Class Diagram and Interface Specification
System Architecture and System Design

for

“Titan Trading”

Report 2: Part 2

By

Group #7

Steven Adler

Kristene Aguinaldo

Timothy Liu

Nicholas Lurski

Avanish Mishra

Safa Shaikh

Brooks Tawil

Kristian Wu

David Zhang

Table of Contents

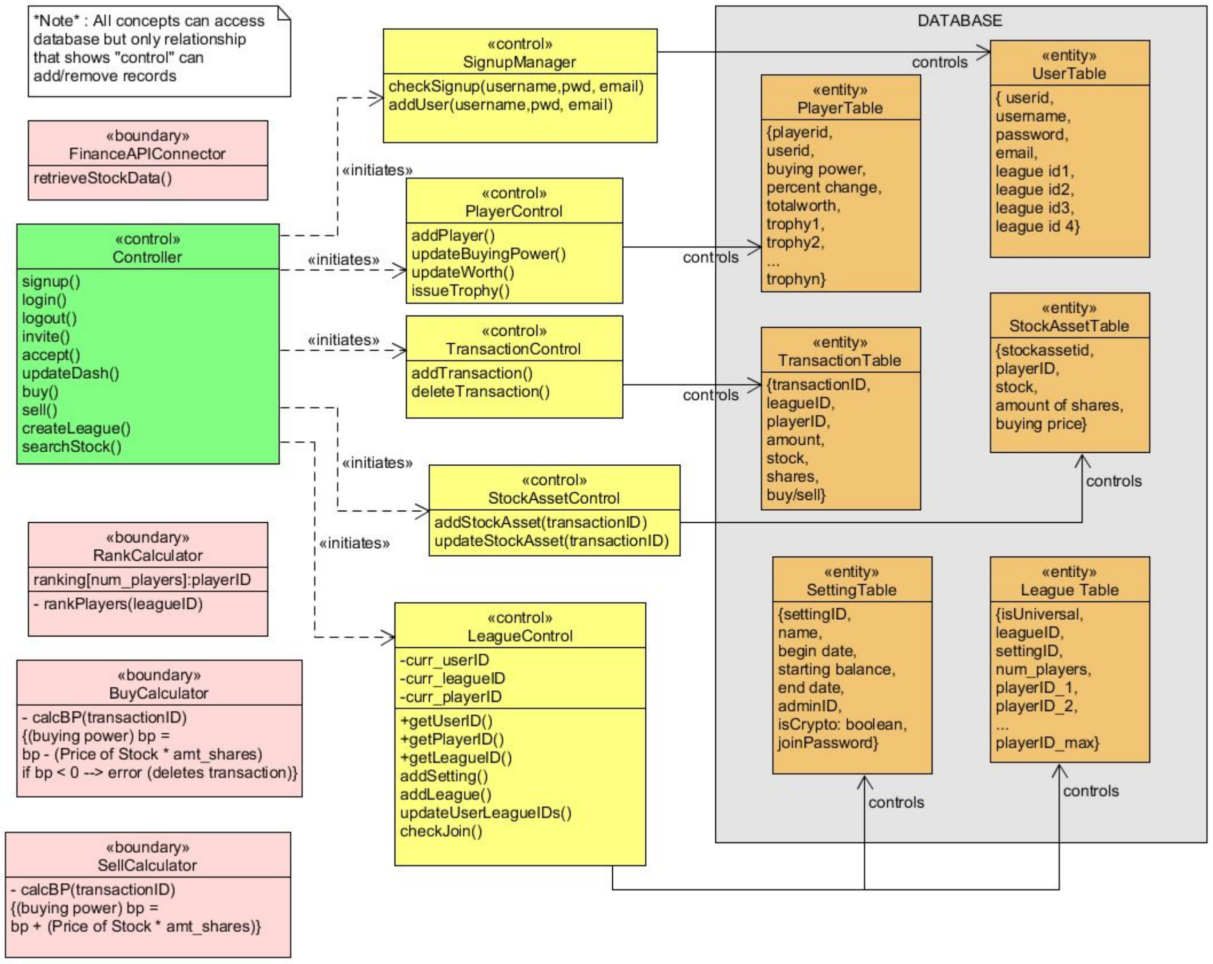
SYSTEM INTERACTION DIAGRAMS

Introduction

Titan Trading is a web app, built with Django and Python and will interface with a PostgreSQL database. The Financial API we will be using is AlphaVantage, which can obtain both stock and cryptocurrency data. Following is an in-depth analysis of use cases and system interaction between domain concepts.

Domain Model

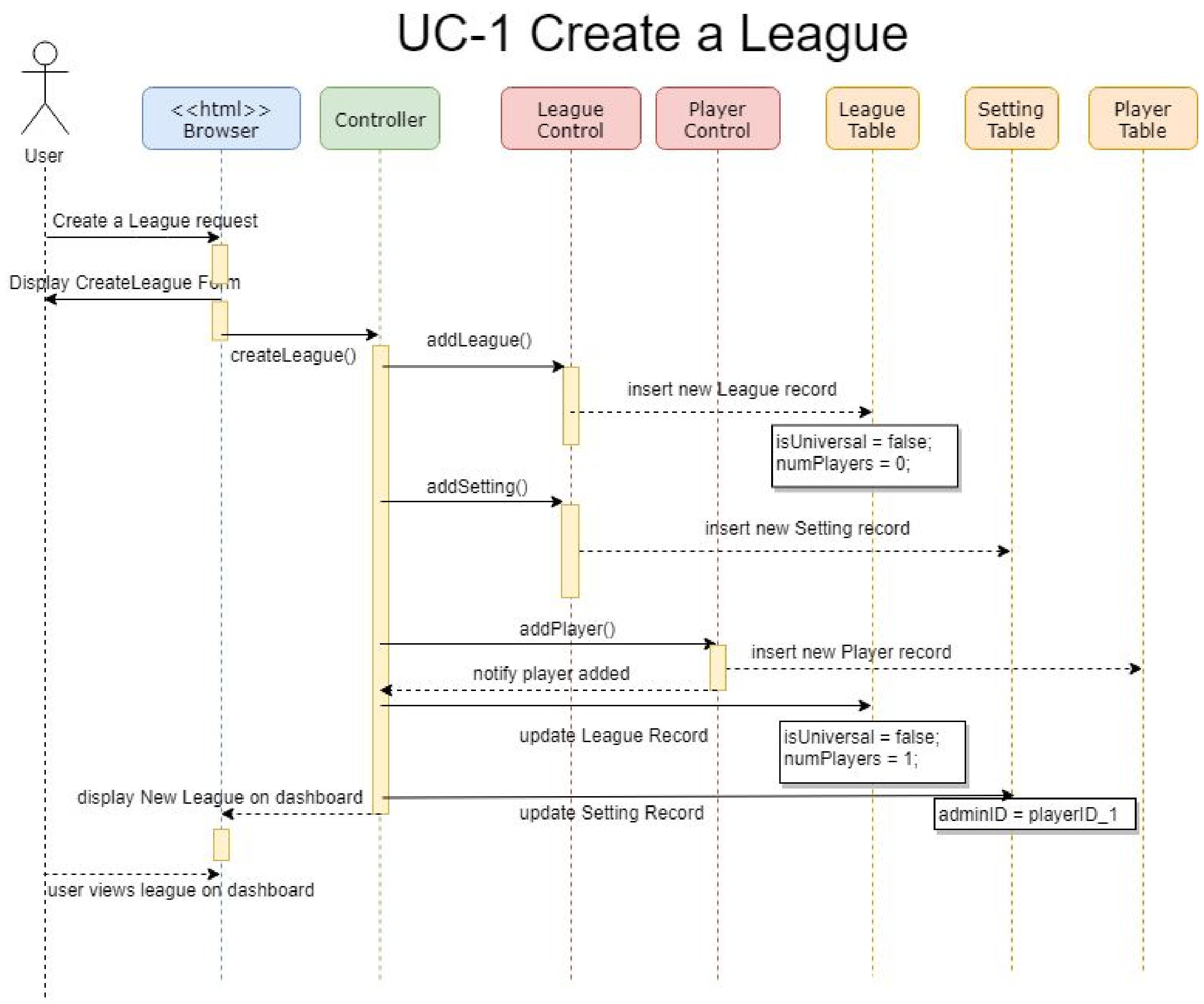
The following is our domain model. The controller is the main interface between the system and the browser with which the user interacts with. The controller allocates responsibilities to the various controls, which respectively update the database.



System Sequence Diagrams

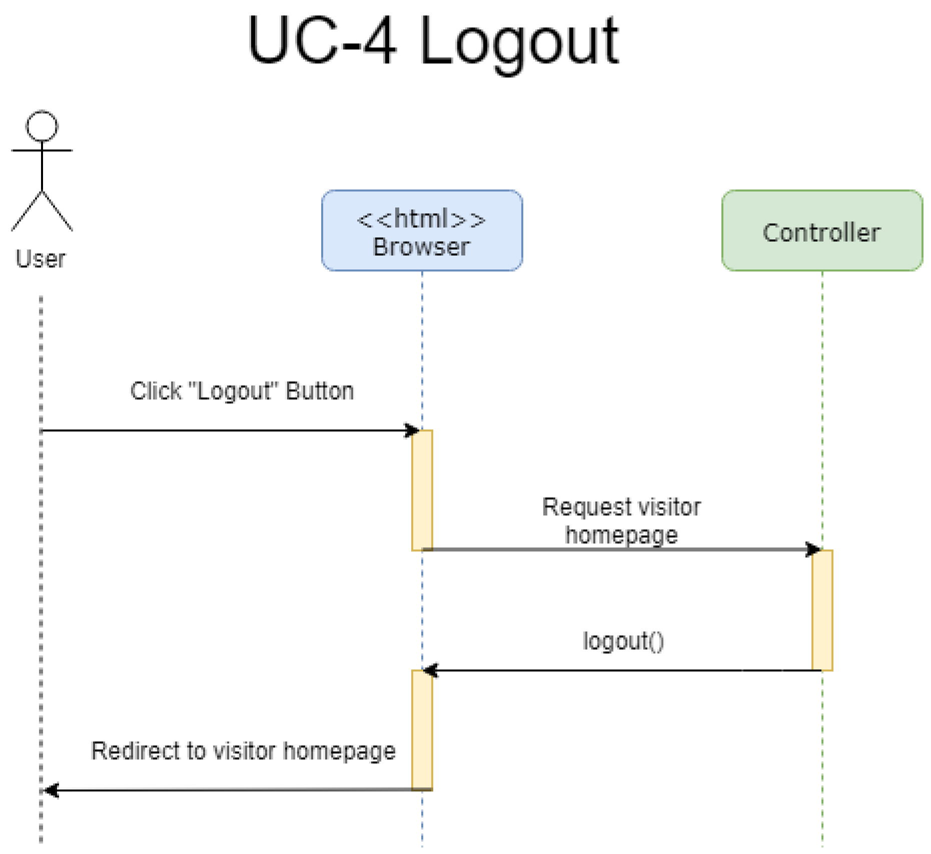
The following diagrams show the system processes of each fully dressed use case. These diagrams showcase the interaction of our domain concepts and how all user data is managed in a database.

UC-1 Create a League



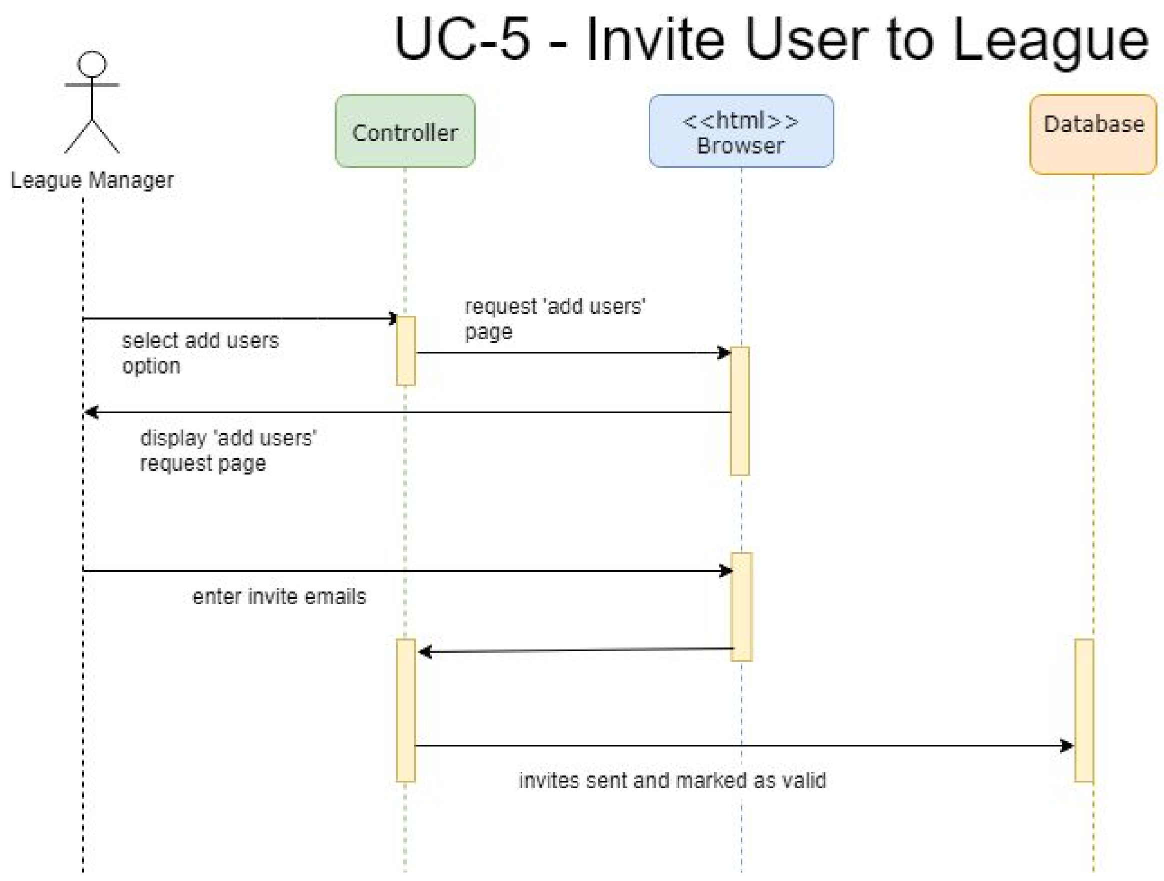
To create a league, a user enters all required data into a form. The controller keeps track of this data and sends it to the League Control and the Player Control. The League Control takes all data associated with the new league and new settings and updates the two league and setting tables in the database. The admin is now the first player, so that data needs to be updated in the player table as well. The player control will create a new player record. This record will now be associated with the prior created league and settings records.

UC-4 - Logout



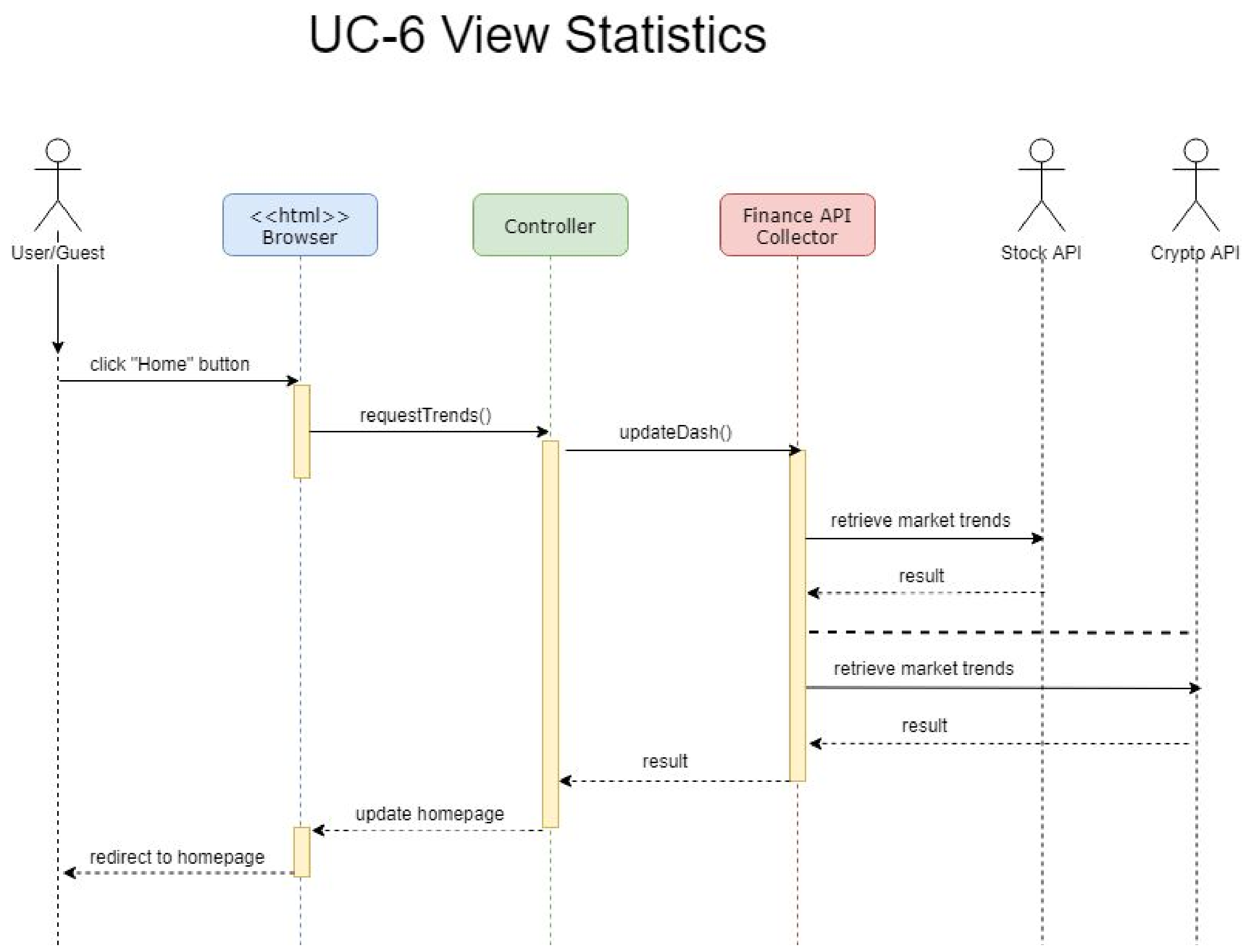
An essential feature is to distinguish visitors from registered users. Users become visitors by logging out and may proceed to initiate this process by clicking the logout button. The framework of the website allows all user data to be saved in the database when the user logs out and that state will be preserved until the user logs in again.

UC-5 - Invite User to League



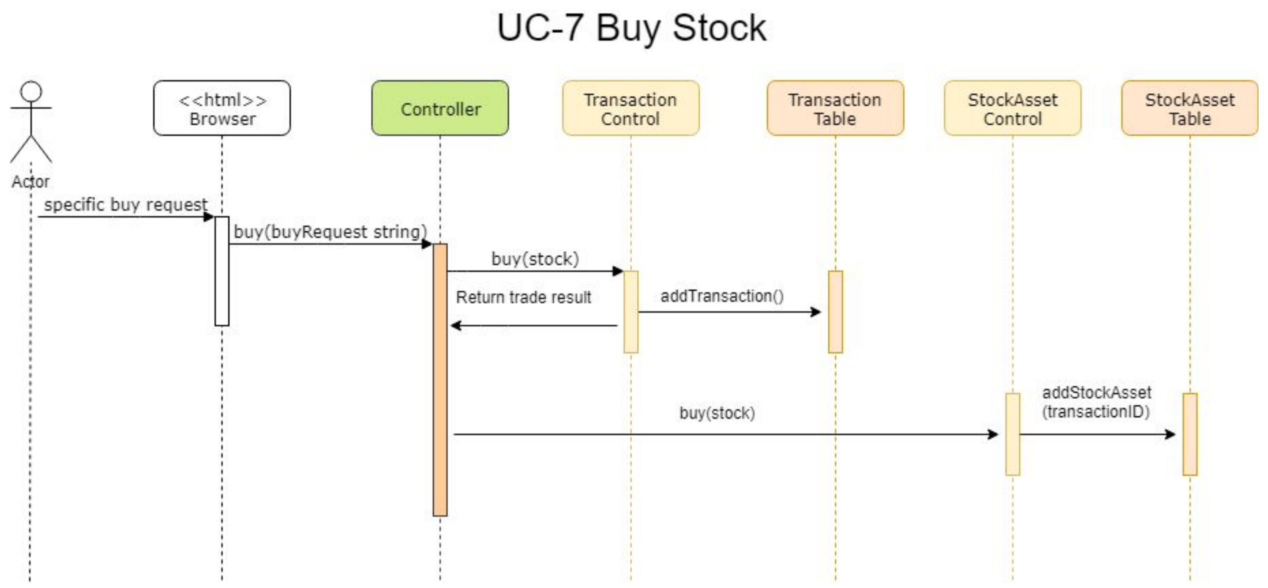
A league manager may decide to invite a user to join a league in order to participate in private competition with that user. The league manager will add users, using the “Add Users” option within the manager’s private league. The Controller will process this request and display the “Add Users” web page to which the league manager will fill in invitees’ emails. The Controller will process and send emails to the invited users. It will then interact with the Database, confirming that invitation emails were delivered.

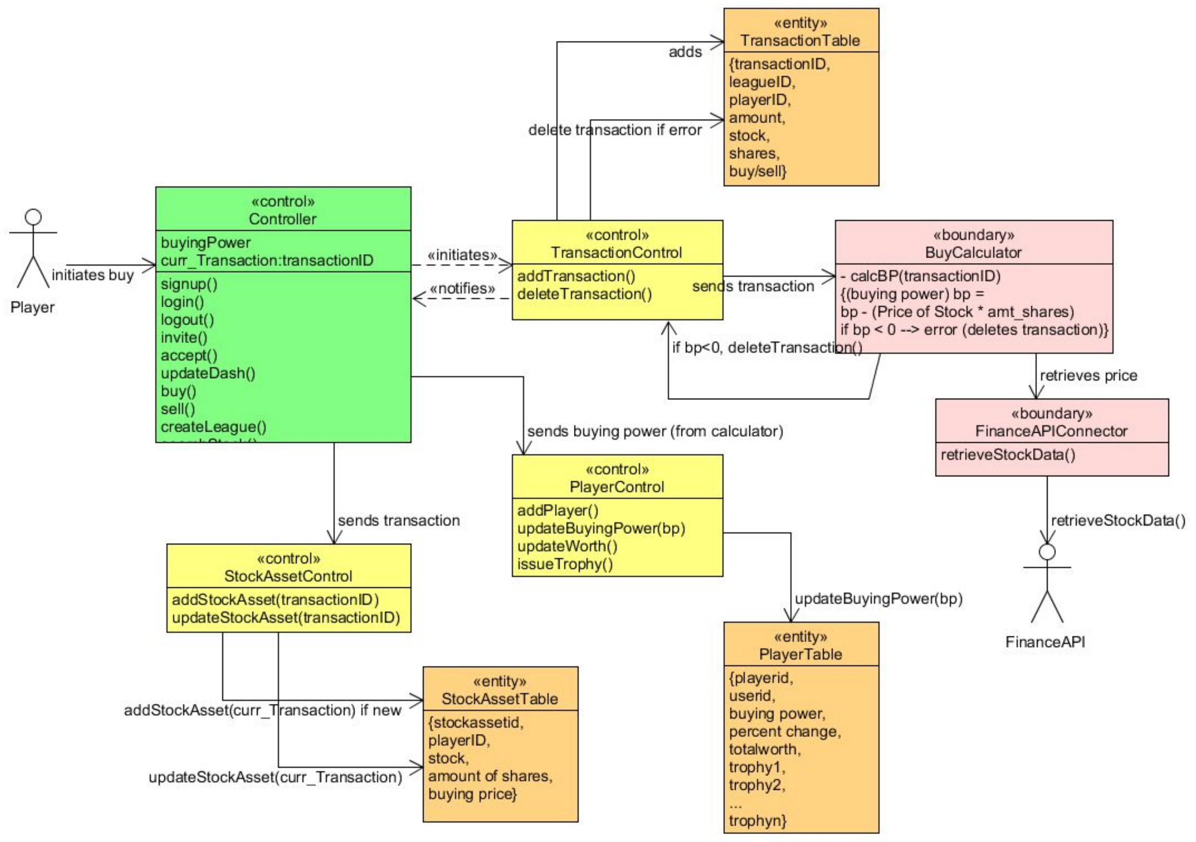
UC-6 - View Market Statistics



Both visitors and users can view market statistics in Titan Trading. The homepage will have updated information on current market trends and this data will be updated any time the user refreshes of clicks the homepage. Clicking the “home” button will retrieve updated data from the stock and cryptocurrency APIs.

UC-7 - Buying a Stock



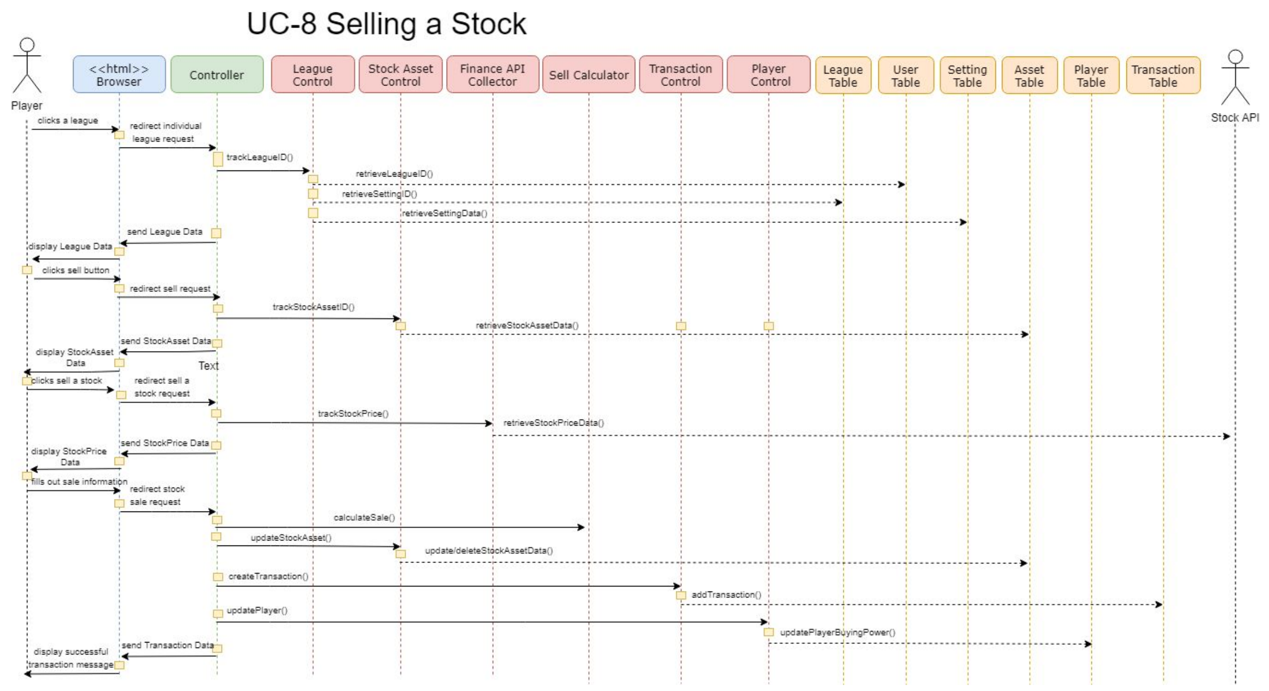


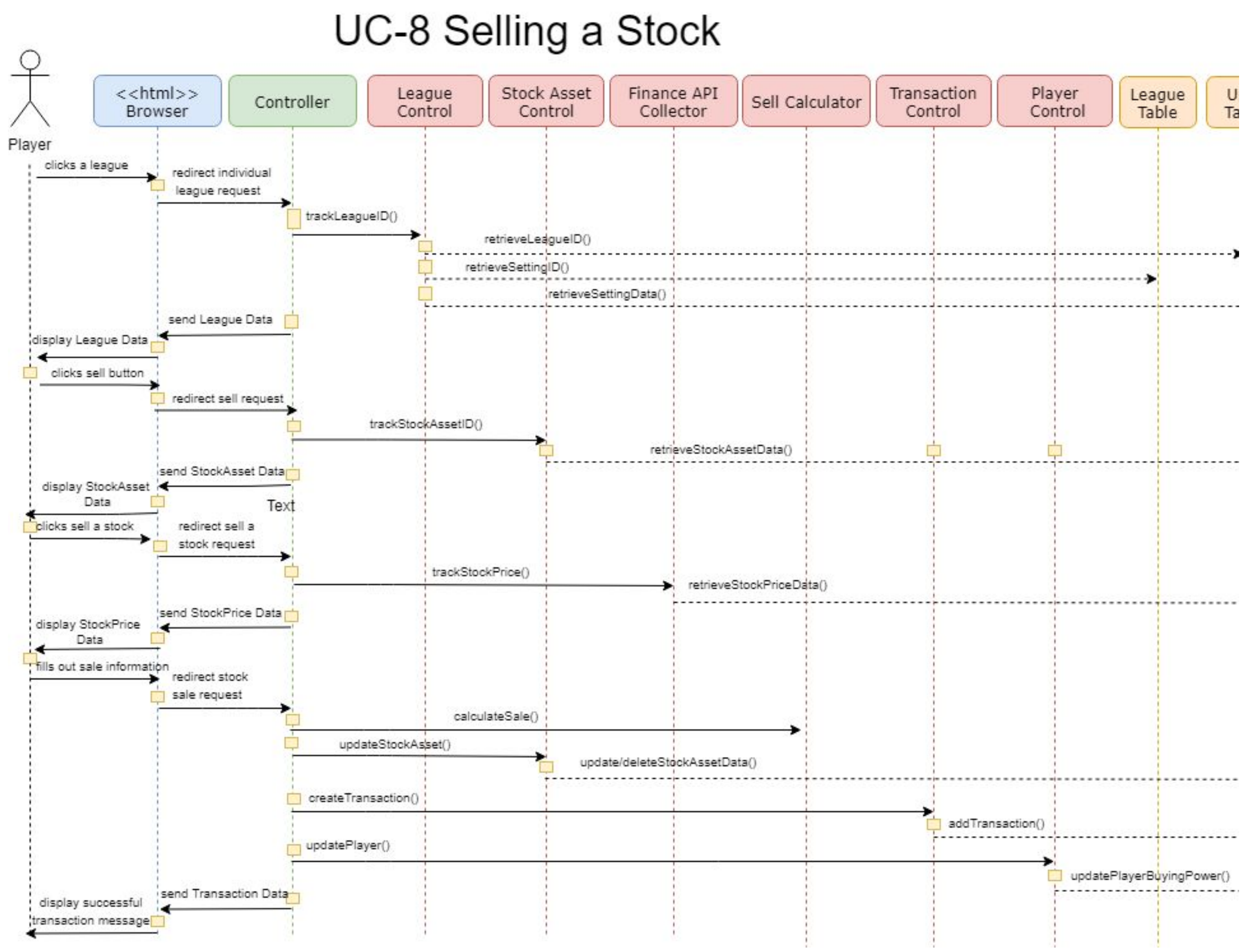
«entity» TransactionTable
{transactionID, leagueID, playerID, amount, stock, shares, buy/sell}
adds
delete transaction if error
Player
initiates buy
«control» Controller
buyingPower
curr_Transaction:transactionID
signup()
login()
logout()
invite()
accept()
updateDash()
buy()
sell()
createLeague()
«initiates»
«notifies»
«control» TransactionControl
addTransaction()
deleteTransaction()
sends transaction
«boundary» BuyCalculator
- calcBP(transactionID)
{(buying power) bp =
bp - (Price of Stock * amt_shares)
if bp < 0 --> error (deletes transaction)}
if bp<0, deleteTransaction()
retrieves price
«boundary» FinanceAPIConnector
retrieveStockData()
retrieveStockData()
FinanceAPI
sends buying power (from calculator)
«control» PlayerControl
addPlayer()
updateBuyingPower(bp)
updateWorth()
issueTrophy()
updateBuyingPower(bp)
«entity» PlayerTable
{playerid, userid, buying power, percent change, totalworth, trophy1, trophy2, ... trophyn}
sends transaction
«control» StockAssetControl
addStockAsset(transactionID)
updateStockAsset(transactionID)
addStockAsset(curr_Transaction) if new
updateStockAsset(curr_Transaction)
«entity» StockAssetTable
{stockassetid, playerID, stock, amount of shares, buying price}

In order to buy a stock, a player must select which stock and how many shares of the stock he/she would like to buy. The controller will take this user input and send it to the Transaction Control, which will create a transaction record with updated prices from the Stock API (or

Crypto API if in a Crypto League). After the transaction is recorded (if successful), the Player Control will update player data, including buying power and player worth (total value of all assets). Stock Asset Control will also add a StockAsset record with the player's ID to keep track of player assets.

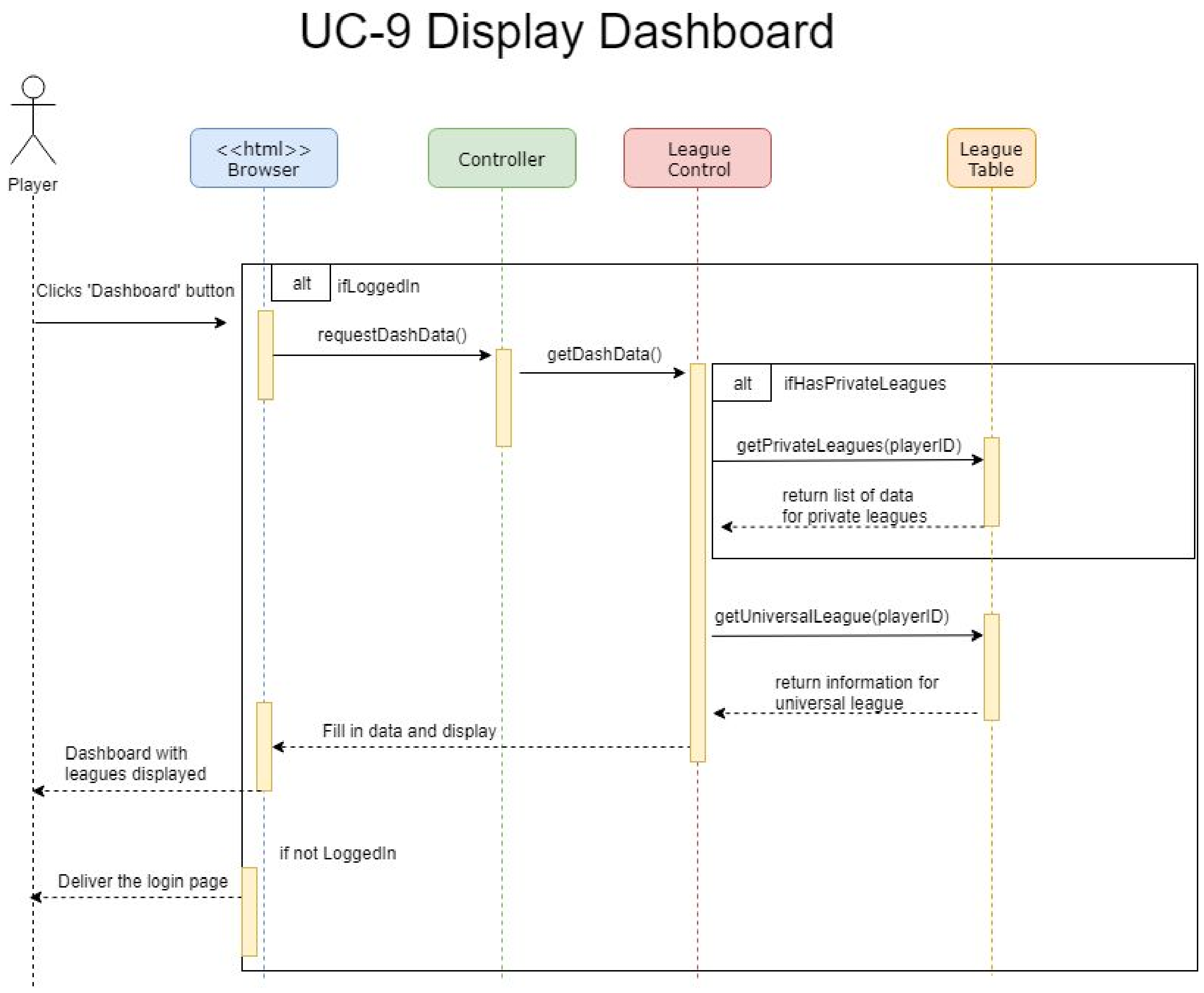
UC-8 - Selling a Stock





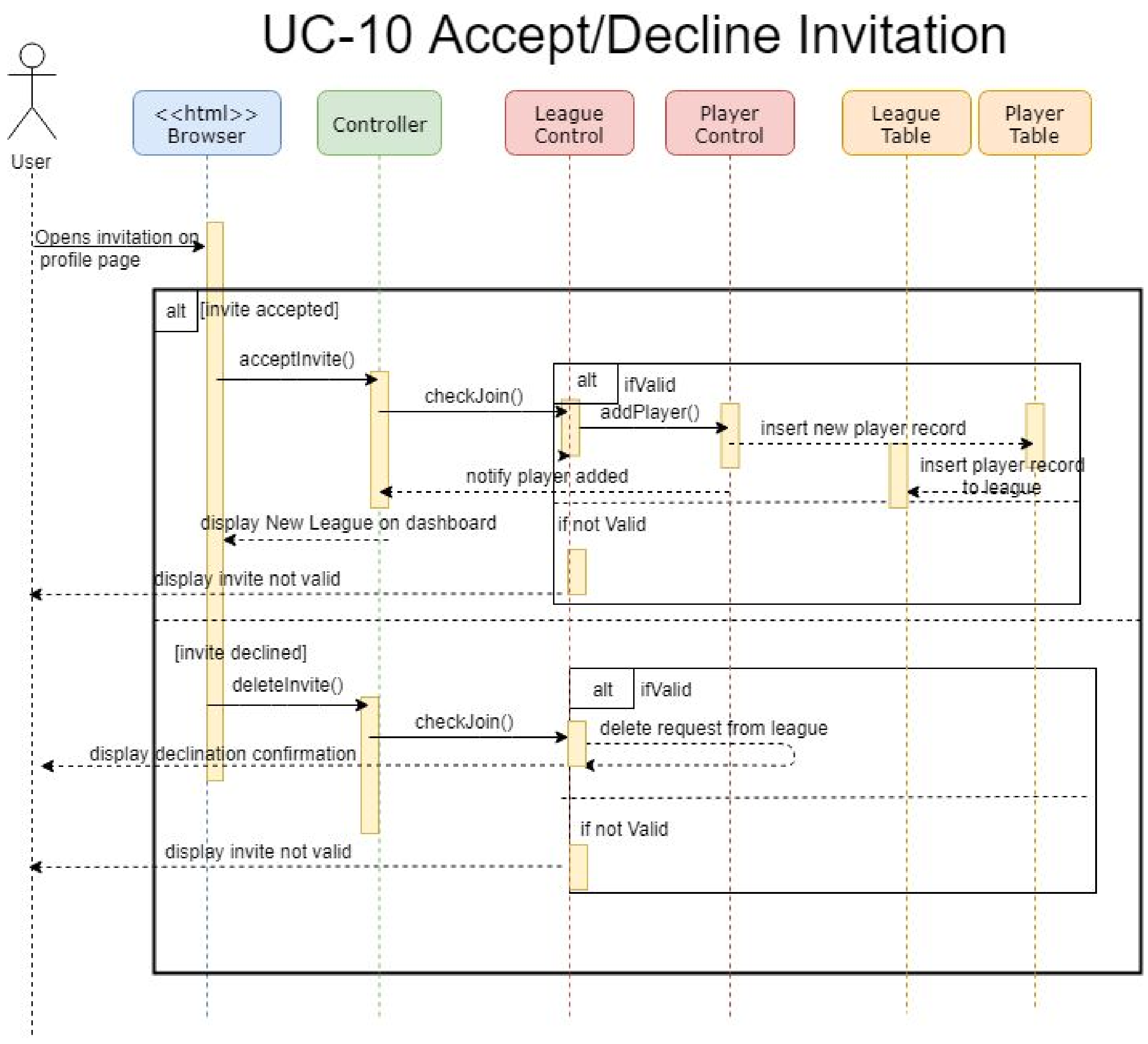
The above system sequence diagram shows the interactions of a player with the system to sell a stock. The player will start at the league dashboard page and will have to click a league where the system will redirect them to the individual league page of the league the player clicked. The redirection process requires the system controller to obtain all the necessary information about the individual league from the League Control. After being redirected to the individual league page with the displayed League Data, the player will have to click the sell button to declare that he wants to sell a stock from the current individual league. When the player clicks the sell button, the system controller will communicate to the stock asset control to obtain all the stocks the current player has within the individual league. The player will click a stock to sell and the system will retrieve the current market price of the stock from the Stock API and prompt the player with a sales form. The player will fill out the sales form with necessary transaction information and the system controller will have to communicate with the Sell Calculator to calculate the sales transaction, Stock Asset Control to updateStockAsset, Transaction Control to create a transaction record, and the player control to update the player's buying power.

UC-9 - Viewing Dashboard and League Leaderboards



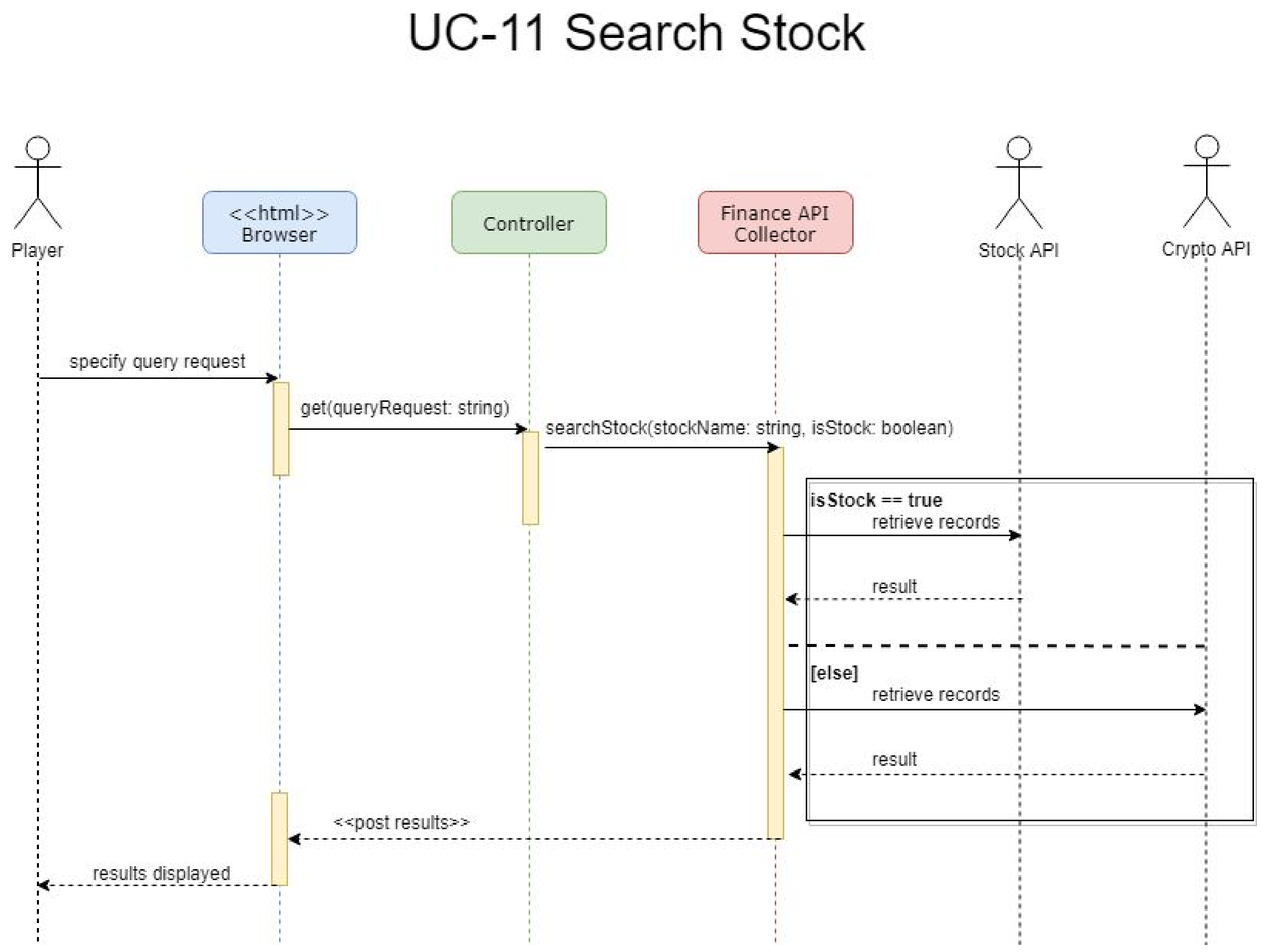
In order to view a user dashboard the user will click the dashboard button. The controller will then display all of the user's active leagues, which are associated with the leagueIDs stored in the user record. The league control will then identify which playerID corresponds to the current user and retrieve the data for that player. The league control will then transfer data to the browser which will display the user's player data in each league.

UC 10 - Accept/Decline Invitation



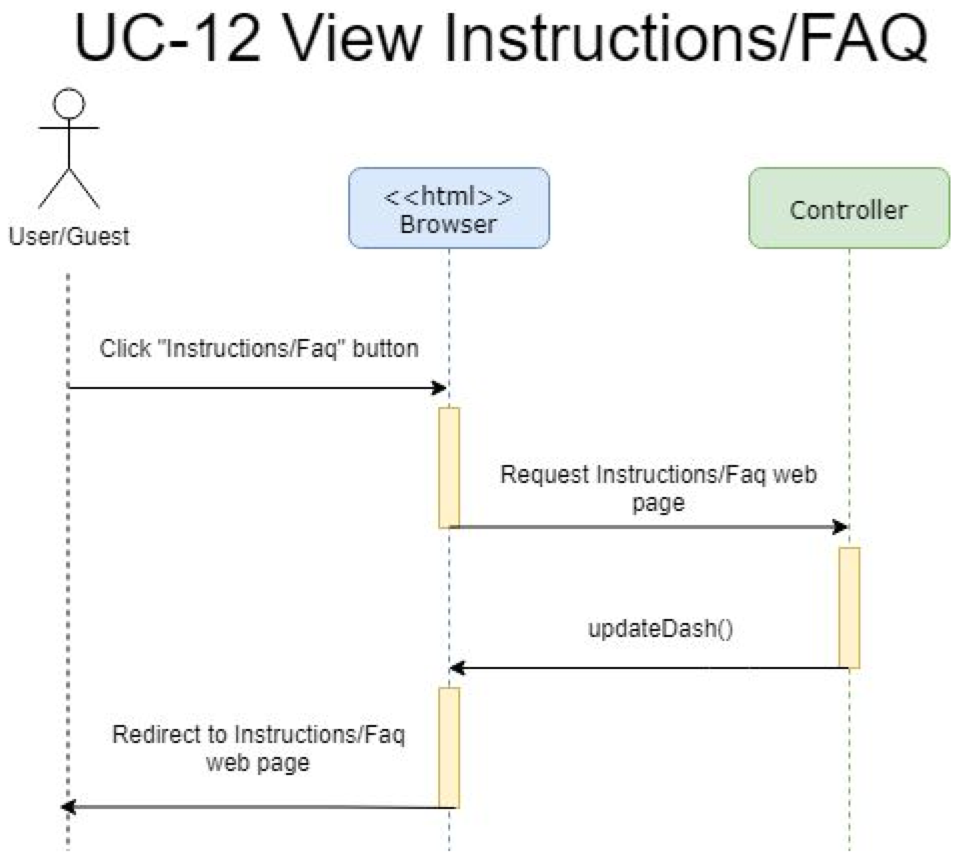
This use case controls whether or not a user invited to a private league accepts or declines the invitation. If they accept the invitation, the controller will communicate with the league control, league table, player control and player tables to create new player profiles and store the new data into the system. If they decline the invitation, the controller will communicate with the league controller to delete this request from the system. In both cases, the league control must check that the invitation request is valid.

UC-11 - Search Stock



A user may want to search for a stock or cryptocurrency in order to find information about its current market value before deciding to buy or sell their assets. When a player is searching for a stock, they initiate a query request which is sent to the Browser. The controller then processes the query request and redirects it to the Finance API Collector to search for most updated market information, which is updated by the minute. A boolean isStock determines if the Finance API Collector will retrieve records for the Stock API or the Crypto API. The result is then passed back to the Browser and displayed to the Player.

UC-12 - View Instructions/FAQ


UC-12 View Instructions/FAQ
User/Guest
<<html>> Browser
Controller
Click "Instructions/Faq" button
Request Instructions/Faq web page
updateDash()
Redirect to Instructions/Faq web page

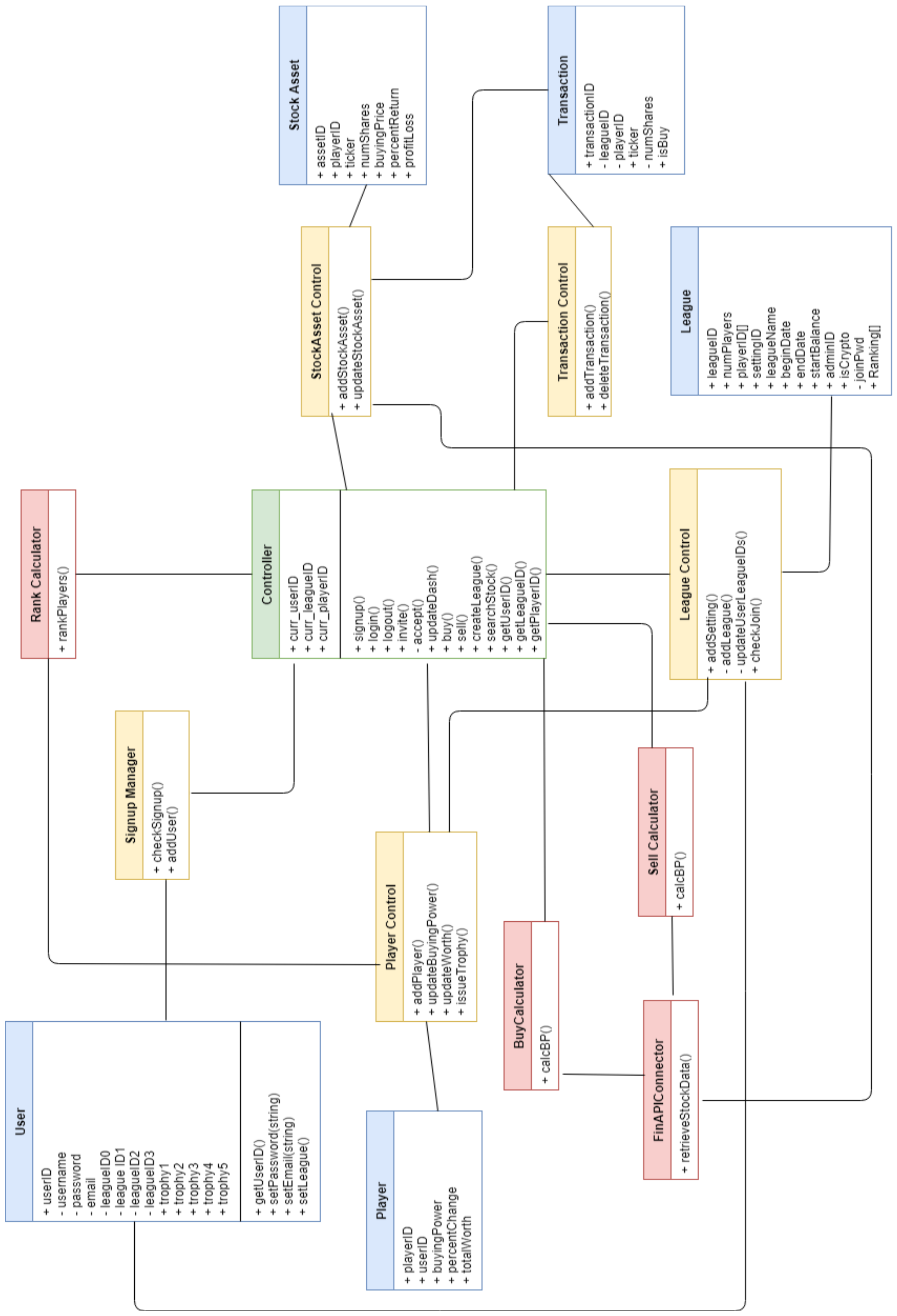
In this use case, a user or guest may decide to view the instructions or FAQ page to learn more about Titan Trading and making investments. A user or guest initiates this use case by clicking the “Instructions/FAQ” button, which is processed by the Browser. The Controller processes this requests and subsequently updates the current webpage to the corresponding web page.

CLASS DIAGRAM AND INTERFACE SPECIFICATION

Introduction

The class diagram has a visualization of all classes and variables that will be used to execute the core functionalities of *Titan Trading*. We employ the use of a primary controller to oversee all activity, and then individual class controllers to separate responsibilities.

Class Diagram


Stock Asset
+ assetID
+ playerID
+ ticker
+ numShares
+ buyingPrice
+ percentReturn
+ profitLoss
Transaction
+ transactionID
- leagueID
- playerID
+ ticker
- numShares
+ isBuy
StockAsset Control
+ addStockAsset()
+ updateStockAsset()
Transaction Control
+ addTransaction()
+ deleteTransaction()
League
+ leagueID
+ numPlayers
+ playerID[]
+ settingID
+ leagueName
+ beginDate
+ endDate
+ startBalance
+ adminID
+ isCrypto
- joinPwd
+ Ranking[]
Rank Calculator
+ rankPlayers()
Controller
+ curr_userID
+ curr_leagueID
+ curr_playerID
+ signup()
+ login()
+ logout()
+ invite()
- accept()
+ updateDash()
+ buy()
+ sell()
+ createLeague()
+ searchStock()
+ getUserID()
+ getLeagueID()
+ getPlayerID()
League Control
+ addSetting()
- addLeague()
- updateUserLeagueIDs()
+ checkJoin()
Signup Manager
+ checkSignup()
+ addUser()
Player Control
+ addPlayer()
+ updateBuyingPower()
+ updateWorth()
+ issueTrophy()
Sell Calculator
+ calcBP()
BuyCalculator
+ calcBP()
FinAPIConnector
+ retrieveStockData()
User
+ userID
- username
- password
- email
- leagueID0
- league ID1
- leagueID2
- leagueID3
+ trophy1
+ trophy2
+ trophy3
+ trophy4
+ trophy5
+ getUserID()
+ setPassword(string)
+ setEmail(string)
+ setLeague()
Player
+ playerID
+ userID
+ buyingPower
+ percentChange
+ totalWorth

Data Types and Operation Signatures

Controller

The controller is responsible for partitioning user actions into a sequence of steps involving the other classes of the system. The controller is the primary control point for all other control classes, and its methods are a precursor to other class actions.

Attributes

- **curr_userID : int**
  Indicates current user initiating an action
- **curr_leagueID : int**
  Indicates current league that is participating in the action
- **curr_playerID : int**
  Indicates current player that is participating in the action

Methods

+ **signup(username : String, pwd : String, email : String) : bool**
  This method is used to signup a new user to become a member of Titan Trading. User data signup data is stored in the *User Table* of the database.
+ **login(username : String, pwd : String) : bool**
  This method is used to log the user in to Titan Trading.
  User data entered will be checked with the data in the User Table of the database.
+ **logout() : bool**
  This method is used for the user to log out of their account.
+ **invite(email : String)**
  This method is used for a league admin to invite new players.
- **accept()**
  This method is used for new players to accept invitations to a league.'
+ **updateDash(c_userID : int)**
  This method is used to update the users dashboard while logged in.
+ **buy(c_leagueID : int, c_playerID : int, ticker : String, numShares : int)**
  This method is for the user to buy shares while in a league.
+ **sell(c_leagueID : int, c_playerID : int, ticker : String, numShares : int)**
  This method is for the user to sell shares from their portfolio within a league.
+ **createLeague(c_userID : int)**
  This method is used to create a new league.
+ **searchStock(ticker : String)**
  This method is used to search for a specific stock
+ **getUserID() : int**
  This method is used to retrieve the UserID for a specific user.

+ **getLeagueID(): int**
  This method is used to retrieve the LeagueID for a specific league.
+ **getPlayerID() : int**
  This method is used to retrieve the PlayerID for a specific player.

SignupManager

The signup manager is responsible for all signup and login activity. It is the primary interface to check username, passwords, and emails.

Methods

+ **checkSignup(username : String, pwd : String, email : String) : bool**
  This method checks whether the data given for a new account is valid. This will contain checking for a valid and unique email, unique username, and valid password.
+ **addUser(username : String, pwd : String, email : String) : bool**
  This method is used to add a new user's data to the User Table in the Database

User

This class represents a user and contains user login information and participating leagues. We currently allow the user to be part of the Universal League and an additional 3 private leagues.

Attributes

+ **userID : int**
  Identifies a unique user.
- **password : string**
  Passcode for a unique user to log-in to their account.
- **email : string**
  Unique email for a user to log-in to their account.
- **leagueID0 : int**
  Unique integer indicating the 0th league they are participating in; typically a Universal league.
- **leagueID1 : int**
  Unique integer indicating the 1st league they are participating in.
- **leagueID2 : int**
  Unique integer indicating the 2nd league they are participating in.
- **leagueID3 : int**
  Unique integer indicating the 3rd league they are participating in.
+ **trophy1 : String**
  Indicates whether a user has achieved trophy 1.
+ **trophy2 : String**
  Indicates whether a user has achieved trophy 2.

- **trophy3 : String**
  Indicates whether a user has achieved trophy 3.
- **trophy4 : String**
  Indicates whether a user has achieved trophy 4.
- **trophy5 : String**
  Indicates whether a user has achieved trophy 5.

Player

This class represents a player in a particular league.

Attributes

- **playerID : int**
  Unique ID associated with a unique player.
- **userID : int**
  UserID who is running the player.
- **buyingPower : float**
  Indicates how much money a player has in order to buy stock or cryptocurrency.
- **percentChange : float**
  Indicates the percent change a player has made on a daily basis.
- **totalWorth : floats**
  Indicates a player's total worth, which is the sum of their buying power and assets.

StockAsset

This class represents a specific stock and all of the data/information associated with it.

Attributes

- **assetID : int**
  Unique integer identifying the asset a player has.
- **playerID : int**
  Identifies which player owns the asset.
- **ticker : String**
  Identifies the ticker symbol for the asset (i.e., BTC, APL).
- **numShares : int**
  Identifies how many shares the player bought.
- **buyingPrice : float**
  Identifies for how much the player bought the asset.
- **percentReturn : float**
  Identifies the current return on investment in terms of a percentage.
- **profitLoss : float**

Identifies the total value of profit or loss on the original purchase.

StockAssetControl

The StockAssetControl class manages the Stock Assets and can perform actions such as adding and updating them.

Methods

+ **addStockAsset(curr_transaction : Transaction)**
  Adds a stock or holding to the StockAsset Table in the database. Each holding is associated with a league player.
+ **updateStockAsset(curr_transaction : Transaction)**
  Updates a holding in the StockAsset Table when a player buys or sells shares of a stock he/she already owns.

Transaction

This class represents a transaction, or a buy/sell operation. Each transaction is associated with the league, player, and specific stock asset.

Attributes

+ **transactionID : int**
  Identifies current transaction.
+ **leagueID : int**
  Identifies the unique league under which the transaction falls.
+ **playerID : int**
  Identifies the unique player that submitted the order for the transaction.
+ **ticker : String**
  Identifies the ticker symbol associated with the specific buy or sell order.
+ **numShares : int**
  Identifies how many shares were bought at the time of the transaction.
+ **isBuy : bool**
  Identifies whether the transaction is a buy or sell order. If isBuy is true, it is a buy order; else, it is a sell order.
+ **buysellPrice : int**
  Indicates the current price of the stock, retrieved from the API. This the latest stock price we will be using for all asset management

TransactionControl

The transaction control manages all the operations pertaining to the transaction table of the database.

Methods

+ **addTransaction(curr_tr : Transaction)**
  Adds a transaction to the database.
+ **deleteTransaction(curr_tr : Transaction)**
  Deletes a transaction from the database.

PlayerControl

The player control handles all player activity, which includes updating buying power, worth, and percent change whenever a player makes a transaction. The player control also keeps track of player trophies.

Methods

+ **addPlayer(curr_user : User, curr_league : League)**
  This method updates a league with any new players to be added
+ **updateBuyingPower(curr_tr : Transaction)**
  This method updates a player's buying power after a transaction
+ **updateWorth(curr_tr : Transaction)**
  This method updates a user's net worth at any point
+ **issueTrophy(trophyName : String)**
  This method gives a trophy to the winning player of a league

League

This class represents all the information for a league. It helps to complete track of which players are competing in the league, who the league manager is, and other league settings.

Attributes

+ **leagueID : int**
  Integer to identify the unique league.
+ **numPlayers : int**
  Indicates how many players are playing in this league.
+ **playerIDs[20] : int**
  Array of playerIDs to identify the players in this league. We currently allow for a maximum of 20 players in each league. (With the exception of the Universal League)
+ **settingID : int**
+ **leagueName : String**
  Indicates the name of the league.
+ **beginDate : datetime**
  Indicates the date and time at which the league was created.
+ **endDate : datetime**

Indicates the date and time at which the league competition will end.

+ **startBalance : float**
  Indicates the starting balance each player in the league will begin at.
+ **adminID : int**
  Indicates the user ID of the league manager.
+ **isCrypto : bool**
  Indicates whether the league will be investing in cryptocurrency or stocks.
+ **joinPwd : String**
  Indicates the password players must enter in to join the league.
+ **Ranking[num_players] : intRanking[num_players] : int (playerID int)**
  Current ranking of all players in terms of the league's player IDs.

LeagueControl

The LeagueControl class manages each specific league and is used whenever players are added or settings are edited.

Methods

+ **addSetting(curr_league : League)**
  Add setting record to database.
+ **addLeague(curr_league : League)**
  Add league record to database.
+ **updateUserLeagueIDs(curr_leagueID : int, curr_userID : int)**
  Update list of the current user's active leagues when adding a player to a league.
+ **checkJoin(c_leagueID : int, c_playerID : int, c_userID : int) : bool**
  Notifies whether a player has successfully joined a particular league.

FinAPIConnector

The FinAPIConnector class uses the Finance API to gather stock data to be used.

Methods

+ **retrieveStockData(ticker : String) : String**
  Retrieves all stock data from the Finance API including the timestamp, open, high, low, close, and volume of the particular stock. Returns a string in JSON format.

RankCalculator

The RankCalculator class will order the players in a league by their ranking based on total earnings.

Methods

- **rankPlayers(leagueID : int) : int ***
  Ranks players in a particular league based on totalWorth. Returns a pointer to the sorted array of players.

BuyCalculator

The BuyCalculator class will update a player's buying power based on the old buying power and and the

Methods

- **calcBP(c_playerID : int, transactionID : int) : int**
  Calculates buying power for a player based on the input transaction. For a buy, subtracts price of stock * amount of shares from current buying power.

SellCalculator

The SellCalculator class will calculate the buying power after a sell order has been made.

Methods

- **calcBP(c_playerID : int, transactionID : int) : int**
  Calculates buying power for a player based on the input transaction. For a sell order, the calculator will add the market price of the asset * amount of shares to current buying power.

Traceability Matrix

						Domain Concepts											
Class	Controller	User Table	Login Manager	Signup Manager	Ranking Calculator	Setting Table	Player Table	Player Control	League Table	League Control	Financial API Connector	Transaction Table	Transaction Control	Asset Table	Stock Asset Control	Buy Calculator	Sell Calculator
Controller	x																
SignupManager			x	x													
User		x	x														
Player							x	x									
StockAsset														x			
StockAssetControl														x	x		
Transaction												x					
TransactionControl												x	x				
PlayerControl			x				x	x									
League									x								
LeagueControl						x			x	x							
FinAPIConnector											x						
RankCalculator					x												
BuyCalculator																x	
SellCalculator																	x

The Controller, RankingCalculator, BuyCalculator, and SellCalculator all have classes with the same name as their domain concept. For each of these classes, we decided that the domain concept name was suitable for one class and so we kept the same names.
The controller class will be the central control point for all other classes that make up the game.
The RankCalculator, BuyCalculator, and SellCalculator are all named for their function, thus they were the only classes derived from their respective domain concepts.
The SignupManager will keep track of all signup and login activity, thus it is derived from the LoginManager and the SignupManager.
The FinAPIConnector comes from the Financial API Connector domain concept. We chose to shorten the name to FinAPIConnector so it wouldn't be as long.
From the User Table and Login Manager domain concepts, we created the User class, which will create the different users in the game and keep track of them.
Next, we created the Player class from the Player Table and PlayerControl concepts, which will create the players represented in a particular league.
From the LoginManager, PlayerTable and PlayerControl domain concepts, we decided to create a PlayerControl class. This class will keep track of all player activity and each players trophies. We also

created StockAsset and StockAssetControl classes derived from the asset table and stock asset control domain concepts to create the actual stock objects, keep track of any information associated with it, and manage the stock assets in the game.
Next, we created a Transaction class from the Transaction Table domain concept to create the individual transactions that take place.
The TransactionControl class, derived from the Transaction Table and Transaction Control domain concepts will keep track of all the transactions that occur in the game, and what league, player, and stock asset the transaction is associated with.
Next, we have the League class, which comes from the LeagueTable domain concept.
The League class will create the individual leagues and their settings, while the LeagueControl class, derived from the LeagueControl, SettingTable, and League Table domain concepts, will be the central point of management for all the different leagues in the game.

SYSTEM ARCHITECTURE AND SYSTEM DESIGN

Architecture Styles

For our project we will be using various Software Architecture styles to help create a reliable and fast product. Many of these styles are well known and used across all different types of software projects, making them well documented and easy to implement in our project. The styles we are using in our project include Representational State Transfer (REST), Data-Centric, Plug-ins, and Client-Server styles.

Representational State Transfer:
Representational State Transfer (REST) type APIs make it very simple to update database information from client side. In our project will be storing a lot of information inside a database, including stocks held and information about sales. When a stock transaction is performed, a REST type API will be used by the client to put this information in the database. Setting this up is streamlined due to the fact that we are using the Django framework with Python.

Data-Centric:
Our application is data-centric by its very nature. We need to keep track of not just what stocks that the user is holding, but the live price of the stock, league standings information. Additionally, the client must be given all of this data in an understandable and easy to digest way. To do this we will be leveraging the REST API system and a SQL database in conjunction with various other stock information APIs to gather price information for the clients. This information will be stored in the databases, where the user can access it by means of a transaction or information query.

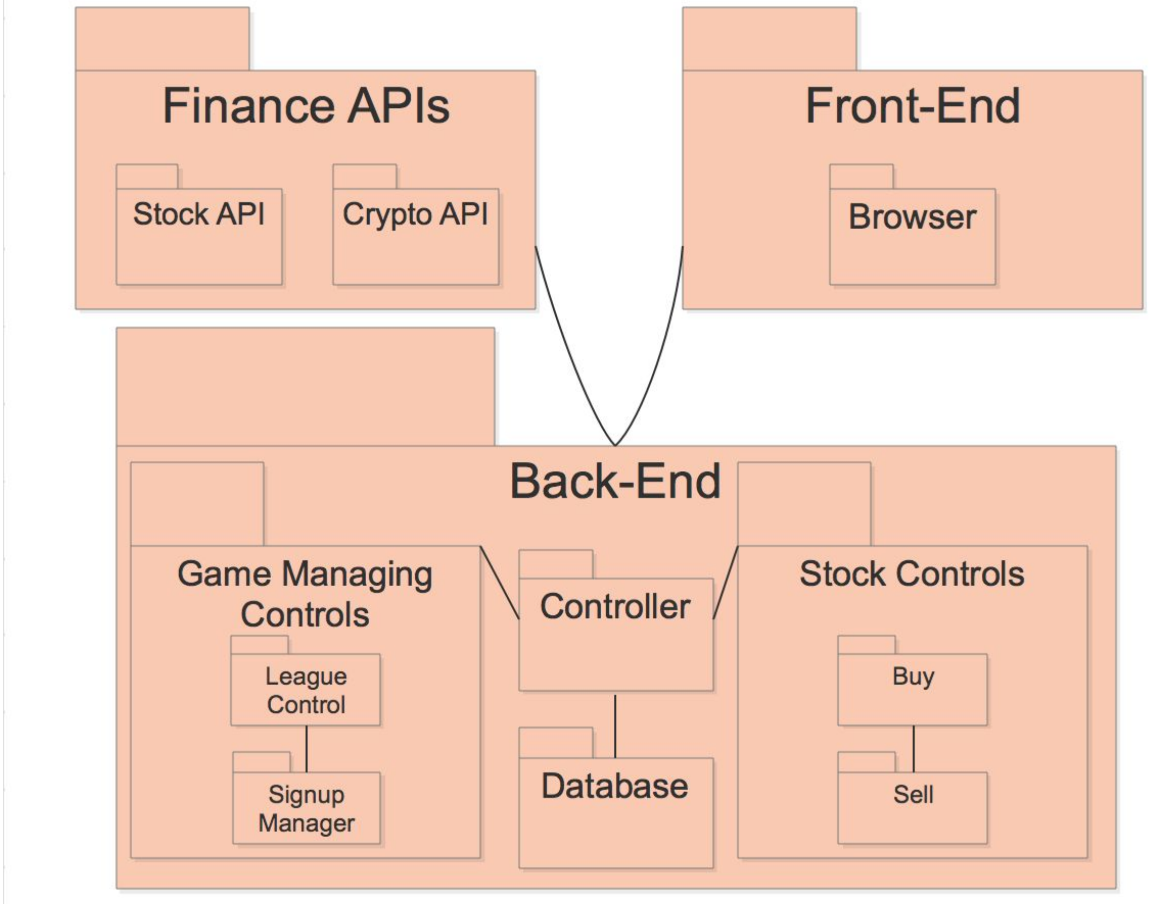
Plugins:
The user will not interact directly with our plug-in system, but will reap the benefits of it when using our website. Python and Django are both plug-in heavy frameworks. Many of the difficult

challenges of other programs can be solved with a package available on the pip repository. This will allow for us to have a speedy development time while being able to concentrate on the end user experience.

Client-Server:
The Client-Server style ties directly in with the Data-Centric and REST system styles. The client will be interracting with the server a lot when browsing and using the website. Almost all of the content we will be serving is dynamic and needs to be updated both by the user and outside data. When it is updated, the user must be able to request and see this new information.

Identifying Subsystems

(created with Edraw Max: https://www.edrawsoft.com/download-edrawmax.php)



The project incorporates three main subsystems: the front-end, the back-end, and the finance APIs.

The front-end encompasses all the user interacts with. It includes the browser subsystem, which lets the user browse the various web pages of the website and interact with the web widgets.

When the user interacts with the web page that requires the system to need to consult the database or finance APIs, the front-end subsystem will communicate the necessary tasks to the back-end system, which handles all the work. The front-end subsystem simply waits for the desired response from the back-end system and presents the updated information to the user.

The finance APIs are used to retrieve real-time stock market data and news. The stock API subsystem gives information relating to stock prices and sector performances while the crypto API does the same for cryptocurrencies. The only subsystem that communicates with the Finance API subsystem is the back-end subsystem, which requests the API subsystem for the required information.

The back-end subsystem handles the majority work in the project. It first holds the controller, which manages communication between the various subsystems. The database subsystem allows for additional, removal, and access of the data in the application. Other subsystems that need use of the database submit their request to the controller, which notifies the database subsystem to query the requested information.
The game managing controls subsystem handles the managerial portion of the league system. It contains the league control subsystem, which handles league creation, settings, players, and invites.
The signup manager is in charge of user accounts on the application. When a user creates an account, it interacts with the database to make sure that the creation request complies with any existing constraints.
The stock controls subsystem checks that any stock operations preserve the integrity of the game. It calculates the purchase/sale, verifies the asset prices with the API, and checks the user balance with the database.

Persistent Data Storage

Titan Trading is a platform that provides Users the ability to create accounts, join leagues, and store stock and crypto assets. Persistent Data Storage is a key function that must be incorporated into the backend to keep track of player data and provide the most realistic trading experience possible.

For the backend, we will be using PostgreSQL, an enhanced version of SQL, to create relational database tables that store specific information. The persistent tables that will be made include: User Table, Setting Table, Player Table, League Table, Transaction Table, and Asset Table.
SQL Database Schema Query with a short description of each table's use of Persistent Data:

```
CREATE TABLE ‘User’(
        ‘userID’ INT,
        ‘username’ VARCHAR(20),
        ‘password’ VARCHAR(20),
        ‘email’ VARCHAR(40),
        ‘leagueID0’ INT,
        ‘leagueID1’ INT,
        ‘leagueID2’ INT,
        ‘leagueID3’ INT,
        PRIMARY KEY(‘userID’)
)
```
Description: The user table will hold the general Titan Trading account information. Fields such as username,password, and email must be persistent because a user’s account information should definitely still exist after a single execution. The leagueIDs relate a User with the specific leagues he is a part of. All of the leagueIDs must also be persistent because once a person joins a league, the person should stay as a member of that league until the league is deleted or the person is kicked out.

```
CREATE TABLE ‘Setting’(
        ‘settingID’ INT,
        ‘leagueID’ INT,
        ‘Begindate’ DATETIME,
        ‘Enddate’ DATETIME,
        ‘Startingbalance’ FLOAT,
        ‘AdminID’ INT,
        ‘Crypto’ BOOL,
        ‘Joinpassword’ VARCHAR(20),
        FOREIGN KEY(‘leagueID’) REFERENCES ‘League’(‘leagueID’),
        PRIMARY KEY(‘settingID’)
)
```
Description: The settings table will hold the general Titan Trading league information. The settings table fields will be filled out and a new Setting tuple will be created when a person creates a private league. All information in the Setting table must be persistent because a league should stay created until the league is deleted by the admin.

```
CREATE TABLE ‘Player’(
        ‘playerID’ INT,
        ‘userID’ INT,
        ‘buyingpower’ FLOAT,
        ‘percentchange’ FLOAT,
        ‘totalworth’ FLOAT,
        FOREIGN KEY(‘userID’) REFERENCES ‘User’(‘userID’),
        PRIMARY KEY(‘playerID’)
)
```
Description: The player table will hold the information on how a user is doing in a particular league. Fields such as buyingpower, percentchange, and totalworth must be persistent so that the progress of the players can be saved.

```
CREATE TABLE ‘League’(
        ‘leagueID’ INT,
        ‘playerID’ INT,
        ‘settingID’ INT,
        FOREIGN KEY(‘playerID’) REFERENCES ‘Player’(‘playerID’),
        FOREIGN KEY(‘settingID’) REFERENCES ‘Setting’(‘settingID’),
        PRIMARY KEY(‘leagueID’)
)
```

Description: The league table relates the playerID with the settingID. While the settingID will describe the information about the league, the playerID will provide the monetary information about the player within that league. This relationship must be kept persistent so that a player can stay part of a private league.

```
CREATE TABLE ‘Transaction’(
        ‘transactionID’ INT,
        ‘leagueID’ INT,
        ‘Amount’ FLOAT,
        ‘Stock’ VARCHAR(20),
        ‘Shares’ INT,
        ‘Buy’ BOOL,
        FOREIGN KEY(‘leagueID’) REFERENCES ‘League’(‘leagueID’),
        PRIMARY KEY(‘transactionID’)
)
```

Description: The transaction table will be a record of all the transactions that a player takes with a league. This must be persistent so that the players can see the history of their transactions.

```
CREATE TABLE ‘Asset’(
        ‘stockassetID’ INT,
        ‘playerID’ INT,
        ‘Stock’ VARCHAR(20),
        ‘Shares’ INT’
        FOREIGN KEY(‘playerID’) REFERENCES ‘Player’(‘playerID’),
        PRIMARY KEY(‘stockassetID’)
)
```

Description: The asset table stores information about the users current stock assets. This information must be kept persistent so that the user won’t lose his assets upon log out.

Network Protocol

The communication protocol Titan Trading has picked is Hypertext Transfer Protocol (HTTP). The main goal of our project is to create a functional stock market game that can communicate with users through a website. HTTP is the communication protocol used by the World Wide

Web and is how web browsers and web servers communicate with each other the internet. Our servers will be able to communicate with the user's browser and respond to user actions with the appropriate commands instantly. HTTP will also enable our application to seamlessly update real-time stock prices and communicate these to our users through our website. Through this protocol, users will be able to access different web pages through hyperlinks by interacting with our unique interface on our website and have the full Titan Trading experience.

Global Control Flow

Execution Order

Titan Trading is a primarily event-driven system because our website is mostly based on user actions. Most of our functionalities and use cases are initiated by the user. As shown in the user interface specifications, there are multiple decisions that a user can decide to take when navigating throughout our website. Therefore, every user will not go through the same steps and web pages every time. Rather, the system will wait in a loop until a user makes a decision to navigate to participate in any of the functionalities Titan Trading offers. For example, after signing in, user1 may decide to go to the Glossary page to read a definition about trading; user1 may then proceed to his League Dashboard and enter the Universal League to view the leaderboard. In contrast, after signing in, user2 may decide to go straight to the League Dashboard and create a new league. User2 will then begin to invite players to join her league.

Although Titan Trading is a primarily event-driven system, some system processes are strictly linear, as shown by the system sequence diagrams. The linearity is present in the interactions between our domain concepts and players. For example, in Use Case 4 (Logout), the user will take initiate an event by clicking the "Logout" button. Then, for every user, control will shift to the Browser, followed by the Controller, followed by the Browser, and back to the user. Other instances of linearity are present for certain events. For example, if a guest wants to begin investing, the guest must first register with Titan Trading, join a league, and traverse to the league's specific buy page in order to start investing.

Time Dependency

Titan Trading is a time-dependent system. Titan Trading uses real-time market prices from our finance API, which can be updated every minute. In addition, our website displays a host of widgets, such as graphs and tickers, that are also dependent on real-time data. Titan Trading utilizes system wide timers, in order to correctly process actions.

For buy and sell orders, there is a Finance API timer and Stock Market Open/Close timer. These two timers are utilized to update real-time market data, which can be described as follows.

- Finance API timer - This timer will notify the system after a minute has passed. After each minute passed, the market data will be updated. This timer is particularly useful for buy and sell orders, so that the market price at the exact time the order was submitted will be used to update critical player parameters, such as total worth and buying power.

- Stock Market timer - This timer will notify the system once the stock market open and closes. Between these intervals of time, any orders submitted will not be accepted by Titan Trading and will notify the user that the market has closed.
- Percent Change timer - This timer will notify the system when to update the percent change for all players. The percent change for each player will be recalculated on a daily basis.

Concurrency

Concurrency constructs are necessary for websites that are managing multiple users. Python's Django already provides a framework for multiprocessing and data safety with synchronization constructs. Within Django, multiprocessing is necessary to process and execute the concurrent actions that concurrent users may decide to take. Synchronization constructs are necessary for data safety, so that at any point in time, all databases used within Titan Trading have consistent data. Specifically, Django uses synchronization locks in order to maintain data consistency (Benita).

Hardware Requirements

Overview:

As a whole, the majority of computational resources are used on the server-side, with the use of Django, a PostgreSQL database, and a python backend that handles front-end interaction. This minimizes the need for large hardware on the user side, and allows as many devices as possible to be supported.

Internet Connection:

The system relies on the use of APIs and widgets that are available for use all over the web. As such, a persistent internet connection is required to use the system to pull data and verify the legitimacy of transactions. This also requires a 99%+ uptime for the server infrastructure, in order to maintain consistency of use for all users. For user connections, a broadband connection should provide enough bandwidth. Since most data transmissions will be done using HTTP and TCP protocols, more bandwidth will result in faster performance.

Server Space and Size:

Thanks to the proliferation of Amazon AWS, Google Cloud, Microsoft Azure and other cloud service providers, the servers that we select are easily scalable and adjustable. Meeting demand and matching required disk space is simple using the modern interfaces of these cloud services. This scaling in size can occur over any period of time. Our PostgreSQL database can be scaled as user count grows and our space usage during a given day can also be scaled to match demand during the busy trading hours every day.

User Device:

Users will be able to access the site through any modern browser that support Javascript, HTML5, and CSS3. This includes, but is not limited to:

- Google Chrome
- Mozilla Firefox
- Microsoft Edge
- Apple Safari
- Opera

The site is optimized to run in a desktop orientation and is best accessed from a laptop or desktop. This does not rule out the use on mobile devices, however the experience may be different and unoptimised for the screen size. As such a 1024 × 768 is the specified minimum screen resolution.

Project Management

Report 2		Team 7								
		Member 1	**Member 2**	**Member 3**	**Member 4**	**Member 5**	**Member 6**	**Member 7**	**Member 8**	**Member 9**
Category	**Points**	**Steven Adler**	**Kristene Aguinaldo**	**Timothy Liu**	**Nicholas Lurski**	**Avanish Mishra**	**Safa Shaikh**	**Brooks Tawil**	**Kristian Wu**	**David Zhang**
Project Management	16	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Interaction Diagrams	30	11.11%	11.11%	11.11%	11.11%	11.11%	11.11%	11.11%	11.11%	11.11%
Class Diagram and Interface Specification	11	25.00%	0.00%	0.00%	0.00%	0.00%	25.00%	0.00%	25.00%	25.00%
System Architecture and System Design	14	0.00%	20.00%	20.00%	20.00%	20.00%	0.00%	20.00%	0.00%	0.00%
Algorithms and Data Structures	4	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
User Interface Design and Implementation	11	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%

Design of Tests	12	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Plan of Work	2	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%	0.00%
Raw Points	100	6.0833	6.1333	6.1333	6.1333	6.1333	6.0833	6.1333	6.0833	6.0833

References

Benita, Haki. “How to Manage Concurrency in Django Models – Haki Benita – Medium.”*Medium*, Medium, 7 July 2017, medium.com/@hakibenita/how-to-manage-concurrency-in-django-models-b240fed4ee2.