

MOCK STOCK

Theodore Hecht III | Jonathan Langston | Luke Orr | Sebastian Schwagerl

Table of Contents

1 - Introduction	2
2 - Project Requirements	2
3 - Project Specification	3
4 - System - Design Perspective	3
5 - System - Analysis Perspective	8
6 - Project SCRUM Report	11
7 - Subsystems	13
8 - Complete System (Source Code Links)	23

Mock Stock - Design Doc

Theodore Hecht III | Jonathan Langston | Luke Orr | Sebastian Schwagerl

1 - INTRODUCTION

Learning how to trade on the stock market can be intimidating. Prospective traders may be anxious to begin, due to the financial commitment and risk. Therefore, an app which allows people to practice trading without any financial commitment is needed. Mock Stock is designed to meet this need.

Mock Stock grants the user a wallet of virtual currency on account creation. It then allows the user to buy and sell “mock” stocks (which are virtual representations of real stocks, which are kept up to date at current market prices.) This can increase or decrease their virtual net worth over time as they practice investing.

Mock Stock also has social features. The primary social features is “Leagues”, in which users can create or join groups of players with a leaderboard showing the individual net worth of every user in the League. This feature is designed to encourage competition and drive user engagement with the application.

Mock Stock is a native mobile application, because this allows easy access for casual users.

2 - Project Requirements

2.1 - Functional Requirements

2.1.1 - FrontEnd Functional Requirements

- ❑ The app can display a registration/login screen.
- ❑ The app can display a user’s portfolio contents.
- ❑ The app can display a searchable and sortable grid of stocks available on the market.
- ❑ The app can allow buy/sell stock requests.
- ❑ The app can display detailed stock information, including historical data with graphs.
- ❑ The app can expose League social activities.

2.1.2 - BackEnd Functional Requirements

- ❑ The api can query real-time financial data from an external financial API.
- ❑ The api can secure routes and authenticate/authorize users to submit requests.
- ❑ The api can store and retrieve information about the user’s portfolio.
- ❑ The api can execute and store financial transaction requests.
- ❑ The api can create, store, modify, and retrieve a user’s “Leagues.”

2.2 Usability Requirements

2.2.1 - FrontEnd

- ❑ Efficiency: minimal clicks/touches required to reach the desired functionality.
- ❑ Intuitive: quick learnability
- ❑ Fast: users of modern mobile applications expect a snappy (non-laggy) user interface.

2.2.2 - BackEnd

- ❑ Efficiency: API calls must be quick and use few system resources.
- ❑ Usability: The API design must be logical and consistently designed for easy integration with the front end.

2.3 System Requirements

The front end will run on iPhones running iOS 12 or higher. The application will be tested on an iPhone 6s Plus, as well as various iOS simulators going up to iPhone XR. This platform was chosen because of developer familiarity with programming for iOS platforms, as well as the large user base iOS devices have.

Microsoft Azure's cloud infrastructure will be used to host the backend api, so the backend will need to run on that system. In particular, it will run on Azure's App Service, which can scale instances up or down depending on the traffic requirements. It also allows for automatic git deployments, which means we can deploy the application with a simple git push (after some configuration in the Azure portal.)

The data storage mechanism will use PostgreSQL, a relational database, hosted on Microsoft Azure's DBaaS platform for PostgreSQL.

2.4 Security Requirements

The api will be secured using a token-based JWT authentication system. In this system, there is a route for requesting a token, in which the user passes their username and password over HTTPS, and the system will generate a token with their username and roles built into the token. It is then signed with a cryptographic signature, to prevent tampering. On future requests, the client sends this token to the server with their request. This allows us to authenticate the user without needing a database hit on each request.

3 - Project Specification

Mock Stock's domain/genre is a finance practice tool. Because of its social features (which utilize gamification principles), the app would fall under the Game / Education genre. It's primary target is the mobile iOS platform.

There are two primary development environments required to produce the application, each of which have different tools and libraries. One is for the front end. One is for the back end. However, some commonly used tools across both environments are: git (for version control), ZenHub (for SCRUM product task management), and Postman (for making API calls.)

3.1 - Front End Technical Specification

- ❑ A computer running macOS. (This is required by Apple to build for iPhones.)

- ❑ XCode. This is Apple's supported IDE for building native iOS applications. It has lots of tools for debugging iOS applications.
- ❑ UIKit. This is Apple's iOS UI library, which will allow us to create a native looking user interface.
- ❑ Swift. This is Apple's new programming language for building iOS apps. It is more modern than Objective-C, which is their prior programming language.

3.2 - Back End Technical Specification

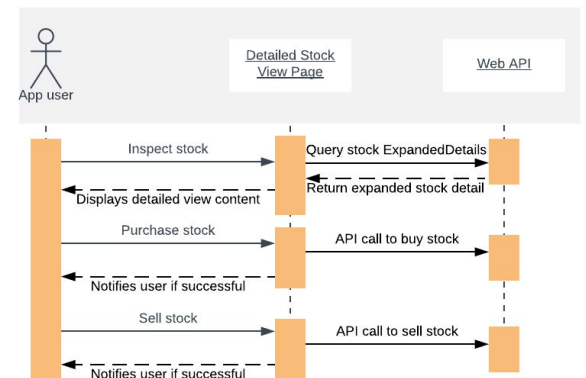
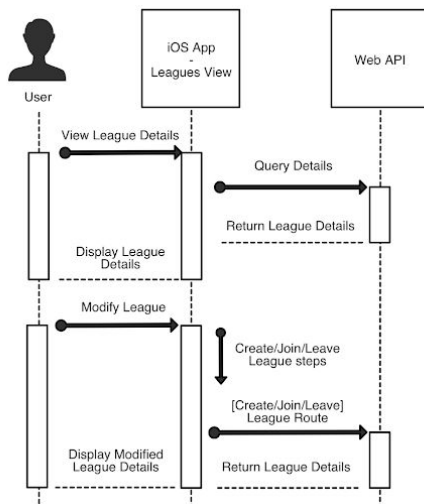
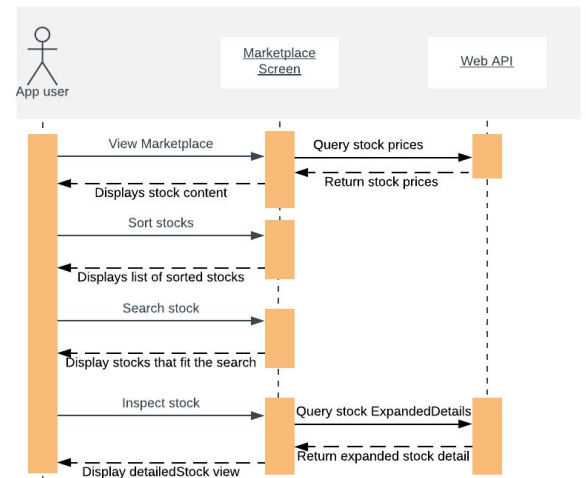
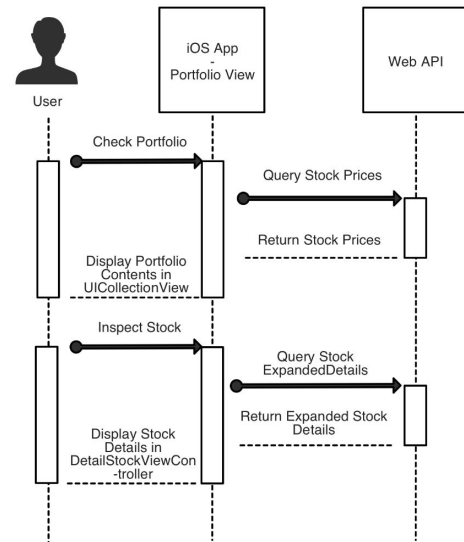
- ❑ C# and ASP.NET Core. This is Microsoft's newest framework for building web api apps, along with their primary supported language. This allows us to build out a REST api very quickly.
- ❑ Visual Studio Code. This is a lightweight, cross platform code editor which has extensions supporting .NET Core development. This was chosen over Visual Studio primarily due to its ability to be used on both Windows and macOS.
- ❑ PostgreSQL. This is a widely used, open source relational database management system. It is capable of storing and retrieving data for our use case, and is capable of being run on the hosting provider in our system requirements (Microsoft Azure's DBaaS platform for SQL databases.)

4 - System - Design Perspective

The front end subsystems are primarily view related, and are organized based on "ViewControllers", the controller part of the MVC pattern in iOS development. The back end subsystems are organized primarily by services and routes. (Routes handle intercepting HTTP requests, and services perform the business logic of the application.) Database access wasn't split into its own subsystem, since we relied largely on Entity Framework Core to provide a data access layer based on our entity models. However, database and entity model design is a backend subsystem.

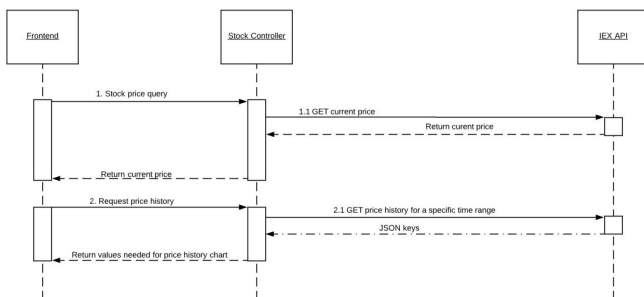
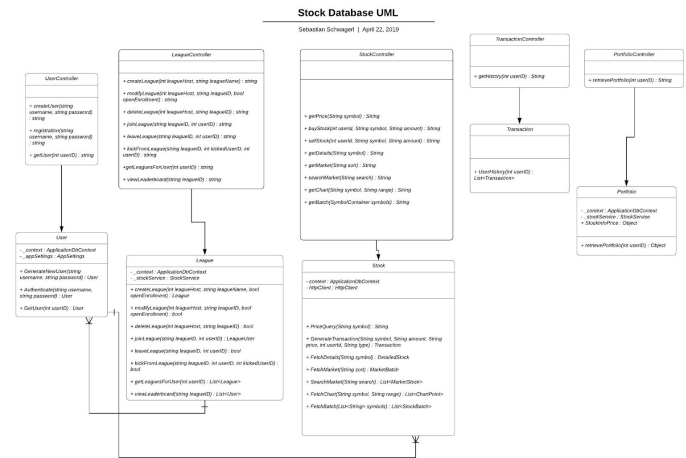
4.1 Frontend Subsystems

- ❑ Registration Screen: This screen displays two text fields for username and password, and two buttons, for registering a new user and logging in to an existing user.
- ❑ Portfolio Screen: This screen displays the user's current financial information (net worth, funds, stock worth), as well as all of the stocks in their portfolio. There is also a log out button for logging out of the system. Clicking on a stock reveals its details in the Detailed Stock Screen.
- ❑ Marketplace Screen: This screen displays a list of stocks currently on the market that are able to be bought and sold. It is a sortable list, and there is a text field to search for stocks you want to purchase. Clicking on a stock brings up more details in the Detailed Stock Screen.
- ❑ Detailed Stock Screen: This screen shows more stock detail, including historical price data on a graph. It includes buttons to let the user buy and sell stocks.
- ❑ Leagues Screen: This screen shows a list of leagues the user belongs to, along with their ID codes (which can be used by friends to join the league).
- ❑ League Details Screen: This screen displays a list of users within a particular league, along with their net worth. It is sorted by ascending order.



4.2 Backend Subsystems

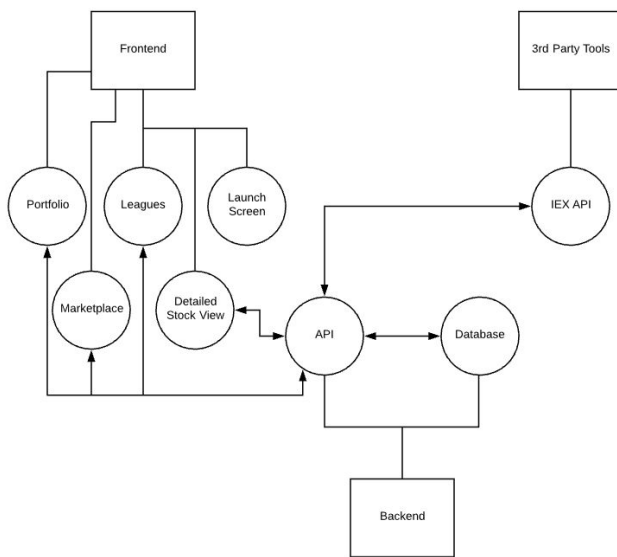
- ❑ **Users:** This subsystem handles user registration and JWT token generation. There is a route to create new users, which involves passing in a username and password and generating the new user which is stored in the database (with a hashed password.) There is a route to request a JWT token, which is then used for most other routes to establish the client's identity. This token is cryptographically signed to prevent tampering, so we can store the user's identity in the token and be sure of their identity when validating the token on each successive request.
- ❑ **Portfolio:** This subsystem handles fetching user portfolio data and calculating the user's portfolio meta data to be returned to the user.
- ❑ **Stocks:** This subsystem handles various stock fetching requests, including price requests, batch price requests, historical price chart data, etc.
- ❑ **Transactions:** This subsystem handles fetching past transactions from the database for admin use.
- ❑ **Leagues:** This subsystem handles routes for creating, joining, leaving, and deleting leagues. It also calculates leaderboard information by evaluating the total net worth of each user within the league, including all their stocks and funds.



4.3 Subsystem Communication

The frontend is made up of multiple screens that interact with the backend API after being verified through JWT token validation. They call the API when some kind of information is needed. Then the API communicates with the database or the IEX stock API to get information and supply it to the front end.

The portfolio and league screens will be sent information about the user from the database through the api. The detailed stock view and marketplace view 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.



4.4 API Routes

GET /api/leagues/id

Returns league details.

POST /api/leagues/createLeague

Creates a new league. Expects the league name.

POST /api/leagues/modifyLeague

Changes enrollment type.

DELETE /api/leagues/deleteLeague

Deletes the league, given the league ID.

POST /api/leagues/join/{leagueId}

Joins the league

DELETE /api/leagues/leave/{leagueId}

Leave a league

DELETE /api/leagues/kick/{leagueId}/{userId}

Kicks the user from the league

GET /api/leagues

Gets a list of all leagues user is in.

GET /api/leagues/leaderboard/{leagueId}

Gets sorted list of users for leaderboard.

GET /api/portfolio

Returns the authenticated user's portfolio details.

GET /api/stock

Retrieves price from stock ticker symbol.

POST /api/stock/buy

Buys a stock. Expects the ticker symbol and quantity.

POST /api/stock/sell

Sells a stock. Expects the ticker symbol and quantity.

GET /api/stock/details

Returns information needed for the stock details view

GET /api/stock/marketplace

Return a large list of stocks to show in the marketplace

GET /api/stock/chart

Returns a price history chart for a given stock in a given time range

GET /api/stock/Batch

Returns a list of stock details for the given symbols

GET /api/transactions

Get a list of previous transactions for the user

POST /api/users

Creates a new user.

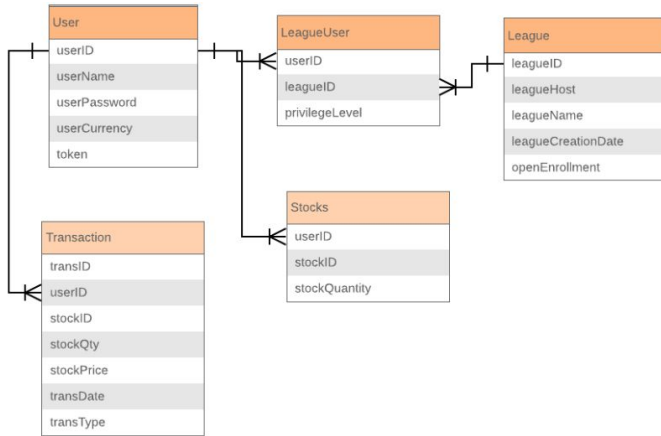
POST /api/users/token

Authenticates the user using their login credentials.

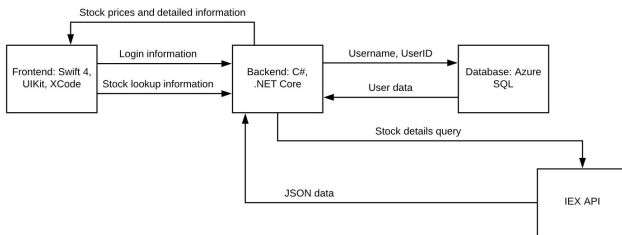
GET /api/users

Obtains user object using their ID.

4.5 - Entity Relationship Model (E-R Model)



4.6 - Overall Operation - System Model



5 - System - Analysis Perspective

5.1 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.

5.2 - 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.

5.3 - 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.

5.4 - Data Dictionary

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	decimal	The current unused funds on a user's account.
token	string	Handles verification of user after initial login.
leagueID	integer	Used for tracking league.
leagueHost	integer	The ID of the user who created the league. The host has special privileges in the league.
leagueName	string	Name assigned to league upon creation.
leagueCreationDate	string	Date and time league was created. For use in record keeping, or potential deadlines.
openEnrollment	boolean	Determines if league is open to everyone.
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.
transactionID	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.

transactionDate	int	Time at which the exchange occurred.
transactionType	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.

5.7 - Algorithm Analysis

5.7.1 - 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 $O(N)$.

5.7.2 - Sort

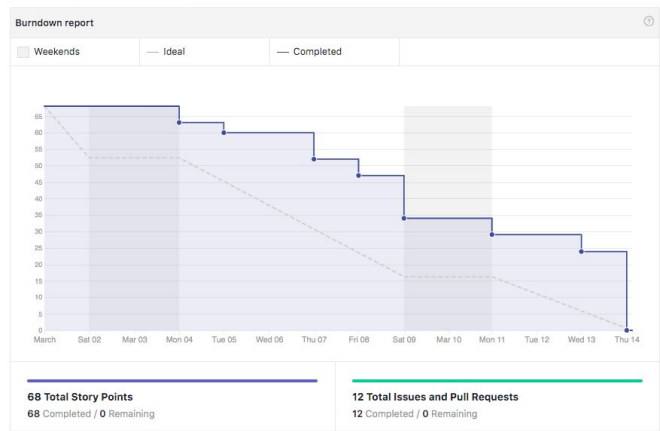
Sometimes users will want to sort stock entries (for example, by current value, or by their growth over a certain period of time.) Merge Sort would give us a worst case of $O(n \cdot \log(n))$ for sorting on current price. Sorting on the growth over a given period of time would be the same complexity, 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 $O(n + (n \cdot \log(n)))$, since we would need to iterate over the list first before sorting.

6 - Project SCRUM Report

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

6.1 - Sprint 1

6.1.1 - Sprint 1 Burndown Chart

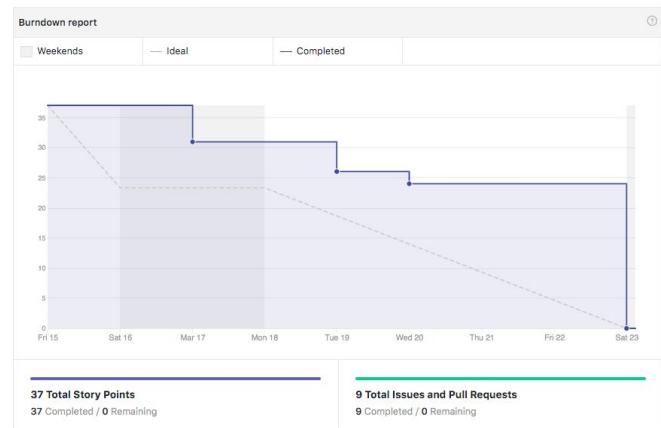


6.1.2 - Sprint 1 Product Backlog

Completed Issues and Pull Requests	Story points
FrontEnd - Feature - Add Tab Bar Controller MockStock #1 Closed ↗ Sprint #1	5
FrontEnd - Feature - Add Portfolio Screen MockStock #2 Closed ↗ Sprint #1	9
FrontEnd - Add Portfolio Collection Items MockStock #5 Closed ↗ Sprint #1	9
BackEnd - Code League Controller MockStock #8 Closed ↗ Sprint #1	3
BackEnd - Code User Controller MockStock #9 Closed ↗ Sprint #1	3
BackEnd - Merge Stock Controller with Transactions Controller MockStock #10 Closed ↗ Sprint #1	9
Frontend - Feature - Add Detailed View Screen MockStock #11 Closed ↗ Sprint #1	13
Frontend - Feature - Add Transaction Subscreen MockStock #13 Closed ↗ Sprint #1	8
Frontend - Add Detailed View Graph MockStock #14 Closed ↗ Sprint #1	5
Backend - Transaction - Stock Price Query MockStock #15 Closed ↗ Sprint #1	3
Backend - Transaction - Buy Stock MockStock #16 Closed ↗ Sprint #1	9
Backend - Transaction - Sell Stock MockStock #17 Closed ↗ Sprint #1	9

6.2 - Sprint 2

6.2.1 - Sprint 2 Burndown Chart

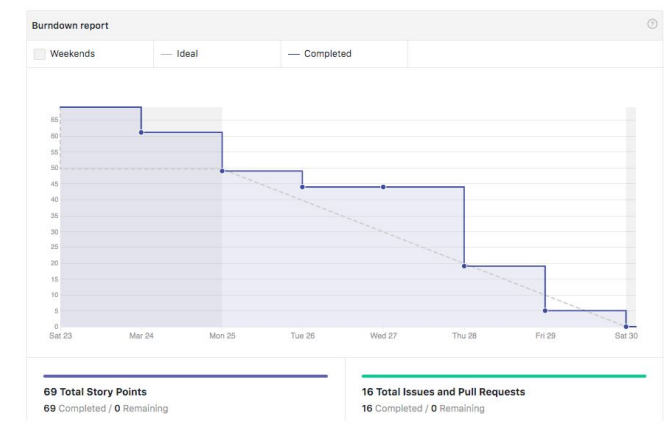


6.2.2 - Sprint 2 Product Backlog

Completed Issues and Pull Requests	Story points
FrontEnd - Configure Portfolio View Collection Items MockStock #18 Closed ↗ Sprint #2	8
BackEnd - Test League Controller Basics MockStock #25 Closed ↗ Sprint #2	3
BackEnd - Code Portfolio Controller & Service MockStock #27 Closed ↗ Sprint #2	8
BackEnd - Rewrite League and User Controller/Services to Standard Format... MockStock #28 Closed ↗ Sprint #2	2
FrontEnd - Transaction View Convert to Small Subscreen MockStock #29 Closed ↗ Sprint #2	3
Backend - Split Transaction Functions into Stock Controller/Service MockStock #31 Closed ↗ Sprint #2	2
Backend - Create Functions for the New Transaction Services MockStock #32 Closed ↗ Sprint #2	5
Backend - Fix Reference Loop Exceptions MockStock #33 Closed ↗ Sprint #2	3
DevOps - Host app on Azure MockStock #39 Closed ↗ Sprint #2	8

6.3 - Sprint 3

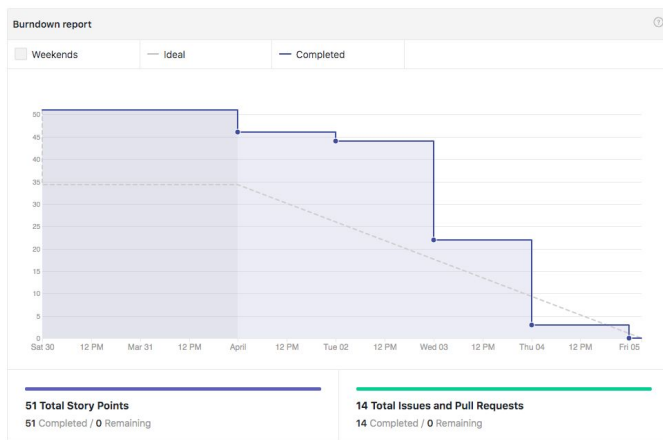
6.3.1 - Sprint 3 Burndown Chart



6.3.2 - Sprint 3 Product Backlog

Completed Issues and Pull Requests	Story points
FrontEnd - User can view leagues in LeaguesViewController MockStock #20 Closed ↗ Sprint #3	5
FrontEnd - user can create new league MockStock #21 Closed ↗ Sprint #3	5
BackEnd - Test User Controller MockStock #26 Closed ↗ Sprint #3	1
FrontEnd - Marketplace View Stock Object Box. MockStock #30 Closed ↗ Sprint #3	8
BackEnd - Use SQL Transaction to Update Database in One Trip MockStock #34 Closed ↗ Sprint #3	5
FrontEnd - Create user registration screen MockStock #35 Closed ↗ Sprint #3	6
FrontEnd - Store username and password in iOS keychain for automatic aut... MockStock #36 Closed ↗ Sprint #3	5
FrontEnd - Add data fetching to the login splash screen MockStock #37 Closed ↗ Sprint #3	6
BackEnd - Bug - User can sell more stock than they currently own. MockStock #38 Closed ↗ Sprint #3	1
DevOps - Host PostgreSQL database on Azure MockStock #40 Closed ↗ Sprint #3	5
BackEnd - Fetch data from IEX for detailed stock view and marketplace MockStock #41 Closed ↗ Sprint #3	2
BackEnd - Add batch stock price requests to the stock controller and stock ... MockStock #42 Closed ↗ Sprint #3	6
Frontend - Marketplace Add todays winners MockStock #44 Closed ↗ Sprint #3	3
Frontend - Marketplace View Add Todays Losers MockStock #45 Closed ↗ Sprint #3	3
BackEnd - Test League Controller Features MockStock #47 Closed ↗ Sprint #3	5
BackEnd - BuyingStocks Error MockStock #48 Closed ↗ Sprint #3	Not estimated

6.4.1 - Sprint 4 Burndown Chart



6.4.2 - Sprint 4 Product Backlog

Completed Issues and Pull Requests	Story points
BackEnd - Refactor/Cleanup Authentication code MockStock #19 Closed ↗ Sprint #4	1
FrontEnd - user can join a league MockStock #22 Closed ↗ Sprint #4	5
FrontEnd - user can leave a league MockStock #23 Closed ↗ Sprint #4	5
FrontEnd - user can inspect league details MockStock #24 Closed ↗ Sprint #4	6
BackEnd - Refactor Portfolio method to also return current stock prices wit... MockStock #43 Closed ↗ Sprint #4	3
Frontend - Marketplace View - Add Search Bar MockStock #46 Closed ↗ Sprint #4	2
BackEnd - Build viewLeaderBoard MockStock #50 Closed ↗ Sprint #4	8
BackEnd - Refactor Data Fetching Code in Stock Service MockStock #51 Closed ↗ Sprint #4	2
BackEnd - Add descending sort functionality for marketplace stocks MockStock #52 Closed ↗ Sprint #4	3
BackEnd - Add stock searching for marketplace MockStock #53 Closed ↗ Sprint #4	3
FrontEnd - Bug - Logging out and registering a new user still shows previous ... MockStock #54 Closed ↗ Sprint #4	2
FrontEnd-Refactor - Embed Portfolio View into a Navigation Controller MockStock #55 Closed ↗ Sprint #4	6
Frontend - Marketplace - Add drop down menu for sorting MockStock #56 Closed ↗ Sprint #4	3
Frontend - Marketplace - Redo marketplace to make use of navigation contr... MockStock #57 Closed ↗ Sprint #4	3

7 - Subsystems

7.1 - Theodore Hecht III

Responsibilities: Registration View, Portfolio View, Leagues View, Backend JWT Authentication

7.1.1 - Initial Design and Model

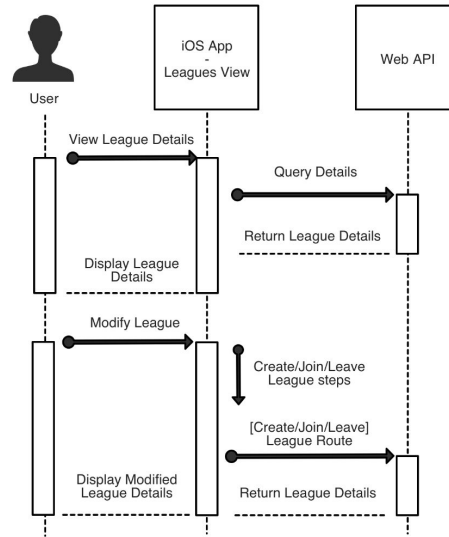
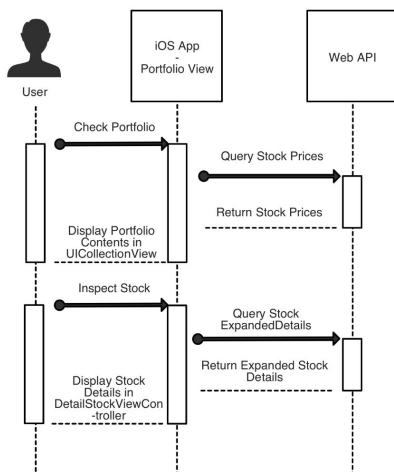
The Portfolio Screen displays the user's portfolio. It's main content is a UICollectionView with custom UICollectionView Cells to display portfolio data. The 'x' button lets you log out of the application.

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.

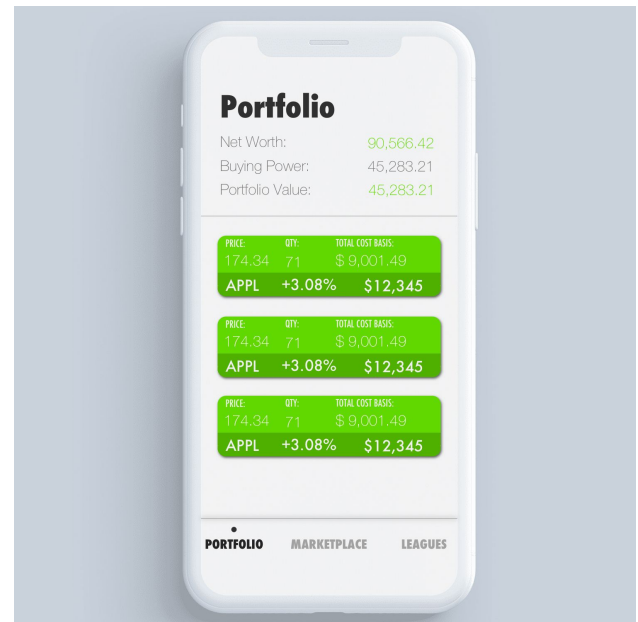
The Registration Screen displays a simple login/register form, where the user can enter their username and password, and register or login to the service, when the appropriate button is pressed.

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 which each view controller displays.

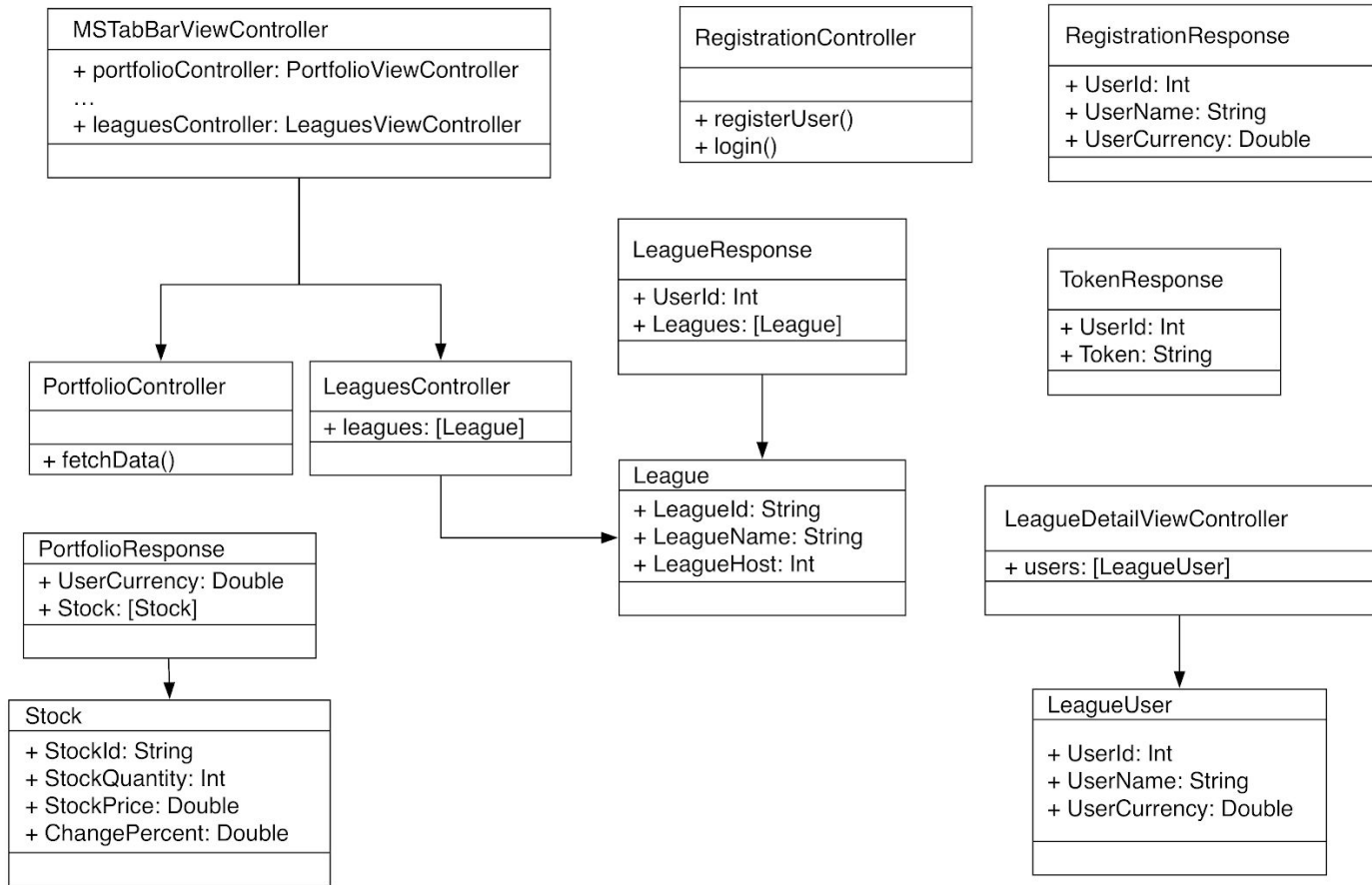
7.1.2 - Sequence Diagrams



7.1.3 - Initial UI Mockup



7.1.4 - Class UML Diagrams



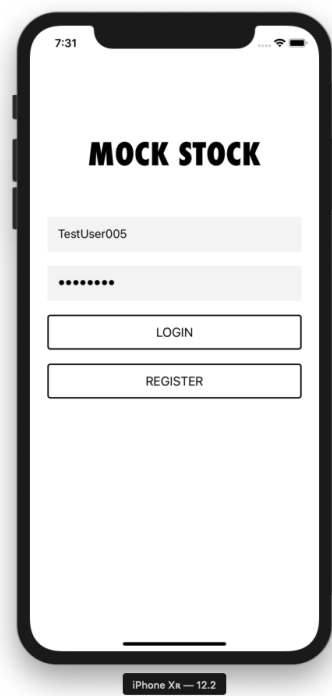
The app as a whole uses the MVC pattern, where the (not pictured) UIKit widgets are the views, which display information to the user, the ViewControllers are the controllers, which handle business logic and routing model data to the views, and the various json-decodable classes are the models.

The **MStabBarController** 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.

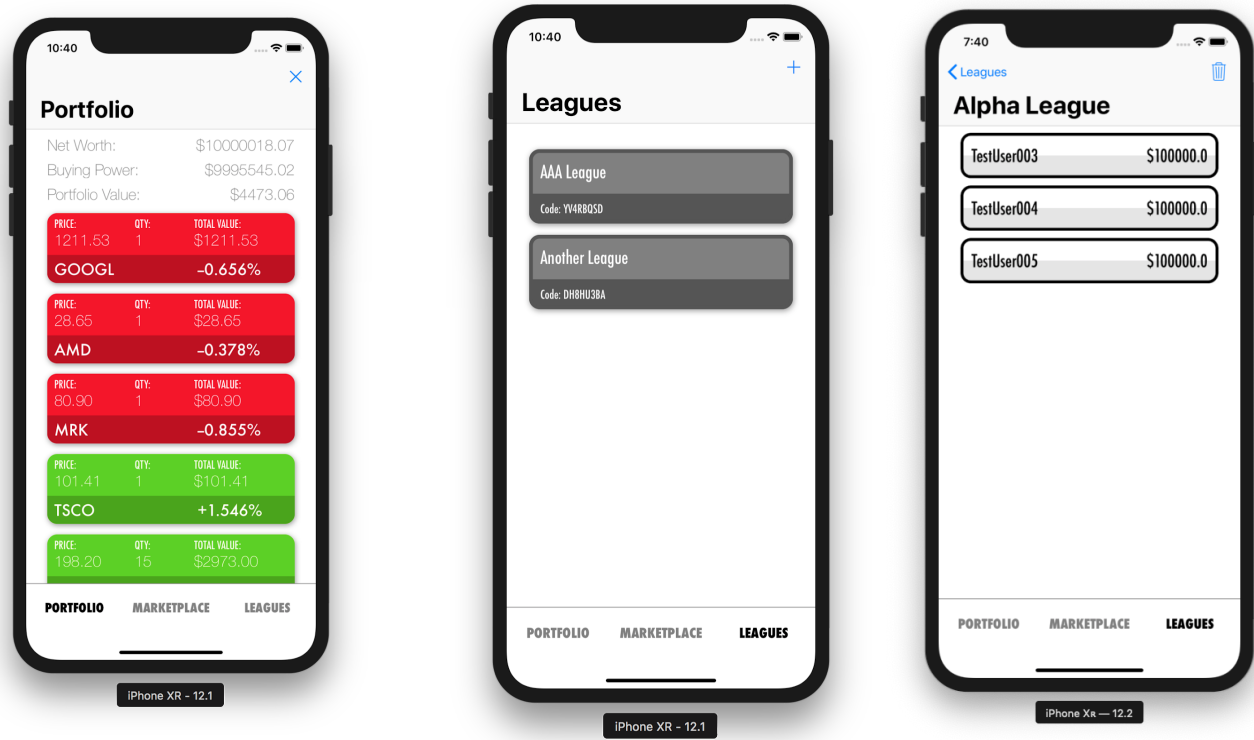
7.1.5 - Refined Over Time

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.



7.1.6 - New Design



7.1.7 - Scrum Backlog

0 Open	17 Closed	Author	Labels	Projects	Milestones	Assignee	Sort
		FrontEnd-Refactor - Embed Portfolio View into a Navigation Controller					
		#55 by thecht was closed 7 days ago					
		FrontEnd - Bug - Logging out and registering a new user still shows previous user's portfolio					
		#54 by thecht was closed 7 days ago					
		DevOps - Host PostgreSQL database on Azure					
		#40 by thecht was closed 17 days ago					
		DevOps - Host app on Azure					
		#39 by thecht was closed 17 days ago					
		FrontEnd - Add data fetching to the login splash screen					
		#37 by thecht was closed 15 days ago					
		FrontEnd - Store username and password in iOS keychain for automatic authentication					
		#36 by thecht was closed 15 days ago					
		FrontEnd - Create user registration screen					
		#35 by thecht was closed 15 days ago					
		FrontEnd - user can inspect league details					
		#24 by thecht was closed 7 days ago					
		FrontEnd - user can leave a league					
		#23 by thecht was closed 7 days ago					
		FrontEnd - user can join a league					
		#22 by thecht was closed 9 days ago					
		FrontEnd - user can create new league					
		#21 by thecht was closed 11 days ago					
		FrontEnd - User can view leagues in LeaguesViewController					
		#20 by thecht was closed 13 days ago					
		BackEnd - Refactor/Cleanup Authentication code					
		#19 by thecht was closed 7 days ago					
		FrontEnd - Configure Portfolio View Collection Items					
		#18 by thecht was closed 24 days ago					
		FrontEnd - Add Portfolio Collection Items					
		#5 by thecht was closed 28 days ago					
		FrontEnd - Feature - Add Portfolio Screen					
		#2 by thecht was closed on Mar 7					
		FrontEnd - Feature - Add Tab Bar Controller					
		#1 by thecht was closed on Mar 4					

7.1.8 - 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.

7.1.9 - 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.)

7.1.10 - 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 - Luke Orr

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

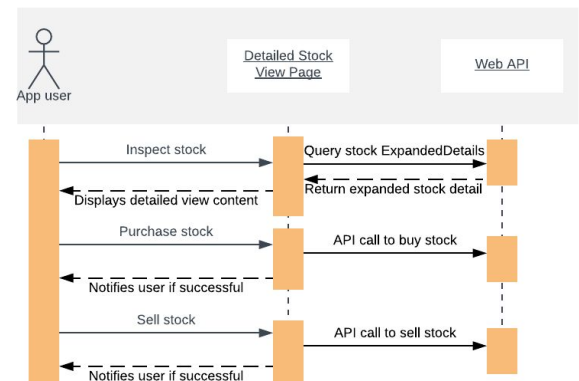
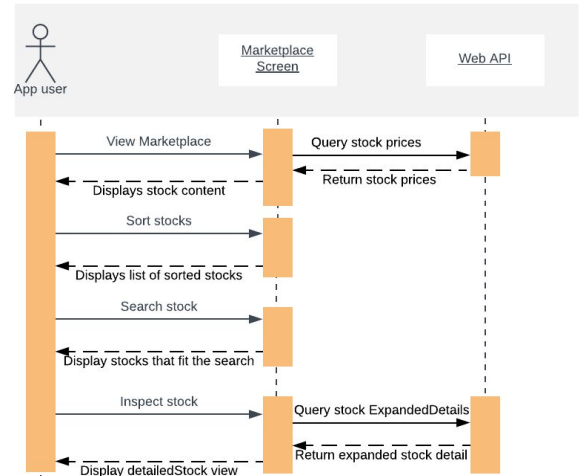
7.2.1 - Initial Design and Model

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.

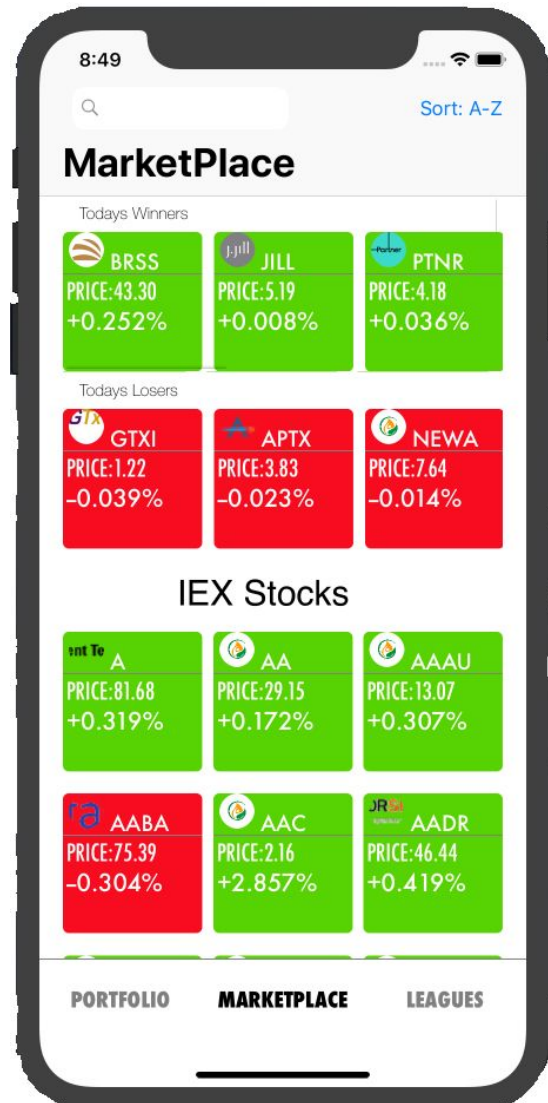
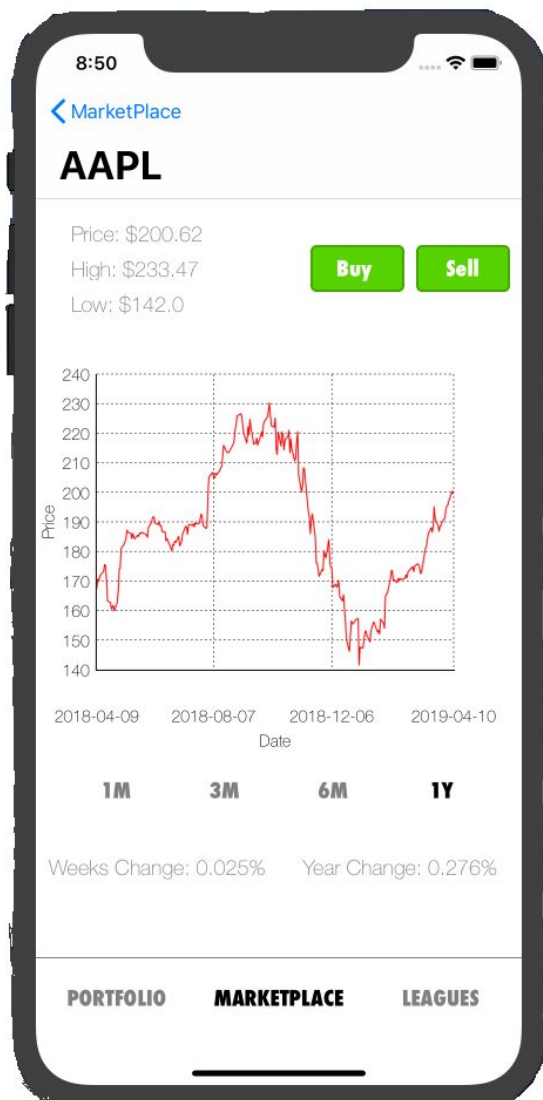
The Detailed View screen has three main features to it. One is that the screen displays more information about the stock that the user selected from either the portfolio or marketplace screens. The second feature is that users can buy or sell stocks from this screen by clicking either the buy or sell 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.

7.2.2 - Sequence Diagrams



7.2.3 - UI Designs














7.2.4 - 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 separate 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.

7.2.5 - SCRUM Backlog

<div> MockStock #12 Frontend - Add Detailed View Data Fetching</div> <div>8</div>	<div> MockStock #29 FrontEnd - Transaction View Convert to Small Subscreen</div> <div>Sprint #2</div> <div>3</div>
<div> MockStock #56 Frontend - Marketplace - Add drop down menu for sorting</div> <div>Sprint #4</div> <div>3</div>	<div> MockStock #13 Frontend - Feature - Add Transaction Subscreen</div> <div>Sprint #1</div> <div>8</div>
<div> MockStock #57 Frontend - Marketplace - Redo marketplace to make use of navigation controller</div> <div>Sprint #4</div> <div>3</div>	<div> MockStock #14 Frontend - Add Detailed View Graph</div> <div>Sprint #1</div> <div>5</div>
<div> MockStock #46 Frontend - Marketplace View - Add Search Bar</div> <div>Sprint #4</div> <div>2</div>	<div> MockStock #11 Frontend - Feature - Add Detailed View Screen</div> <div>Sprint #1</div> <div>13</div>
<div> MockStock #45 Frontend - Marketplace View Add Todays Losers</div> <div>Sprint #3</div> <div>3</div>	
<div> MockStock #30 FrontEnd - Marketplace View Stock Object Box.</div> <div>Sprint #3</div> <div>8</div>	
<div> MockStock #44 Frontend - Marketplace Add todays winners</div> <div>Sprint #3</div> <div>3</div>	

7.2.6 - Coding

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

7.2.7 - 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.

7.2.8 - 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.

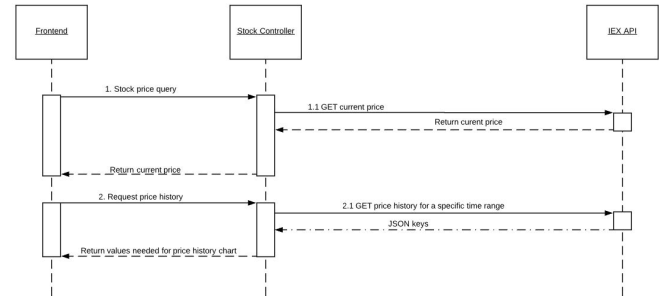
7.3 - Jonathan Langston

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

7.3.1 - 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.



7.3.2 - 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.

7.3.3 - 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

Stock Service
<ul style="list-style-type: none">- <code>_context: ApplicationDbContext</code>- <code>_appSettings: AppSettings</code>- <code>httpClient: HttpClient</code>
<ul style="list-style-type: none">+ <code>PriceQuery(str symbol)</code>+ <code>GenerateTransaction(str symbol, str amount, str price, int userID, string type)</code>+ <code>FetchDetails(str symbol)</code>+ <code>FetchMarket(str sort)</code>+ <code>SearchMarket(str search)</code>+ <code>FetchChart(str symbol, str range)</code>+ <code>FetchBatch(List<str> symbols)</code>

Transaction Service

Transaction Service
<ul style="list-style-type: none">- <code>_context: ApplicationDbContext</code>- <code>_appSettings: AppSettings</code>
<ul style="list-style-type: none">+ <code>UserHistory(int userId)</code>

7.3.4 - SCRUM Backlog

<input type="checkbox"/> 0 Open <input checked="" type="checkbox"/> 14 Closed	Author	Labels	Projects	Milestones	Assignee	Sort
<input type="checkbox"/> Backend - Add stock searching for marketplace	#33 by JTLangston96 was closed 5 days ago			Sprint #4		
<input type="checkbox"/> Backend - Add descending sort functionality for marketplace stocks	#32 by JTLangston96 was closed 7 days ago			Sprint #4		
<input type="checkbox"/> Backend - Refactor Data Fetching Code in Stock Service	#31 by JTLangston96 was closed 8 days ago			Sprint #4		
<input type="checkbox"/> Backend - BuyingStocks Error	#48 by SebastianSchwagerl was closed 14 days ago			Sprint #3		
<input type="checkbox"/> Backend - Add batch stock price requests to the stock controller and stock service	#42 by theicht was closed 13 days ago			Sprint #3		
<input type="checkbox"/> Backend - Fetch data from IEX for detailed stock view and marketplace	#41 by JTLangston96 was closed 12 days ago			Sprint #3		
<input type="checkbox"/> Backend - Bug - User can sell more stock than they currently own.	#38 by theicht was closed 12 days ago			Sprint #3		
<input type="checkbox"/> Backend - Use SQL Transaction to Update Database in One Trip	#34 by JTLangston96 was closed 12 days ago			Sprint #3		
<input type="checkbox"/> Backend - Fix Reference Loop Exceptions	#33 by JTLangston96 was closed 22 days ago			Sprint #2		
<input type="checkbox"/> Backend - Create Functions for the New Transaction Services	#32 by JTLangston96 was closed 19 days ago			Sprint #2		
<input type="checkbox"/> Backend - Split Transaction Functions into Stock Controller/Service	#31 by JTLangston96 was closed 21 days ago			Sprint #2		
<input type="checkbox"/> Backend - Transaction - Sell Stock	#17 by JTLangston96 was closed 27 days ago			Sprint #1		
<input type="checkbox"/> Backend - Transaction - Buy Stock	#16 by JTLangston96 was closed on Mar 8			Sprint #1		
<input type="checkbox"/> Backend - Transaction - Stock Price Query	#15 by JTLangston96 was closed on Mar 5			Sprint #1		

7.3.5 - 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.

7.3.6 - 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 - Sebastian Schwagerl

Responsibilities: Portfolio & Leagues backend controllers and services, and database schema design.

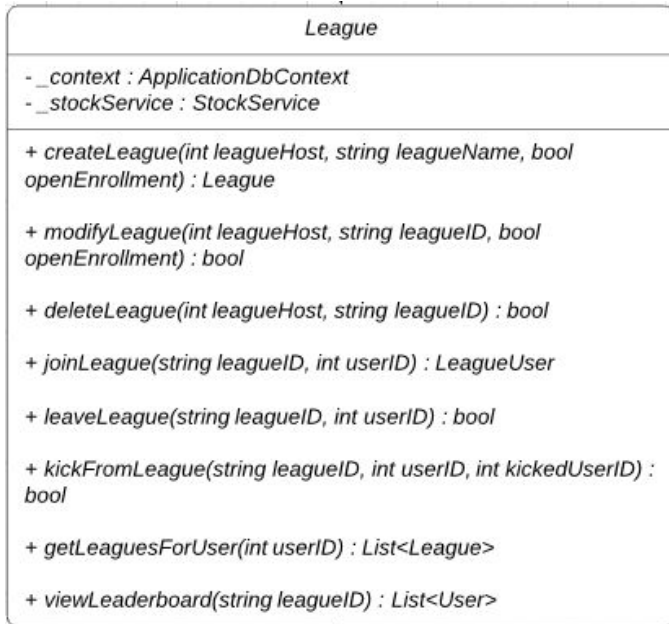
7.4.1 - Initial Design and Model

Initially, we planned for 5 Controllers and Services (User, League, Company, Stock, and Achievements)

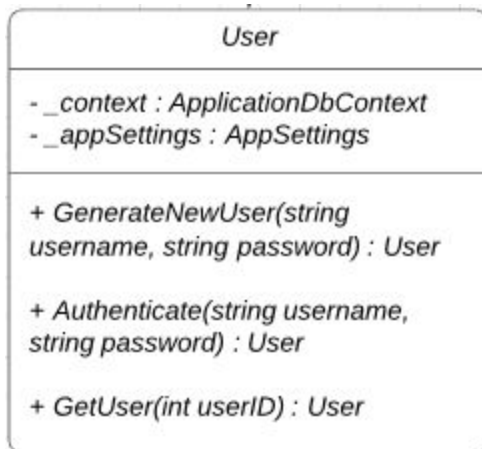
Portfolio was then added to handle displaying data for a screen of the same name, while company and modifyingFunds from User was placed within the stock controller.

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.

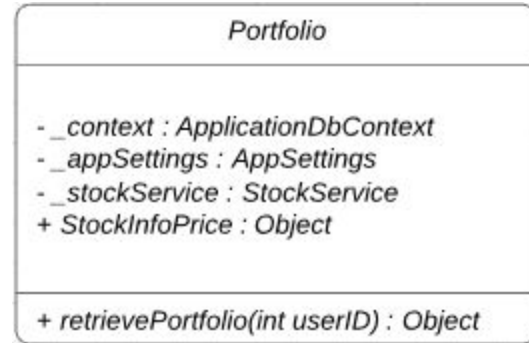
League Service:



User Service:



Portfolio Service:



7.4.2 - 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 net worth.

7.4.3 - 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.

7.4.4 - SCRUM Backlog

<input type="checkbox"/>	BackEnd - Merge Stock Controller with Transactions Controller		#10 by SebastianSchwagerl was closed 22 days ago		updated 22 days ago		Sprint #1	
<input type="checkbox"/>	BackEnd - Code League Controller		#8 by SebastianSchwagerl was closed 22 days ago		updated 22 days ago		Sprint #1	
<input type="checkbox"/>	BackEnd - Code User Controller		#9 by SebastianSchwagerl was closed 22 days ago		updated 22 days ago		Sprint #1	
<input type="checkbox"/>	BackEnd - Rewrite League and User Controller/Services to Standard Formatting		#28 by SebastianSchwagerl was closed 22 days ago		updated 22 days ago		Sprint #2	
<input type="checkbox"/>	BackEnd - Code Portfolio Controller & Service		#27 by SebastianSchwagerl was closed 14 days ago		updated 14 days ago		Sprint #2	
<input type="checkbox"/>	BackEnd - Test League Controller Basics		#25 by SebastianSchwagerl was closed 14 days ago		updated 14 days ago		Sprint #2	
<input type="checkbox"/>	BackEnd - Test League Controller Features		#47 by SebastianSchwagerl was closed 12 days ago		updated 12 days ago		Sprint #3	
<input type="checkbox"/>	BackEnd - Test User Controller		#26 by SebastianSchwagerl was closed 12 days ago		updated 12 days ago		Sprint #3	
<input type="checkbox"/>	BackEnd - Refactor Portfolio method to also return current stock prices with the list of stocks		#43 by thecht was closed 6 days ago		updated 6 days ago		Sprint #4	
<input type="checkbox"/>	BackEnd - Build viewLeaderBoard		#50 by SebastianSchwagerl was closed 6 days ago		updated 6 days ago		Sprint #4	

7.4.5 - 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.

7.4.6 - User Training and Testing

As this is the backend, user training is not required.

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

8.1 - Source Code:

- ☐ Main repo: <https://github.com/thecht/MockStock>
- ☐ Backend: <https://github.com/thecht/MockStock-Backend>
- ☐ Frontend: <https://github.com/thecht/MockStock-FrontEnd>

User manual is in the main GitHub repo, along with all source code and previous presentations.

8.2 - Team Contributions

- ☐ Theodore Hecht III - Registration View, Portfolio View, Leagues View, Backend JWT Authentication
- ☐ Luke Orr - Marketplace, Stock Detail, Graph and Transaction frontend views.
- ☐ Jonathan Langston - Stock & Transaction backend functionality, Communication with the IEX API.
- ☐ Sebastian Schwagerl - Portfolio & Leagues backend controllers and services, and database schema design.