TTT Stock Market League

Report 2
Software Engineering
14:332:452

Group 4:

Albert Joshua Capistrano -- ac1695
Yong Ho Danny Lee -- ydl7
Ajeet Malhotra -- am2140
Raffay Khan -- rnk38
Antoni Chrobot -- ac1733
Ping Lin -- pyl12
Christopher Young -- cy258
Aryeh Ness -- akn45

Project Repository - https://github.com/amalhotr/SMIL_project
Report Repository - https://github.com/amalhotr/SMIL_report

March 3, 2019

Revision History:

Version Number	**Date**
V.1	03/3/2019
V.2	3/10/2019
V.3	3/17/2019

Project Management:

Category	Albert Joshua C.	Yong Ho Danny L.	Ajeet M.	Raffay K.	Antoni C.	Ping L.	Chris Y.	Aryeh N.
System Interaction Diagram	20%	20%	20%		20%	20%		
Class Diagram				100%				
Data Types and Operation Signatures		10%		90%				
Traceability Matrix		100%						
Architectural Design			75%		25%			
Identifying Subsystems			100%					
Mapping Subsystems to Hardware								100%
Persistent Data Storage						100%		
Network Protocol			100%					
Global Control Flow		50%						50%
Hardware Requirement	100%							
Algorithms and Data Structures			100%					
User Interface Design and Implementation			100%					
Test Cases	25%				75%			
Test Coverage					100%			
Integration Testing						100%		
Plan of Work	25%	50%		25%				
Project Management	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%	12.5%

Contents

1 System Interaction Diagrams

1.1 Introduction

The following sequence diagrams will display the framework of a user's interaction with the program's key use cases. For each of the listed use cases, we have outlined steps that are taken, and analyzed the different scenarios possible when a user is initiating an action. This will show how the system will be handling the success and failure cases. Since our software is mostly web-based, we will be utilizing the database and its controller very frequently. The following diagrams will precisely plan how this will be achieved within the software of our system.

1.2 Sequence Diagrams

Use Case 3: Join League

Figure 4.1 displays the sequence diagram when joining a stock market league as a registered user, acting as a Player. The Player initiates the action by requesting a list of public leagues. The DatabaseConnect facilitates this request by fetching the available leagues from the database and returning them to the Player. The Player will then select a league to join and call joinLeague, passing the LeagueID and PlayerID to the LeagueController. The LeagueController will then verify that the LeagueID refers to a valid public league. Upon verification, the LeagueController will call on the DatabaseConnect to add the PlayerID to the list of members of the given LeagueID. The LeagueController will also call the LeagueManager to add the given PlayerID to it's list of league members.

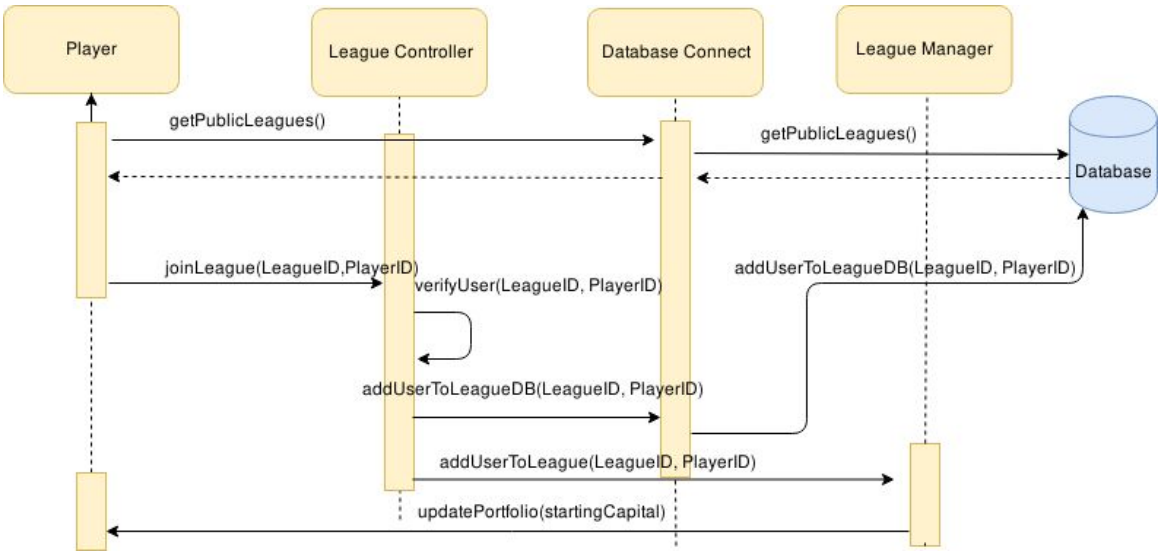


Figure 4.1: Use Case 3: Join League

Use Case 4: Research Stock

Figure 4.2 displays the sequence diagram when researching a stock, and adding it to a personal watchlist. The sequence begins when the player initiates the researchStock in the trade view. The trade view then sends a request to the transaction controller to provide information for the desired stock. Once the transaction controller pulls the information from the finance API, it returns the information to the trade view for the player to see. If the player chooses to add the stock to their watchlist, the trade view sends the request to the transaction controller. The transaction controller makes a database connection to store the stock into the users watchlist.

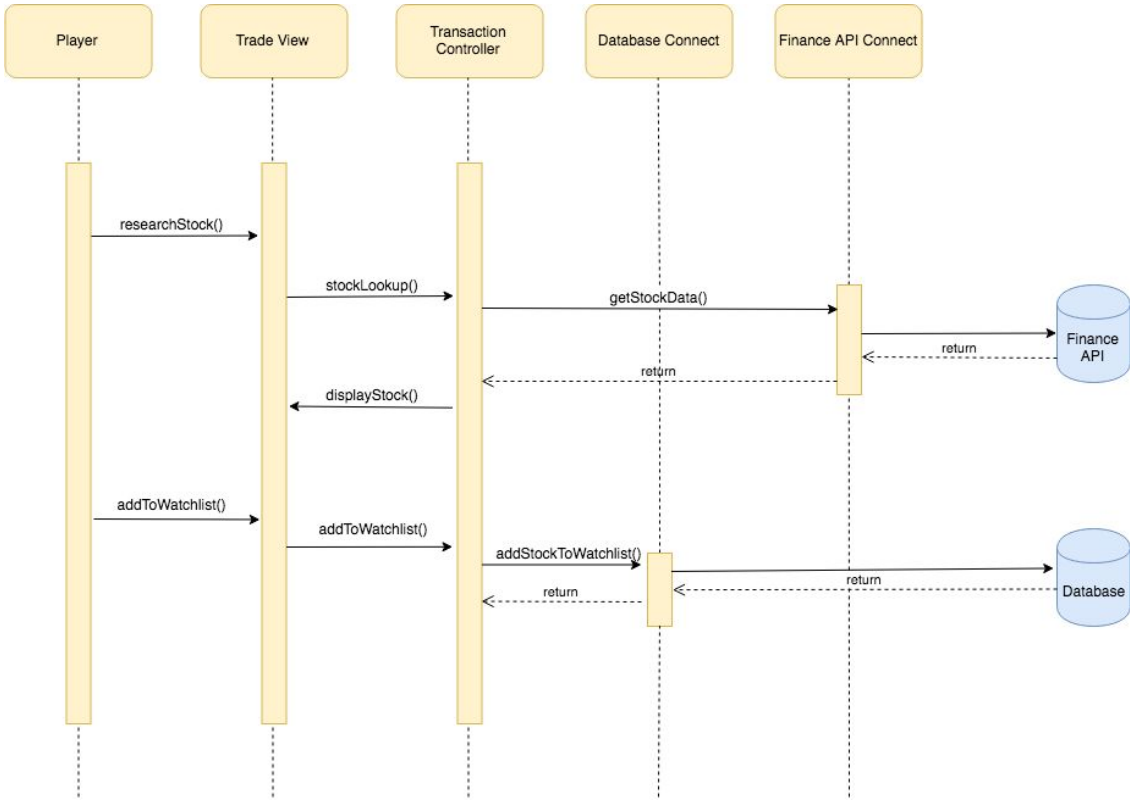


Figure 4.2 Use Case 4: Research Stocks

Use Case 5: Buy/Sell Stocks

Figure 4.3 displays the sequence diagram for when a user is buying/selling a stock. When a user attempts to buy/sell a stock, the Transaction controller will receive a transaction request consisting of the stock symbol and the amount of that stock. The Transaction Controller will get information about that stock through the Finance API and update the user's stock portfolio and currency balance to the database through the function "updateSetting()." Afterwards, it will display the user's current portfolio and balance to the user.

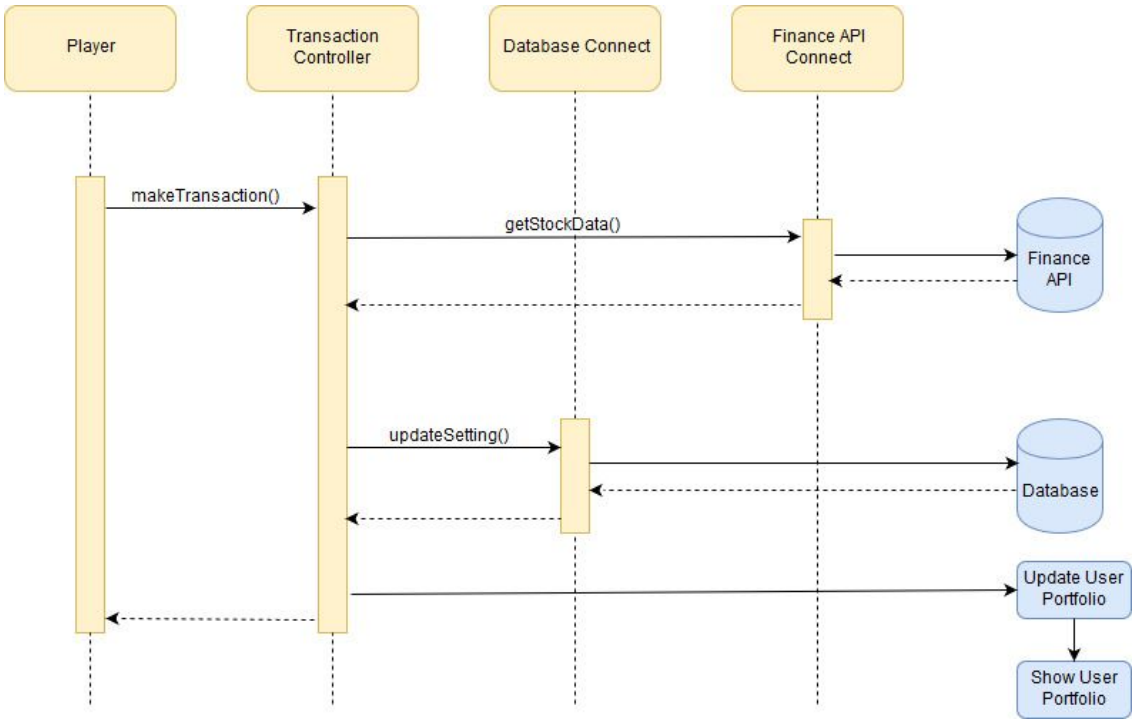


Figure 4.3 Use Case 5: Buy/Sell Stocks

Use Case 7: Create League

Figure 4.4 displays the sequence diagram when creating a stock market league. When a user attempts to create their own league the league controller will be informed and get involved in the process. In order to create a new public or private stock market league, the program must update the “Profile VIew” setting of the user initiating the action. Once this is done, a function, called UpdateSetting() which simply updates a user’s profile, is called and will lead to creating the league and connecting to the database as well, done through the database connection.

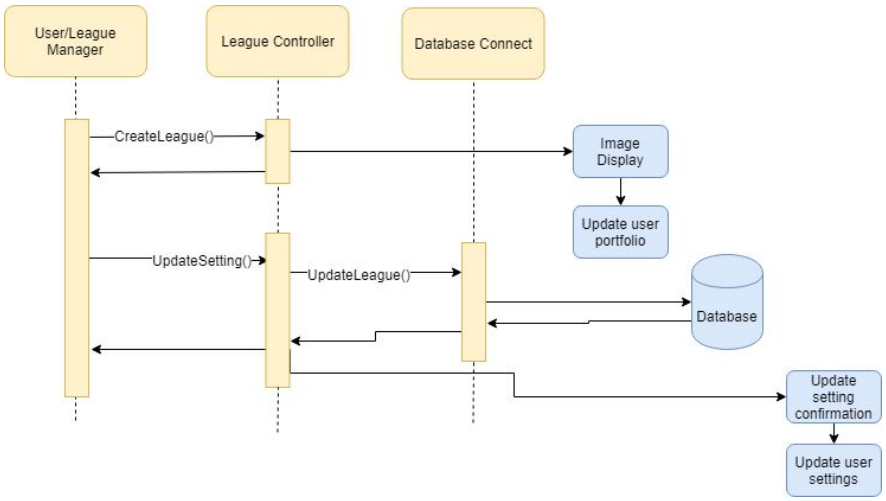


Figure 4.4 Use Case 7: Create League

Use Case 8: Reinvest Dividends

Figure 4.5 displays the sequence diagram when the player chooses to reinvest the dividends of a stock currently in the player’s portfolio. When the player is viewing a stock on the home view, the player has the option to click on the “Reinvest dividends” button. The clicking of this button initializes the use case. When the button is clicked, the Transaction Controller will receive a request that involves stock ticker, number of shares owned, and reinvestment dividends option. The Transaction Controller will then call the Finance API to obtain relevant information on the stock. After this process is completed, a pop-up on the screen will ask the player to confirm the reinvestment request. If the player declines the request, the use case is terminated. If the player confirms the request, the database and the player’s portfolio will be updated. The

“Reinvest dividends” button will turn green at the end of the use case.

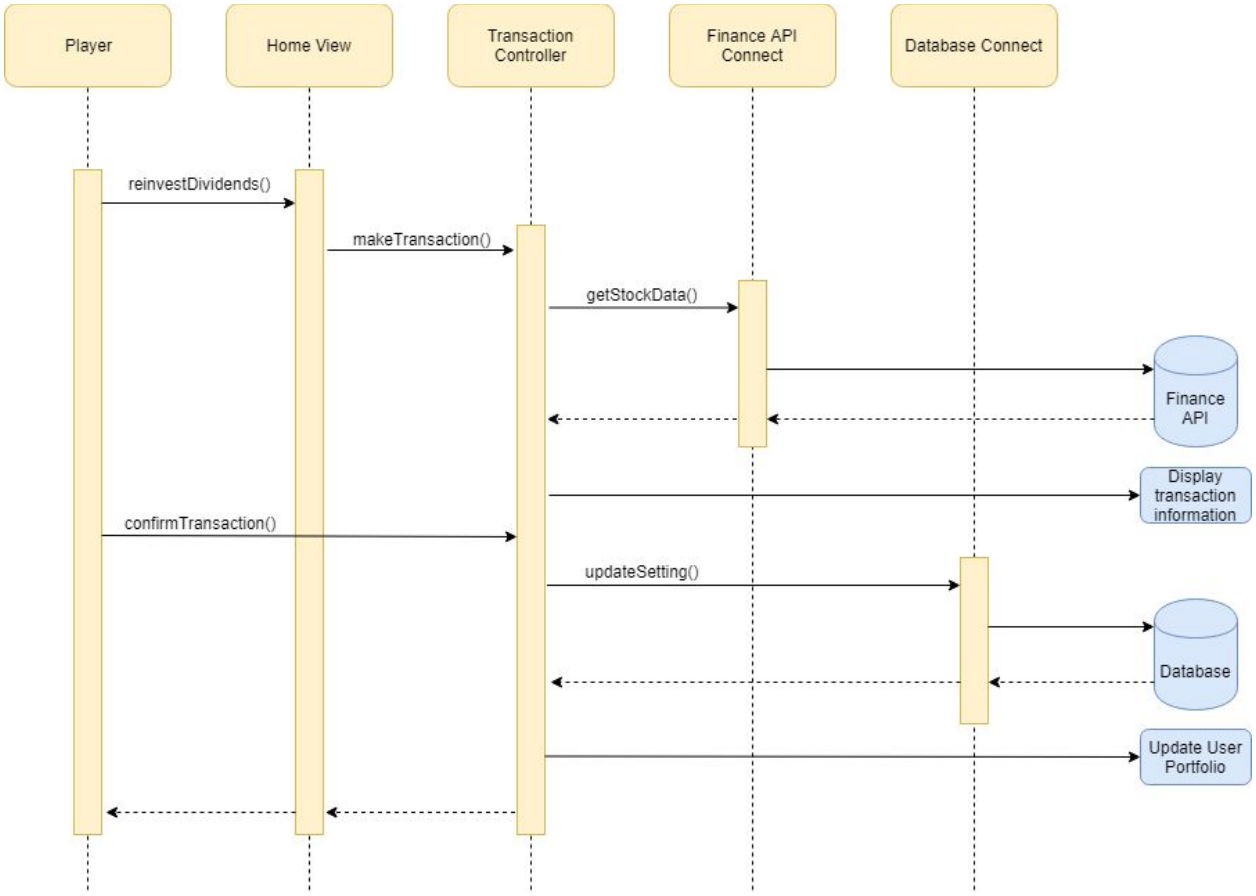
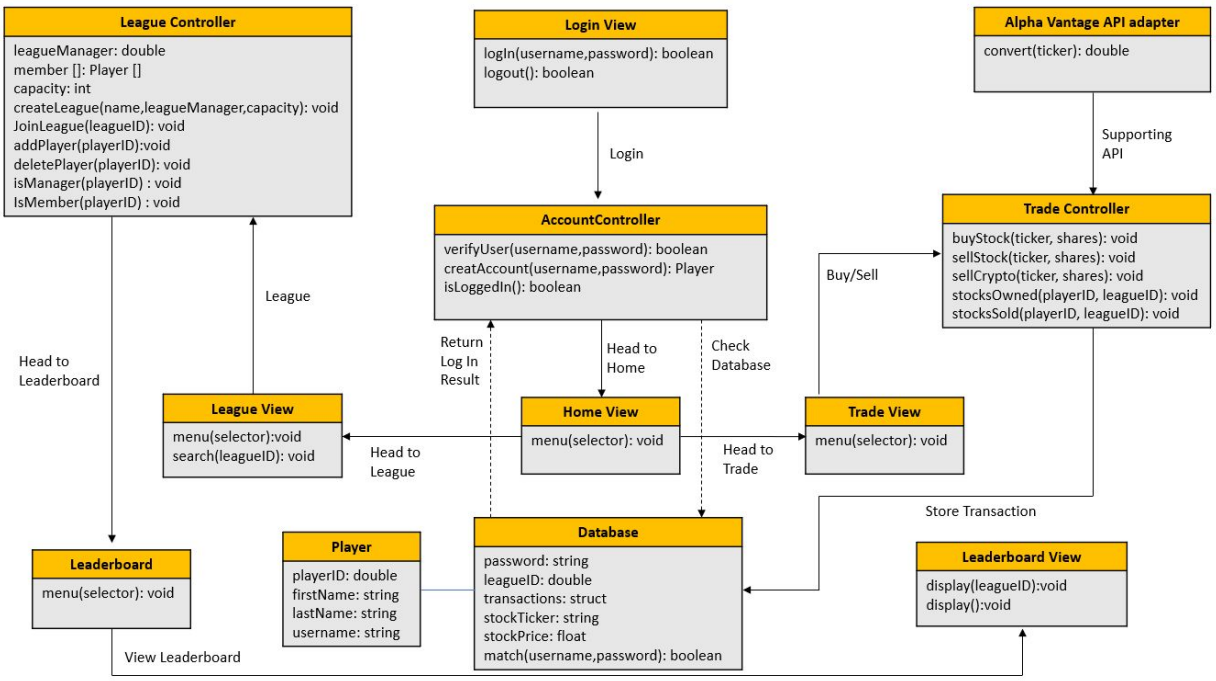


Figure 4.5 Use Case 8: Reinvest Dividends

2. Class Diagram and Interface Specification

A. Class Diagram



B. Data Types and Operation Signatures

LoginView: This class is used to represent the main screen the players will be greeted with when opening the application. The login() function will simply take a players username and password and allow them to login. It will return a boolean true or false depending on whether the AccountController class is able to verify the user. Additionally, this menu will be able to allow a user to logout if he is already logged in using the logout() function. If the user has no account, a "No_Account" button will transfer the user over to the AccountController class.

AccountController: The AccountController class is essentially the middle man in the log in process. The verifyUser function will take in a username and password and forward it along to the Database class. The Database class will confirm the authenticity of the username and password combination and return a boolean value that the verifyUser function will return. A false flag will prompt the user that the login information was invalid and a true flag will send the user to the "Home View". The creatAccount function is how a player is able to create his player account for the first time. All a user must do is provide a username and password and the AccountController will send it to the database for storage. The database also automatically

generates a playerID via this function for simpler mapping. Finally, isLoggedIn() is a simple interrupt function to validate that the user is logged in when first coming in from login view. If they are not, they will be prompted to head back to Login View to login or to creatAccount.

Home_View: Home View is the "main menu" class of our program. It lays out the map and functionality of our application. Through the menu function, one is able to select whether to head to the League View screen or the Trade View screen. The selector input will be based on a button click on our graphical user interface.

Trade_View: The Trade View class presents a menu on what further transactions a user would want to make based upon his current portfolio. The menu() function will allow a user to buy stick, sell stock, sell crypto, and view their portfolio. Clicking any of the following buttons will call their respective function from the Trade Controller.

League_View: The League View class presents a menu on selecting leagues you are currently a member of or searching for a new league. This is accomplished through it's menu function and it will forward the user to the League controller. The search function on the other hand will search for a specific league based upon a leagueID. This is found via communicating with database which stores leagueIDs as the data type double. Once the corresponding league ID is found, the League View class will forward you to the League Controller. If the league could not be found, the system will display a message to re-input the ID as it was not valid.

Database: The database class stores the applications most pertinent data. A user's password is stored as "password" under the data type string. An additional attribute stored is the array of leagueIDs stored as the data type double. This allows the league to match any leagueID to an existing LeagueController class. The next attribute we have is transactions and this in the form of a data structure. The reason being is that this will contain the transaction history of every playerID on the application. The transaction history will be a string array matched to a playerID array. The stockTicker Attribute stores what ticker is attributed to what stock. This is of data type string. Additionally, the stockPrice attribute defines the price of a specific stock as data type float. This stockPrice will be matched with a specific stock ticker.
The match function is what the database will use to log a user in when a signal is forwarded from the AccountController. It will use a given username and password, see whether they match via a hashing algorithm, and return a boolean signal that will be forwarded back to the AccountController. Evidently, the value will be true if the username and password matched and false if there was no match.

Player: Player is a subclass of the superclass Database. It stores all of the player relevant data. The playerID is exactly how it is labeled, the ID of the player. This is stored in the data

type of a double. The following attributes also function exactly as labeled. firstName stores a players first name as a string. The attribute lastName stores a players last name as a string. Finally, username stores the players username as a string.

League_Controller: The LeagueController class is primarily the heart of the league aspect of the application. Each LeagueController class is associated with a leagueID which can be found in the Database. The first attribute the LeagueController is leagueManager. This is a double which indicates a playerID that is the manager of the league. The "member" attribute is an array of data type Player which indicates all of the players that are currently in the league. The integer type attribute capacity refers to the maximum amount of players allowed in a certain league. For example is the capacity was set to 5, the league would not be able to hold more than 5 players. Moving on to the functions present in the class, the function createLeague does exactly as described an creates a league. It does this by taking in a league name, a league manager, and capacity value and storing it to the database. The function join league will take the users playerID and associate it with an inputed leagueID, thus essentially joining an existing league. The addPlayer function allows a league manager to directly add a player to the league using their playerID. This allows the player to join the league without even having to search for it on their own. The deletePlayer function likewise will remove a player from the league based on their playerID on the managers behest. LeagueController also has two utility function to make sure that the right permissions are given to the right users. isManager checks for whether the player is a manager of the league and isMember checks for whether the player is a member of the league. Both utilize the playerID to make this conclusion.

Leaderboard: The Leaderboard is the menu where a user can select to view the leaderboards. This is done through the menu function which takes a button press input called selector. Depending on the button press, the the class will call for different functions found in the Leaderboard_View class. A Global button will call for display() while a League button will call for display(leagueID).

Leaderboard_View: The Leaderboard View class is designed to output the standings of the league of global standings amongst every player on the platform. The display function will take in either a leagueID input or no input. If it is provided a leagueID input, it will output and display the leaderboard standings of the leagueID passed. If the display function is provided no input, the default case is for it to display the global leaderboard standings of every player on the application.

Trade_Controller: The Trade Controller class handles all of the financial transactions of the application. The function buyStock allows a user to buy stocks based upon an inputted ticker and the amount of shares. It will also check the current capital of the player to ensure the

transaction is valid. Our buyStock function will additionally allow a user to purchase cryptocurrency as we are simply assigning unique tickers to them to allow the use of the same function for multiple purposes. The function sellStock does exactly as it is titled and sells shares of stock depending on the specified amount of shares and ticker. It also checks the portfolio of the player to ensure that the number of shares of stocks involved in the transactions are provided in a sufficient amount in the portfolio. sellCrypto will sell cryptocurrency based upon the ticker (type of cryptocurrency) and shares (which in this case is simply the amount of cryptocurrency. Once again, the portfolio will be checked to ensure the currency is owned. The stocksOwned function will allow a player to open up the stocks owned by any player in any league via a playerID and a league ID. It will return that players respective portfolio in the respective league. We have also decided for ease of use to set the default blank case for an input to display the current users portfolio as it is much faster then searching their owner user ID and league ID. Finally, stocksSold will show the transaction history of any playerID searched in their respective league. Similar to stocksOwned, we will once again use the blank case (default case) to show the current users transaction history.

Alpha_Vantage_API_Adapter: The Alpha Vantage API is what will allow our system to retrieve the price of a specified stock. The class has only one method and it is convert(ticker). We will be able to search for the stock through the ticker and it will return the price of the stock as a double. This will allow us to buy, sell, and check the prices of the stocks at any given moment for whatever transaction is required.

C. Traceability Matrix

	League Controller	Login View	Alpha Vantage API Adapter	Account Controller	Trade Controller	League View	Home View	Trade View	Leaderboard	Player	Database	Leaderboard View
League Controller	X											
Login View		X										
Alpha Vantage API Adapter			X									
AccountController				X								
Trade Controller					X							
League View						X						
Home View							X					
Trade View								X				
Leaderboard									X			
Player										X		
Database											X	
Leaderboard View												X

The only name changed was Transaction controller to Trade controller. We can see how the domain concepts have evolved into the class diagram by looking at the Use Cases mentioned earlier. The use case for Join League involves the League controller, Database connect, League manager, and classes pertaining to the player and league. The use case for Research Stock involves Trade view, Transaction Controller, Database connect, Finance API Connect, and stocks classes. The use case for Buy/Sell Stocks involves Transaction controller, Database connect, Finance API connect, and stocks classes. The use case Create League involves, League

controller, Database connect, league classes, and player classes. The use case Reinvest Dividends involves Home view, Transaction controller, Finance API connect, Database connect classes.

3. System Architectural and System Design

A. Architectural Styles

We will be synchronizing multiple architectural styles in our implementation of TTT Stock Market League. The main architectural styles that will be used are Model View Controller, Database-Centric Architecture, RESTful API, along with using the Client-Server model.

Model View Controller

The Model View Controller is going to help us break the application into three interconnected parts. The first part is the Model, which is essentially the central component of the application, as it manages the data, logic, functions and the tools for the application. The second part of the MVC is the View, which supports the user interface end of the application, it is responsible for the charts, diagrams, tables, and any other user interface element that is to be displayed. Lastly comes the Controller, which is the takes the input, and communicates it to the model and/or the view using different commands. The three components in the MVC work together to ensure that the application runs as intended and makes the development simple to implement.

Database-Centric Architecture

The data that is stored in the TTT Stock Market League is critical to the operation of the application. Each user will have data that represents their account information, portfolio, leagues, watchlists, and UID and password. This is crucial data that is required whenever a user logs into their account, and has to be easily and quickly accessible. Therefore we will be using a standard relational database which makes accessing data simple, along with making the access time very fast.

RESTful API

We will be using a RESTful API to facilitate communication between a client and server. The client will issue GET HTTP method to retrieve information from our server. This information will include requested stock prices/charts, leaderboard information, a news feed etc.

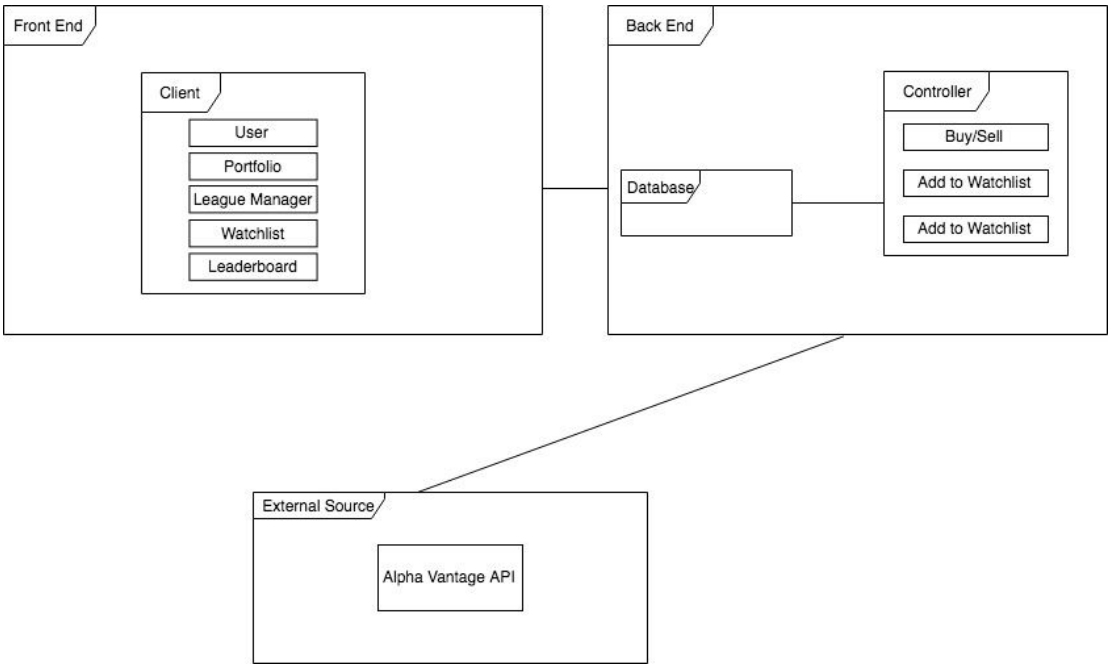
The client would also issue POST HTTP method to enter data such as number of stocks purchased/sold and requests to join leagues.

Using a RESTful API in our application gives us the advantage of having a layer of abstraction between client and server. It also would simplify developing our app on further platforms, such as an Android or iOS app, should we choose to expand our project.

Client-Server Model

The TTT Stock Market League will also use the Client-Server Model, which essential partitions tasks between the server, which is the resource, and the client, which is the requester. The web server will provide web pages for the client to interact with and in return, the client will send requests back to the web server.

B. Identifying Subsystems



TTT Stock Market League aspires to set its platform and system across several interfaces. This means that there must be subsystems of the application that help break the project into smaller systems. The two main subsystems are the front end and the back end.

The front end of the application consists of the user interface, where the user is able to interact with the application, and make decisions on their accounts and leagues, all on a visual interface. The front end of the application is so important as the user interface is the only method of communication between the user and the back end of the system, where all of the data is stored and accessed. The user interface is universal for all devices and browsers, including smartphones, tablets, and computers. However, the application will be optimized for computer

browsers. The front end communicated with the back end of the application to make sure that data is send and is consistent.

The back end of the application is where the inputted data from the front end is stored, accessed, and interpreted. The back end will consist of the database, where all of the data is stored, and the controller. The back end will also support calls to the Alpha Vantage api, where all of the stock market and cryptocurrency data will be retrieved from. The back end system has many responsibilities, and its proper functionality is essential to the success of the application.

C. Mapping Subsystems to Hardware

The TTL Stock Market League will run on multiple machines, with two main subsystems: a client (web browser) UI subsystem, and a back-end server (web server) database. The server is constantly running on a remote machine that a client can connect to at any time from any machine. When a user goes to the website on their web browser, they access the UI subsystem and are then connected to the server subsystem running on the remote machine. The user uses this UI to communicate with the back-end server and access the database as needed. Through this communication, the user's actions in the UI update and change the database as it relates to the actions they take. Since all this information is stored on a remote machine server, a user can log into their account from any web browser and have access to the same main database.

D. Persistent Data Storage

The TTL Stock Market League will need to save data that outlives a single execution of the system. The persistent object will be stored in a relational database through MySQL. Without persistent data storage, the application will not be able to function, as the player will not be able to login or reinvest dividends. Thus, the application will read and write data from a MySQL database.

The queries to the MySQL database will be send by the back-end subsystem. Multiple use cases involve a query to the database, such as viewing the stocks in the player's watchlist. The player's actions on the user interface indirectly trigger the queries to the MySQL database. Thus, the player's action, such as adding a stock to the watchlist, will will trigger the controller in the back-end to formulate and send a query to the MySQL database. Upon the return of this request, the controller will process the information and update the relevant elements in the user interface. Hence, the player will always receive real-time information about his/her account, as persistent data is stored in a MySQL database.

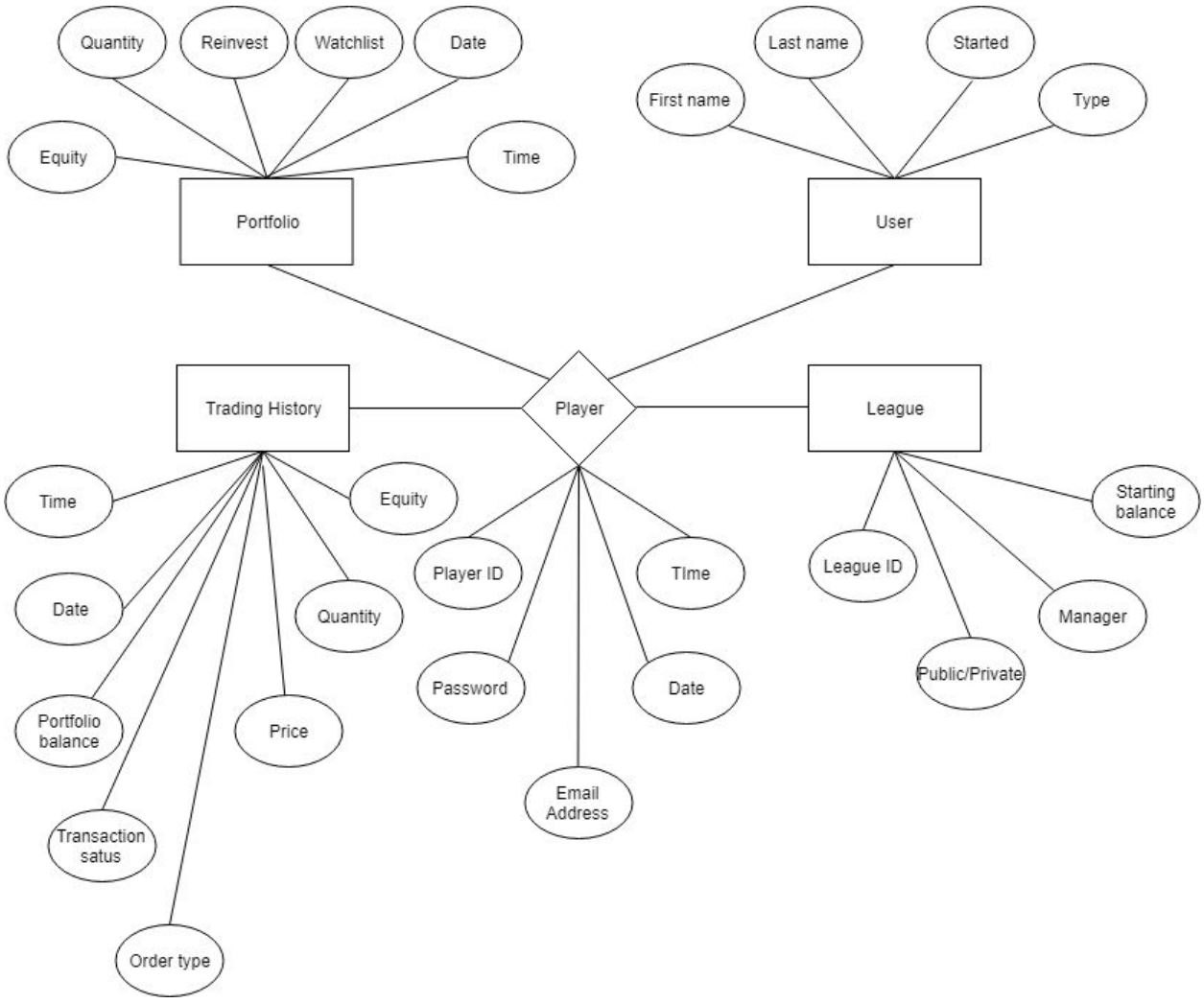


Figure #.# Persistent Database Entity Relationship Model

E. Network Protocol

The primary protocol used in TTT Stock Market League will be HTTP, or Hypertext Transfer Protocol. This is the main protocol that is used to transmit data over the world wide web (WWW), which is what this application will run on. When a user enters the URL for the application in a browser, a command/request will be sent through HTTP. This will be the main communication and protocol that will be used between the software interface and the user. Through HTTP, users will be able to access and perform different tasks like buying/selling stock, adding stocks to their watchlist, and making/participating in leagues.

F. Global Control Flow

Execution order: The TTL Stock Market League will mostly use an event-driven system. Once the user is logged in, they have the freedom to pick whatever actions they wish to take, and the system must wait for whichever action the user picks. The order of the actions taken can vary with each user once they're on the website; the user can select to buy or sell certain stocks from their own portfolio, or by searching for the stock directly and choosing to invest. Multiple actions have different possible routes that can be taken to complete those very actions, and so the system must wait in loops to see what the user does. There are still a few procedure-driven systems on the platform: one of them will be the process of the user having to first create an account with a unique username and password, and then log in to their account. Another is the updating of each of the leaderboards, which the system will do on its own as the rankings of users change. Finally, the fetching of relevant stock market information once the user selects a stock to look at will be handled in a procedure-driven process.

Time Dependency: The TTL Stock Market League will be run in real time alongside the Stock Market to simulate real life transactions. There will be a few timers running through the program as players are participating in the stock market league.

- League timer: This timer will keep track of the length of the league. Once this timer is over, the league will end and a winner will be chosen.
- Leaderboard Timer: This timer will keep track of the leaderboard and how often it will be refreshed. Since we cannot have the same players on the leaderboards at all times, it will reset every week or month so that other players have a chance of joining the rankings.
- Inactivity timer: This timer will keep track of how long a player has not performed an action. If this timer ends without being interrupted by the user performing an action, the player will be logged off due to inactivity.
- Stock Info Update Timer: This timer will be used to update the information and prices of stocks and cryptocurrencies for the players. This will keep all of the information up to date so that all transactions will provide the correct information to the database.
- Stock Market Timer: This timer will keep track of the start and end of the stock market for the TTT Stock Market League according to the operation times of the real stock market. This will be the window in which the players can perform transactions.

G. Hardware Requirements

The required specifications to run our software will be quite minimal since a majority of this heavily depends on the servers that are implemented. TTT Stock market league will be compatible on any computer that will be able to run an internet browser, as long as there is connection to the web.

Internet Connection

To have our stock market league to be fully functional, the player must have internet connection, or else they would not be able to access any of our software's functions. In order to have our data to be transferred from the database, the player would need to have a computer that is able to send and receive a low band frequency but ideally the higher the frequency the more efficient. Our software requires to have a network connection between our servers and the finance API during trade hours, or else any participating players would be able to access any transactions. Other functionalities that comes with our software would also require internet connection, like player registration, creating/joining leagues, and others. Without any sort of connection, any interaction the player is having with our software would not be saved in our servers.

Required Memory

Considering our software is in the beginning stages of development, it is difficult to estimate the appropriate overall system performance of the system that needs to be supported. Our software will need to take the data stored in the database to permit any possible interactions. To increase performance, our group would ideally want to use the least recently used system to maintain the necessary information in the system memory. Implementing this scheme will dispense any unused/unaccessed bits that has not been used in a long period of time. Any of the functions used that is loaded into the system will need some memory space as well. An approximate of 512MB would be necessary for any testings with onto our system.

Disk Storage Space

Having hard drive availability is essential to our software because we need to store any information that is coming from the database onto the system that is being used. Approximately 10-15 GB would be optimal storage space for our system, and any program instruction that may be needed for storing data.

Hardware Requirements

As mentioned previously, our software is not very hardware demanding. We would only need the very minimum to run our software on any player's system. With that said, for a quintessential user experience with our stock market league a player would need just the basics.

A functional mouse, keyboard, and a display screen to view their interactions would be satisfiable. The screen does not need to be any sort of high quality, a simple 800x600 pixel is acceptable.

4. Data Structures & Algorithms

a. Algorithms

N/A (No Complex Algorithms Involved)

b. Data Structures

There are multiple data structures that TTT Stock Market League will use to maximize functionality and efficiency, these include a queue, hash table, and the python list.

Queue

The queue data structure is that follows the FIFO (first in first out) logic. This will be used when TTT Stock Market League handles trade requests, either buying or selling. When multiple users decide that they would like to make a trade, they will be entered into the queue and the first user to enter the queue will have their trade executed first, assuming they have sufficient funds, and the last user to enter the queue will have their trade executed last. This will help the application handle multiple trade request at the same time, by putting them into a queue and executing them one at a time.

Hash Table

The hash table data structure is a data storage method that makes accessing data very fast using a hashing function. This will be used in the case of storing the users stock and cryptocurrency holdings. For the users stock holdings, there are always new stock position that are added, modified, or sold/removed. Because of this, short access time in crucial. For the hash table, the average case scenario is constant time, or O(1) time complexity, which is significantly quicker compared to an array or a list where the access time is O(n), or linear time.

Python List

The python list is a data structure that will be used in many different parts and functions in TTT Stock Market League. A python list is a dynamically allocated array, which means that the python list has the ability to grow, and does not have to be set to a certain size at initiation. This has so many applications in out project, one in particular is the leaderboard or watchlist which both continue to increase in size, and can be perfectly implemented using a python list as it can keep on increasing in size as more stocks are added, or there are more users.

5. User Interface Design and Implementation

The user interface design is still similar to Report 1, as there has not been many functional changes that will affect the UI, other than color and style changes. The User Interface was designed to be very intuitive and easy to use from the beginning. The majority of the User Interface and front end has not yet been implemented, however as we implement and develop it we expect there to be small changes made in the design, however it will follow our earlier renderings and designs.

The User Interface is designed with simplicity as the first priority, which translates into a very simple to use interface that consists of very few pictures and not many different contrasting colors. All of the functions are a few clicks away, and are easily accessed by the navigation bar at the top.

6. Design of Tests

a. Test Cases

Login view

TC-1: login(username, password): boolean

Test Procedure	Expected Results
Call Function (Successful)	The given username/password pair is passed to Database.match() to see if the pair is present in the Database. Returns true if found, else false.
Call Function (Unsuccessful)	Function returns true under the condition that username/password pair do not exist in the Database. Returns false under the condition that the username/password pair does exist in the Database.

TC-2: logout(): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if player is logged out, otherwise returns false.
Call Function (Unsuccessful)	Returns false if player is logged out, or returns true when player is still logged in.

Account controller

TC-3: verifyUser(username, password): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if username/password pair is present in Database. Returns false if username/password pair is not present in Database.
Call Function (Unsuccessful)	Returns false even if username/password pair is present in

	Database. Returns true even if username/password pair is not present in Database.

TC-4: createAccount(username, password): Player

Test Procedure	Expected Results
Call Function (Successful)	Returns a Player object filled with the passed arguments upon successful creation of account. Returns 'None' on otherwise.
Call Function (Unsuccessful)	Returns 'None' despite a valid username/password pair, or returns a Player object with uninitialized values or incorrect values.

TC-5: isLoggedIn(): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if user has been verified and is logged in. Returns false otherwise.
Call Function (Unsuccessful)	Returns false despite user being logged in. Returns true when user is not logged in.

League controller

TC-6: createLeague(name, leagueManager, capacity): boolean

Test Procedure	Expected Results
Call Function (Successful)	Checks if a league with 'name' already exists in Database. Returns false if it already exists. Creates a league by writing the passed arguments to the Database. Returns true upon successful creation, else returns false
Call Function (Unsuccessful)	Returns true and writes to Database despite another league with the same name already existing in the Database.

TC-7: addPlayer(playerID): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if Player is added successfully to member set, else returns false
Call Function (Unsuccessful)	Returns false despite Player being added successfully to member set, and returns true despite Player not being added successfully to member set

TC-8: deletePlayer(playerID): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if Player is removed successfully from member set, else returns false
Call Function (Unsuccessful)	Returns false despite Player being removed successfully from member set, and returns true despite Player not being removed successfully from member set

TC-9: isManager(playerID): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if playerID matches the playerID of the leagueManager, returns false if IDs do not match
Call Function (Unsuccessful)	Returns false if playerID matches the playerID of the leagueManager, returns true if IDs do not match

TC-10: isMember(playerID): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if playerID is in the member set. Returns false if playerID is not in the member set
Call Function (Unsuccessful)	Returns false despite playerID being in the member set. Returns true despite playerID not being in the member set

Trade controller

TC-11: buyStock(playerID, leagueID, ticker, shares): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if the Player referenced by playerID in the given League referenced by leagueID, has enough money to purchase the given number of shares of a stock. Database is updated to reflect the new number of stocks that the player owns. Returns false if the Player does not have enough capital to purchase the given number of shares.
Call Function (Unsuccessful)	Returns false despite the Player referenced by playerID in the given League referenced by leagueID, has enough money to purchase the given number of shares of a stock. Returns true if the Player does not have enough capital to purchase the given number of shares.

TC-12: sellStock(playerID, leagueID, ticker, shares): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if the Player referenced by playerID in the given League referenced by leagueID, has at least as many shares as they desire to sell. Database is updated to reflect the new number of stocks that the player owns. Returns false if the Player does not have as many shares as they desire to sell.
Call Function (Unsuccessful)	Returns false despite the Player referenced by playerID in the given League referenced by leagueID, having at least as many shares as they desire to sell. Returns true despite the Player not having as many shares as they desire to sell.

TC-13: buyCrypto(playerID, leagueID, ticker, shares): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if the Player referenced by playerID in the given League referenced by leagueID, has enough money to purchase the given number of shares of a cryptocurrency. Database is updated to reflect the new number of cryptocurrency that the player owns. Returns false if the

	Player does not have enough capital to purchase the given number of cryptocurrency.
Call Function (Unsuccessful)	Returns false despite the Player referenced by playerID in the given League referenced by leagueID, has enough money to purchase the given number of cryptocurrency. Returns true if the Player does not have enough capital to purchase the given number of cryptocurrency.

TC-14: sellCrypto(playerID, leagueID, ticker, shares): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if the Player referenced by playerID in the given League referenced by leagueID, has at least as many cryptocurrency as they desire to sell. Database is updated to reflect the new number of cryptocurrency that the player owns. Returns false if the Player does not have as many cryptocurrency as they desire to sell.
Call Function (Unsuccessful)	Returns false despite the Player referenced by playerID in the given League referenced by leagueID, having at least as many cryptocurrency as they desire to sell. Returns true despite the Player not having as many cryptocurrency as they desire to sell.

Database

TC-15: match(username, password): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if username/password pair exist in the Database, else returns false
Call Function (Unsuccessful)	Returns false despite username/password pair existing in the Database, returns true despite username/password pair not existing in Database

League view

TC-16: menu(selector): void

Test Procedure	Expected Results
Call Function (Successful)	Shows all possible options that is available for user to select. Will be successful if all selections are fully functional and operating with no errors.
Call Function (Unsuccessful)	Does not show any/missing selections for the user, and functionalities does not work.

TC-17: search(leagueID): boolean

Test Procedure	Expected Results
Call Function (Successful)	Returns true if the searched league ID is found and is available.
Call Function (Unsuccessful)	Returns false if searched league was not found.

Home view

TC-18: menu(selector): void

Test Procedure	Expected Results
Call Function (Successful)	Shows all functionalities with stock market league, such as buy/sell stocks, view profile etc.,to user in home view.
Call Function (Unsuccessful)	If unsuccessful, user will not be able to view all/some options on home screen.

Leaderboard

TC-19: menu(selector): void

Test Procedure	Expected Results
Call Function (Successful)	Shows the current status of a specific league,

	if successful the leaderboard will show the accurate standings of each player, and rank all players in numerical order.
Call Function (Unsuccessful)	If unsuccessful, players would not be able to view an accurate leaderboard and cannot see the correct standings of the current league.

Trade view
TC-20: menu(selector): void

Test Procedure	Expected Results
Call Function (Successful)	Shows the available stocks for trading by the player. If successful, players will be able to accurately view which stocks are available for trade and any information needed about that particular stock.
Call Function (Unsuccessful)	If unsuccessful, player will not be able to view the status of the stock that a player wishes to trade.

b. Test Coverage

Given that many of our functions return boolean values, Equivalence Testing will be a logical choice to implement. Equivalence Testing groups input types into categories. Only a small number of inputs from each category would need to be tested to gain confidence in the workings of our functions. An example of this would be the testing of our buyStock() and sellStock() functions. Focusing on buyStock(), we need test that this function returns true if and only if the player calling this function has enough money to purchase the number of shares that they request. Say that the user requests to buy $100 worth of shares. The transaction could only occur if the user has at least $100. We could write a unit test simulating the user having $100 and trying to buy the $100 worth of shares. If this test passes, we can be confident that it will also work if the user has instead $200. This is because a value of $100 and $200 are in the same equivalence group. The same can be said for a scenario where the user has $99 and wants to buy $100 worth of shares. If our test passes with these numbers (the function would return false in this case),

then we can be confident that the function will also work if the user had $1. Again, this is because $99 and $1 are in the same equivalence group.

By recognizing these equivalence groups, we can assess the outcomes of many inputs by only testing a few inputs. The inputs that we test would include boundary values (in the example in the previous paragraph these boundary values would be $100 and $99), as well as some other values in the equivalence group (such as $200 and $1 as in the example above).

c. Integration Testing

For integration testing, hybrid sandwich integration testing will be implemented. The hybrid sandwich integration testing is an iterative combination of bottom-up and top-down integration testing. Repetitive unit testing and integration testing of the components will be conducted until the target level of integrated modules is achieved. The hybrid sandwich integration testing strategy was chosen, as it fits with the goals of Demo 1. In Demo 1, the objective is to develop the website user interface, database and website assimilation, Alpha Vantage API, email system, and profile registration. Since these goals intertwine opposite ends of system hierarchy, hybrid sandwich integration testing best accommodates testing for Demo 1. Thus, within each use case, unit tests will be conducted as we progress through the project. After the use cases pass the unit tests, integration tests for these use cases will be run to ensure that the individual components work together to form an integrated unit. The process will then by repeated again and again with different unit testing for other use cases and integration testing of these components with other previously integrated use cases.

d. User Interface Testing

For user interface testing, the user effort and figures for user interface specification will be the criteria of the testing. In Report 1, the user effort estimation for the use cases was listed in terms of clicks and keystrokes. In order to ensure the user interface meets the criteria, testing will be conducted by following the steps outlined in Report 1 for each use case and verifying that clicks and keystroke requirements are met for main successful and alternative outcomes. Moreover, the information displayed by the user interface will be compared to user interface specification, particularly the figures, to ensure that the user interface displays all the desired information and functionality. .

7. Project Management and Plan of Work

a. Merging the Contributions from Individual Team Members

In order to make compiling the final report easier on all of the team members, we utilized the group editing technology of Google Docs. This allowed each team member to simply log on using their emails to the group document and edit which part they wanted to contribute to. This lessened the burden on the group overall and, by the time we all finished contributing, the work would already be compiled with everyone's work. This also made uniform formatting and consistency very easy as each member could see how the other members formatted their own portion. We could simply copy another member's format and use it as a template for every other section.

One of the first issues we had was communication. In the beginning, team members would start working on whatever parts they wanted to and would finish before other members had a chance to contribute to the parts they enjoyed/had the most knowledge in. This led to less contributions from some members of the team. The solution to this issue was to use the mobile application "GroupMe" to be able to communicate which parts we wanted to work on so that each member could gain some contribution. This also helped to resolve another issue which was the differences in everyone's schedules. Some members had clubs, religious activities, and extra classes they partake in so we were unable to collaborate altogether and would have some inconsistencies in our separate parts. We used GroupMe to coordinate a time for each member to get on Google Docs simultaneously and contribute together. This reduced the number of inconsistencies in our reports and gave each member more time and better ways to contribute.

We will also be utilizing Github as a tool for collaboration in order to piece together the first demo. To ensure uniform formatting and syntax, we have already decided that Python/Django will be the choice of language we will use to create our stock market league. Github will allow each collaborator to push / merge their work onto the repository so that there is no confusion or overlapping of the same code. As this is a more complicated task than the reports, it's believed more issues will arise while building our first demo.

b. Project Coordination and Progress Report

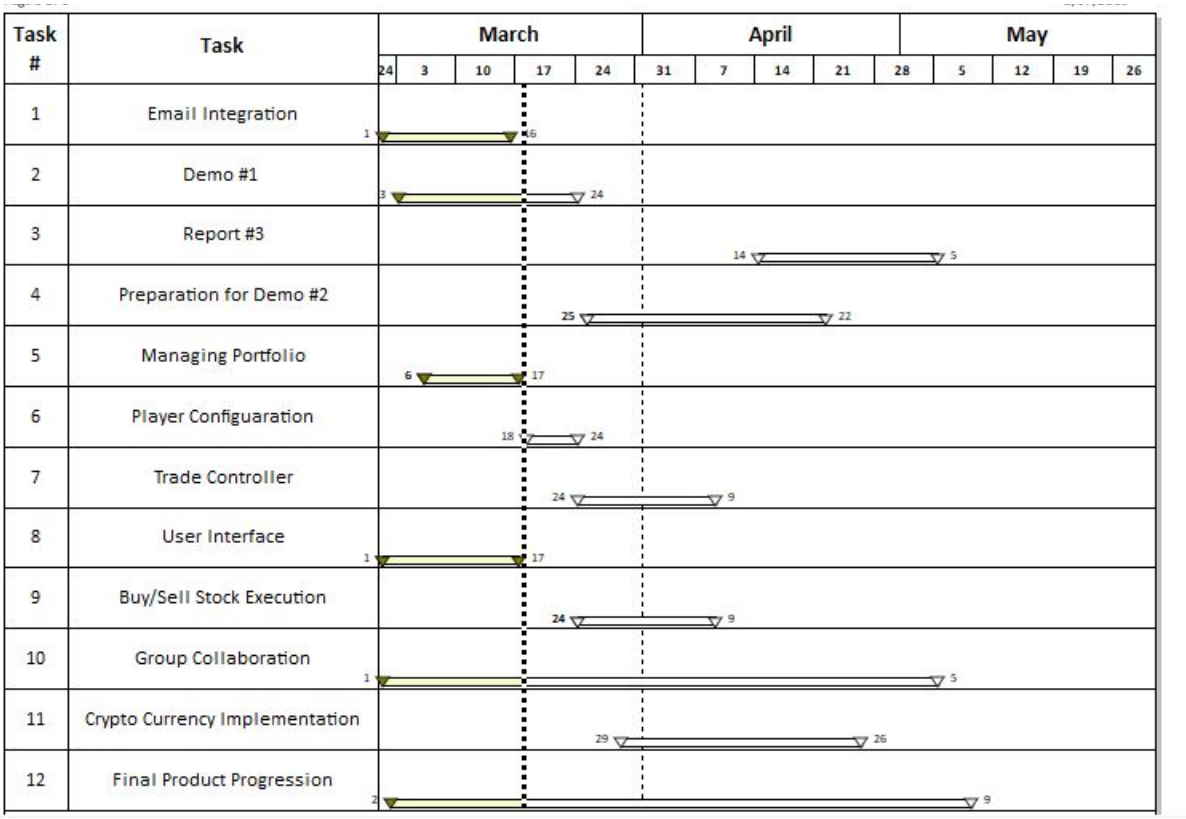
Throughout the course of the semester, the team has been working on the TTT Stock Market League little by little. Since we've been figuring out the details of the projects during the write-ups for report 1 and 2, most of the components we've been able to complete were mainly with the user information such as the registration, user information, and the user profile. Having finished the user portion, we will be able to utilize this to test all of the other components that must be finished for the first working demo. Although it may not be our final product, the UI has also been created. The current

UI allows the user to navigate through the interface with ease and also gives a simple, non-confusing perspective. The user must be able to log on and create a league or join a league with no issues. This is currently underway and will be finished soon as it is another portion we can utilize to test all other parts of the stock market league.

Before our deadline for the first demo, we must also finish the transactions between the user and the stock database. Being able to buy and sell stocks is the most integral portion we must complete in order to have a working demo of the stock market league. If none of our users are able to make any transactions, this will result in the entire project being unusable. In order to get the transactions working, we will also have to be able to pull stock information from a file or the api. This is another portion a sub-team will be working towards finishing so that the user will be able to get current stock information and make transactions reliably and without any faults. Currently, we have no eta on the first working prototype of the project but we hope to be able to complete a working version with just the essential functions by the end of this week. This will give us enough time to add other extra features and to give us enough time to debug any issues that may arise.

In order for all of the essential functions to work, we must have a strong database that will be able to handle all of the transactions from all of the users. This will be the key focus for the coming week before we really get into working on the features. The team also hopes to work towards implementing the learning process before the first demo as we have stated that the TTT Stock Market League will be for both intermediate users and beginners alike. The main goal at this moment, is to be able to have all of the essential functions working with a UI that will be easy for new users to navigate through. Although the team does not know when the first prototype will be finished, we hope to be able to create a functioning preliminary model very soon.

C. Plan of Work



d. Breakdown of Responsibilities

Due to the fact that our project is ambitious and there are multiple functions to each utility it provides, it was vital we split up every team member's responsibilities appropriately. Our distribution of the project was based upon individual selections. Every member decided what section they wanted to work on depending upon their strengths and ease of collaboration with respective team mates in their sub groups. This allowed us to be comfortable with what we were working on and provide flexibility within sub-groups to meet up. Furthermore, since there are so many operations and tasks we need to cover, we allowed members to be in multiple groups. The breakdown is as follows:

Joshua, Danny, Antoni and Raffay will be working on the active stock market trading, peer comparisons, and stock tracking. This includes buying and selling stocks/cryptocurrency, and creating a system of leaderboards that illustrate an individual players standing in contrast to his competitors. On the active stock market trading side, the main obstacle will be utilizing the Alpha Vantage API to pull stock prices in the processes of trading and selling. On the leaderboards side, compiling and comparing each players portfolio also presents a challenge on a display side since we must be able to

present all the information in the most user friendly manner. This sub group will create and be responsible for the trade controller, trade view, leaderboard, and leaderboard view class along with the integration of the Alpha Vantage API. This can be considered the core of the project as there is no application if players can not successfully buy and sell stocks freely.

The group of Danny and Ajeet will be responsible for account registration and league management. This entails players logging in and logging out successfully and creation of player accounts. Since our applications require player accounts for a player to join a league, we must ensure this is functional. The group is responsible for finding an efficient and user friendly way to sign up for the application and create leagues. The class associated with this subgroup is the AccountController class, League Controller, League View, and LoginView class.

Antoni and Josh will be responsible for spreadsheet exporting or essentially the database of our application. Since our application will be accumulating a lot of data, we want to make sure it is stored and retrieved seamlessly with no error. Some examples of data values we need to store are usernames, passwords, stock portfolios, and leagues. Without a functioning database, the application will simply not be able to function as we will be unable to store anything transactions or information received. Thus, along with active stock market trading, a working database is vital in a function project and application. This group will work on the HomeView and Database classes.

Ping, Aryeh and Chris will work on cryptocurrency and integration and the dividend. Since our application does not exclusively work with stocks, we must ensure our selling point of cryptocurrency integration is properly taken care of. It will allow users to buy and sell cryptocurrency, which is a current market trend. Additionally, this group is assigned with the task of making sure the dividend function operates properly and adds to a player's portfolio. This group will be involved in both the trade controller class and the database class.

Ping and Ajeet will work on stock market recommendations. This consists of presenting stocks and articles on stocks you are viewing in the trade view. This is to provide a better user experience for players who don't exactly have the experience or knowledge to make their own decisions on which investments to make. Being another selling point of our product, it is essential for it to be functioning in our application as we are marketing our app as being beginner friendly. The classes associated with this subgroup is the Trade Controller class and the TradeView class.

The final group consists of Aryeh and Chris will be collaborating on the educational element of the project. This consists of providing informational material to our players to get them accustomed to the stock market and allow them to proficiently compete in the league. The main component of this subsection will be utilizing credible sources such as investopedia as it will be our primary resource in creating the educational

tool component.We need to ensure we our giving our players the best on correct information. This group will be involved in the HomeView and TradeView classes.

8. References

[1] Investopedia - Investopedia Stock Terms - https://www.investopedia.com/categories/stocks.asp
[2] Nasdaq - Glossary of Stock Market Terms - https://www.nasdaq.com/investing/glossary/
[3] Rutgers - Software Engineering Project Report- https://www.ece.rutgers.edu/~marsic/Teaching/SE/report2.html
[4] Tutorial Points - Estimation Techniques https://www.tutorialspoint.com/estimation_techniques/estimation_techniques_use_case_points.htm