c. Design Patterns

1. Singleton Pattern for Database_Manager and YahooFinanceAdapter:
    - Use Case: These classes seem like they should have only one instance in the system to manage database interactions and financial data retrieval, respectively.
    - Benefits: Ensures a single point of access to the database and the Yahoo Finance service, reducing the overhead of multiple instances and ensuring data consistency.

2. Factory Method for User and OrderInfo Creation in Account Controller and Order_Manager:
    - Use Case: The creation of User and OrderInfo objects seems to be a complex process, and may vary depending on the context.
    - Benefits: Allows for more flexible object creation. By encapsulating the creation process, it's easier to manage and modify how objects are instantiated.
3. Strategy Pattern for ML_Prediction:
    - Use Case: This class might need different algorithms for market data prediction, which could change dynamically.
    - Benefits: Enables the ML_Prediction class to switch between different prediction algorithms at runtime, providing flexibility and making it easy to introduce new algorithms.

4. Observer Pattern for StockPrediction Updates:
    - Use Case: When a stock prediction is updated in Database_Manager, there might be several other components (like Portfolio) that need to be notified.
    - Benefits: Allows for automatic update propagation. When a StockPrediction changes, all dependent objects are automatically informed and updated.

5. Decorator Pattern for Extending Quote Functionality in YahooFinanceAdapter:
    - Use Case: Additional features might be needed for a Quote, like adding historical data or analyst ratings without altering the existing Quote class.
    - Benefits: Provides a flexible way to add functionalities to objects dynamically. The Quote object's behavior can be extended at runtime by wrapping it with decorator classes.

6. Command Pattern for Order_Manager Operations:
    - Use Case: Operations like placing, deleting, and executing orders involve encapsulating a request as an object.
    - Benefits: Allows for parameterizing clients with different requests, queueing requests, and implementing undoable operations. It also decouples the class that invokes the operation from the one that knows how to perform it.

7. Adapter Pattern for YahooFinanceAdapter:
    - Use Case: To interface with the Yahoo Finance API, which might have a different interface than what your system expects.
    - Benefits: Converts the interface of a class into another interface that the client expects. YahooFinanceAdapter makes the Yahoo Finance API compatible with your system.

8. Facade Pattern for Simplifying Client Interaction:
    - Use Case: To provide a simplified interface to a complex subsystem, such as a combination of Database_Manager, Order_Manager, and ML_Prediction.
    - Benefits: Provides a unified interface to a set of interfaces in a subsystem, making the subsystem easier to use.

d. Object Constraint Language

1. Database_Manager

- Invariant: The database connection must always be active when an operation is called.

OCL: context Database_Manager

inv: self.databaseConnection.isActive()

- Precondition for get_user_info(user_id: int): The user_id must be valid and exist in the database.

OCL: context Database_Manager::get_user_info(user_id: int)

pre: user_id > 0 and self.existsUser(user_id)

- Postcondition for update_user_info(user_id: int, new_info: UserInfo): The user's information must be updated in the database.

OCL: context Database_Manager::update_user_info(user_id: int, new_info: UserInfo)

post: self.getUserInfo(user_id) = new_info

2. Order_Manager

- Precondition for place_order(order: OrderInfo): The order must not be null and must have valid data.

OCL: context Order_Manager::place_order(order: OrderInfo)

pre: order <> null and order.isValid()

- Postcondition for delete_order(order_id: int): The order with the given order_id should no longer exist.

OCL: context Order_Manager::delete_order(order_id: int)

post: not self.existsOrder(order_id)

3. ML_Prediction

- Invariant: The prediction model must be initialized before any prediction is made.

OCL: context ML_Prediction

inv: self.model.isInitialized()

- Postcondition for update_MarketData(data_id: int, new_data: MarketData): The market data must be updated accordingly.

OCL: context ML_Prediction::update_MarketData(data_id: int, new_data: MarketData)

post: self.getMarketData(data_id) = new_data

4. Account Controller

- Precondition for log_in(username: string, password: string): The username and password must not be empty.

OCL: context Account_Controller::log_in(username: string, password: string)

pre: username.size() > 0 and password.size() > 0

- Postcondition for create_account(user_info: UserInfo): A new user account must be created.

OCL: context Account_Controller::create_account(user_info: UserInfo)

post: self.existsUser(user_info.user_id)

5. YahooFinanceAdapter

- Precondition for get_quote(symbol: string): The stock symbol must be valid and not empty.

OCL: context YahooFinanceAdapter::get_quote(symbol: string)

pre: symbol.size() > 0 and self.isValidSymbol(symbol)

- Postcondition for update_MarketData(data: MarketData): The market data in the system should be updated.

OCL: context YahooFinanceAdapter::update_MarketData(data: MarketData)

post: self.getMarketData(data.data_id) = data

8. 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.

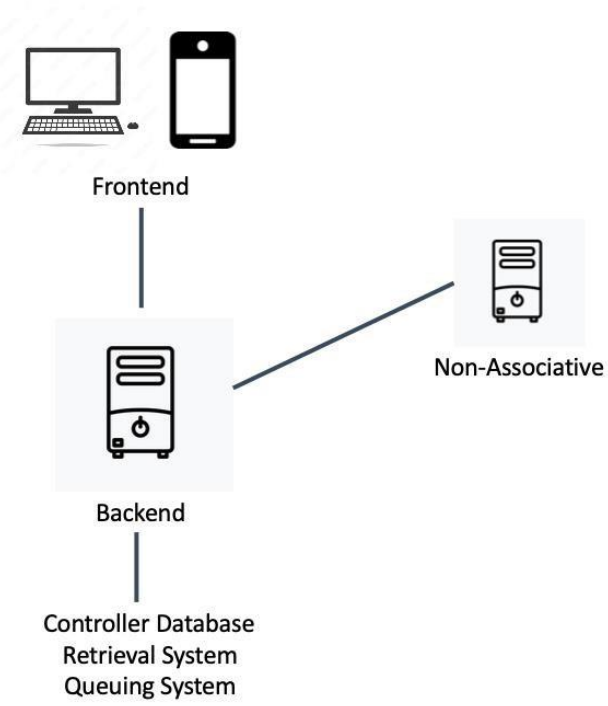


Figure 3.1: Diagram of the network protocol.

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.

9. User Interface Design and Implementation

a. Updated Progress

As of the latest update, there hasn't been significant progress from the initial user interface (UI) mockups presented in the first report. However, we are preparing for minor yet impactful changes to the final website, based on upcoming user interaction studies. Minor adjustments are also anticipated for the initial demonstration, though these are not yet executed or finalized.

Enhancing Site Efficiency

Our priority is ensuring that the website performs efficiently for all users, irrespective of their device or internet connection. Our strategy includes:

- **Client-Side Caching**: Breaking down the website to enable caching of elements like headers that remain constant across the site. This approach minimizes repeated loading.
- **HTML and CSS Emphasis**: To reduce load, the UI relies heavily on HTML and CSS, limiting image use. Images used are optimized in size and format.
- **Responsive Design**: Using frameworks like Twitter Bootstrap CSS to ensure the application is accessible across various devices, including mobiles, tablets, and desktops.

We acknowledge the trade-off in not supporting older browsers incompatible with HTML5, CSS3, or modern web standards. However, this is a calculated decision, as our target audience primarily uses modern browsers.

b. **User Interface Specification**

Preliminary Design

The ML-Stock Prediction UI, serving as a “command center,” will provide users with a comprehensive view of stock predictions and current prices. Key design considerations include:

- **Lightweight Design**: Ensuring the UI is not cumbersome for mobile and tablet users.
- **Color Scheme**: A palette of grey, black, blue, white, and possibly red, focusing on pastel and web-friendly colors.

- **Framework Utilization**: Building on open-source frameworks, with Bootstrap CSS as a potential choice, to cater to desktop, mobile, and tablet browsers.

Landing Page and Login

- **Goal**: Facilitate user engagement by requiring registration or using a zero-effort registration system in the future (e.g., Google account login).
- **Features**: Displaying a random graph showing historical data and future stock predictions.

Global Header

- **Persistence**: The header remains consistent across the website once the user logs in.
- **Navigation Views**: Offering 2-3 views, including User Information, Stock Selection, and Analyze/Predict Stock.

c. Integration of Progress and Specification

The integration of the updated progress and preliminary design specifications into the ML-Stock Prediction project aims to create a user-friendly and efficient interface. The focus on lightweight design, strategic caching, and responsive layout caters to a broad user base. The adoption of modern web standards, despite the trade-off of excluding older browsers, aligns with our target audience's technological preferences. This approach, combined with user feedback from upcoming studies, will guide the refinement of our UI to best serve the needs of our users in accessing and analyzing stock prediction information.

Enhanced Use Case Integration with User Interface

Use Case UC-4: Pick Stock and ML-Language Prediction

Enhanced Flow with UI Integration

1. **Stock Selection and Order Placement:**

    - UI Feature: A dedicated section in the "command center" UI allows the Investor to select stocks. This is facilitated by an intuitive search and selection interface.
    - Interaction: The Investor picks a stock through a dropdown or search bar, then places a market order using a simple form.
2. **Data Request to Yahoo! Finance:**
    - UI Indication: While the system sends the request, a loading indicator or a temporary message appears in the UI, enhancing user experience by providing feedback on the system's status.
3. **Data Reception:**
    - UI Update: Once the data is received, the UI updates to show the latest stock data, including ML-based predictions, in an easy-to-read format, possibly using graphs or tables.
4. **Record Order:**
    - UI Confirmation: After the order is recorded in the Database, a confirmation message or notification is displayed to the Investor, ensuring them of the successful transaction.
5. **Display Portfolio Changes:**
    - UI Portfolio Update: Changes in the Investor's portfolio are dynamically displayed in the UI, providing real-time feedback (to be implemented in future versions).

Use Case UC-2: View Market Data

Enhanced Flow with UI Integration

1. **Initiating Stock Search:**
    - UI Feature: A search bar in the global header or a dedicated section in the UI for stock search.
    - Interaction: The Investor inputs the stock symbol or keywords, and the UI dynamically suggests symbols as they type.
2. **Database Query:**
    - UI Indication: A brief loading symbol or message while the system processes the request.
3. **Suggested Symbols Retrieval:**
    - UI Display: The suggested stock symbols are displayed in a dropdown or a list format beneath the search bar.
4. **Display of Suggestions:**
    - UI Interaction: The Investor can scroll through the suggestions and select one, with the UI responding fluidly to the selection.

5. **Symbol Selection:**
    - UI Feedback: Once a stock symbol is selected, the UI transitions to a detailed view of the stock, including charts and data.
6. **Data Request:**
    - UI Indication: Similar to UC-4, a loading indicator is shown while the system fetches data.
7. **Data Update and Integration:**
    - UI Update: The UI updates to show real-time market data and ML-based predictions, integrating visuals and data in a comprehensible manner.
8. **Investor Presentation:**
    - UI Final Display: The aggregated data is presented in a user-friendly layout, with options to delve deeper into specific data points or return to the search.

Overall Integration with User Interface

The integration of these use cases with the user interface is vital for providing a seamless and intuitive experience. The UI design will cater to the needs of the Investors, allowing them to make informed decisions with ease and precision. By combining efficient data presentation, responsive design, and clear navigational elements, the ML-Stock Prediction platform is poised to offer a robust and user-centric solution for stock market analysis and trading.

Note on Feature Availability for Demo 2

Limited Feature Set for Upcoming Demonstration

As we progress towards Demo 2, it's important to note that not all the features outlined in the use cases and their integration with the user interface will be fully ready or implemented at this stage. Specifically:

- **Use Case UC-4 (Pick Stock and ML-Language Prediction)** and **Use Case UC-2 (View Market Data)** will have limited functionality in the upcoming demonstration.
- Certain advanced UI elements, particularly those involving dynamic data presentation and real-time updates, may not be fully operational.
- The integration of ML-based predictions and real-time market data from Yahoo! Finance will be in a preliminary phase, with more comprehensive functionality to be expected in future versions.

Focus for Demo 2

For Demo 2, our focus will be on showcasing the core capabilities and the foundational framework of the ML-Stock Prediction platform. This includes:

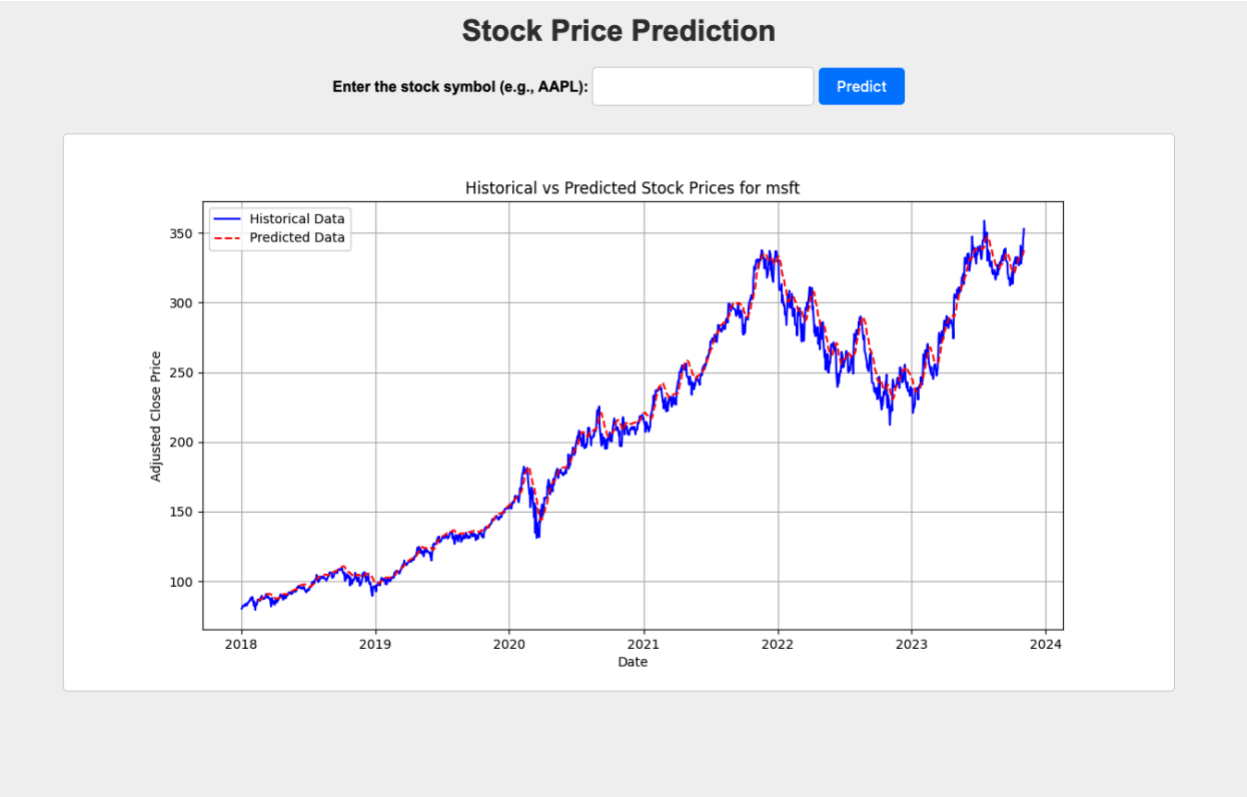
- Basic stock selection and order placement processes.
- Initial implementation of the search functionality for stock data.
- A primary version of the user interface, emphasizing layout and basic interactivity.

Future Development and Full Implementation

Post-Demo 2, our team will continue to refine and enhance the platform, working towards the full implementation of all features. We will integrate user feedback from Demo 2 to improve the platform's functionality, usability, and overall user experience.

This phased approach ensures that we prioritize quality and stability, gradually building towards a comprehensive and robust trading platform. We appreciate your understanding and look forward to presenting our progress in Demo 2.

The following Screenshot is the basic look of the site allowing the user to select a stock and see the price and prediction model.



10.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.

e. Updated Tests for ML_Prediction Class

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.

These updated test cases are crucial for verifying the functionality and reliability of the ML_Prediction class within the ML-Stock Prediction platform. They ensure that the platform's core capabilities related to market data management are robust and error-free.

11. **History of Work, Current Status and Future Work**

a. History of Work, Current Status, and Future Work

Work History: The Haystack Group embarked on a venture to develop a web application for stock prediction using machine learning, an endeavor that was challenging to execute within the desired timeframe. The challenge stemmed from the team's expertise and the decision to minimize substantial financial expenditures for the application's operation.

The project commenced with an exploration and understanding of various stock prediction models, leading to the development of models based on these insights. Once a suitable model was established, the team crafted a local application that displayed both the actual and predicted stock prices using historical data obtained from the Yahoo Finance API. The application visually represented the data, aligning two distinct lines on a graph. Initial issues arose when an unexpected stock market rally in October 2023 caused a deviation in the predictions. Upon correcting for this anomaly, the team proceeded with selecting the date range for stock predictions, choosing start dates in either 2018 or 2020. This range ensured the inclusion of companies with a substantial historical record to bolster the model's accuracy. However, when transitioning the local application to a web platform, the team recognized the need to refine the handling of the model and its predictions.

Current Project Status: The primary challenge currently faced involves the preservation and retrieval of models within a database for accurate stock prediction. Issues occur when models are inadvertently overwritten by user actions, resulting in inaccurate forecasts. The team is also working on presenting forecasts for various future intervals, including 7, 30, 60, 90, and 180 days. While nearing a solution for the issue related to array dimensions, this remains a work in progress.

Future Directions: Moving forward, the intention is to implement a user login system to enable stock selection for prediction and potentially allow portfolio creation. The team aspires to migrate from reliance on local resources to a cost-effective, deployable web application. The goal is to integrate real-time trading data and predictive modeling through an alternative API, enhancing the application's utility for investors.

b. Roadmap

Haystack Group Roadmap:

Completion of Future Stock Prediction: Targeted for November 24/25. The deliverable will include a locally executable application accompanied by a detailed README for user guidance, available for download on GitHub.

Demo 2: The team aims to finalize the feature for error-free model saving and retrieval by the date. Additionally, if resources permit, we will endeavor to deploy a web application to offer guests a trial experience of the stock prediction functionality.

The Haystack Group's roadmap has evolved in response to the specialized expertise required for this application's development. Despite significant obstacles, including learning new programming languages, mastering APIs, and understanding financial terminology, the team is proud of its progress to date. The strategic shift to focus on accurate forecasting and reliable model saving and retrieval has been pivotal. Without these features, the web application would merely replicate basic functionalities available through the Yahoo Finance API. The team prioritizes offering guests the ability to utilize the predictive and forecasting features on a local machine, over the development of a user login system and a web application that simply displays stock data graphically.

References

1. https://bitflyer.com/en-jp/s/regulations/security
2. https://getbootstrap.com/2.0.2/
3. https://flask.palletsprojects.com/en/3.0.x/
4. https://www.geeksforgeeks.org/stock-price-prediction-using-machine-learning-in-python/#