Mock Stock

Project Team: Luke Orr, Theodore Hecht, Sebastian Schwagerl, Jonathan Langston

Table of Contents

**1. Project Definition (**100 - 200 words**)** – *Group responsibility*

*Some people want to learn about and practice stock trading without any financial commitment. Because of this, we have decided to create a mobile app that will allow people to learn about the stock market. Our users will be able to learn using a “hands on” approach, allowing them to trade “mock” stocks with virtual currency. This project will be created as a mobile app for the iOS operating system using Swift 4 and XCode, with a restful web API backend in .NET Core with a SQL database layer to hold user details, all created by the Mock Stock team.*

**2. Project Requirements** – *Group responsibility*

* Functional
  + The backend API must be able to query real-time financial data (stock prices) from a third party API.
  + The backend API must have middleware to authenticate users who make requests from the mobile app. (Our authentication solution may also have database requirements, like storing usernames/emails and hashed passwords.)
  + The backend API must be able to store and retrieve information about the user’s virtual portfolio, exposed as a set of REST API endpoints that the frontend can access.
  + The frontend should let users buy/sell virtual stocks at real-world prices.
  + The frontend should let users examine their portfolio contents.
  + The frontend should allow users to display detailed information on specific companies.
  + The frontend should have a search/sort button to find specific stocks that the users wants.
  + The frontend should have a button to change the graphical timeline of the price history of the stock.
* Usability
  + User interface
    - Efficiency of use
    - Intuitive design
  + Performance
    - UI must be fast and responsive
    - API calls needs to be efficient
    - Database calls need to run quickly
* System
  + Hardware
    - Frontend: iOS device (phone), 6s and above
  + Software
    - Frontend: iOS current version
    - Backend: PaaS (Azure App Service)
  + Database
    - SQL database options: MySQL, PostgreSQL, MariaDB
* Security
  + Basic authentication middleware over HTTPS. Frontend sends username/password combo on each request encrypted over HTTPS, and backend middleware authenticates the request using this information.

**3. Project Specification** – *Group responsibility*

* Focus / Domain / Area
  + Stock market learning tool
* Libraries / Frameworks / Development Environment
  + Frontend: Swift4, UIKit (and other native iOS libraries), XCode
  + Backend: C#, .NET Core 2.X
* Platform (Mobile, Desktop, Gaming, Etc)
  + Mobile (iOS)
* Genre (Game, Application, etc)
  + Game/Education

**4. System – Design Perspective** – *Group responsibility*

* Identify subsystems – design point of view

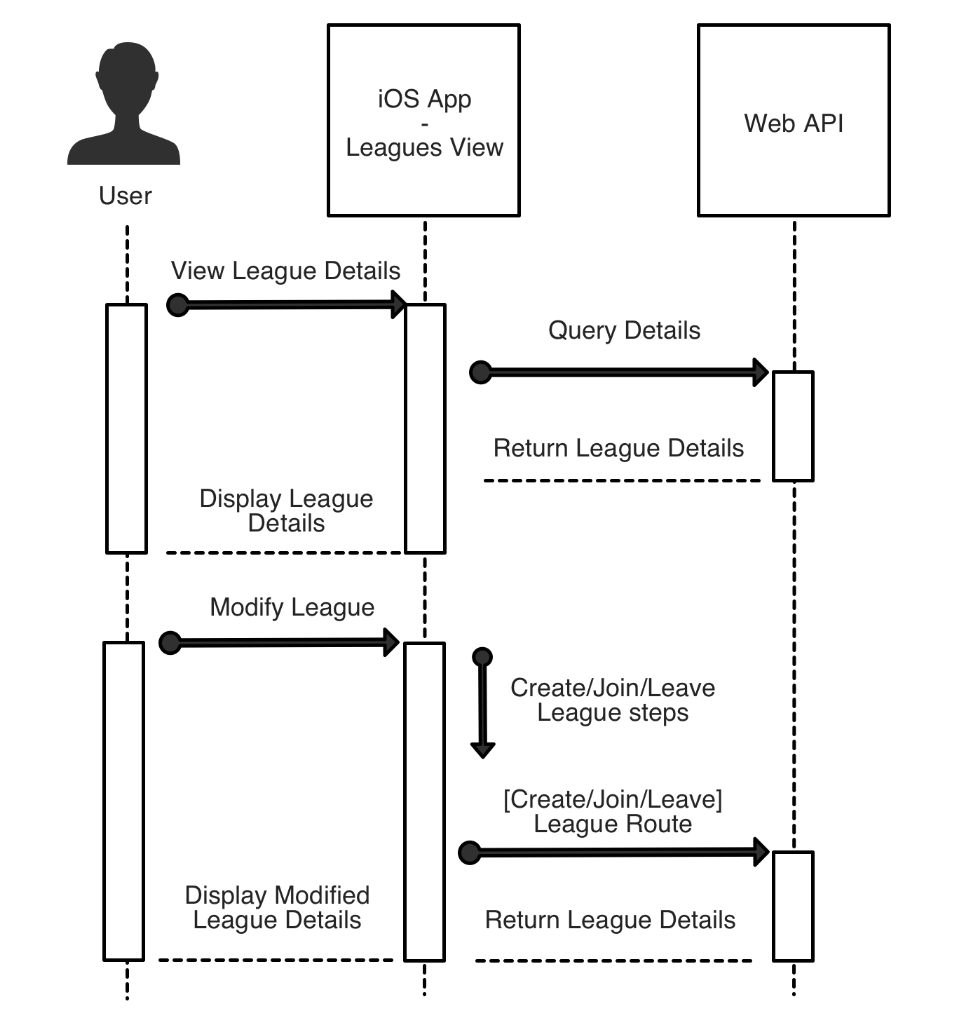
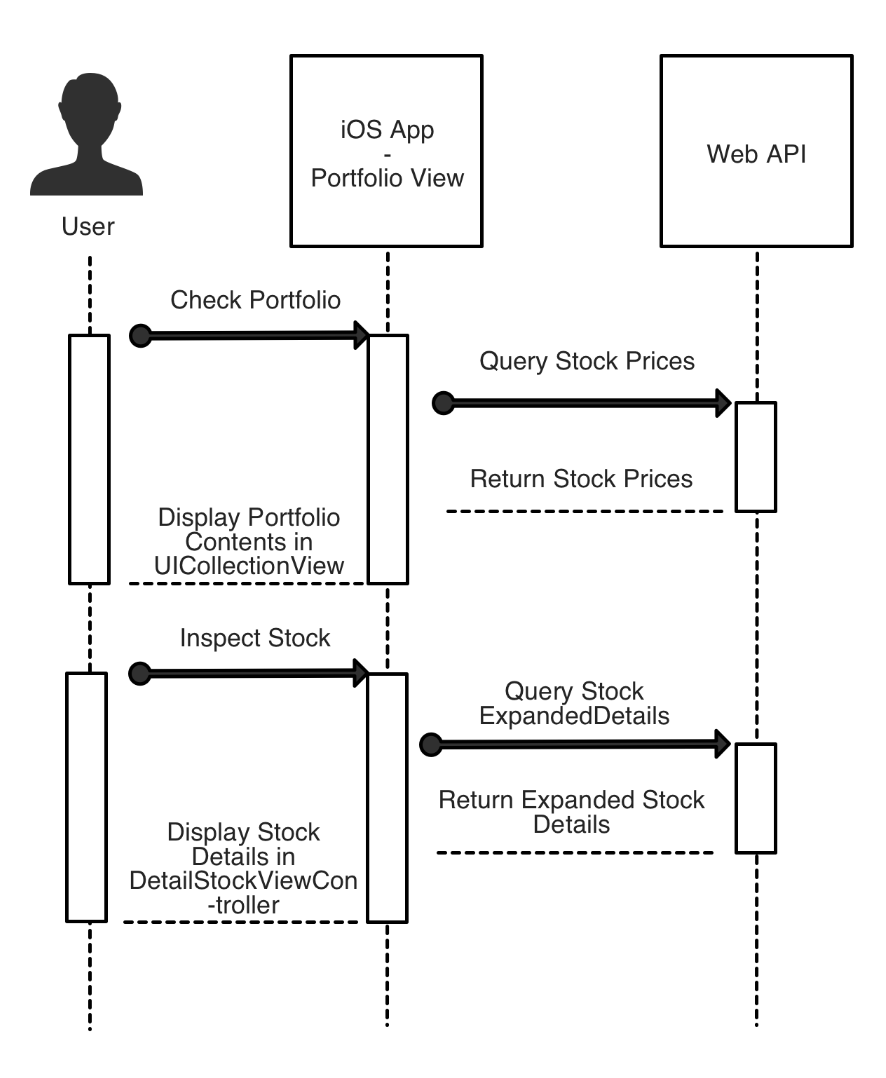
Front End Screens: (Portfolio, Leagues, Marketplace, Detailed Stock View).

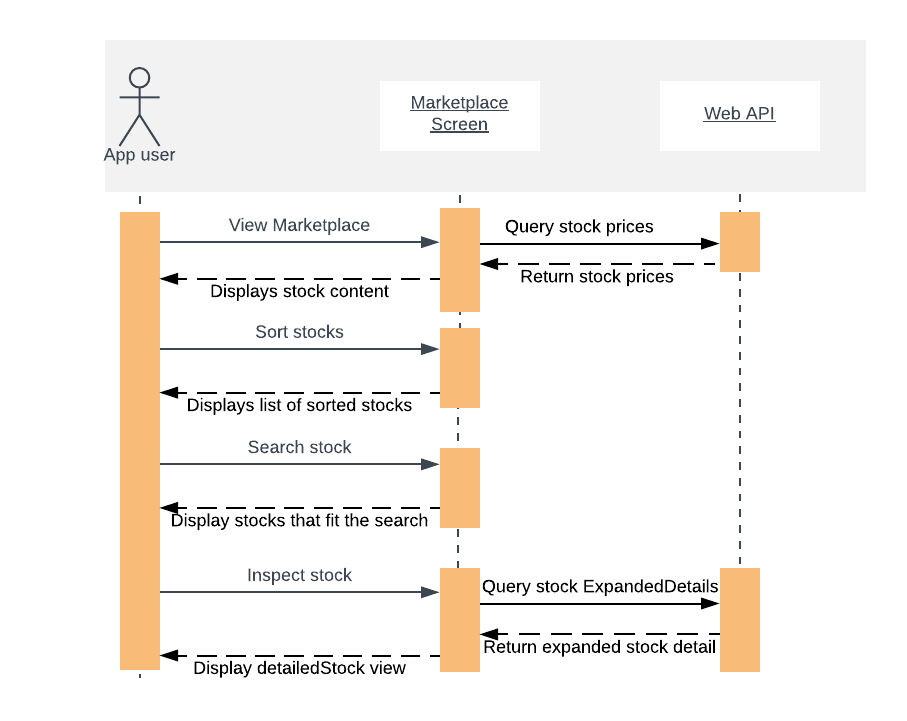
Portfolio Screen: This screen shows the user’s portfolio. It contains a list of stocks the user owns, along with their current prices and the entire portfolio worth. It gets the current details for the portfolio from the backend API. The user can click a specific stock to see its expanded details (price history, etc.).

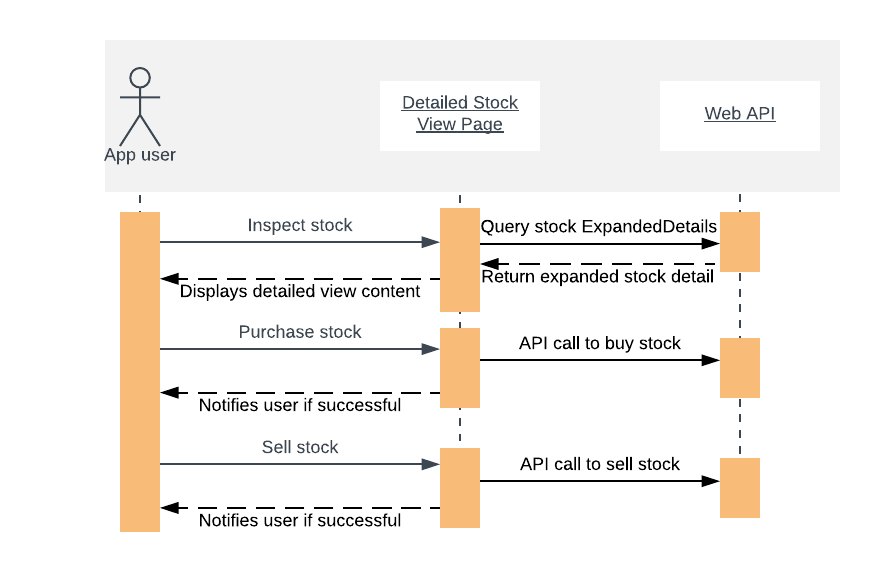
Leagues Screen: This screen shows all the leagues a user is a member of. A league is a competitive leaderboard with one or more users. Users can click on a league to see the current league details (a list of league members and their portfolio values.) Users can create, join, or leave leagues from this screen.

Marketplace Screen: This screen shows the stock marketplace. It contains a list of sorted stocks based on the users sort selection. The user will also be able to search stocks based off company name or ticker symbol. Current stock information will be obtained via the IEX api. The user can click a specific stock to see its expanded details (price history, etc.).

Detailed Stock View Screen: This screen shows the detailed view of a specific stock. The screen will display stock details such as price, today's change, change in the past year, as well as a graphical representation of the stocks price fluctuation. From this screen users will also be able to buy and sell stocks.





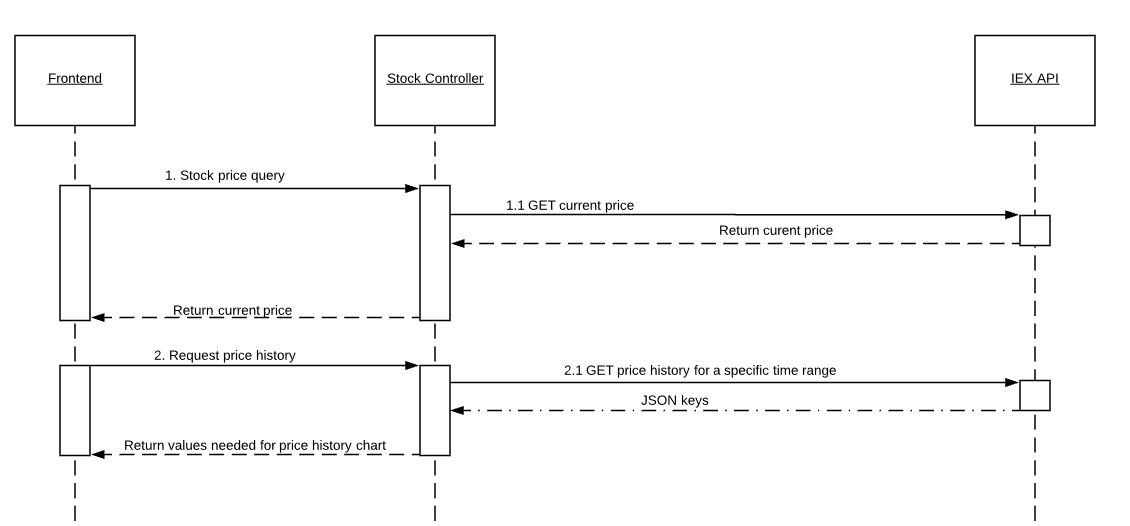


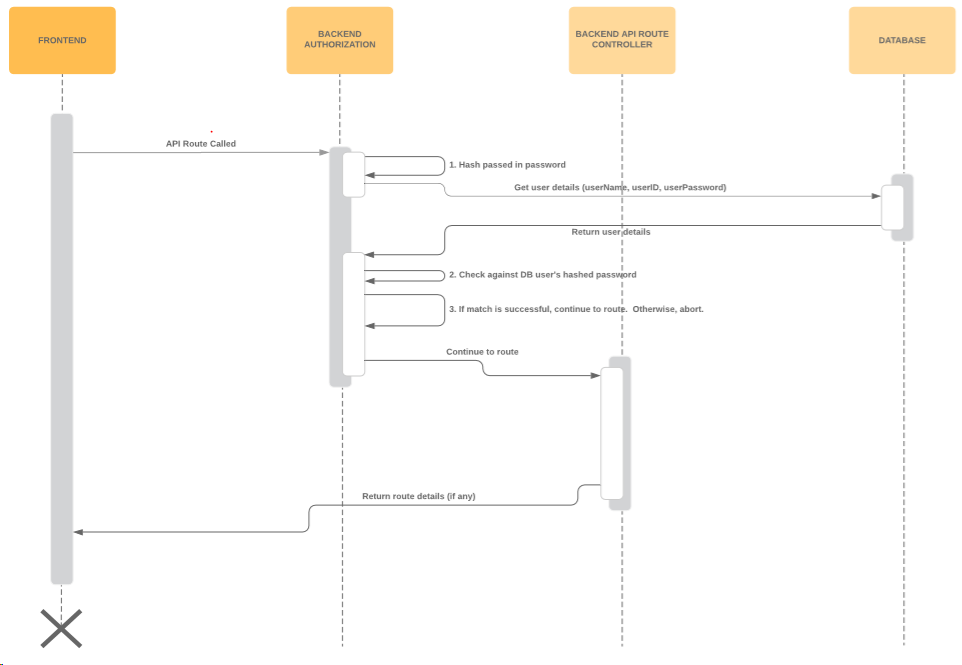
Backend Subsystems: (Authentication Middleware, Database Layer, 3rd Party API)

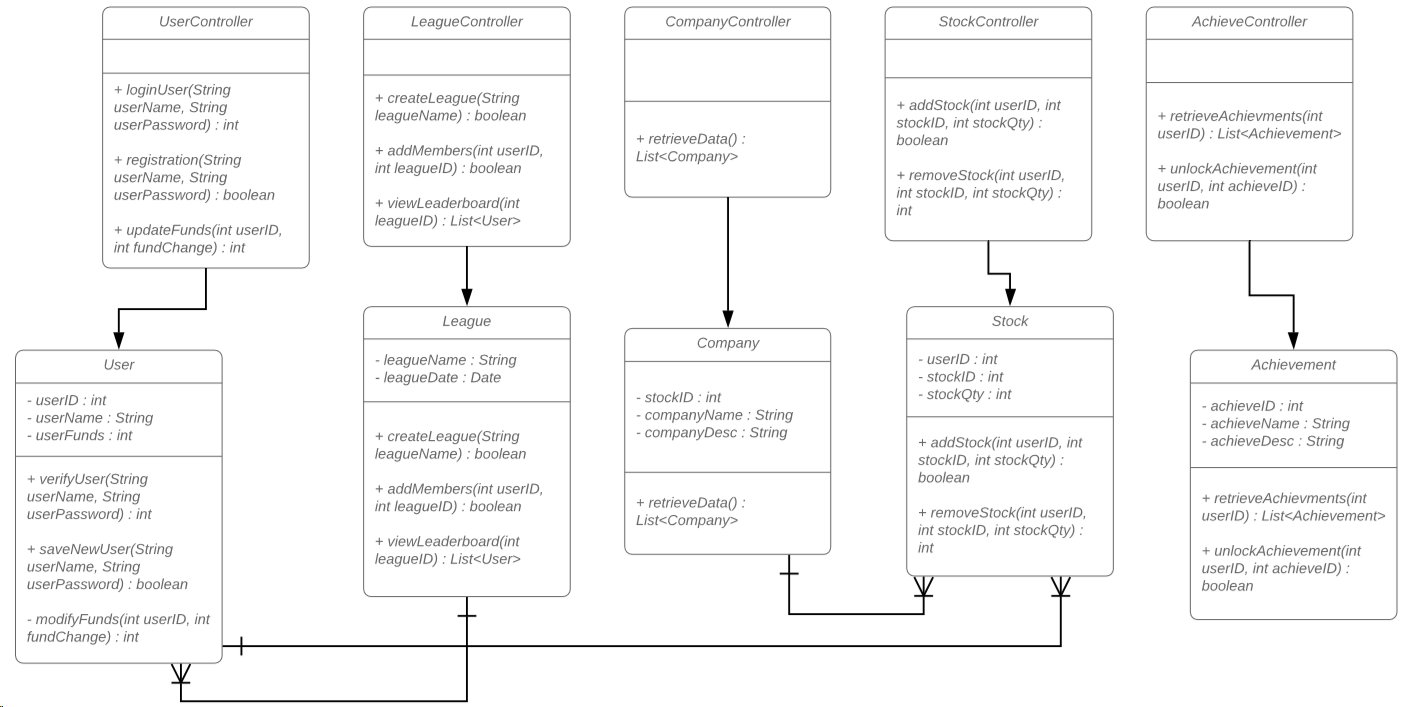
3rd Party API: A 3rd party API is needed in order to gather the information needed on stock prices and their details. The IEX API is used for this purpose since it can provide real-time data for free. This will be called from the backend API when the frontend needs the stock information.

Authentication Middleware: Authentication is performed before any call to the backend API route controllers, to verify that the user is who it is. It will hash the password provided, and compare it to the one that is stored on the database. If they match, it will continue to its intended destination on the backend, though if it fails, it will abort and throw an error.

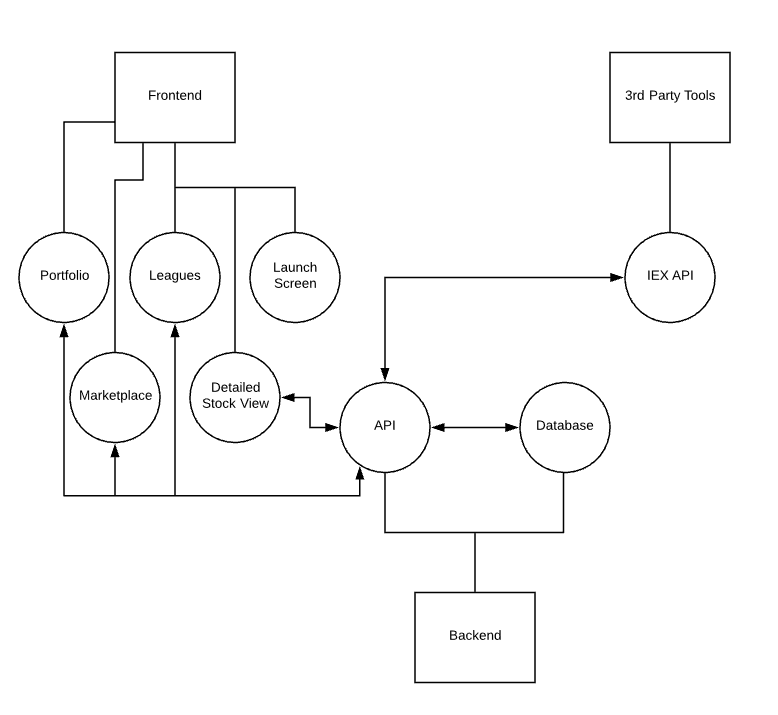
Database Layer: The database classes are split into 5 categories: User, League, Company, Stock, and Achievements. They will be reachable through their respective controllers, depending on the actions on the view. There may likely be a model that handles SQL connections that all 5 other models will connect to, though it will depend on the complexity of the SQL formulas.







* Sub-System Communication

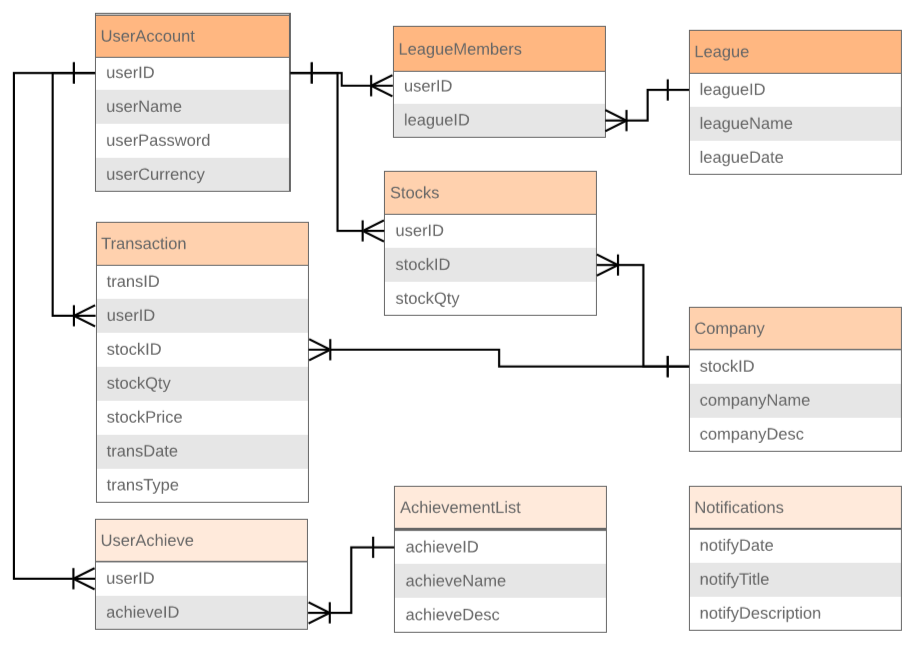


* The frontend is made up of multiple screens that interact with the backend API after being verified through the middleware authentication. They call the API when some kind of information is needed, which it will then communicate with the database or the 3rd party stock API (IEX) to get that information and supply it to the frontend. The Portfolio and League screens will be sent information on the user from the database. The Detailed Stock View and Marketplace screens will be sent information on current stock prices from the IEX API. When stocks are bought or sold, the API will need to interact with the database to update information on the user’s ownership of the stocks and IEX to get the current price to update the user’s currency.

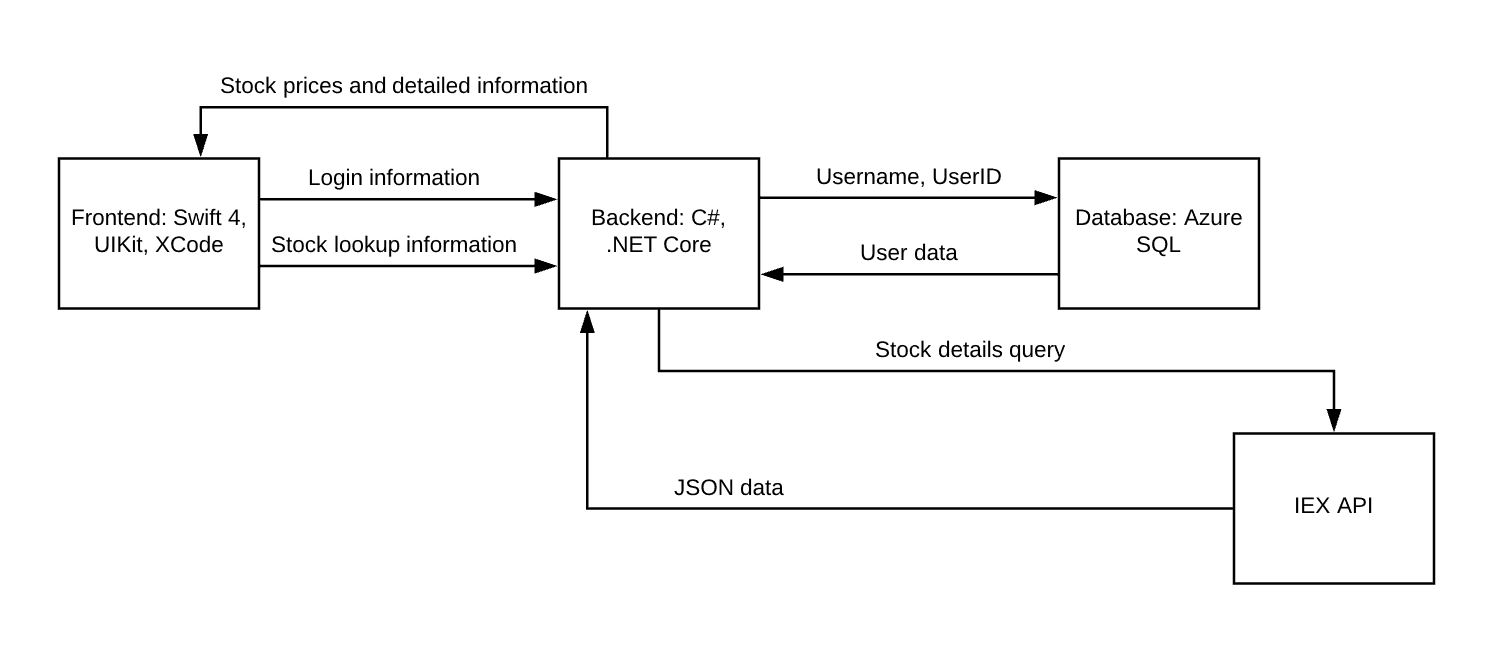
API Routes (Front to Back end communication. All data passed is in JSON.)

|  |  |
| --- | --- |
| Route | Description |
| GET /api/portfolio | Returns the authenticated user’s portfolio details. (User currency, list of current stocks owned + amount of each owned, prices of each stock at the moment.) |
| GET /api/stock/symbols | Returns a list of stock details (prices, etc.) for the given symbols. |
| POST /api/stock/buy | Buys a stock. Expects the ticker symbol of the stock to buy and the amount of stock to buy. |
| POST /api/stock/sell | Sells a stock. Expects the ticker symbol of the stock to sell and the amount of stock to sell. |
| GET /api/leagues/id | Returns league details. (Members and portfolio values.) |
| POST /api/leagues/create | Creates a new league. Returns the uniquely created league id, and registers the authenticated user as the league owner. |
| DELETE /api/leagues/ | Deletes a league, if the authenticated user is the owner. Expects the league id. |
| POST /api/leagues/join/{leagueid} | Joins a league with the authenticated user. Expects the league id. |
| DELETE /api/leagues/leave/{leagueid} | Leave a league |
| DELETE /api/leagues/kick/{leagueid}/{userid} | Kicks a user from a league |
| GET /api/transactions | Get a list of previous transactions for the user |

* Entity Relationship Model (E-R Model)



* Overall operation - System Model



**5. System – Analysis Perspective** – *Group responsibility*

* Identify subsystems – analysis point of view
* Backend: Web API + Database
  + C# and .NET Core were chosen because of a combination of developer familiarity (with C#) and cross platform ability. We can run a .NET Core backend on Windows and Mac, meaning our front end iOS team can develop with the backend code running for development/testing purposes. (This wouldn’t be possible with older versions of .NET, which are Windows only.)
  + There are several different options for hosting our web application, the primary ones being Microsoft’s Azure and Amazon’s Web Services. We ended up choosing Azure, due to strong integration with Microsoft’s .NET ecosystem and its pricing. Azure App Services allows 10 free applications, with no need to micromanage virtual machines, and allows us to scale those applications out horizontally with increased instances if we need them. Azure also provides a month of free credits to be used for cloud PostgreSQL database services, which we can activate near the delivery date of the project. However, beyond the free month, PostgreSQL would cost money to host on Azure, whereas Azure SQL could be hosted free for a year. But since our frontend development team will develop with MacOS for iOS development, it would be ideal to use a database capable of running locally on those computers, which Azure SQL was not made to do.
* Frontend:
  + Swift4 and UIkit were chosen for the front end for a couple of reasons. One is that overall the development team is more familiar with IOS development compared to Android. Another reason is because IOS beats out android when it comes to development complexity. Android has more devices and OS versions to worry about compared to IOS. In 2018 over 50% of IOS devices were running on IOS 12 (the most current version) compared to android where only .1% of users were running on Android 9 Pie and 14.6% on Android 8 and 8.1 Oreo. Because of those reasons, developing for IOS would result in a less complex app building process.
* System (Tables and Description)
  + Data analysis
    - Data dictionary (Table - Name, Data Type, Description)

|  |  |  |
| --- | --- | --- |
| Name | Data Type | Description |
| userID | integer | Used for tracking the user with one value. |
| userName | string | The entered username. Paired with password to access. |
| userPassword | string | Will be hashed before being stored. Entered password is compared to stored hashed password. |
| userCurrency | double | The current unused funds on a user’s account. |
| leagueID | integer | Used for tracking league. |
| leagueName | string | Name assigned to league upon creation. |
| leagueDate | DateTime | Date and time league was created. For use in record keeping, or potential deadlines. |
| stockID | integer | Used for tracking companies and stocks. |
| stockQty | integer | The number of stocks owned by a user for a specific company. |
| companyName | string | Name of Company |
| companyDesc | string | Short description of company, like what they sell, or what services they provide. |
| transID | integer | Used for tracking a transaction; for when a user buys or sells a stock. Transaction History. |
| stockPrice | double | Price paid or earned from buying/selling a stock. |
| transDate | DateTime | Time at which the exchange occurred. |
| transType | string | Determine if the stocks were bought, sold, or any other options. |
| achieveID | int | Used for tracking various achievements. |
| achieveName | string | Title of Achievement. |
| achieveDesc | string | Description of Achievement Requirements |
| notifyDate | DateTime | Date at which notification was created. May last a certain amount of time beyond creation. |
| notifyTitle | string | Header of notification. |
| notifyDesc | string | Body of notification. For updates to the product, or for ongoing events. |

* + Process models

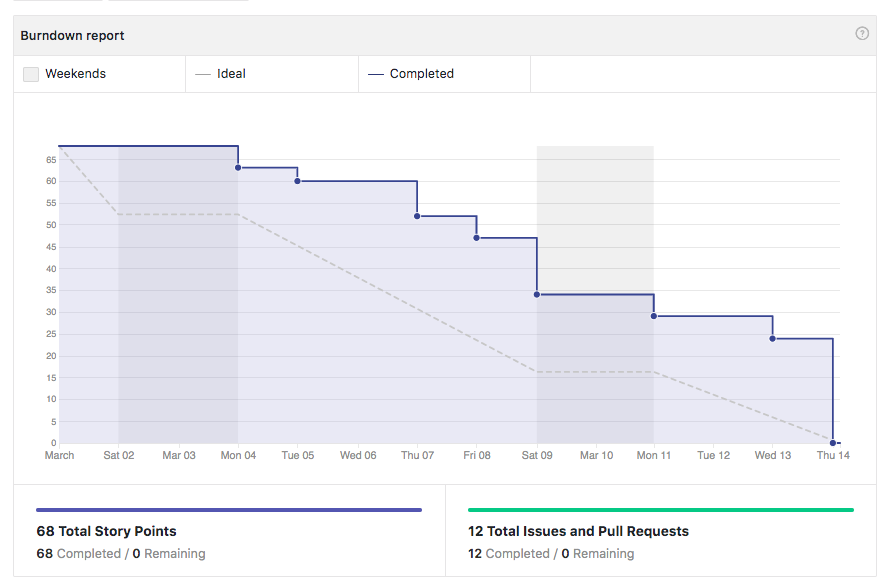
The process model we will be using is Agile and Scrum. Agile was chosen because of its emphasis on less documentation and more developing. This process model will us to focus more on productive work compared to other more plan heavy models such as waterfall. We will also be using the Scrum process framework which is a subset of Agile. Scrum was chosen because of the benefits it provides, such as providing estimates on how long certain features will take to implement with the product backlog. It will also allow us to be in control of the project schedule with the use of a burn-down chart and product backlog.

* Algorithm Analysis
  + Search: When searching, we will have a list of stocks in a users portfolio or a list of stocks in the market stored on the device. Searching through this list would take at most the entire length of the list, meaning the time complexity of searching will be O(N).
  + Sort: Sometimes users will want to sort stock entries (for example, by current value, or by their growth over a certain period of time.) Merge Sort would give us a worst case of O(n\*log(n)) for sorting on current price. Sorting on the growth over a given period of time would be the same complexity, since computing growth-over-time for each sorted stock could be done within the sorting algorithm and would just add extra constant-time computations, which we could ignore. But if we computed the growth-over-time outside of the sort, it would be O(n + (n \* log(n))), since we would need to iterate over the list first before sorting.

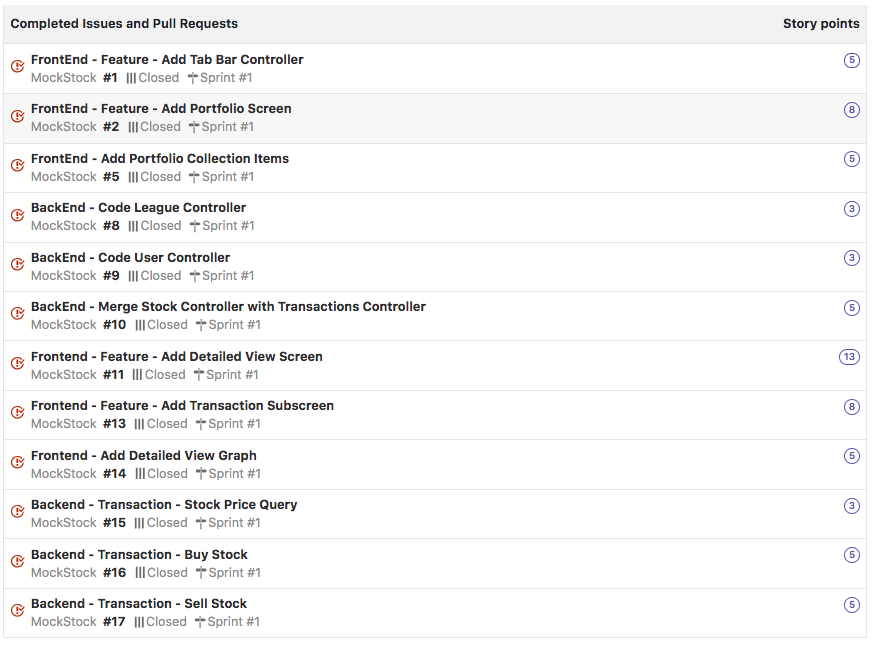
**6. Project Scrum Report -** *Group Responsibility*

Below is the backlog of tasks and burndown chart for each sprint. Our overall product backlog is simply the sum of these sprint backlogs.

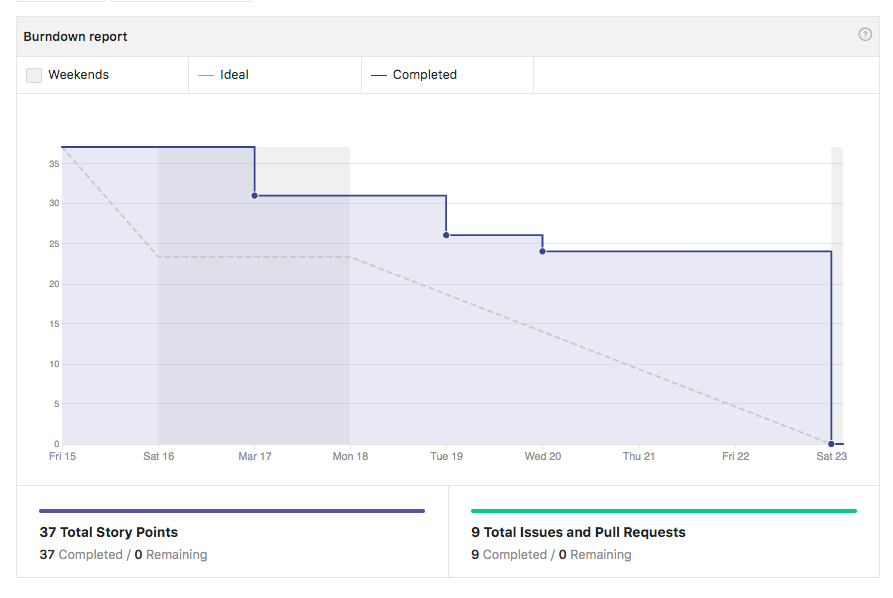
Sprint #1 Burndown Chart:



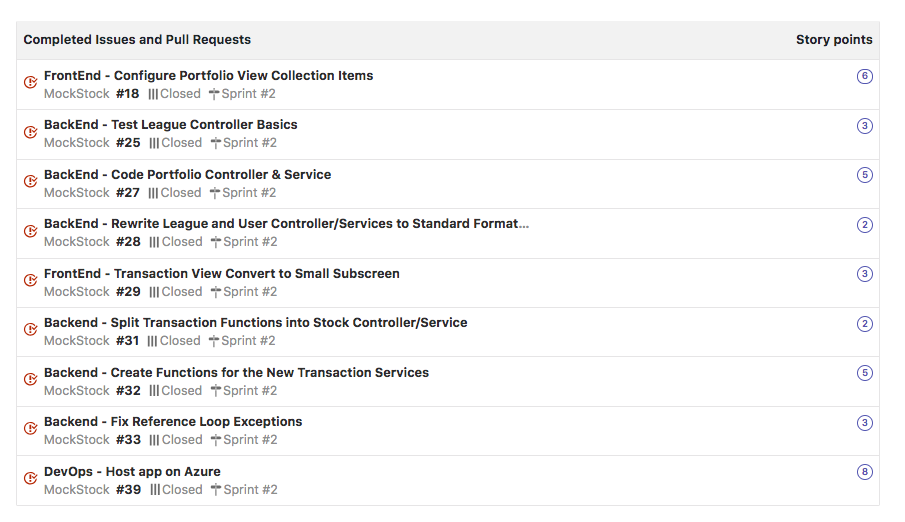
Sprint #1 Product Backlog



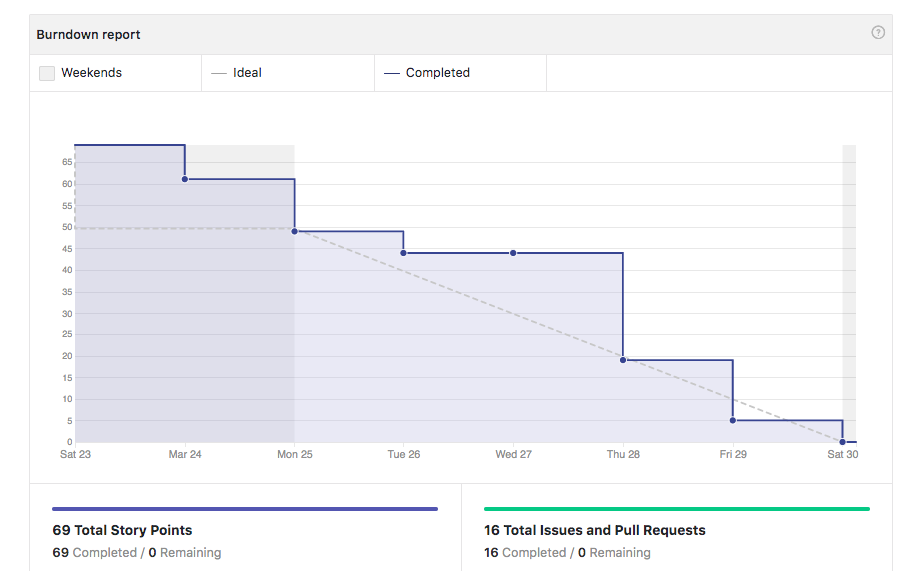
Sprint #2 Burndown Chart:



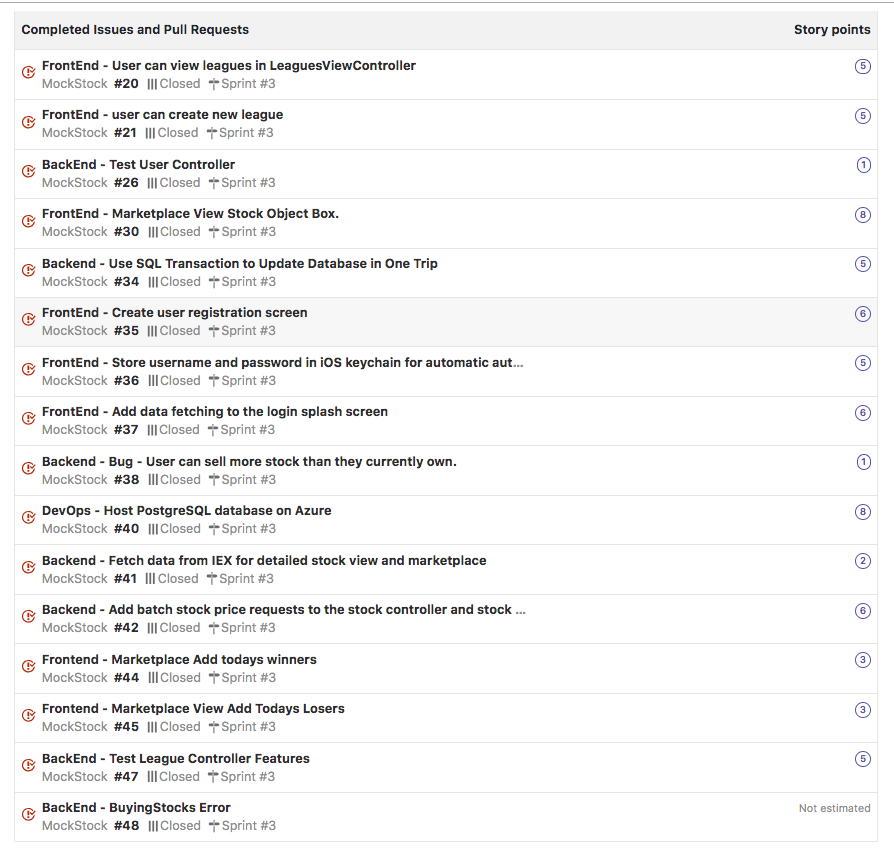
Sprint #2 Product Backlog



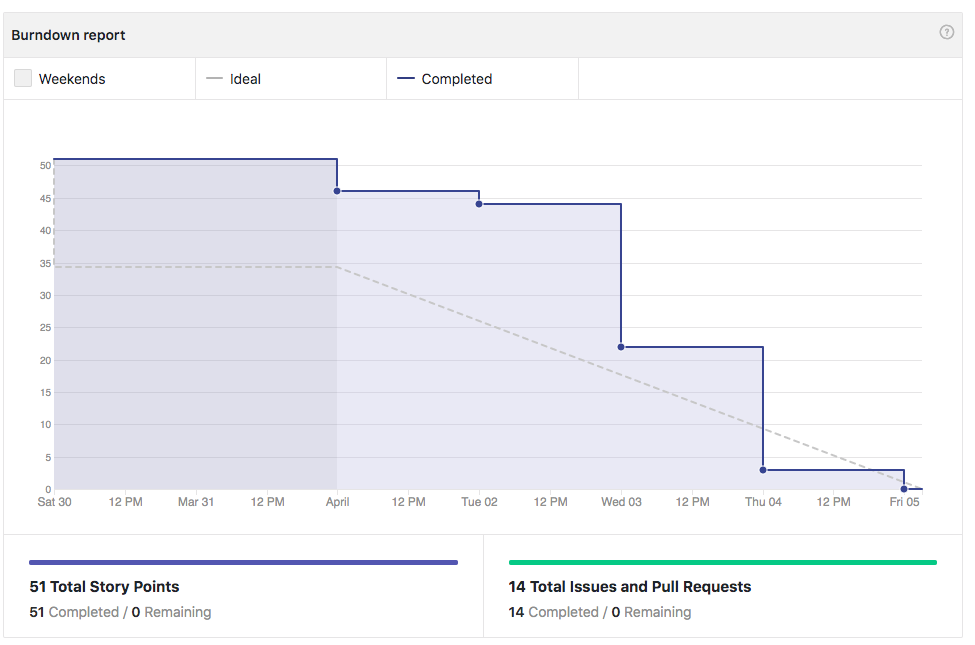
Sprint #3 Burndown Chart:



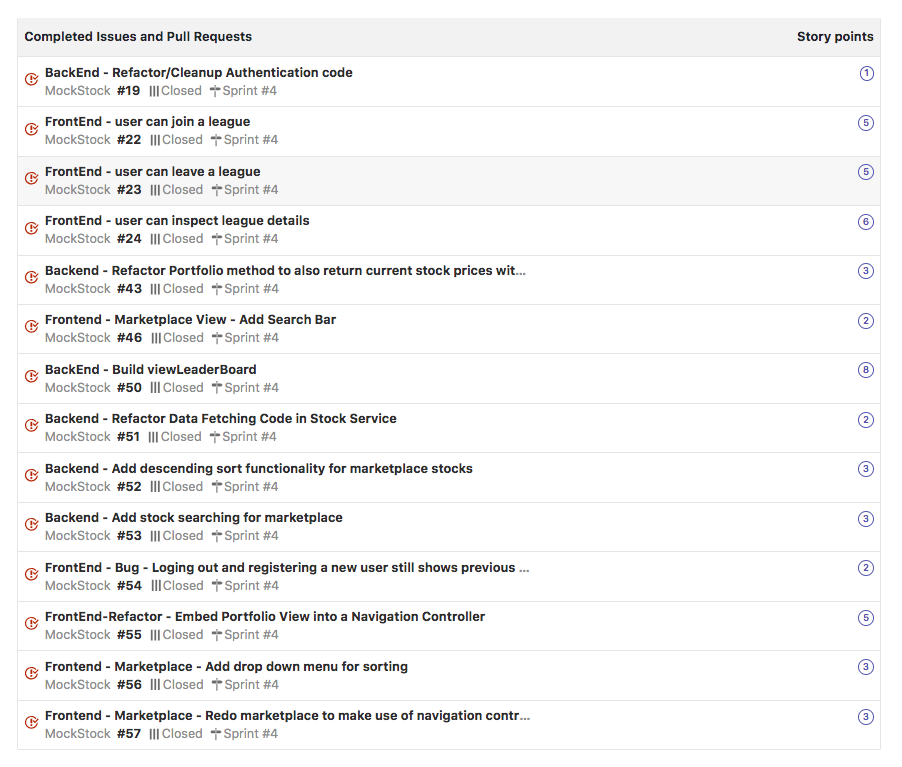
Sprint #3 Product Backlog:



Sprint #4 Burndown Chart:



Sprint #4 Product Backlog:



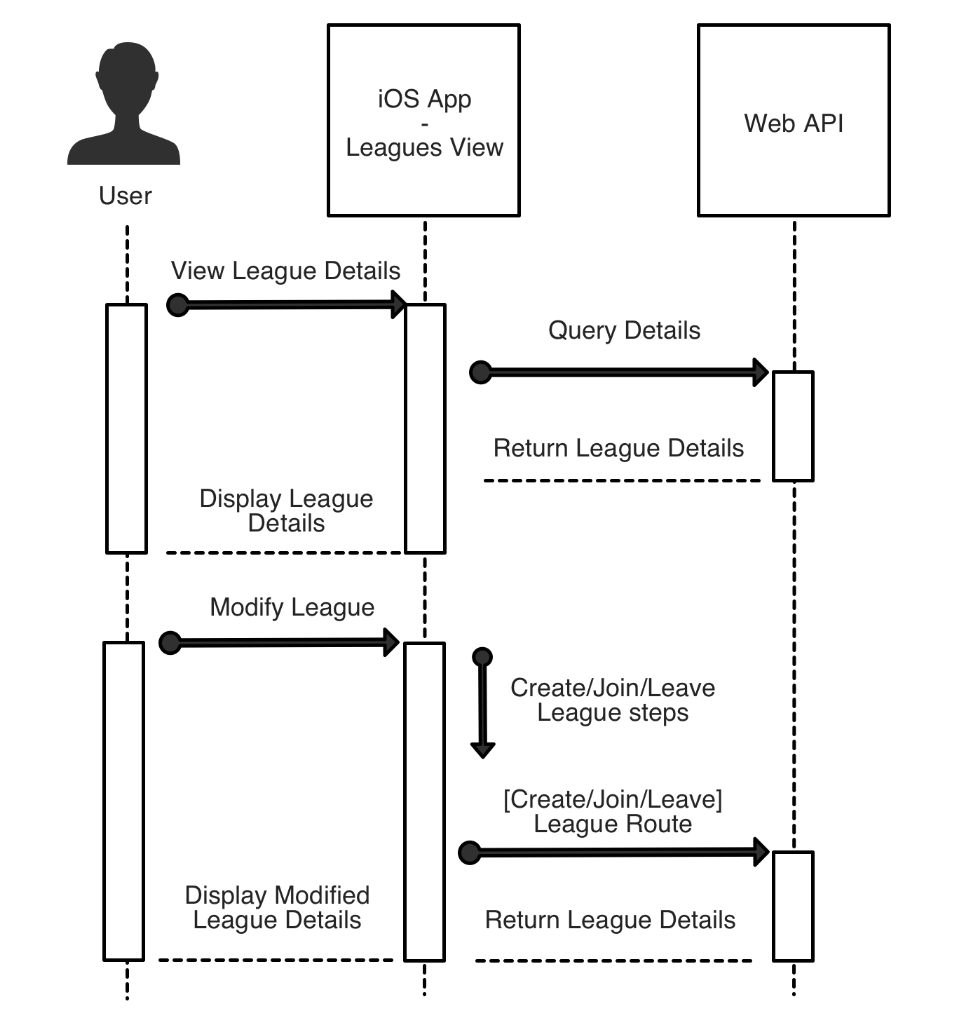
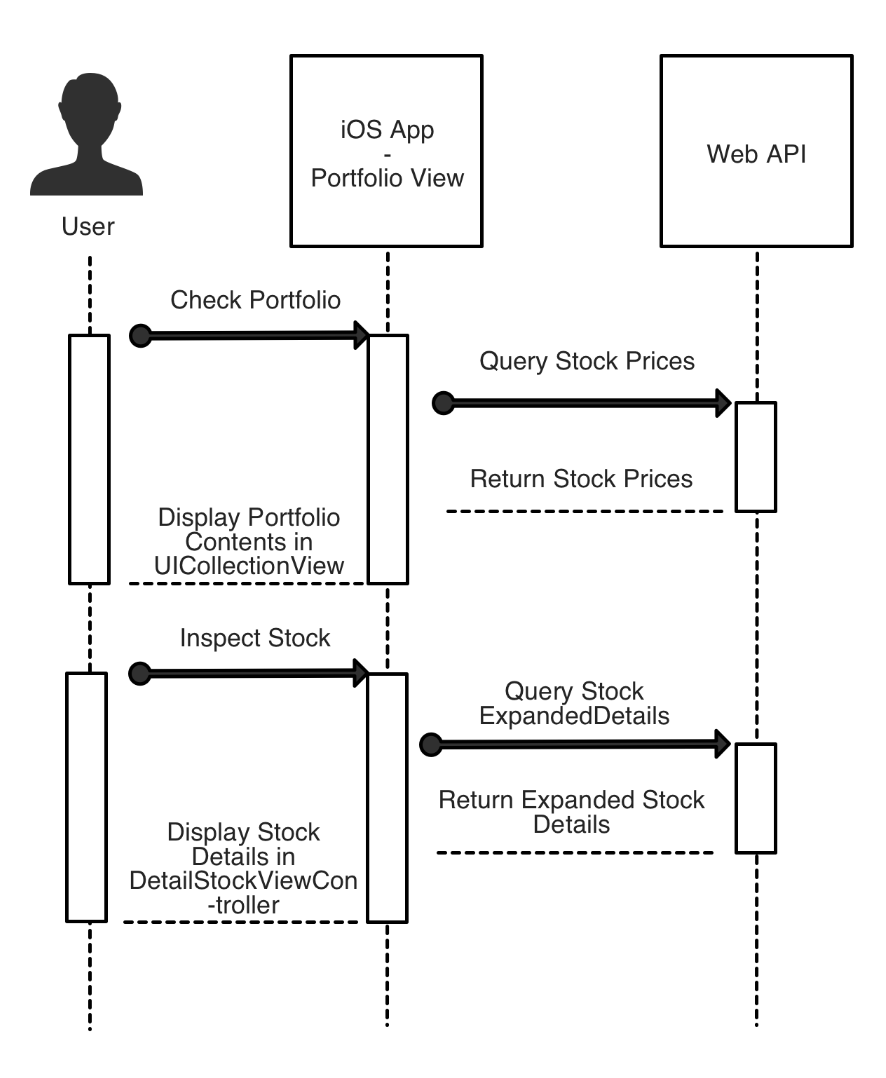
**7. Subsystems**

**7.1 Subsystem 1** – Theodore Hecht III

Responsibilities: *Portfolio & Leagues frontend views + backend authentication*

* Initial design and model
  + The Portfolio Screen displays the user’s portfolio. It’s main content is a UICollectionView with custom UICollectionView Cells to display portfolio data. The ‘x’ button lets you log out of the application.
  + The Leagues Screen displays all leagues in which the user has membership. This is also implemented with a UICollectionView. There is a detailed league screen that displays each member of the league sorted by income level. The + button lets you create or join a league.
  + The Registration Screen displays a simple login/register form, where the user can enter their username and password, and register or login to the service, when the appropriate button is pressed.
  + Each of these screens fetches data from the backend API when the screen appears. It decodes the response into one of the model objects, which implements the Decodable protocol, allowing JSON properties to be decoded into the object. This data is then used to populate the UIView widgets each view controller displays.

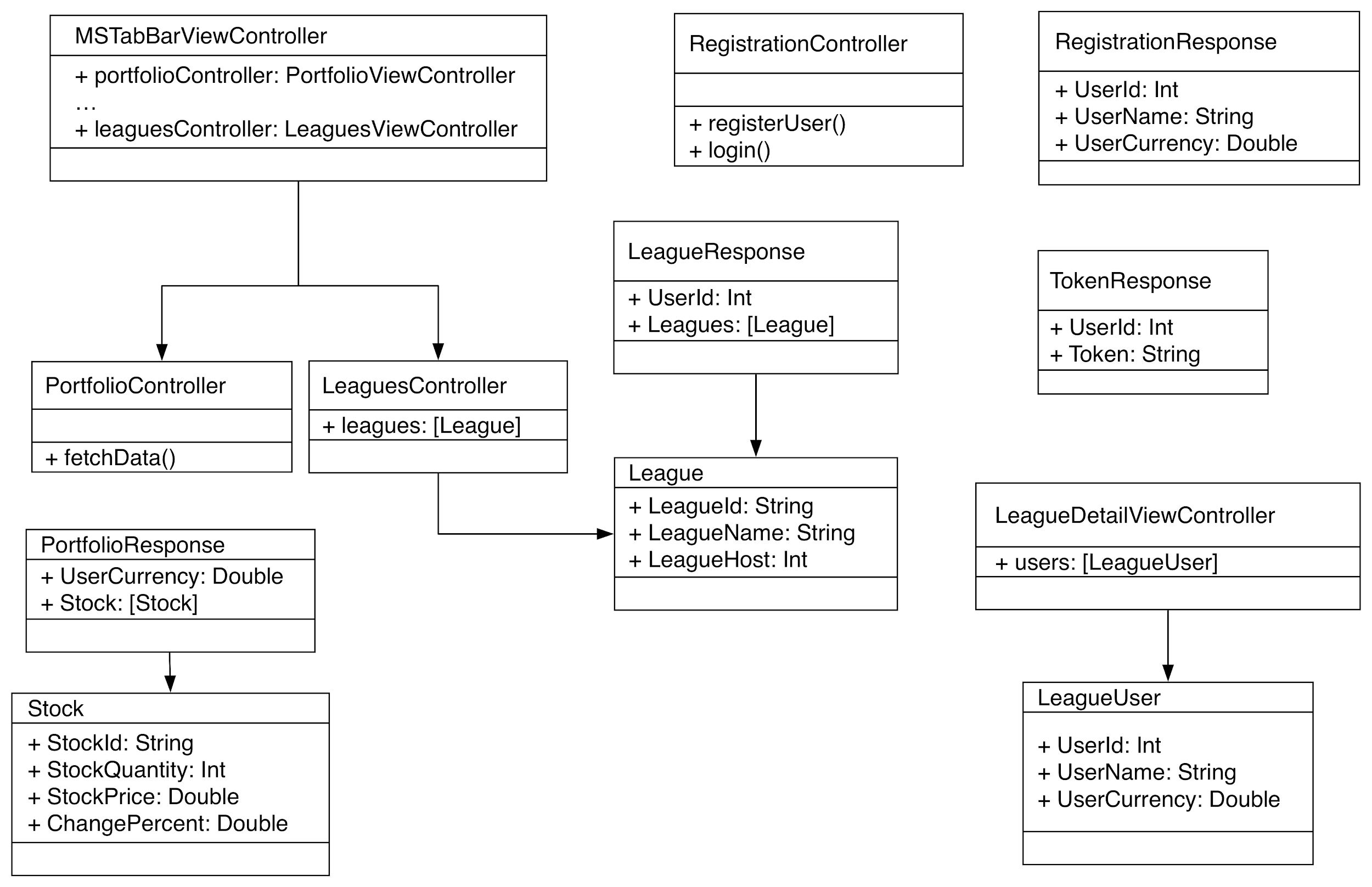
Sequence Diagrams:



* + Initial UI Mockup:



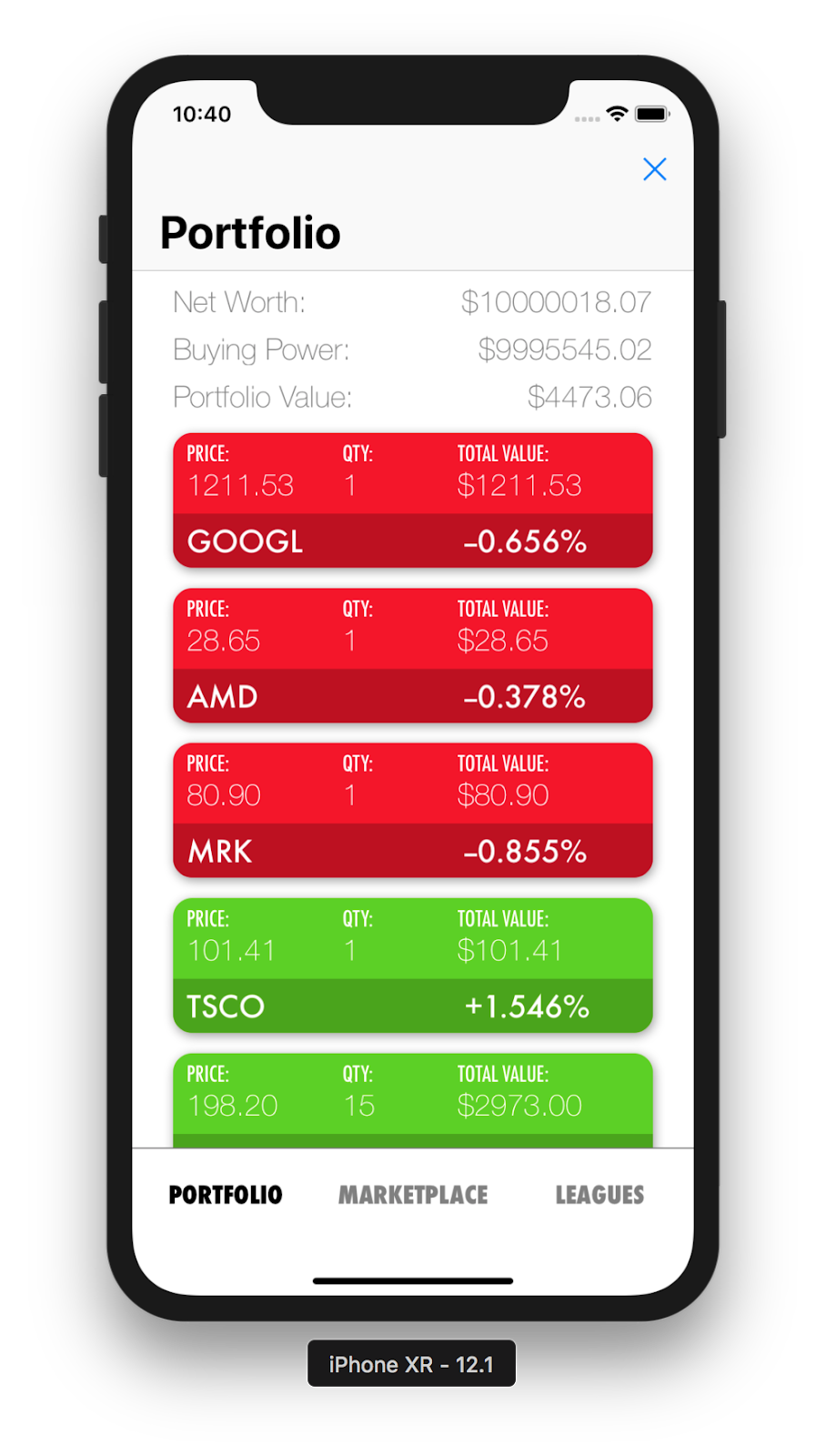
Class/UML diagrams for code-behind.

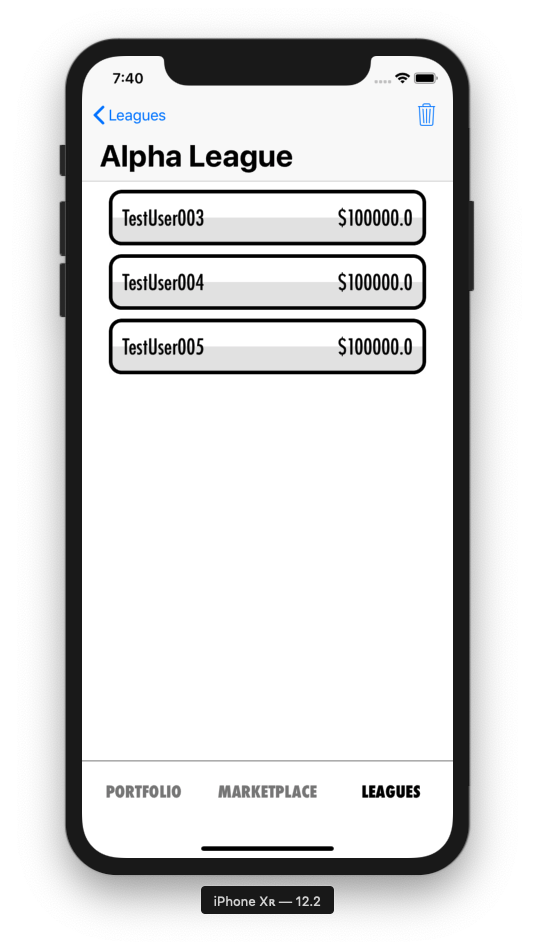
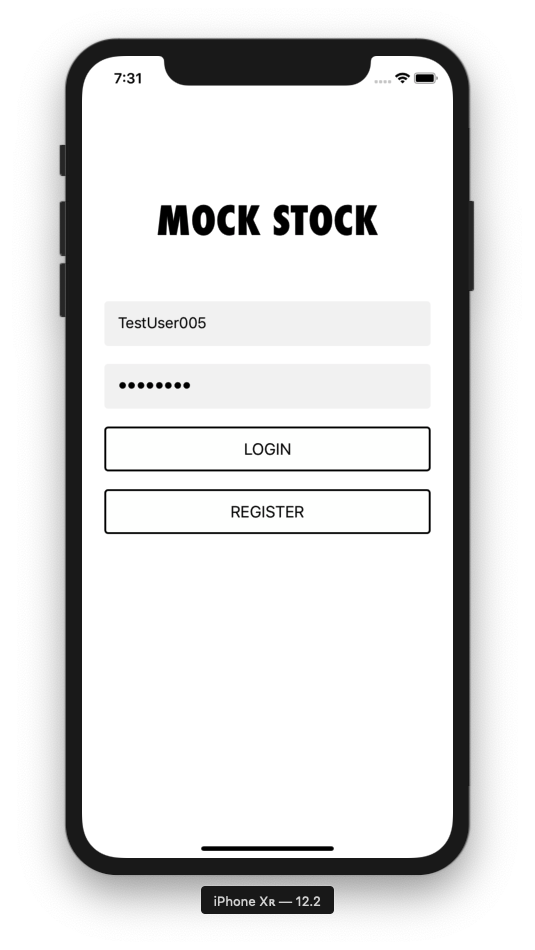


The app as a whole uses the MVC pattern, where the (not pictured) UIKit widgets are the views, which display information to the user, the ViewControllers are the controllers, which handle business logic and routing model data to the views, and the various json-decodable classes are the models.

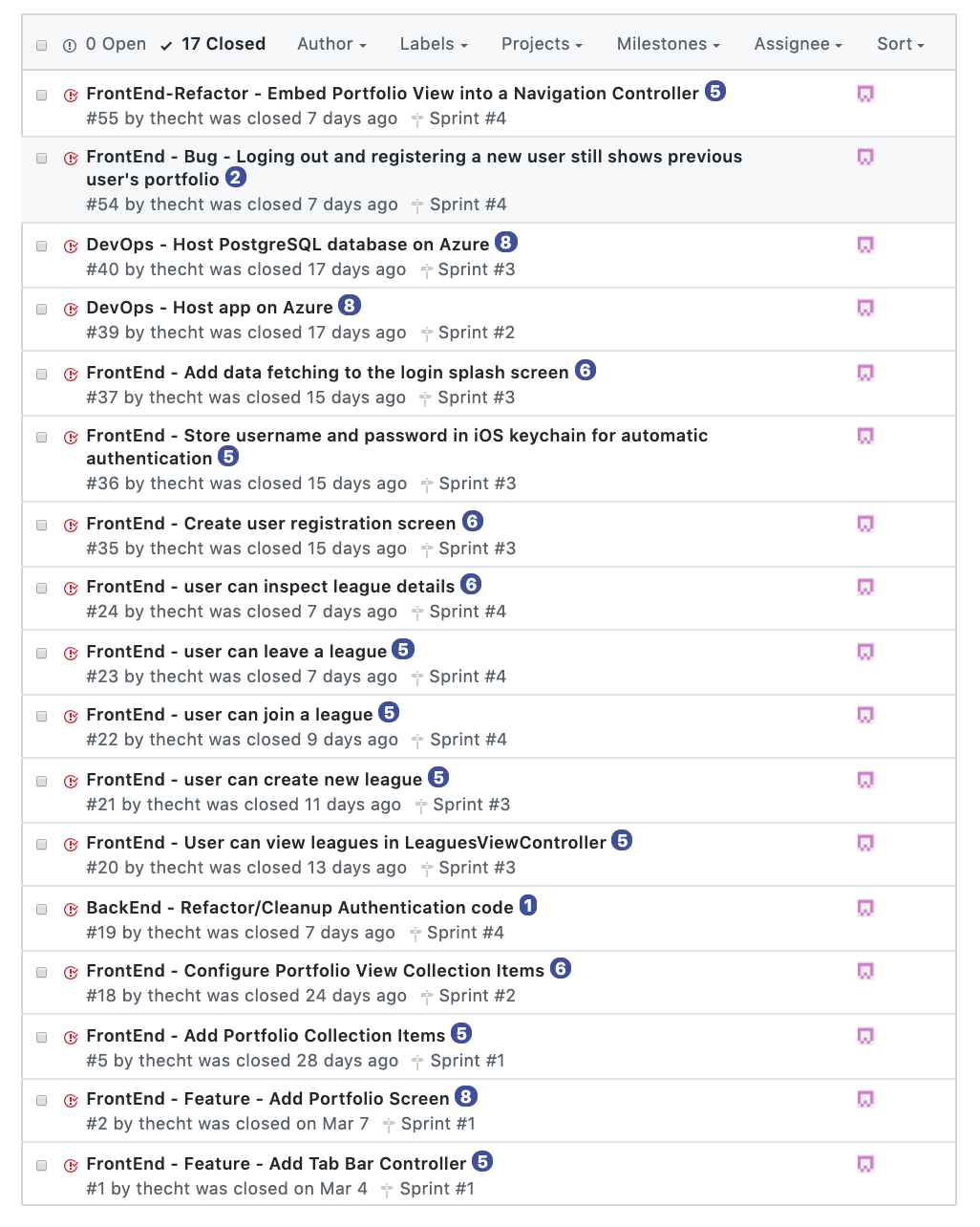
The MSTabBarViewController handles displaying each major view subsystem (Portfolio, Marketplace, or Leagues controller.) Each ViewController has various UIKit widgets (not pictured), including labels, collection views, and collection view cells. It also has various model objects (displayed above) which implement the Decodable interface, which allows them to be easily deserialized from JSON, which is returned by our RESTful API.

* If refined (changed over the course of project)
  + The primary change from the initial frontend model was the inclusion of navigation controllers, which the primary view controllers are now embedded inside. (You can see these headers in the screenshots on the next page.) The reason for this refinement was to more easily transition from one view controller to an expanded details view controller, with smooth animations already built in. It also allows for a more consistent look and feel across the application. Unfortunately, this comes at the cost of customizability, since the navigation bar isn’t as customizable as an entirely custom built solution. Fortunately, this design change didn’t significantly alter the user interaction flow of the application.
  + Another change was the backend authentication mechanism. The new one, which I wrote, uses JWT token authentication, instead of “Basic authentication.” The reason for this refinement was to prevent the need to check the database on every request for the user’s password. With JWT tokens, this is avoidable. Instead, the username/password is checked once when a token is requested, and then the token is sent with future requests to the backend. This is advantageous for two main reasons: one, it increases performance by reducing the need for an extra call to the database on each request; two, it reduces the attack surface for compromising someone’s account, because tokens expire, so a sniffed token has only a small window of time where it can be used (though we are using HTTPS, so this shouldn’t be a common occurrence.) The disadvantage is if we were to revoke permissions for a user, they would still have those permissions until their token expires, since the permission claims are contained within the token itself.
  + New Design:





* Scrum Backlog (Product and Sprint - Link to Section 6)

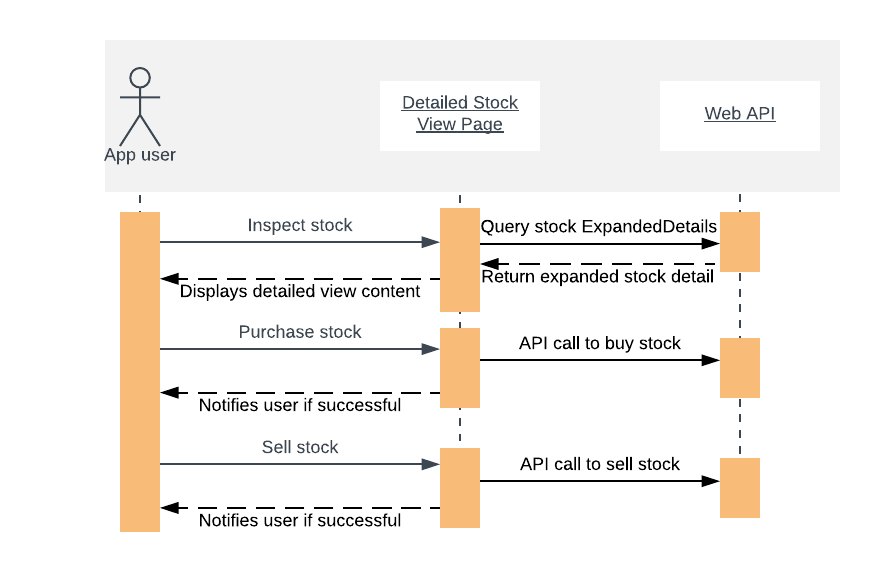
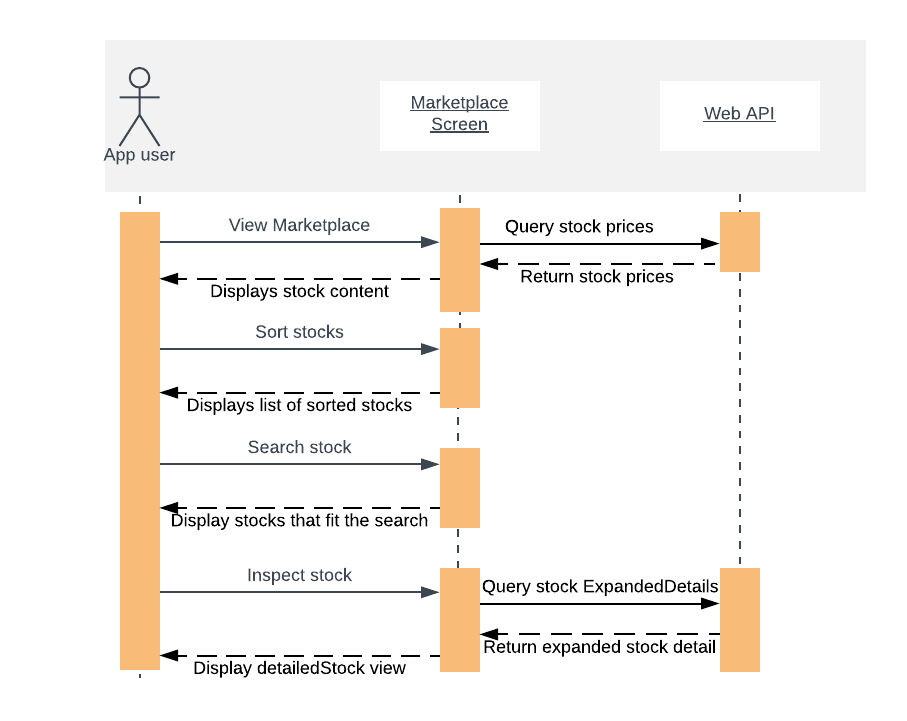


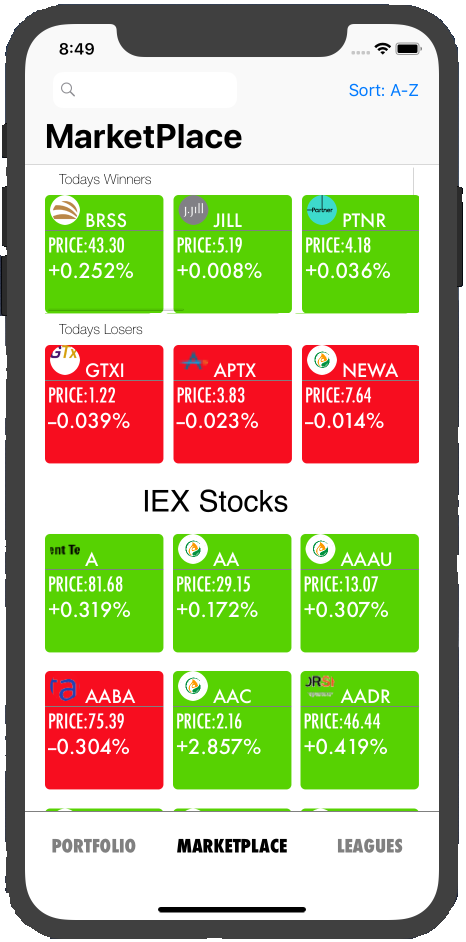
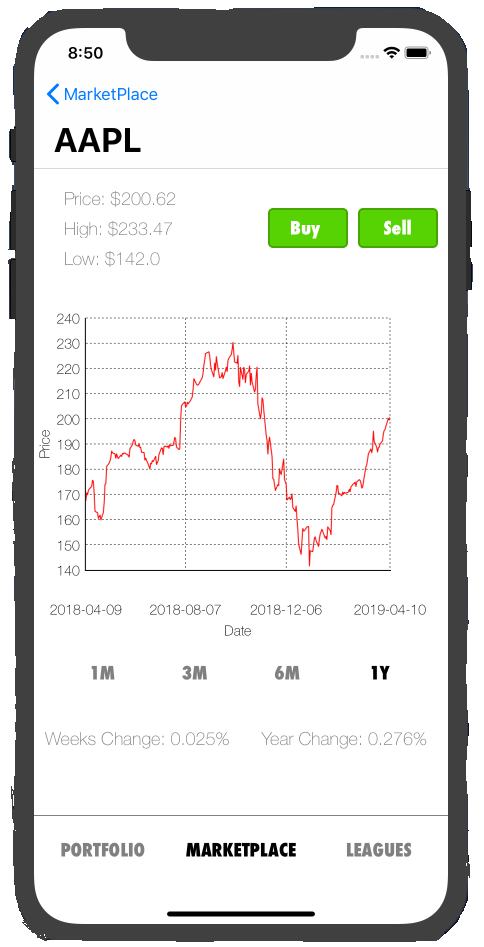
* Coding
  + The app was developed with the Swift 4 programming language, an object oriented programming language developed by Apple. This language was chosen because it is Apple’s preferred way of making native iOS apps.
  + We use CocoaPods as a dependency manager so that we could install third party libraries with ease.
  + We made extensive use of the MVC design pattern, as well as the delegation pattern, both of which are heavily used in iOS development.
* User training
  + How to Login or Register: At the Login/Registration screen, enter your username and password details into the boxes, and press either Login or Register.
  + How to View Portfolio: Click the Portfolio button on the bottom tab bar to access the portfolio view. Scroll up or down to see the contents. (A spinning activity indicator in the navigation bar signifies network activity.)
  + How to Inspect Owned Stock Details: In the portfolio view, click on a stock to bring up a detailed stock view to inspect stock details.
  + How to Logout: In the portfolio view, click the ‘x’ button in the top right corner in the navigation bar.
  + How to View Leagues: Press the Leagues button in the bottom tab bar to access the leagues view. Scroll up or down to see the leagues the logged in user belongs to.
  + How to View League Leaderboards: In the leagues view, click on a league to open the league details, which displays a list of all members of a league along with their total net worth (sorted in descending order).
  + How to Create League: In the leagues view, click on the ‘+’ button in the top right of the navigation bar. In the popup, press “Create League”, and then enter the name of the new league.
  + How to Join League: In the leagues view, click on the ‘+’ button in the top right of the navigation bar. In the popup, press “Join League”, and then enter the league code of the league you wish to join. (Note: codes are case sensitive.)
  + How to Leave League: In the league details view (which displays the league leaderboards), press the trashcan icon in the top right corner of the navigation bar. When prompted, press the option to leave the league. (Note: if you are the host/owner of the league, leaving the league deletes the league.)
* Testing
  + The front end application was tested using XCode and various iOS simulators. We used .NET Core’s open source kestrel web server for local testing with the backend, and then moved to a development staging environment in Azure to test the app as we moved to integrate the various pieces.
  + At each major integration, each of the actions in the above section (User training) were tested to see if we got the expected results. If we didn’t, finding the bug became a top priority.

**7.2 Subsystem 2** – Luke Orr

Responsibilities: Marketplace, Stock Detail, Graph, and Transaction frontend views

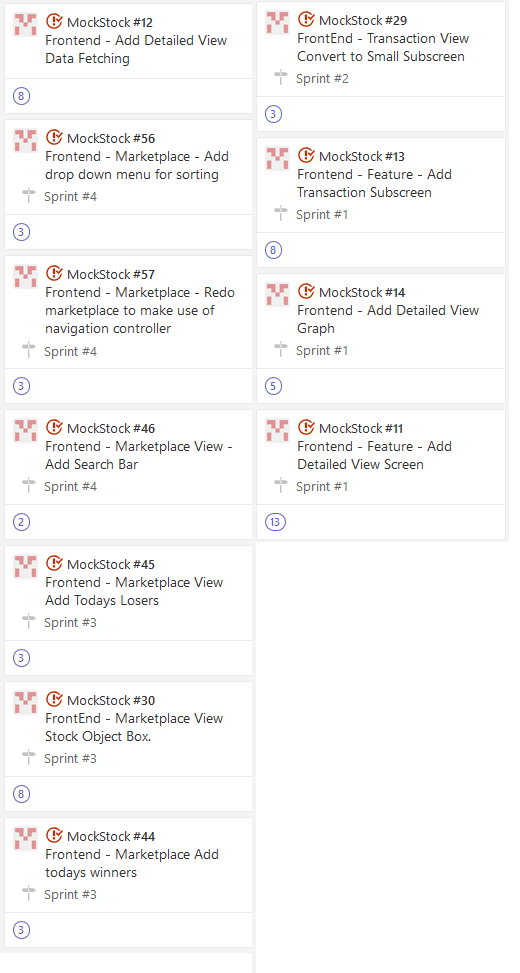
Initial design and model

* + The Marketplace screen displays three separate lists of stocks each its own uicollectionview. One list is the “Today’s Winners” list which is a collection of stocks pulled from the backend API. Another list is the “Today’s Losers list which is also a collection of stocks pulled from the backend API. The winners and losers sections are displayed in a horizontal uicollectionview that is inside of the main uicollectionview. The last list is the main marketplace list pulled from the backend API. The marketplace allows for users to search for either a specific stock or for stocks beginning with the letters that the user typed into the search field. This not only allows for users to find the stock that they want, but also allows users to refine the list of stocks displayed to them. The last feature of the marketplace screen is the sort feature. The sort button allows users to sort the third list of stocks by either a-z or z-a.
  + The Detailed View screen has three main features to it. One is that the screen displays more information about the stock that the user selected from either the portfolio or marketplace screens. The second feature is that users can buy or sell stocks from this screen by clicking either the buy or sell button. The last feature is that the detailed view screen also contains a graphical representation of the stocks price history. The graph can be changed by selecting one of the four buttons located below the graph view. This allows users to see the stocks price history in an easy to understand way. The four graph options are one month, three months, six months, and one year. Once a user selects either the buy or sell button, a subview will pop up which is the transaction subview. This subview is where the users will type in the amount of the stock they wish to either buy or sell. Upon success the system notifies the user with a popup.
  + Sequence Diagrams  
    
  + UI Designs



Refined Systems

* + Marketplace
    - The marketplace view went through a couple changes. Originally the marketplace view was going to be a single view of a list of stocks. The user would be able to sort the stocks based off symbol and price. However there was no way for us to be able to add a sort by price given what was available to us from the IEX API. After discussing this issue, we decided to replace the sort by price function with two separate lists of stocks called “Today’s Winners” and “Today's Losers”. The winners and losers stocks are displayed in a horizontal scroll view that is embedded in the main marketplace view. The pros of this change is that it makes the marketplace view have an app market look. The users will be able to quickly see “featured” stocks that they may not have seen with a simple sort by price feature. A con for this change is that there is no way for a user to be able to easily see the most and least expensive stocks.
  + Detailed View
    - Our initial design for the detailed view was that users would be able to purchase and sell stocks directly from that view. However after initially designing the detailed view, it looked very cluttered. We then decided to move the transaction functions over to a seperate view called the transaction view. This way once users click on the buy or sell button contained in the detailed view, a separate sub screen will pop up that allows the users to purchase/sell the stock. A pro for this change is that it made the detailed view screen less cluttered which allowed us to make the graph image larger and more legible. A con to this change is that now users have to go through an additional screen in order to buy or sell the stock.

Scrum Backlog (Product and Sprint - Link to Section 6)

Coding

* + The frontend was designed using the swift programming language which is an object oriented language.

User training

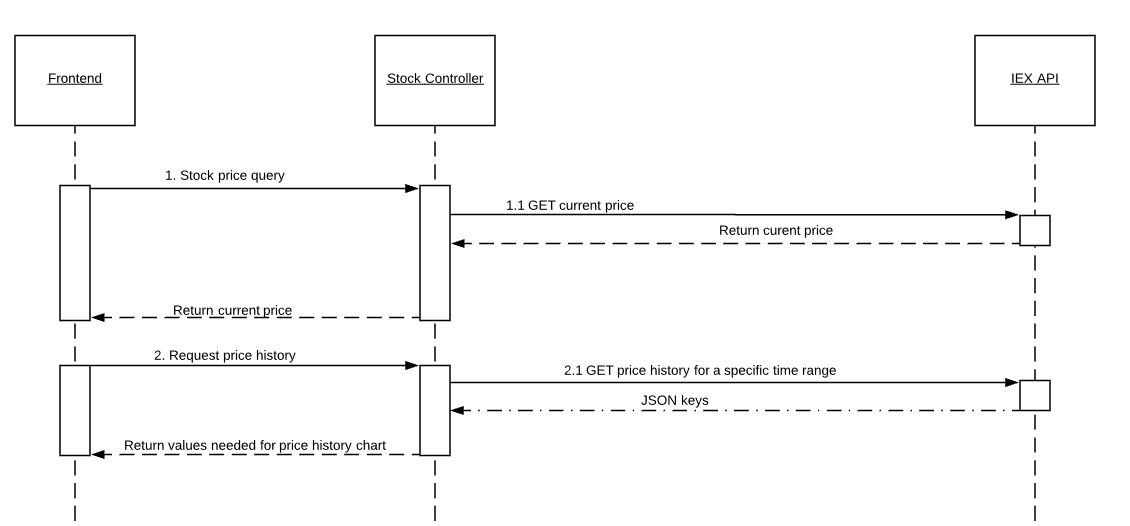
* + The use of the mock stock application would not require much training on the part of the user. We designed the front end UI to be simple and easy to understand. One of our goals was to make it so that users would be able to easily understand and use the mock stock application. The most complicated thing for the users involving my subsystems would be knowing that they can click on the top right button in the marketplace to change the sort.

Testing

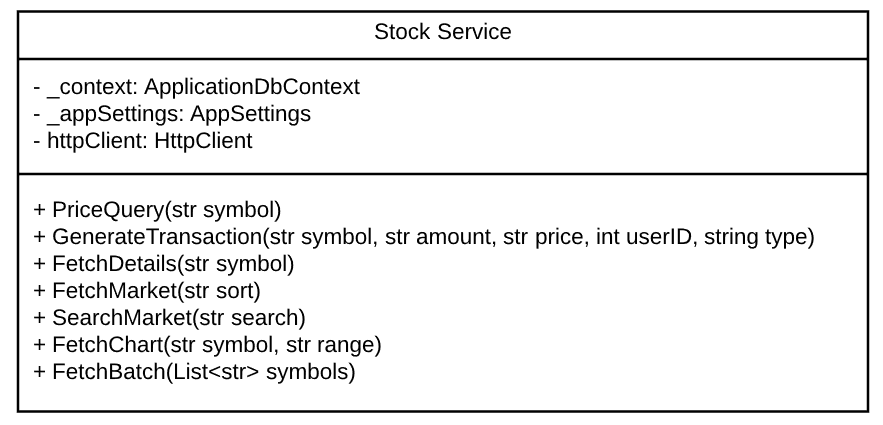
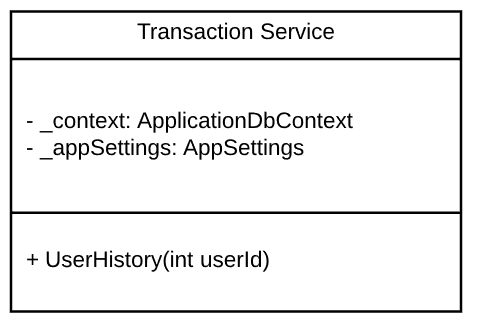
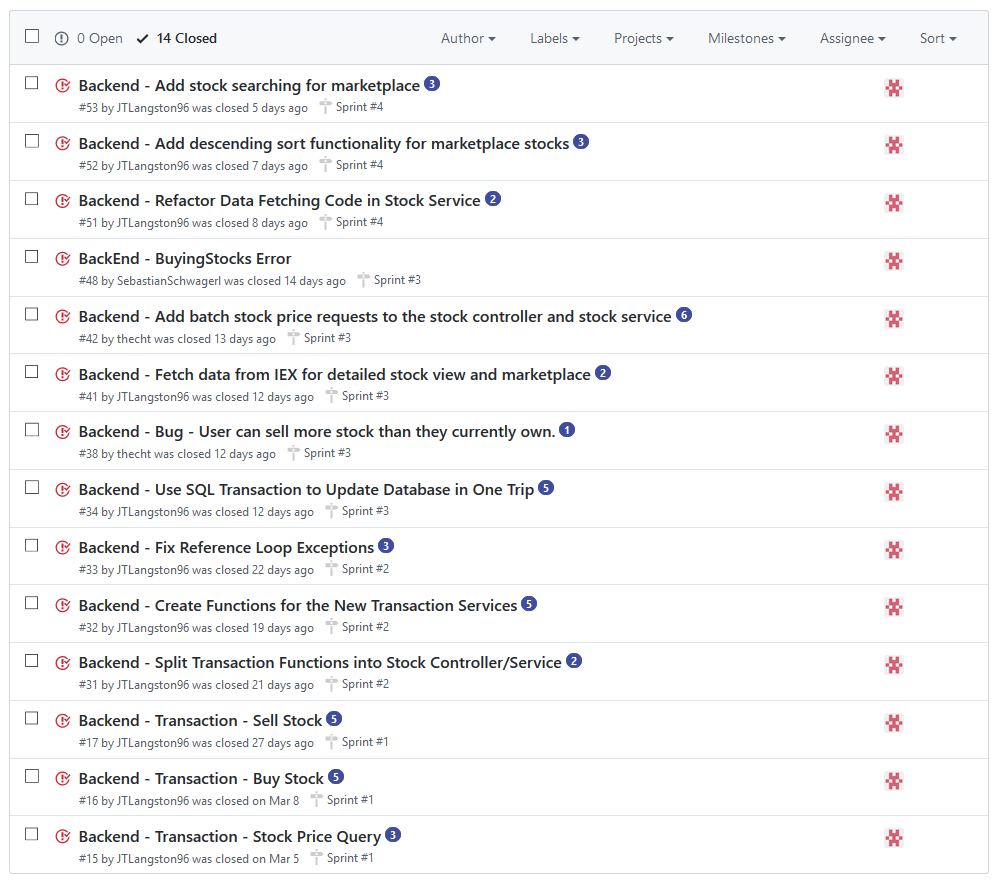
* + The frontend subsystems were mostly tested using xcode and an iphone simulator. The api calls were tested using postman to see if the data displayed by the frontend systems were accurate. Each subsystem was tested thoroughly using the above methods.

**7.3 Subsystem 3** – Jonathan Langston

Responsibilities:*Stock & Transaction backend functionality, Communication with IEX API*

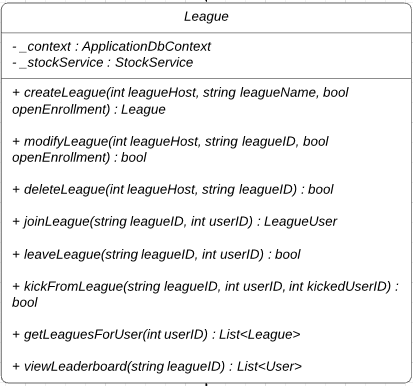
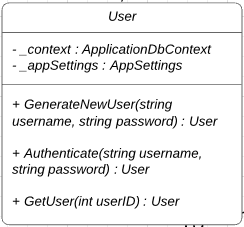
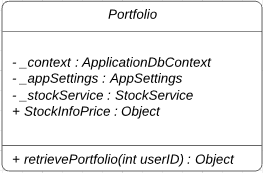
* Initial design and model
  + At first there was no Transaction controller, only the stock controller to handle all kinds of stock related functions and purchase functions. The controller has methods to buy stocks, sell stocks, get a price query, and get detailed information like price history to create a graph with.
  + Each controller has a corresponding service (i.e. Stock Controller and Stock Service) in order to separate the business logic.
  + Interaction with IEX API:  
    
* Data dictionary

|  |  |  |
| --- | --- | --- |
| Name | Data Type | Description |
| StockID | integer | The stock symbol. Also used as a composite key for tracking owned stocks. |
| StockQuantity | integer | The number of stocks owned by a user for a specific company. |
| Price | double | Price paid or earned from buying/selling a stock. |
| TransactionID | integer | Used as an index for the transactions in the database. |
| TransactionDate | DateTime | Time at which the exchange occurred. |
| TransactionType | string | Determine if the stocks were bought or sold. |

* Refined design
  + Eventually a Transaction controller needed to be added for functions like getting a transaction history for a user or any future functions that could be added after development is finished that relate specifically to transactions. This change was made since it would not make much sense to have transaction related functions in a controller meant to be used for stock related actions. It also improves future refinement of the app if any additional transaction functions are added in post-development patches.
  + Stock Service:  
    
  + Transaction Service:  
    
* Scrum Backlog (Product and Sprint - Link to Section 6)  
  
* Coding
  + The Stock and Transaction parts of the backend are written in C# and uses the .NET Core framework. Entity Framework Core is used to work with the database and make changes to it along with making changes to created objects being sent to the frontend.
  + Since C# was the language of choice, a more object-oriented approach was taken when building the backend to reduce the amount of repeated coding that was needed. Entity Framework Core helped accomplished this with their entity models.
* Testing
  + Once the code was written, it was tested using Postman to send http requests locally and using pgAdmin to handle the local database that was used for the testing environment.

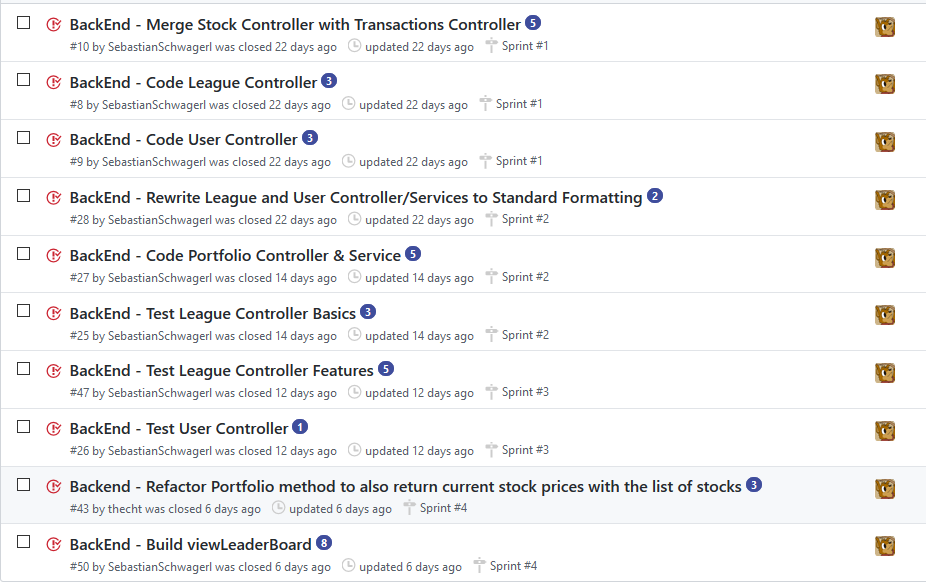
**7.4 Subsystem 4** – Sebastian Schwagerl

Responsibilities: *Portfolio, Leagues, and User backend methods.*

* Initial design and model
  + Initially, we planned for 5 Controllers and Services (User, League, Company, Stock, and Achievements)
  + Portfolio was then added to handle displaying data for a screen of the same name, while company and modifyingFunds from User was placed within the stock controller.
  + The services were intended to have global variables, though the controllers passed in all the values they needed by being auto-wired by .NET Core’s dependency injection system, which handles injection dependencies into constructors when configured in the app’s startup class.
  + League Service:  
    
  + User Service:  
    
  + Portfolio Service:  
    
* Data dictionary

|  |  |  |
| --- | --- | --- |
| Name | Data Type | Description |
| user | User | Contains the entirety of user data. |
| userID | integer | Used for tracking the user with one value. |
| userName | string | The entered username. Paired with password to access. |
| userPassword | string | Will be hashed before being stored. Entered password is compared to stored hashed password. |
| userCurrency | double | The current unused funds on a user’s account. |
| league | League | An entry for a league. Contains leagueID, Host, Name, CreationDate, and Enrollment type. |
| leagueUser | LeagueUser | A user within a league. Stores each of their IDs as well as their privilege level, though unused. |
| leagueID | integer | Used for tracking league. |
| leagueName | string | Name assigned to league upon creation. |
| leagueDate | DateTime | Date and time league was created. For use in record keeping, or potential deadlines. |
| stockID | integer | Used for tracking companies and stocks. |
| stockQty | integer | The number of stocks owned by a user for a specific company. |
| stockPrice | double | Price paid or earned from buying/selling a stock. |
| changePercent | decimal | The percentage change of a stock over a span of time. |
| stockInfoPrice | StockInfoPrice[] | An array of all stocks for a user, including quantity owned, price, and percent change. |
| leaderBoard | List<User> | An unsorted list of users with a known net worth. |
| sortedLeaderBoard | List<User> | An ordered list of users, sorted by their net worth. |

* Refinements
  + Most of the methods in the service are self-sufficient and do not rely on any global values outside of an ApplicationDBContext object and an object connecting to StockService.
  + Portfolio also became its own Controller and Service, split apart from User.
  + The CompanyController and Service were removed from the project as they were handled instead by the API in the Stock methods.
  + Achievements were not implemented due to time constraints for the frontend.
* Scrum Backlog



* Coding
  + Language: C#, with Linq database connections using Visual Studio Code.
  + Using C#, we went with an Object Oriented approach to programming. Doing this, we were able to easily keep everything in order, and along the lines of our EntityModels method, which tracked the parts of every object.
  + Every controller method passes back a serialized object version of what was returned from their service, so it always returns a string.
* User training
  + As this is the backend; user training is not required.
* Testing
  + Testing was handled through Postman and a Postgres Database using PGAdmin 4. Using these, I was able to perform tests without any frontend assistance.

**8. Complete System** – *Group responsibility*

* Final software/hardware product
* Source code and user manual – screenshots as needed - Technical report
  + Github Link
* Evaluation by client and instructor
* Team Member Descriptions

***This is just a guide, and use it to create/improve your report. Feel free to add sections. You are responsible for your own subsystem/s, not other members. You have to contribute to the team’s goals and objectives, and develop your subsystem/s, write your documents and slides.***

**Things to Consider:**

What to do if player has dropped to very low $ in the Stock game? Give chance to reset game? Have them deal with it like real life?

* That's a good question. We certainly don’t want them to keep making new accounts. Perhaps we have weekly achievements & milestones that are completable even with low $, which will give them more virtual currency. Perhaps some virtual currency can be given each day they log in, to incentivize them using the app often. We can come up with a list of ways to address this issue in the next week or two as we plan the app.

Shall we have a chart of stock prices available for every company, if possible to import?

* We might could keep up-to-date data on a certain number of high profile companies (like those in the S&P 500) from whatever financial API we use, and record those regularly. If a user searches for a specific ticker name not in the list, we can make another API call to grab that data in real time (or if they want to see the historical price of the stock, to display in a chart, perhaps there is a way to grab that from the financial API. Or maybe there are open databases out there with historical stock price information? In that case we could import it and store in our database.)

Complete Before Progress Report #2

**1 - DESIGN:**

Subsystems:

* Use-case, sequence, & UML Diagrams w/ descriptions if necessary. **(Everyone)**
* ~~Subsystem Communication + Overall System Diagrams.~~ **~~(Tyler)~~**
* ~~API Route list (GET, POST, etc.)~~ **~~(Trey)~~**
* ER Model **(Sebastian)**

**2 - ANALYSIS:**

* ~~Backend Systems Analysis~~ **~~(Trey)~~**
* Frontend Systems Analysis **(Luke)**
* Data Dictionary **(Sebastian)**
* Process Models **(Luke)**
* ~~Algorithm Analysis - Search & Sort.~~ **~~(Luke & Trey)~~**

**3 - Update 1-3 on Doc (Low Priority):**

* In the introduction, remove the tech information, replace with general functionality information.
* In the Project Requirements, go into more detail for “System.”
* In the Project Specifications, explain in more detail what each chosen piece of tech actually does for us and why (briefly) we chose it.

By Thursday:

1. UI Designs done
2. Product Backlog done
3. User + League controllers & services
4. Stock controller

**Product Backlog** :

FrontEnd:

* Create UI Designs
  + Portfolio Screen
  + Marketplace Screen
  + Leagues Screen
  + Detailed Stock View Screen
* Search for stocks in the Marketplace
* Leagues (shows a list of leagues.)
  + Create or Join League (popup).
  + Inspect League (tap league -> popup).
  + Leave League (in inspector popup)
  + Disband League (in inspector popup if owner.)

BackEnd:

* Add Authentication/Authorization
* Add a Controller method for every necessary API route. (Add Services to perform the business logic / work).
  + LeagueController
    - Create league -- allow user to create a league - 1
    - Join league -- allow a user to join an already created league
    - View leagues -- displays list of open leagues for user.
    - View leaderboard -- display sorted list of users within league.
  + TransactionController
    - Buy stock -- allow user to purchase a stock
    - Sell stock -- allow user to sell an owned stock
      * Update stock -- handles actions that add to or remove from existing stock quantity.
      * Delete stock -- if the stock qty reaches 0, it will be removed.
  + UsersController
    - Create user -- adds user and credentials to database.
    - Modify funds -- change the available currency of the user.