
User & Design Specifications
for
“Capital Games”

Report 3: Part 1
Software Engineering
14:332:452

Team 2:

Jeff Adler
Eric Cuiffo
Nick Palumbo
Jeff Rabinowitz
Val Red
Dario Rethage

May 3, 2013
Version: 1

Contents

Contents	2
1 Customer Statement of Requirements	4
1.1 Problem Statement	4
1.2 Glossary of Terms	6
2 System Requirements	8
2.1 User Stories	8
2.2 Nonfunctional Requirements	10
2.3 On-Screen Appearance Requirements	11
3 Functional Requirements Specification	13
3.1 Stakeholders	13
3.2 Actors and Goals	13
3.3 Use Cases	14
3.4 System Sequence Diagrams	21
4 User Interface Specification	28
4.1 Preliminary Design	28
4.2 User Effort Estimation	28
5 Effort Estimation	44
5.1 Background	44
5.2 Unadjusted Use Case Points	44
5.3 Technical Complexity Factors	46
5.4 Environmental Complexity Factors	47
5.5 Calculations	47
6 Domain Model	48
6.1 Concept Definitions	48
6.2 Association Definitions	49
6.3 Attribute Definitions	50
6.4 System Operation Contracts	53
6.5 Economic and Mathematical Models	54
7 System Interaction Diagrams	56
7.1 Introduction	56

7.2	Financial Data Retrieval Subsystem	56
7.3	Asynchronous Processing Subsystem	61
7.4	Design Patterns	65
8	Class Diagrams and Interface Specifications	67
8.1	Financial Adaptor Class Diagram	67
8.2	Financial Adaptor Data Types and Operation Signatures	68
8.3	Financial Adaptor Traceability Matrix	70
8.4	Design Patterns	72
8.5	Asynchronous Subsystems	73
9	System Architecture and System Design	76
9.1	Architectural Styles	76
9.2	Identifying Subsystems	77
9.3	Mapping To Hardware	78
9.4	Persistent Data Storage	78
9.5	Network Protocol	80
9.6	Global Control Flow	80
9.7	Hardware Requirements	81
10	Data Structures	85
10.1	Table	85
10.2	Queue	85
10.3	Tree	85
11	User Interface Design and Implementation	86
11.1	Finalized Product	86
12	Design of Tests	94
12.1	Test Cases	94
12.2	Test Coverage	95
12.3	Integration Testing	96
12.4	Test Cases	96
13	History of Work, Current Status, & Future Work	98
13.1	History of Work	98
13.2	Current Status	99
13.3	Key Accomplishments	99
13.4	Future Work	100
13.5	Project Management	101
	References	102

1 Customer Statement of Requirements

1.1 Problem Statement

Perhaps nothing portrays capitalism better than the Stock Market. The ability for individuals and collectives to gain equity in international corporations, trade that equity, and perhaps even gain a profit, has piqued the imagination of a nation for well over a century. One could even say that owning stock is part and parcel of The American Dream.

However, there is a barrier that separates this dream from reality for many would-be investors: an understanding of the market. The stock market has myriad intricate ways of bundling and exchanging instruments, most of which will be beyond the ken of an economic novice. An economist may be interested in the differences between Mutual Funds and Exchange-Traded Funds; a banker may have the judgment to decide between a Stop Order and a Market Order. These financial techniques offer greater flexibility and control over investments to experienced investors and scientists, who are masters of the field. The beginner does not care to be bothered by these techniques, as they can turn a straightforward process into an overwhelming headache.

With Capital Games we are interested in developing a learning platform for these students - a stock market simulation program.

Capital Games is marketed at two primary classes of user; students and novice investors, each of whom have different needs. Students require a social aspect to their experience - shared simulation instances with global rules and social features. Novice investors require performance metrics and research tools. Both require interactive tutorials, visualization tools, and email updates, in addition to the core requirement of being able to execute various types of trades.

At its simplest, Capital Games is about exchanging stocks and managing investments. This is done through the respective menus for each Research, Trading, and Managing Portfolios. Research allows investors to analyze relevant financial metrics of publically traded corporations. Trading allows investors to place market, stop, and limit orders for their various portfolios. Managing Portfolios allows investors to view their investments and performance metrics for each of their simulations. In all menus, data can be visualized and interactively examined, in addition to being tabulated. This unprecedented level of accessibility will ease accessibility to market trend analysis.

Portfolios and trades only exist in the context of leagues, or market simulation instances. Each league has with its own rules, administrators, and varying privacy levels. Investors can participate in both public leagues, which anyone can join but offer less social interactivity, and private leagues, which require private email or Facebook invitations but which have expanded social features. Leagues are social because they include Trade Streams of executed trades from league members, Investor Profiles containing trade history and portfolio performance of investors, and a Comments Board. Additionally, each league will have a scoreboard for its members portfolio performances. Top investors will have their names and net worth displayed prominently on league

pages.

As a site with social content, it is also important to have the ability to moderate and review submissions by users. This is provided by having two classes of moderators, League Managers and Site Administrators. League Managers are, by default, the users who create a given league, and can ban, invite, and promote users within their leagues, as well as being able to delete comments and create league-wide announcements. League Managers, by default, are also participating in a given league. Site Administrators can delete leagues, ban users and delete comments, add front page announcements, view reports about abusive users, and view other various statistics about users, trades, and leagues. As mentioned previously, even tighter social network integration is a long-term goal.

With the continuing influx of mobile browsing and computing devices to the personal computing market, it is increasingly important to have a single unified interface for users. This is accomplished through the use of Responsive Design, in which a single web page automatically and intelligently reflows itself to accommodate devices of any screen size. This revolutionizes the trading experience – users don't want to use a dozen odd applications and browsers to access their favorite sites, they want to just click-and-go. These changes are made possible by improvements in mobile browsers, which now universally support Javascript. Therefore one site really will be enough for all users.

As alluded to previously, a strong emphasis of this platform is the use of interactive portfolio graphs. Previous systems have failed to speak directly to users because they presented static images that were impossible to manipulate or interact with. This is a core design feature of Capital Games. We employ the newest, most state-of-the-art graphing tools to allow a user to see any and every stock and portfolio over an indefinite time span with the finest degree of granularity.

Another way of enhancing user experience is by letting users opt to receive daily or weekly email updates about their portfolio performance. This is a feature set that all financial investors have access to and therefore is something that novice investors who use our platform should not be denied. Previous learning platforms have failed to develop a respectable e-mail service – their demos barely covered assets, and certainly didn't mention trends. We will provide the first fully functional e-mail system.

These features, together with other core capabilities such as email updates and interactive tutorials, provide the most cutting-edge and modern platform for both individual and collaborative efforts to conduct financial simulations.

1.2 Glossary of Terms

Margin – Borrowed capital used to execute trades, i.e. “buy on margin”. Although leveraging margin is possible for normal buy and sell orders, they are critical to short orders, in which the entire stock is sold without actually being owned. [1] Margin can refer to both the act of purchasing a stock on credit, and to the percentage of a stock’s equity value required to be held in capital against the risk of the stock decreasing in value. [2]

League – an instance of a market simulation with a predefined rule-set and containing many *investors*. All leagues are created by a *League Manager* There are two types of leagues:

- **Public** – Any Investor can join this type of League
- **Private** – A private league can only be seen by its members and administrators. A User does not join this league, rather they are placed into it by a League Manager.

Order – An *investor* must place an order for the purchase or sale of a *stock*.

- **Stop Order** – A type of order used to protect gains or limit losses. Stop loss orders are activated if a stock drops below the stop price and buy stop orders are activated if a stock rises above the stop price. [3] When activated, a Stop Order becomes a *Market Order*.
- **Limit Order** – A type of order used to prevent trades from occurring except at indicated prices. Buy limit orders will only be executed at or below the indicated price, and sell limit orders will be executed at or above the indicated price. Limit orders are not guaranteed to ever be executed and expire after a specified duration. [4]
- **Market Order** – An order to be executed as soon as possible at current market prices.[5]
 - **Short Order** – A type of transaction in which an Investor symbolically borrows a certain number of stocks (using their existing Margin) and sells them at market price, expecting the stock value to decrease and to make a profit when exiting the position. Exiting is called a *cover*. [6]
 - **Cover Order** – A type of transaction in which an Investor purchases stocks to *cover* the symbolic loan of stocks created by a *short order*. [7]

Portfolio – A detailed account of the *stocks* associated with an *investor* in a given league. Portfolios are unique.

Stock – A type of asset that represents equity in a company.

- **Ask Price** – The price at which a trader is willing to sell a stock.
- **Bid Price** – The price a trader is willing to pay for a stock.
- **Bid-Ask Spread** – The bid-ask spread describes the difference in price between the bid and the ask. These two prices are marginally different, but always with the ask being the more expensive of the two. It represents the friction inherent in trading a stock. [8]
- **Ticker Symbol** – an abbreviation used to uniquely identify publicly traded shares of a particular stock on a particular stock market.
- **Symbol List** – a list of a market/several market’s ticker symbols.

User Roles – Each user with an account can have one or more of the following roles:

- **Investor** – A instance of the User, who commits capital expecting to see it grow in value. Users Instances are referred to as *investors*.
- **League Manager** – A League Manager is an *investor*. A user does not necessarily have this role for every league they are in. Only ones in which they created the League or were given a League Manager role from another League Manager of that League. League Managers control settings of leagues.
- **Site Administrator** – This is the most powerful role. A Site Administrator is a user with elevated privileges, to ban users and delete offensive comments.
- **Suspended** – A user with this role is currently pending losing their Suspended Role, or granted a *Banned* role. While suspended an *Investor* cannot do anything with their Account other then login and view the duration of their suspension, The reasoning behind their ban, and an appeal form if the situation permits.
- **Banned** – A banned user can never be unbanned, this occurs after a rejected suspension appeal.

2 System Requirements

2.1 User Stories

Identifier	User Story	Weight
ST-1	As a user, I can register an account so that I may participate in Capital Games.	10 pts
ST-2	As a user, I can join or create leagues so that I may compete with others in a simulated stock market environment based on real-time stock data.	10 pts
ST-3	As a user, I can search for companies both by company name and stock symbol so I may scout companies I would like to invest in.	6 pts
ST-4	As a user, I can browse a companies profile and view the performance data over a configurable span of time so that I may determine whether or not I want to invest in them.	6 pts
ST-5	As a user, I can buy or sell stocks within a fantasy league I am a member of so I may build my fantasy league portfolio.	10 pts
ST-6	As a user, I can manage my portfolio within a league to track my investments.	8 pts
ST-7	As a user, I can visually track my finances via graphs and charts so I may more easily manage my portfolio.	4 pts
ST-8	As a user new to the stock market, I will have access to tutorials that teach about the stock market via a specially created novice fantasy league.	6 pts
ST-9	As a user, I can see the performance of stocks I invested in via a stock-ticker like marquee so I may have a quick overview of my day-to-day performance.	3 pts

ST-10	As a user, I can see an activity stream of recently executed trades by other users in my leagues so I am always up to date.	5 pts
ST-11	As a user, I can see the performance of other users' portfolios so I may observe the investment habits of others.	2 pts
ST-12	As a user, I can view a list of all members in each of my leagues so I know how many others I am competing with.	1 pt
ST-13	As a user, I can view a portfolio leaderboard so I may have a summary of relative performance between users in my league.	1 pt
ST-14	As a user, I can submit abuse reports on users so I may continue having a positive fantasy league experience. See section 6.5 for details.	5 pts
ST-15	As a user, I can message other users so I may interact with people I am playing within and out of my league.	4 pts
ST-16	As a user, I can post, edit, or delete comments to league pages so I may communicate with leagues en masse.	2 pts
ST-17	As a user, I can opt to receive periodic e-mail notifications of my stock performance or trades so I may be kept up to date even when not actively viewing the site.	3 pts
ST-18	As a user, I can additionally link my account with Facebook so I may share my fantasy league experience with friends.	1 pt
ST-19	As a user, I can recover or change my password so I may always have access to my own account.	5 pts
ST-20	As a user, I can access my profile and settings on a dashboard on the top of every page within the site.	8 pts
ST-21	As a user, I may opt to create a league and become a league manager so I may have my own personal league.	10 pts
ST-22	As a league manager, I can add league rules, a league name, and a league logo to personalize my league.	8 pts
ST-23	As a league manager, I may manage players within the league so I may invite players I want to join, ban players that are being abusive, and assign other league managers.	8 pts

ST-24	As a league manager, I can moderate and delete comments in the league page.	5 pts
ST-25	As a league manager, I can create league announcements.	4 pts
ST-26	As a site administrator, I can view reports of and delete leagues that are abusive in nature.	2 pts
ST-27	As a site administrator, I can delete abusive/offensive comments and ban users or IP addresses so the website remains a clean, positive stock market fantasy league experience.	6 pts
ST-28	As a site administrator, I may post front page news or announcements.	3 pts
ST-29	As a site administrator, I may have access to a user count, number of active leagues, total leagues, quantity of daily transactions, the most/least popular stocks, and newly created or banned users so I may have reliable site statistics.	9 pts

Notes

For representations of use cases that relate to visual requirements, see: [4](#).

ST-18: "Experience" refers to things such as stock purchases, current capital, position within the league, etc.

For other details on the specifications of these user stories, refer to [3.3](#).

2.2 Nonfunctional Requirements

Functional

Additional features for security could be enabled through the use of various third-party plugins. There exists several packages for the purpose of authentication and authorization of applications. Key authentication features to are the the ability to encrypt and store passwords, provide recovery options for users that have forgotten their password and store a cookie to validate the session. Other plugins may provide authorization features. These will allow for a user to perform different actions based on their position. For example, a user will be able to comment and delete their own comment, but an admin will be able to comment and delete all comments on the league they are an admin of.

Usability

A key point in the design of this application is ease of use and appeal to the users. Through the use of CSS and Bootstrap, we will be able to make the theme of our application consistant and pleasing. With CSS, we will create a universal header and navigation bar that each page will build off of. Javascript will provide for responsiveness and it will be the key framework for which we build our interactive tutorials upon. The interactive tutorials are meant for inexperienced users in

the topic of stocks to learn the fundamentals of the game. Any user that finds themselves lost later on can always view these tutorials again or browse through any specific topic.

Reliability

In order to ensure that there is no confusion to the user in the case of the internet or server failure, all transactions end with a final confirmation, and no changes to the account are made until after this confirmation. A user that leaves the application and returns later will still be logged in. Server failures should be dealt with by the application's host.

Performance

The performance of the site is mostly maintained by an appropriate technology stack consisting of a web framework, database, web server, and assets server. Performance management tools are built into the server for maintenance by application developers (not necessarily site administrators).

Supportability

Various measures and plugins exist for supportability. Between a combination of plugins enabling test-driven and behavior-driven development, supporting and modifying should be relatively painless. The project should be highly portable in the sense that a user will be able to access the website on all major browsers and mobile devices, and have a specialized appearance for both. For maintainability, there is the option of a user to be a site admin. These users can view details about the entire site, such as activity and user feedback. There also exists an internationalization framework for translating to provide multi-language support.

2.3 On-Screen Appearance Requirements

The on-screen appearance requirements fall into three general areas including utilizing responsive design, conforming to most popular screen resolutions and refraining from the use of non-universally supported client-side technologies. [9] As more and more devices are becoming capable of browsing the web, one of the main on-screen requirements is to implement responsive client-side markup that can intelligently adapt to the clients UI capabilities. As shown in Figure 2.1, a single page intelligently redistributes its elements to provide a unified interface across devices (in this case, a desktop browser and a smartphone). These capabilities include screen size, screen resolution and input methods. With these points in mind, Capital Games will be built to be usable on traditional desktop browser environments as well as mobile platforms. Javascript will be used to determine the best presentation of a page depending on the users browser.

While a number of standards are emerging in the mobile market in regards to standard screen resolutions, there is still great variability present in conventional monitor sizes and resolutions. According to w3schools.com, as of February 2013, less than 10% of Internet users have a screen resolution less than 1024x768. Therefore, an additional on-screen appearance requirement make Capital Games usable with screen resolutions greater than or equal to 1024 x 768. Finally, client side technologies must also be restricted to ones that are universally supported. Adobe Flash technology will not be used, as it isnt universally supported. Flash can also become pretty sluggish on the clients browser. This constraint will most likely lead to faster content loading and a more fluid user experience. Instead, HTML5, CSS and Javascript will be used to facilitate interactivity and determine the most suitable presentation of content.

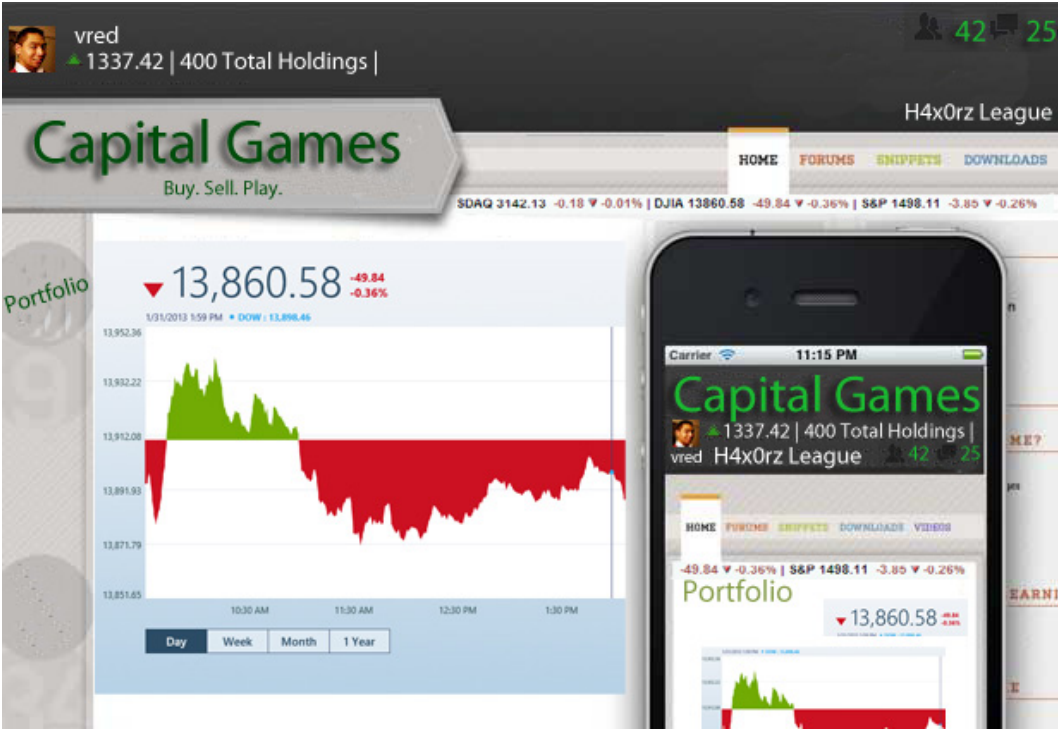


Figure 2.1: This mockup illustrates the concept of Responsive Design with our initial UI mockup. The Responsive Design provides a single unified page which reflows content to match the size of various user devices. Preliminary design elements include interactive graph, scrolling marquee, and persistent navigation bar, corresponding with ST-4, ST-7, and ST-9.

3 Functional Requirements Specification

3.1 Stakeholders

As identified previously, the primary parties interested in this platform would be students and novice investors. However, due to the popularity of related platforms, it is not unrealistic that a future incarnation of this application could be marketed actively towards target groups. For instance, students would be promoted to this service to host various competitions; introductory texts on finance could also place references here. To that end, it behooves us to cater to those primary demographics.

At the same time, gaining a sufficient user base would also open the possibility of discreetly placed advertisements throughout the application. Therefore, we can consider marketing agents to be stakeholders as well, with the caveat that the site will not initially be designed with commercial product placement in mind. Our decision reflects a popular business model for firms today, in which an easily monetizable application does not compromise its rollout with commercials which can easily be implemented later. Yet another reason is the consideration of the various business expenses associated with a commercial rollout – notably the licenses and fees associated with having a commercial (as opposed to free) service.

3.2 Actors and Goals

User

A visitor who has registered and logged in to an account.

- Join/create leagues
- Take part in competitions
- Change personal settings

League Manager

A user who created or controls a league.

- Create a league competition
- Edit league settings

Site Administrator

A user who can control key aspects of the site

- Change global settings
- Create/edit global events
- View statistics of site

Database

The unit that holds all site-relevant data

- Push data back to views about users/events
- Store new data about about users/events

Browser

The middleman between user and system

- Present data to the user
- Retrieve data from the user

Yahoo! Finance

The unit that knows about current financial statistics

- Retrieve data about stocks

Queueing System

A subsystem for scheduling orders so as not to block user interactions.

- Place orders to be executed or canceled asynchronously
- Schedule events and mailings for system

3.3 Use Cases

Preface

From our user stories, we have derived seven use cases to be fully elaborated upon. These use cases do not necessarily fully encompass all of the requirements for our application, but they touch upon the most important functionalities while also covering a breadth of different aspects. Justification for why each use case was included is written along with it. Another thing worth noting is that the browser is an actor in each of the use cases as it is the medium through which the user interacts with the system. As such, it will be included under the term "System" for the fully-dressed use cases to avoid redundancy and superfluous or wordy additions to the flow of each event.

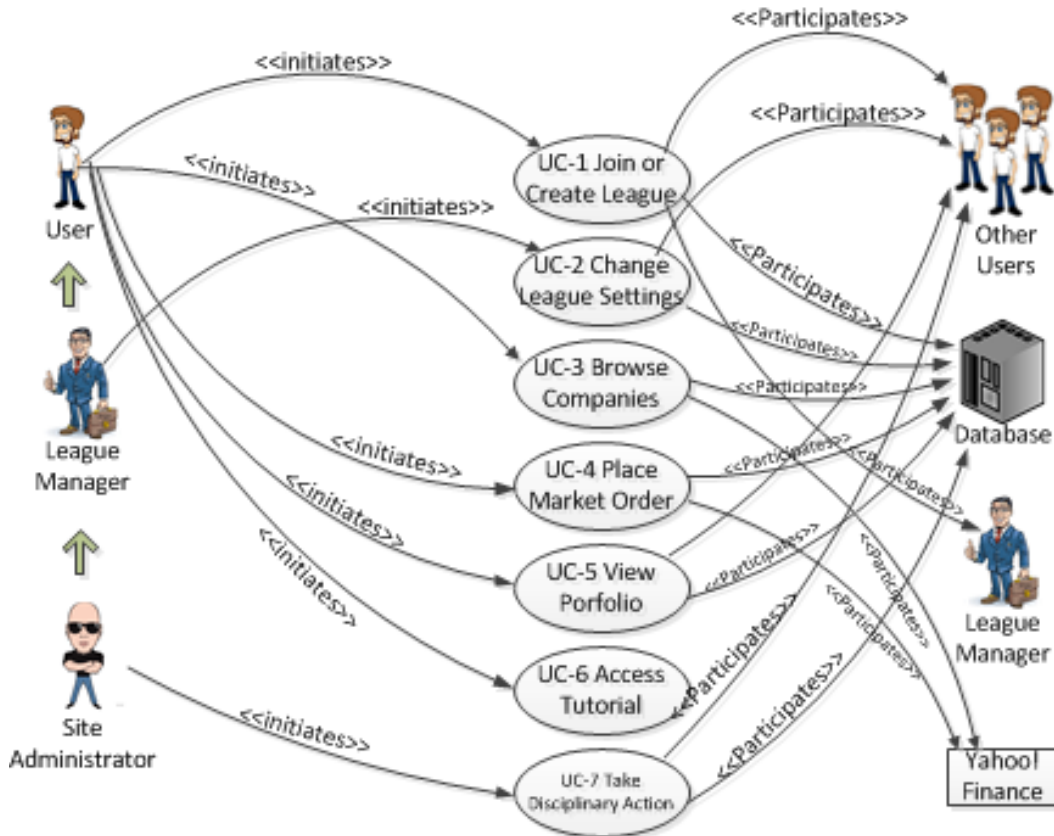


Figure 3.1: This graphic illustrates the relationships between the core actors of our platform.

Fully-Dressed Use Cases

Before a user can participate in most of the functionality of our site, the user must first join or create a league. To the user, creating a league is very similar to joining a league, the notable differences being that the user becomes League Manager of a league that they create and then must also invite users to join said league. Therefore, we detail joining/creation as a single use case. User invitation, as a responsibility of the League Manager, will be explored in a later use case. One relevant aspect of the responsibility of a League Manager to the use case though, is whether a league is made public or private; that is, whether it shows up in a public league listing page or can only be joined by direct invitation from the League Manager. Thus, our first use case involves a business policy:

CG-BP01: So that a user may create a join leagues with only their friends, leagues marked as private will not show up on the league listings unless a user is a current or pending member.

Thus, a user will only be able to browse listings of public leagues or private leagues to which they have access.

Use Case UC-1 Join or Create League

Related Requirements:	ST-2, ST-21
Initiating Actor:	User
Actor’s Goal:	To join or create a fantasy finance league
Participating Actors:	Database, Other Users, League Managers
Preconditions:	-If joining a league, either a public league exists and has open positions or player has been invited to a private league -User is logged in
Postconditions:	-The Database is updated to reflect the creation of or addition to the league
<i>Flow of Events for Main Success Scenario:</i>	
→	1. User navigates to leagues listing page
←	2. System displays public and pending private leagues available for the User , sorted by date created
→	3. User selects join on a league in which they are interesting in joining and to which they have access
←	4. System authorizes user and registers User into that league, notifying Database to update to reflect this change
<i>Flow of Events for Extensions (Alternate Scenarios):</i>	
3a. The user selects create league rather than join league	
→	1. User inputs desired league name and settings
←	2. System (a) creates the league and inputs it to the Database and (b) registers the User into that league as League Manager
4a. The user attempts to join or create a league without permission	
←	1. System rejects request and delivers reason for rejection

It is important here to note another business policy of our site relevant to the user’s experience:

CG-BP02: A user is able to join an unlimited number of leagues and become League Manager for as many leagues as the user wishes to create.

Though the settings are selected when creating the league, any League Manager can change certain settings of their league at any time. These settings are comprehensive, including such items as name, privacy, number of spots, and duration. In addition, the League Manager can also manage members from the settings. However, certain settings cannot be changed after the league enters active competition, such as starting capital, commission rate, and margin, because changing primary competition rules mid-game would be unfair.

Use Case UC-2 Change League Settings

Related Requirements:	ST-22, ST-23
Initiating Actor:	League Manager
Actor's Goal:	To change the settings of a league and manage its members
Participating Actors:	Database, other Users
Preconditions:	-User is the League Manager of the league -User is logged in
Postconditions:	-The league settings have been changed as desired and the Database reflects the changes
<i>Flow of Events for Main Success Scenario:</i>	
→	1. League Manager selects the league settings option from the league page
←	2. System requests the current settings from the Database and presents them to the League Manager along with options to change select settings
→	3. League Manager updates the settings, such as privacy, league name, number of spots, and managing users
←	4. System sends the updated settings to the Database
<i>Flow of Events for Extensions (Alternate Scenarios):</i>	
1a. The User selecting league settings is not the League Manager	
←	1. System requests the current settings from the Database and displays them, but does not provide ways to change them
4a. The League Manager has altered the status of a league member	
←	1. System will request the Database to update the User's status in the league, be it becoming league manager or removing that User's instance from this league (banned)

It is of some concern that League Managers may become abusive of their powers, and therefore it is important to create on a policy to explicitly state how this power is treated. In modern fantasy leagues (such as football, baseball, etc.), the League Manager does not typically have their power moderated, and this has not caused any problems in the success of these fantasy websites. The ability to leave a league and join another is left to the users if they feel that their league manager has become abusive. Their joining of the league acts as an implicit contract to accept of the League Manager's settings. However, if this League Manager becomes a problem and users bring it to an administrator's attention, disciplinary action may be taken. Thus we generate the next site policy:

CG-BP03: A League Manager is able to change the status of users in their league without moderation. However, if a League Manager is deemed abusive, a site administrator may take disciplinary action against them.

Core to our site is the ability of the user to have access to information about companies so that the user may make informed decisions on how he would like to invest. As this is so crucial to the functionality of this project, it is absolutely necessary to make information easily available

to the user and presented in a way that is clear and easy to understand. Therefore, the search of companies as mentioned in ST-3 should be simple to use and intuitive and the display of company profiles as mentioned in ST-4 should be such that a user can easily access any desired information about the company's financial performance.

Use Case UC-3 Browse Companies	
Related Requirements:	ST-3, ST-4
Initiating Actor:	User
Actor's Goal:	To bring up information on a desired company
Participating Actors:	Database, Yahoo! Finance
Preconditions:	-Yahoo! Finance is accepting inquiries -User is logged in
Postconditions:	-None worth mentioning
Flow of Events for Main Success Scenario:	
→	1. User begins entering a search term
←	2. System makes suggestions for companies in real-time
→	3. User enters search term or selects a suggestion
←	4. System (a) requests information from Yahoo! Finance and (b) displays the information to the user in a clear and interactive manner
Flow of Events for Extensions (Alternate Scenarios):	
1a. The User selects a direct link to a company rather than enter a search term	
←	1. Same as step 4 above
3a. The search term is invalid, i.e. the company does not exist	
←	1. System informs user company does not exist and offers similarly titled companies as links

Note that the exact way in which the information requested from the Yahoo! Finance is displayed to the user is not specified in this use case. This will be described instead in later sections about on-screen appearance requirements as to try to separate the functionality of the site and design of the site as separate as possible.

The goal of browsing companies ultimately is for the user to gain the knowledge needed to place market orders. Market orders are the atomic action of our site; i.e. the center point of every league is the user's ability to initiate transactions in an attempt to invest their money as best they can.

Use Case UC-4 Place Market Order	
Related Requirements:	ST-5

Initiating Actor:	User
Actor's Goal:	To place a market, stop, or limit order
Participating Actors:	Database, Yahoo! Finance
Preconditions:	-Yahoo! Finance is accepting inquiries -User is logged in -User is a member of a league
Postconditions:	- User's portfolio is updated to reflect change in position
Flow of Events for Main Success Scenario:	
→	1. User selects the league in which they would like to place the order
←	2. System displays prompt for market order, including type, amount, and company
→	3. User fills out form and requests the order be placed
←	4. System (a) requests market price from Yahoo! Finance and (b) places the order into the Database
←	5. The order either resolves or expires, and the System updates the User's position in the Database accordingly
Flow of Events for Extensions (Alternate Scenarios):	
1a. The User chooses to place a market order from a company's profile rather than from the league page	
→	1. The User selects which league in which to place the order
←	2. The System takes the User to league marker order prompt as described in Step 2 above, with the prompt for company already filled out
→	3. Go to Step 3 above
4a. The User does not have enough money or margin to place the order	
←	1. System informs the User that they do not have enough money or margin to place the order and returns them to the market order prompt

The potential kinds of orders referenced in the above use case are defined in the glossary. The details on the necessary computations to enact these orders will be defined in a section later on.

In order to keep track of their own finances and any of their fellow league member's finances, a user must be able to view member portfolios. This keeps with the competitive nature of our site in addition to allowing the user to track their own progress.

Use Case UC-5 View Portfolio	
Related Requirements:	ST-6, ST-7, ST-11
Initiating Actor:	User
Actor's Goal:	To view one's own finances or another's finances

Participating Actors:	Database, other Users
Preconditions:	-User is a member of a league -User is logged in -Database is tracking user's position
Postconditions:	-None worth mentioning
Flow of Events for Main Success Scenario:	
→	1. User selects a league member's profile
←	2. System requests that member's information from the Database and displays it in an organized and graphical manner to the User
Flow of Events for Extensions (Alternate Scenarios):	
2a. User is viewing their own portfolio	
←	1. System gives the User options to place market orders related to their existing positions

Once again, the exact display of information is not defined in the use case, but rather will be explored further in the section about user interface specifications. Next to discuss is the tutorial as referenced in ST-8. We consider this to be one of the main aspects that separates us from previous iterations of fantasy stock leagues; our site will educate users new to finance and enable them to learn all about the world of finance and how to invest, in addition to how to these subjects relate to the use of our site.

Use Case UC-6 Access Tutorials	
Related Requirements:	ST-8
Initiating Actor:	User
Actor's Goal:	To become educated in finance
Participating Actors:	None
Preconditions:	-User is logged in
Postconditions:	-None worth mentioning
Flow of Events for Main Success Scenario:	
→	1. User selects the tutorial option from the site's main page
←	2. System displays possible topics on which the User may be educated on
→	3. User selects topic
←	4. System presents an interactive tutorial to the User , which will be further elaborated upon in a later section

In order to maintain a clean fantasy finance experience for our regular users, site administrators will reserve the ability to moderate other users—issuing warnings, suspensions, or even bans

for abusive activity. To put it explicitly:

CG-BP04: Site administrators will warn, suspend, or ban users for abusive activity—this includes aggressive behavior on league comments or user messages, spamming users, joining numerous leagues without active participation, and anything else that is deemed to harm the experience for other users.

Use Case UC-7 Take Disciplinary Action	
Related Requirements:	ST-27
Initiating Actor:	Site Administrator
Actor's Goal:	To take action against an abusive User
Participating Actors:	Database, Users
Preconditions:	-Initiating actor is a Site Administrator -There are outstanding abuse reports
Postconditions:	-The Database is updated to reflect any actions taken against the User The abuse report shows that it has been resolved on the administration page
Flow of Events for Main Success Scenario:	
→	1. Site Administrator selects the site administration page option from the main screen (only viewable by Site Administrators)
←	2. System makes a request to the Database and displays all outstanding abuse reports
→	3. Site Administrator (a) selects an abuse report, (b) reviews the report, and (c) selects what action is to be taken (if any)
←	4. System implements the action selected by the Site Administrator and updates the Database accordingly
←	5. System notifies the offending User of any actions taken against them

3.4 System Sequence Diagrams

In the following sequence diagrams, we describe exactly the interactions between the key actors our system. It is important to note that most of the interaction between the user (or league manager or site administrator) and system is facilitated by the browser. The user, through filling forms and button clicks, instructs the browser which requests to make to the system. In turn, the system communicates with the database to request the desired data, takes any required actions, and delivers the data to the browser for presentation to the user.

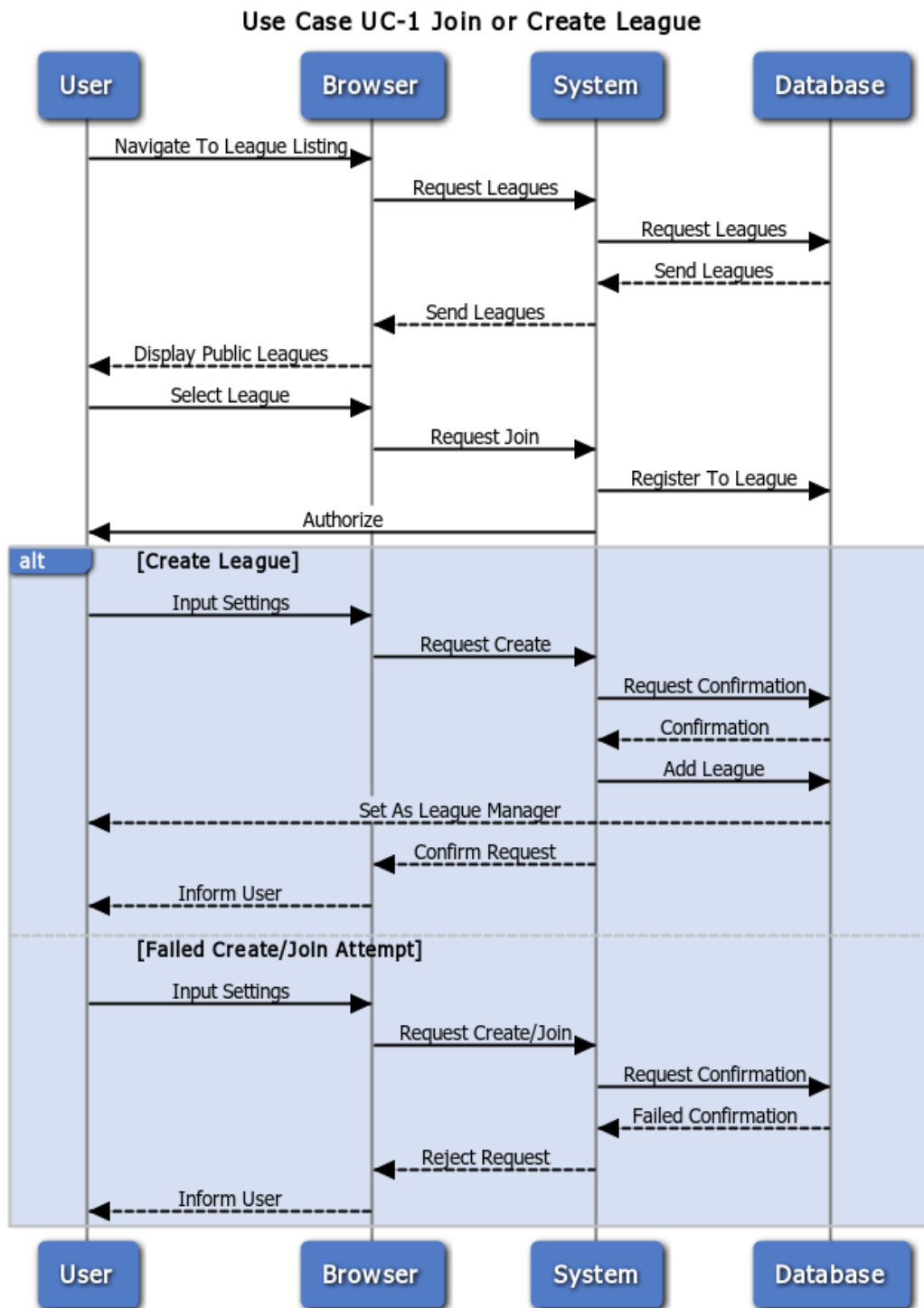


Figure 3.2: See UC-1 on page 15. When the user navigates to the league listing page, they invoke this use case. The user initiates a request to view all the public leagues and the system retrieves them from the database. Then, they are presented to the user who is given the option to join any valid league.

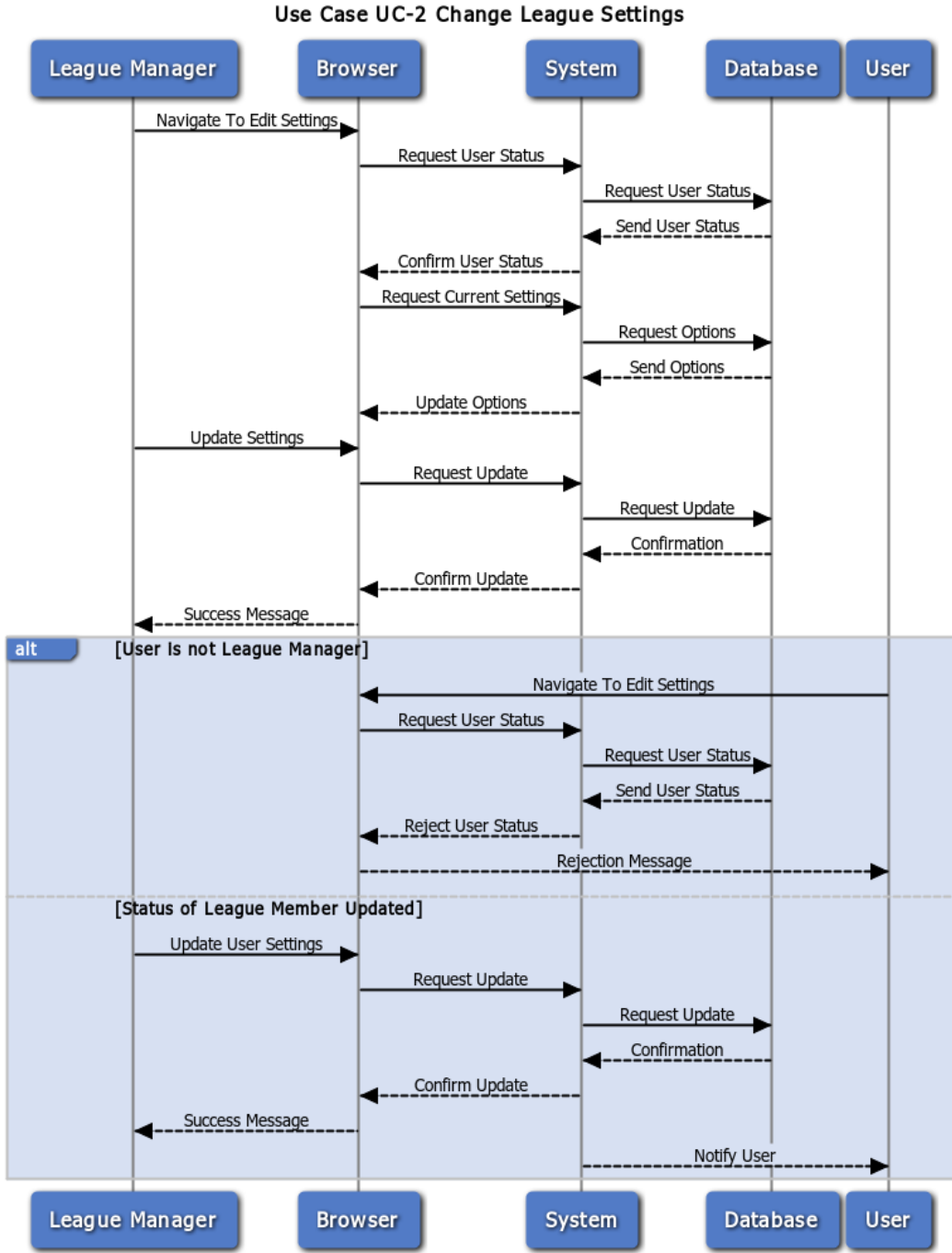


Figure 3.3: See UC-2 on page 16. This is sequence of events that occur when a league manager alters the league settings. The system fetches the current settings from the database and returns them to the browser. It also ensures that the user attempting this change is a league manager. Then, the user can initiate a request to change the settings which will be enacted out by the system. If the league manager changes the status of a user within the league, the system notifies that user.

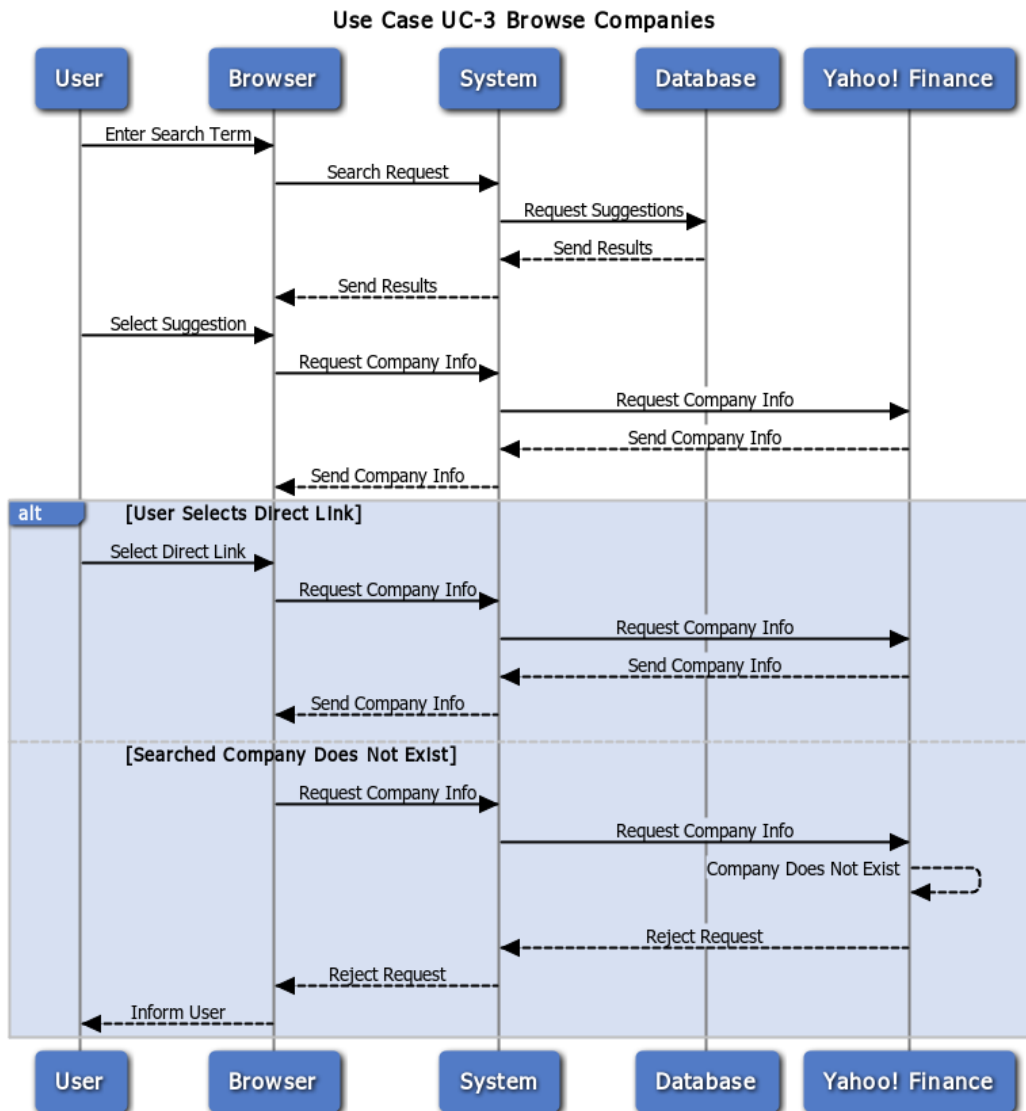


Figure 3.4: See UC-3 on page 18. When the user desires to research companies, this is the sequence that follows. The user is able to search and browse for companies. They can also get to a company’s page through a direct link. Yahoo! Finance responds to requests and delivers data to our system which is then transferred to the browser and fills out a company profile.

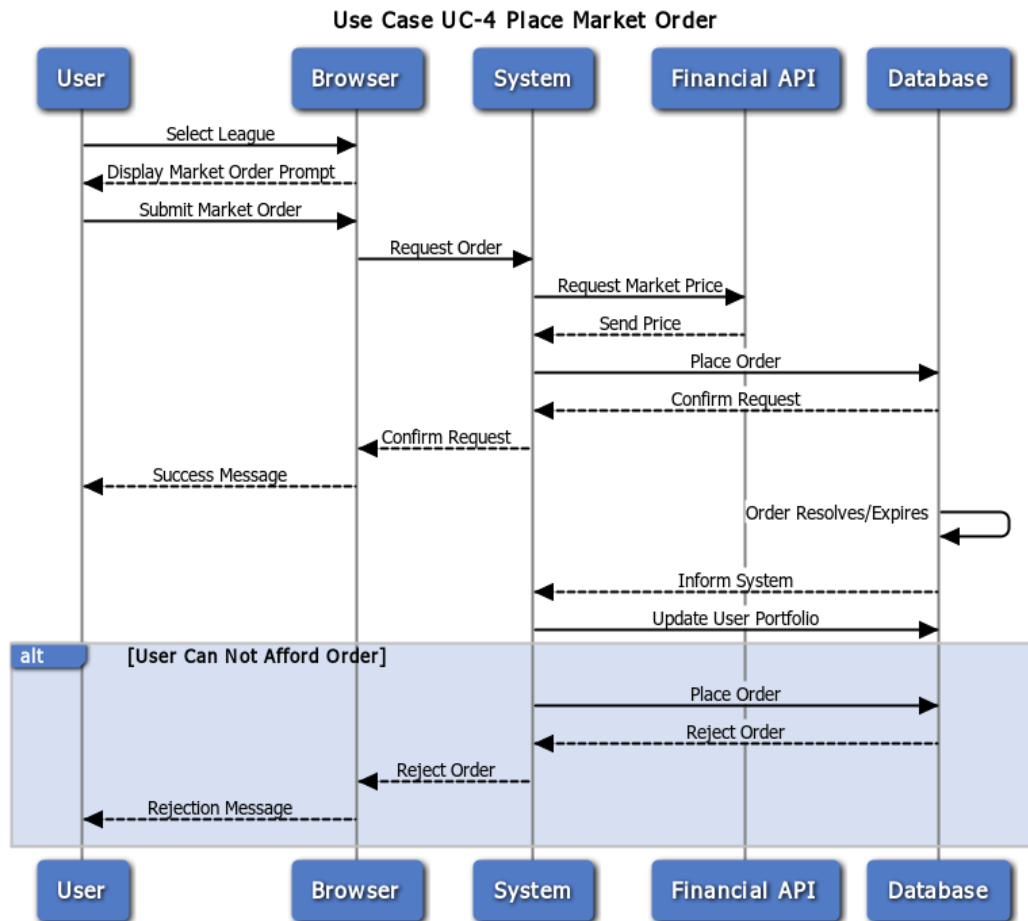


Figure 3.5: See for UC-4 on page 18. This sequence encompasses the bread and butter of our application—market orders. The user selects a league in which to place an order, fills out a prompt in the browser which then submits the request to the system. The system inserts the order into the database and enqueues it (the queuing system will be elaborated upon in a later section). Once the order resolves or expires, the database notifies the system and the user’s portfolio is updated accordingly.

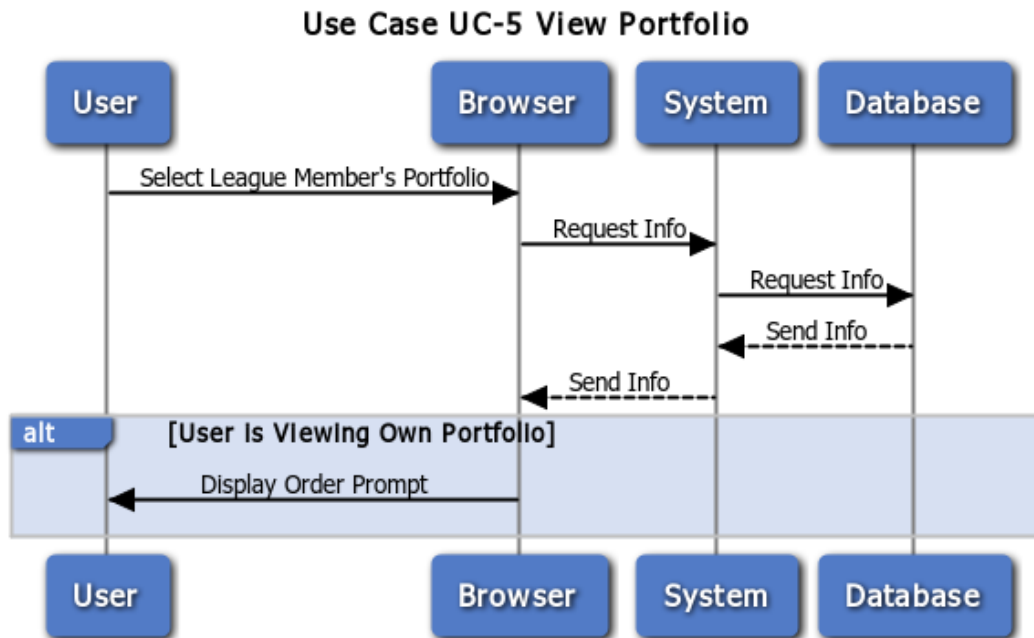


Figure 3.6: See UC-5 on page 19. This use case is relatively straightforward. The user browses to a league member’s portfolio, and the browser submits a request to the database for that portfolio’s information.

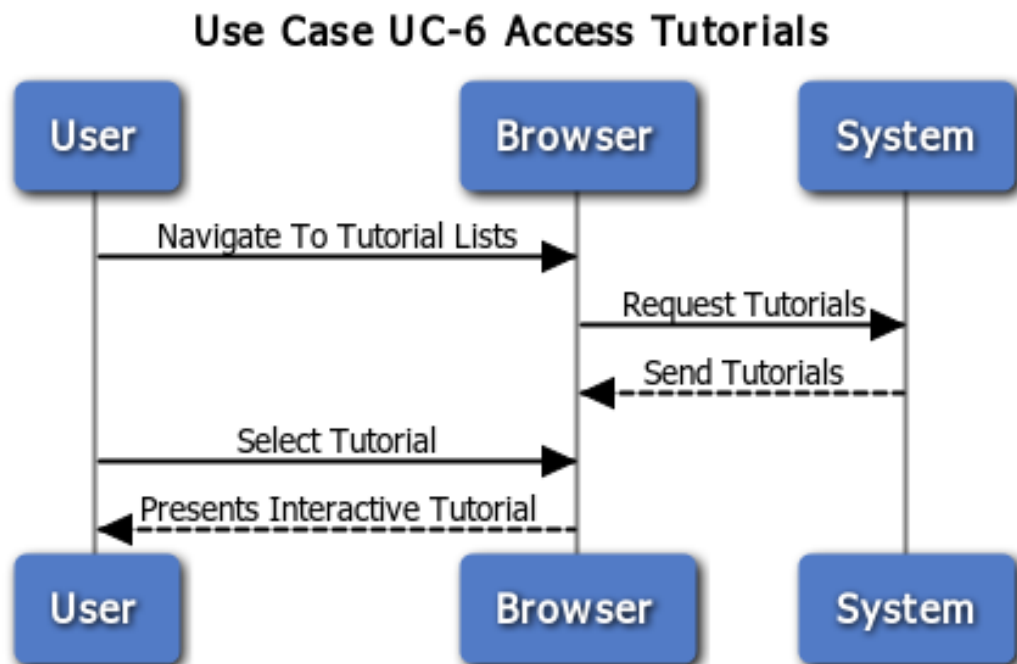


Figure 3.7: See UC-6 on page 20. Another simple use case. The user simply navigates to the tutorial page, which is populated by the system.

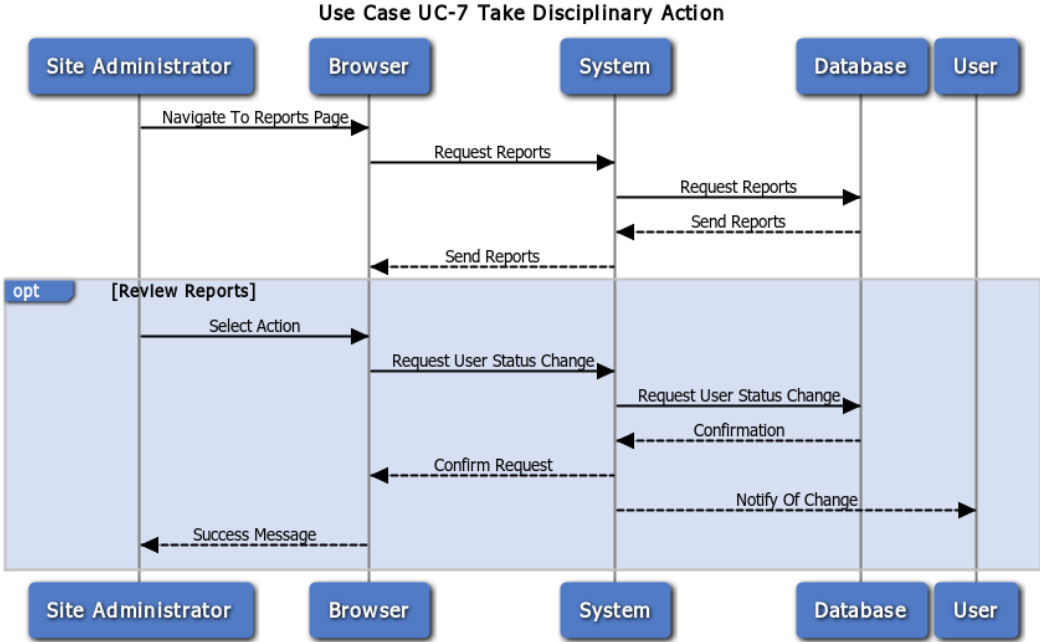


Figure 3.8: See UC-7 on page 21. When a site administrator navigates to observe any outstanding abuse reports, this flow of events is initiated. The database delivers all outstanding reports to the system which then populates them in the browser. If the administrator decides to take an action, the status change is reflected in the database, and the system notifies the user of whatever action was taken against them.

4 User Interface Specification

4.1 Preliminary Design

The user interface of CapitalGames emphasizes easy to understand graphical representations of financial metrics pertaining to various aspects of trading and the economy in general. In addition, color harmony and adequate space distribution is a priority to provide a pleasant user experience. A UI design built on top of the responsive Bootstrap UI framework was chosen due to its extensive support for many UI components needed in the application. The center of the CapitalGames experience is the dashboard where a user can quickly see an overview of his/her performance in all leagues, join new leagues and learn more about finance. Each primary view is presented below with particular attention having been put into a consistent and uniform user experience. Each view is annotated with applicable use cases so that a sequence of views can be determined for each use case.

4.2 User Effort Estimation

Capital Games will utilize a streamlined user interface that has become rampant in modern web design. Essentially, all user interaction regarding login/sign-up and even actual interactions with their fantasy leagues and league portfolios can all be done within at most ten clicks and 50 keystrokes for data entry, and most of these interactions are in the registration process.

1. Login/Registration: 2 mouse clicks and 50 keystrokes
 - a) Click Login/Register on the corner of the header.
 - b) Data entry (20 keystrokes for username and password). (In addition, for registration, 10 to confirm password, 15 for e-mail address, and 5 for CAPTCHA for spambot control over the login/registration interfaces) In addition, all these keystrokes can be simplified

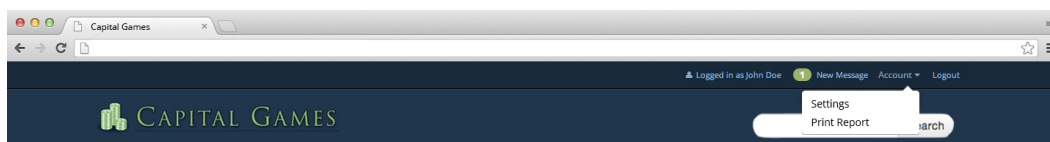


Figure 4.1: The header was designed to provide a persistent area for search and account management. It also establishes the branding of our application which is important for future recognition. It features the search field with auto-complete functionality.

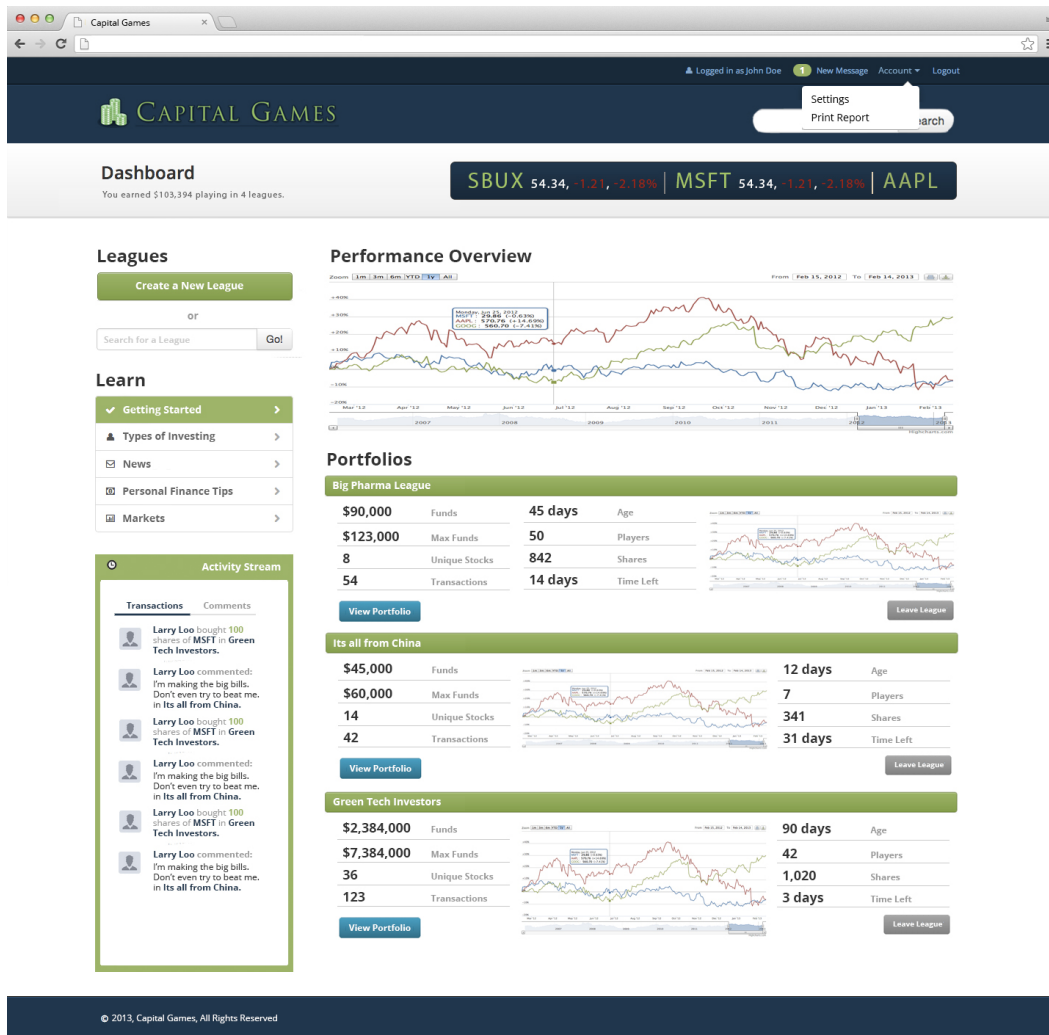


Figure 4.2: The dashboard was designed to conveniently notify the user of his/her performance in all leagues. This is done with the prominent graphic at the top of the content body as well as with all the brief views of active portfolios.

in one click with Facebook integration, in which the site will pull their Facebook login data and use it for Capital Games.

2. League Portfolio Interaction: Buy/Sell, 3 clicks and 10 keystrokes
 - a) Click Portfolio tab in navigation menu header.
 - b) Hover over any company listing on the page with your cursor and click Buy/Sell.
 - c) Input amount of shares you want to buy/sell.
 - d) Click to confirm.
3. Create Fantasy League: 10 clicks and 10 keystrokes
 - a) Click League tab in navigation menu header.
 - b) Click Create New League in submenu.

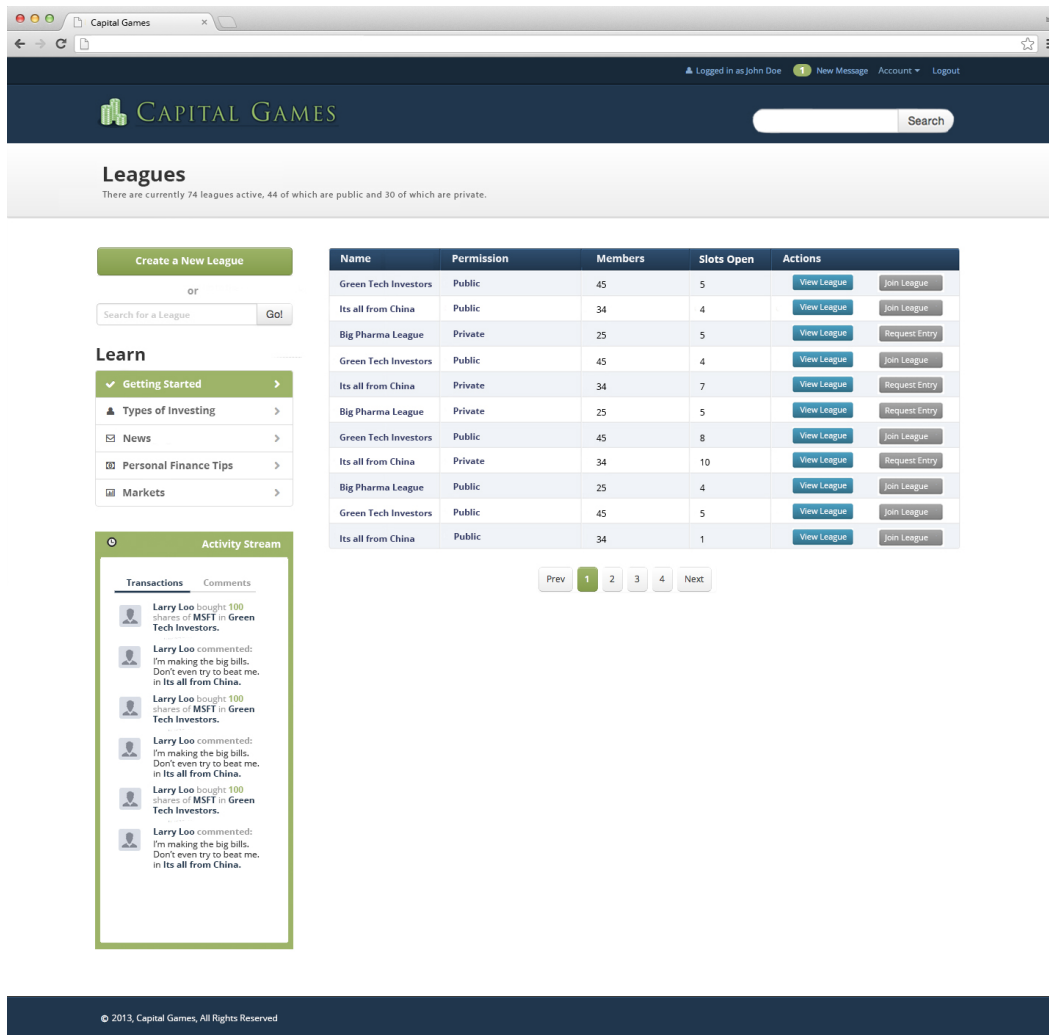


Figure 4.3: The leagues view was designed to cleanly display all leagues which a user might be interested in. It is used both when browsing and when searching for a particular league.

- c) Enter League Name and Click Checkboxes for desired rules.
 - d) Click to confirm.
4. Company: 3 clicks and 4 keystrokes
 - a) Press the Trade button
 - b) Select Buy/Sell
 - c) Type in number of stocks
 - d) Click to accept
 5. League Users: One click
 - a) To view all members of the league, click See All Members
 - b) As an Administrator or League Manager, deleting comments is one click on the delete button next to a comment

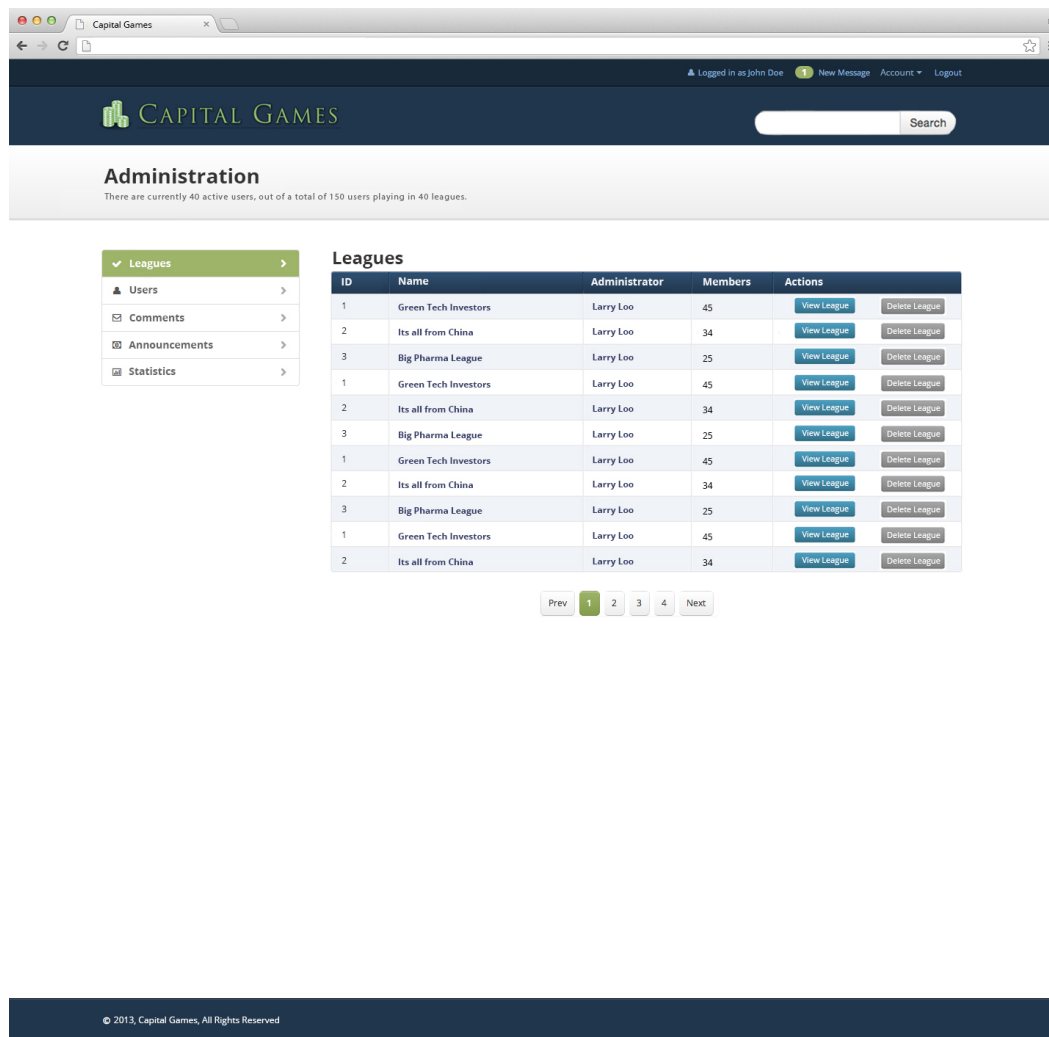


Figure 4.4: The administration area is a multi view component enabling administrators to manage all aspects of the system from user management to viewing overall system statistics.

6. League Manager: Four clicks
 - a) Click League Settings
 - b) Click Users tab
 - c) Edit anything
 - d) Click Save Changes
7. Messages: 2 clicks plus message
 - a) Click friend from drop down list
 - b) Enter message
 - c) Click send

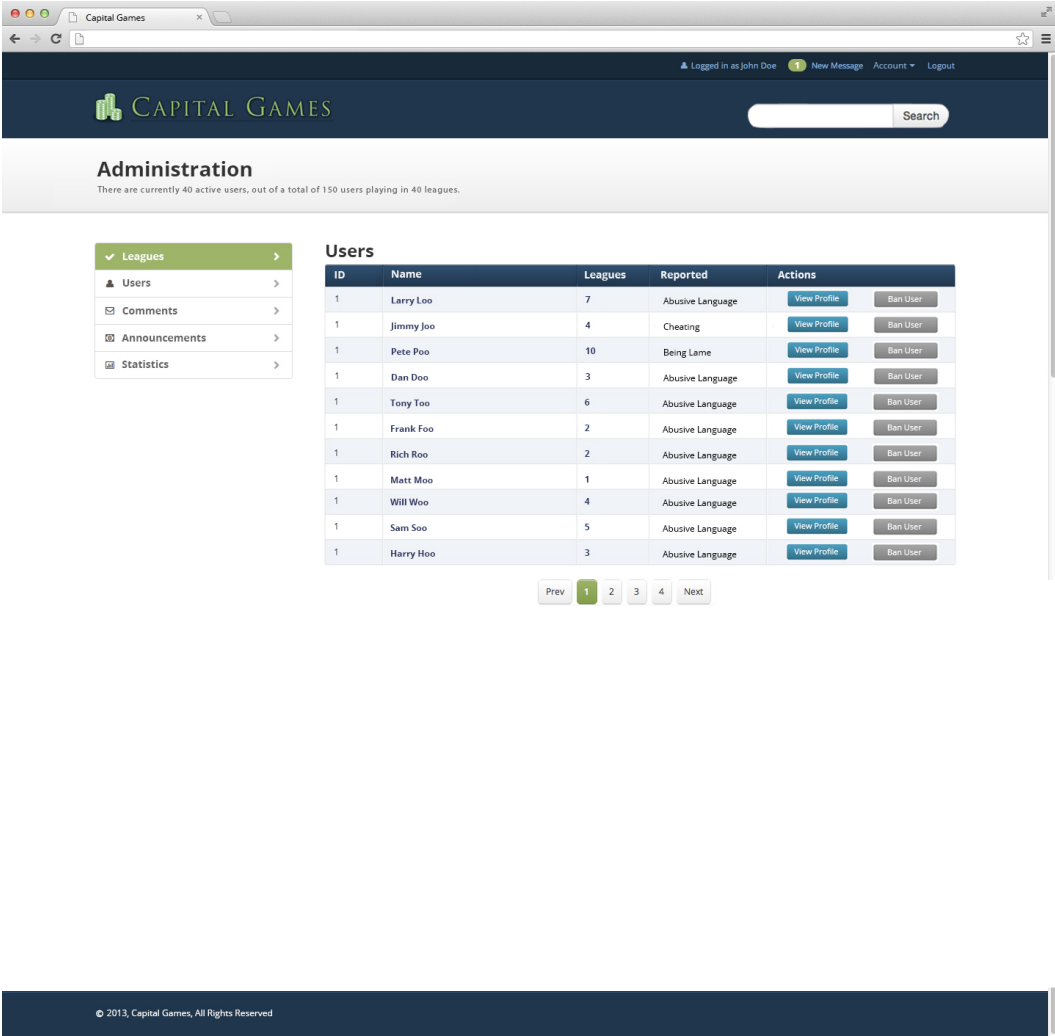


Figure 4.5: This portion of the administration area contains a top-down view of comments posted by users in various locations, and the ability to ban them with a single click.

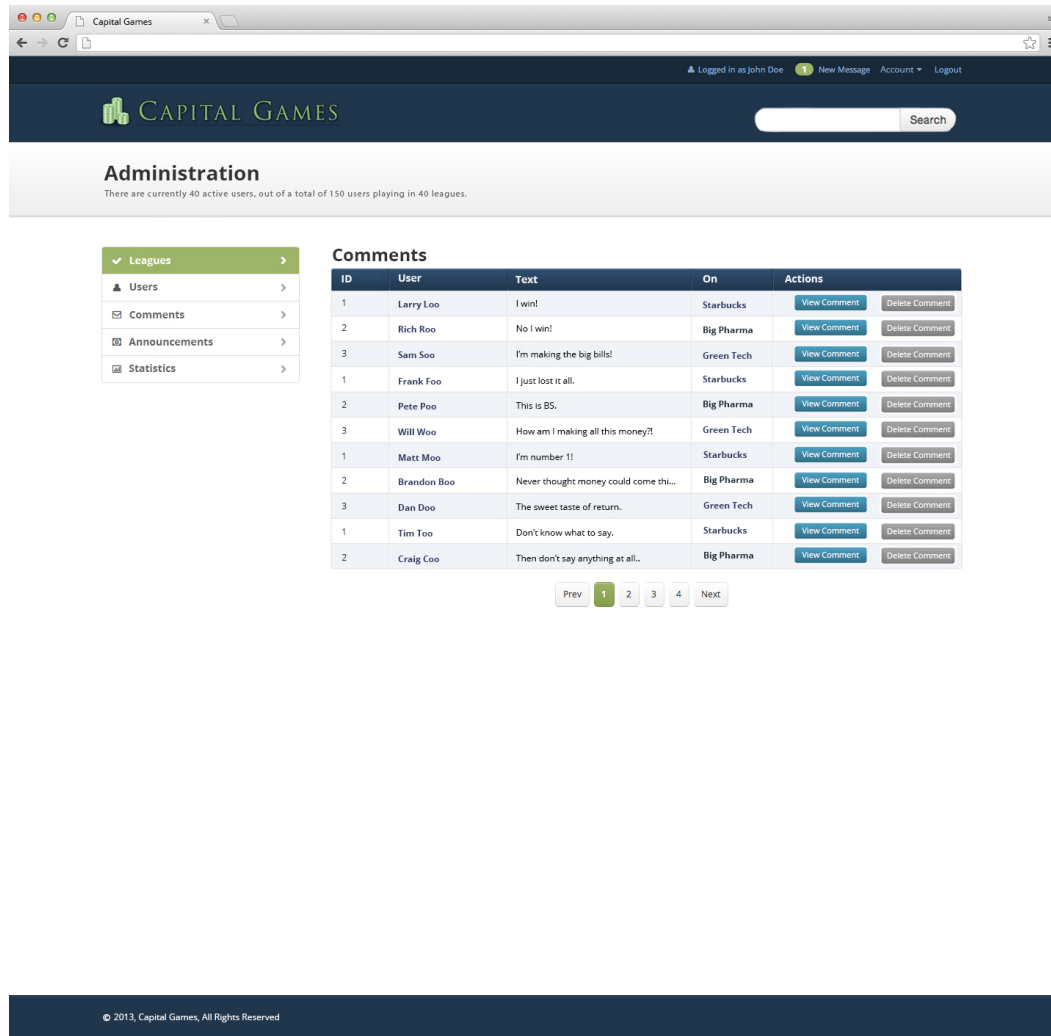


Figure 4.6: This portion of the administration area contains a top-down view of comments posted by users in various locations, and the ability to delete them with a single click.

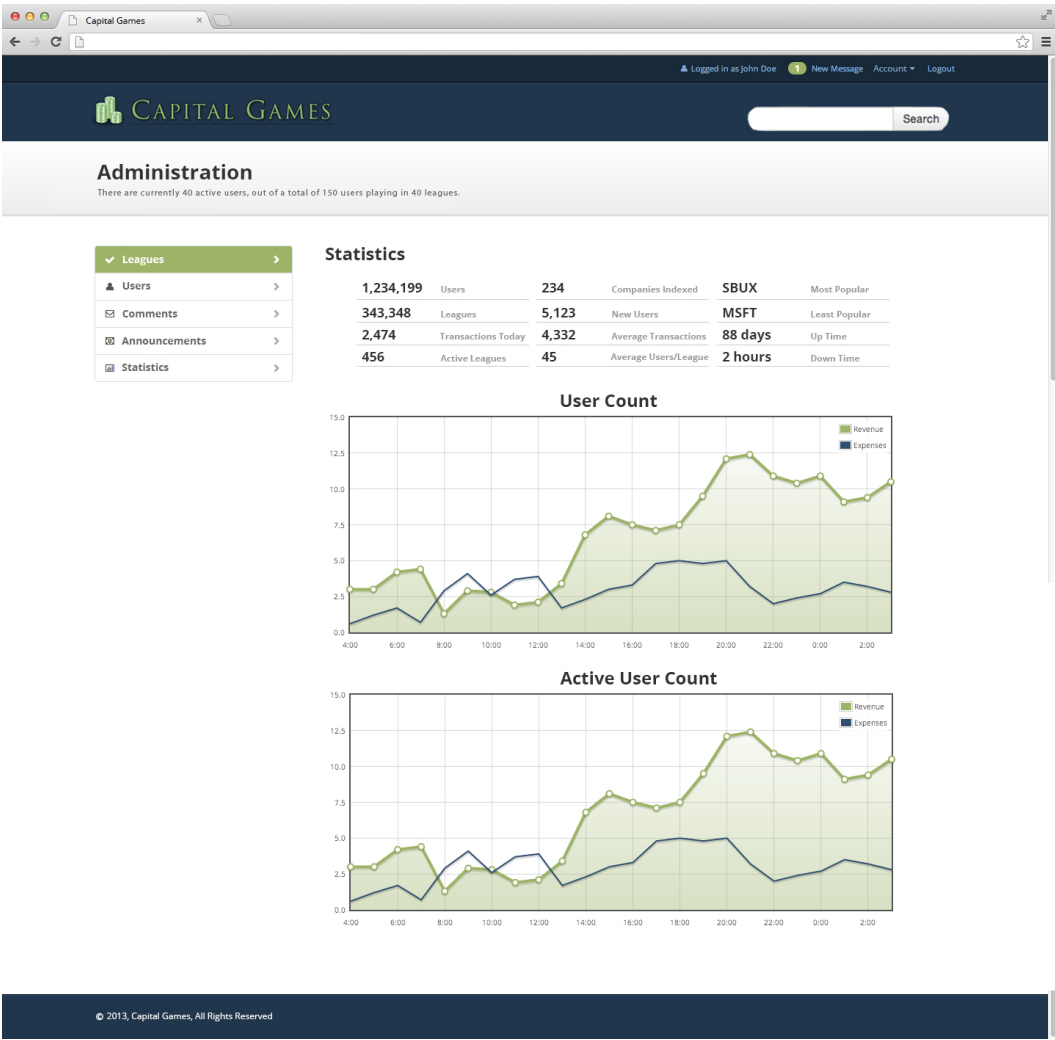


Figure 4.7: This section of the administration area contains statistics of interest.



Figure 4.8: From the Company page, you are able to view details statistics about certain companies after being linked to it or searching for it. Major details, such as the quote, are up at the top while further details are at the bottom. A user can comment on the company on the bottom right. If you decide that you want to trade, you simply press the trade button and a box will pop-up, giving you options for the trade.

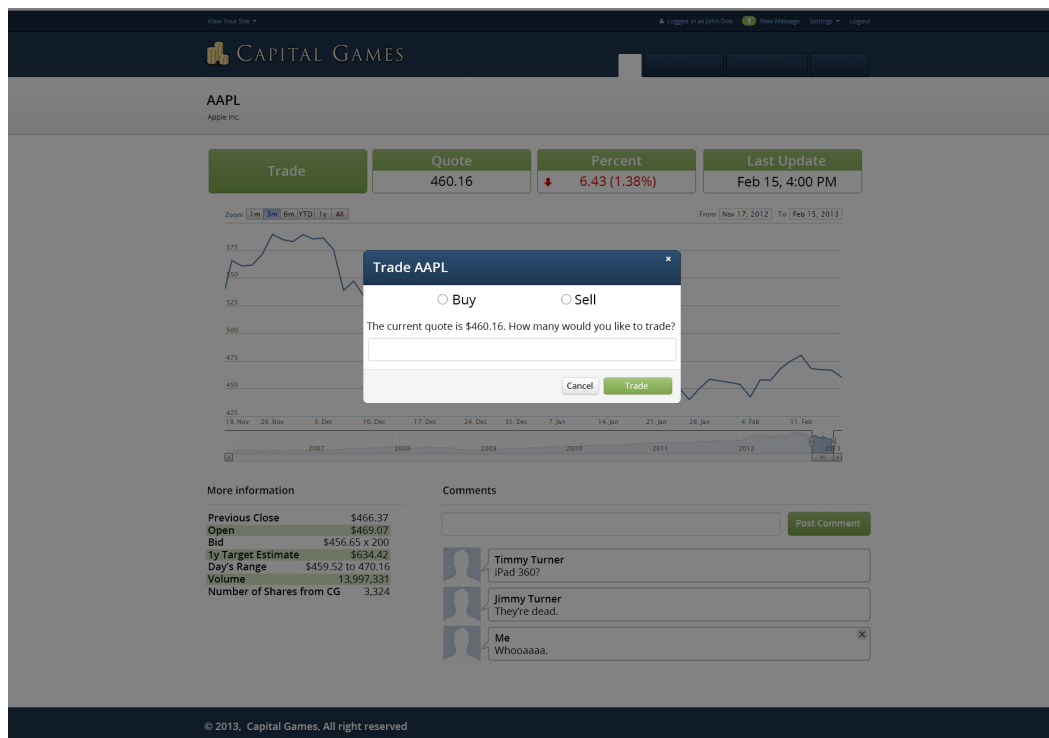




Figure 4.9: From this page, you are able to view a certain league. Up at the top are the main facts about the league as well as a button to join/quit the league and the icon for it. In the middle, there is a ranking system to show the users in the highest standing. Down at the bottom, you can see the activity and also comment on the league itself.

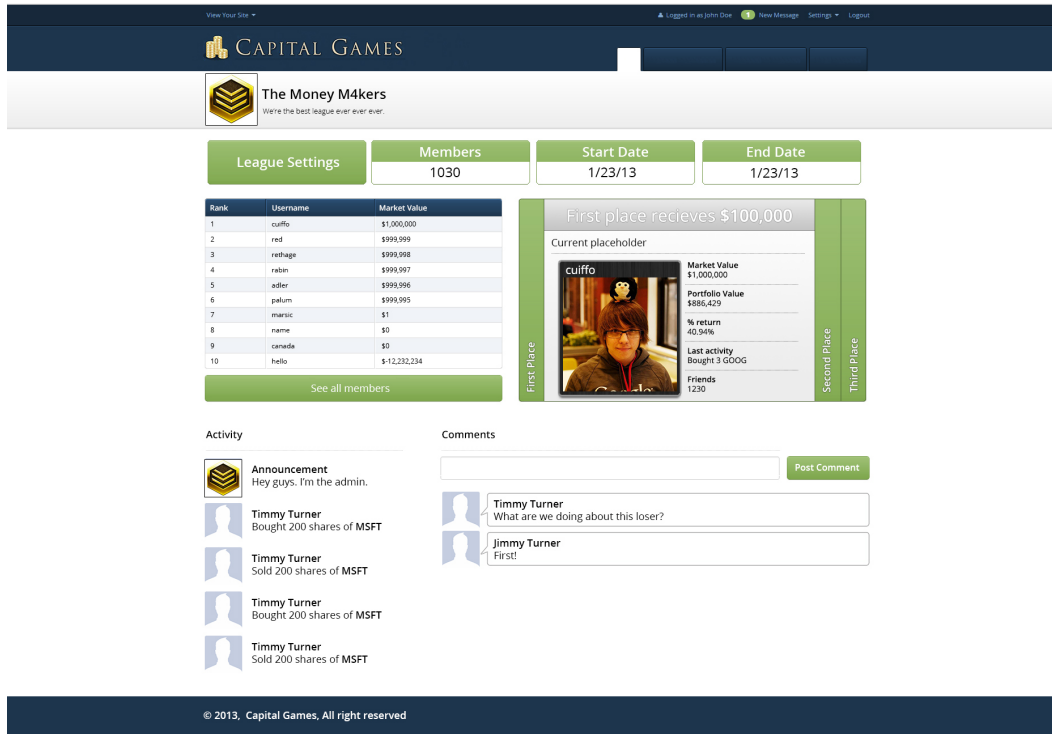
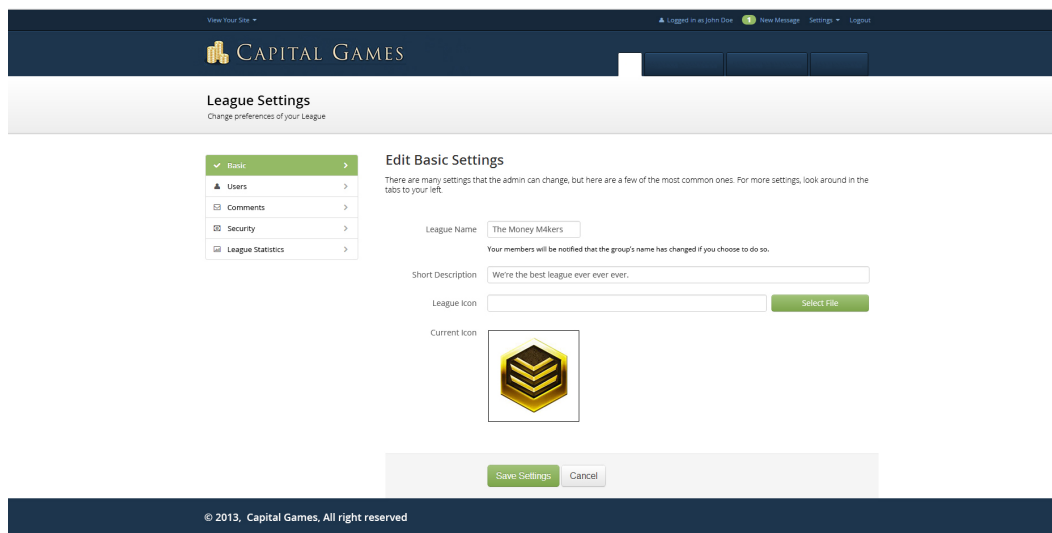


Figure 4.10: A league admin will see the join/quit button on a league as the settings page for them. When they click on that, they are brought to a page that gives them many settings they can change for the league, the most typical being the name, description and icon.



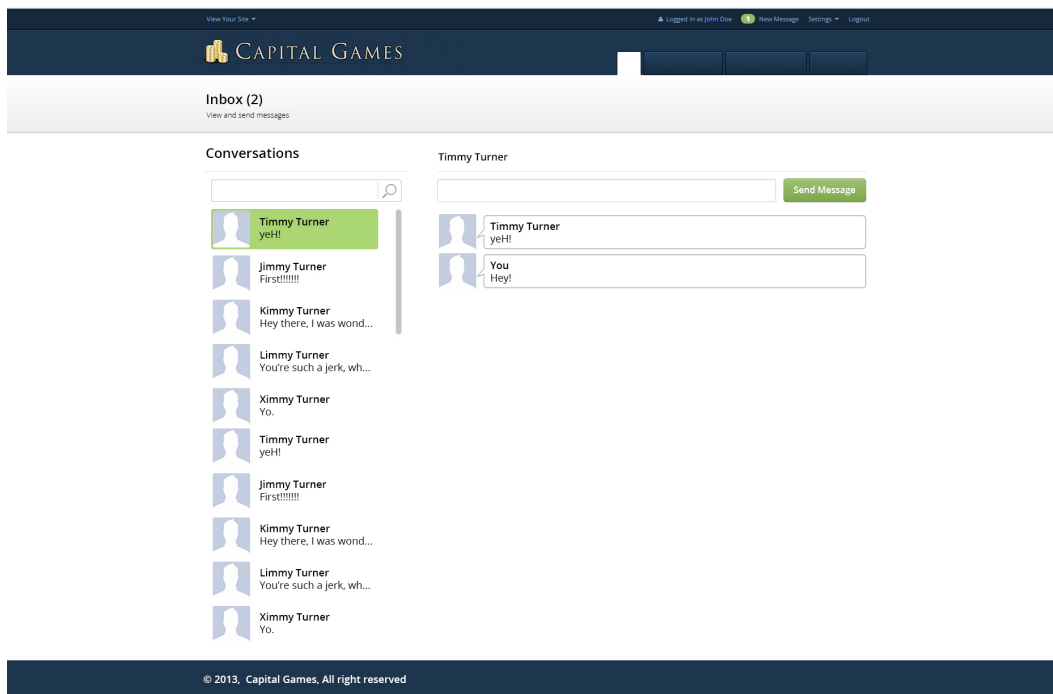


Figure 4.11: A league admin will see the join/quit button on a league as the settings page for them. When they click on that, they are brought to a page that gives them many settings they can change for the league, the most typical being the name, description and icon.

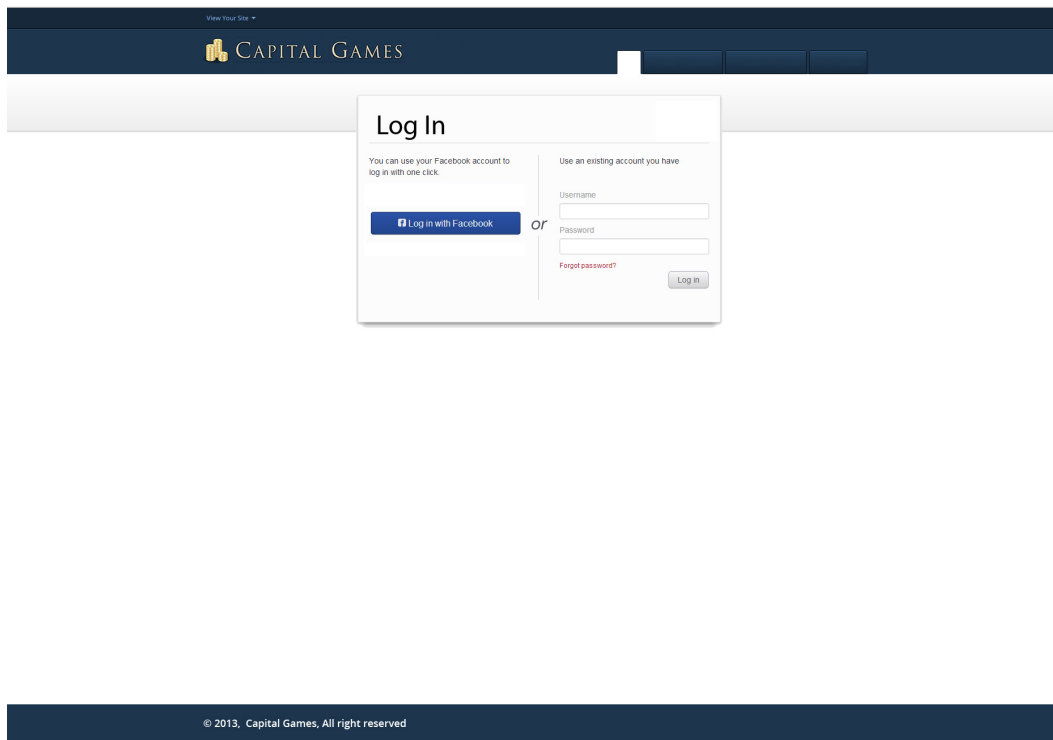
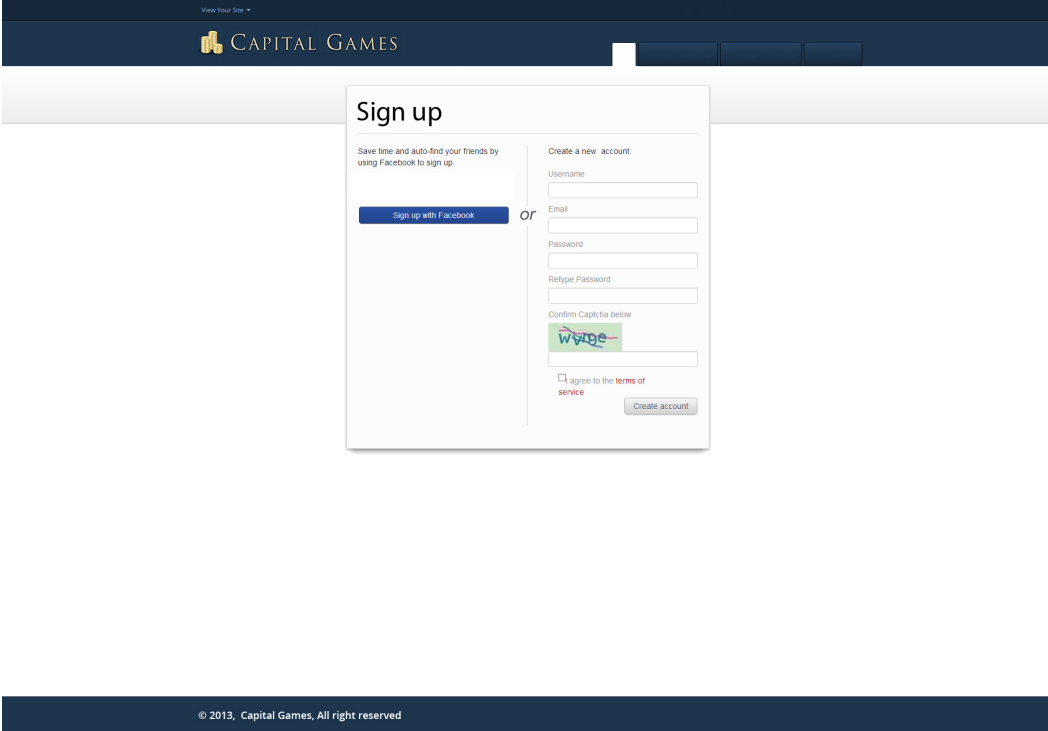


Figure 4.12: Users would be able to login simply by clicking a login button on the top-right hand corner of the screen, which would take them to a prompt in which they can enter their username and password. This would only require one click and about 20 keystrokes from any page of the website. Users logged in to facebook may also take advantage of Facebook integration and instantly log in with 1 click.



The image shows a web browser window with a dark blue header. The header contains the text "View Your Site" and the "CAPITAL GAMES" logo. A white "Sign up" modal form is centered on the page. The form is divided into two sections: "Sign up with Facebook" and "Create a new account". The "Sign up with Facebook" section includes a blue button labeled "Sign up with Facebook". The "Create a new account" section includes input fields for "Username", "Email", "Password", and "Retype Password". Below these fields is a "Confirm Captcha below" section with a green and blue captcha image showing the word "wage". At the bottom of the form, there is a checkbox labeled "I agree to the terms of service" and a "Create account" button. The footer of the browser window contains the text "© 2013, Capital Games, All right reserved".

Figure 4.13: Users who are not logged in will also have the “Sign Up” button available to them in the header that will enable them to register for Capital Games. This can be accomplished within 1 click and 50 keystrokes. A user logged in to facebook may also instantly register within 1 click.

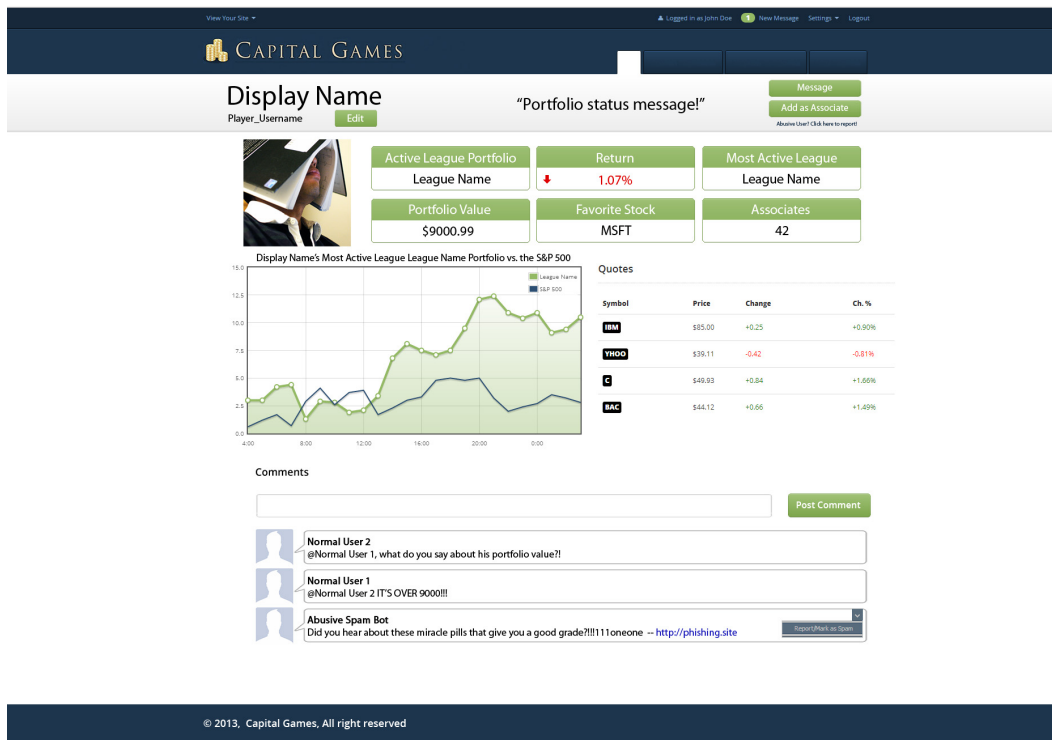


Figure 4.14: Users may access their Portfolio by clicking a menu tab in the top header of the website. This view enables them to conveniently see a summary of their return, active league, portfolio value, stock, and other data pertaining to their stock. They would be able to edit it in one click via the edit button.

View Your Site

CAPITAL GAMES

Player_Username
Profile info

Display Name

Firstname

Lastname

Birthdate

Gender

City

State

Zip

Description

Paragraph B I U S Quote Link Video Paste Less

☒ Spoiler ☒ Spoiler Block ☒ List Remove Format Clean

I am an undergraduate Computer Engineering student.

image

© 2013, Capital Games, All right reserved

Figure 4.15: Upon clicking the “Edit” button on their portfolio page, users will also be able to manage profile items such as their display name, e-mail address, and other optional information they may choose to disclose, such as their name.

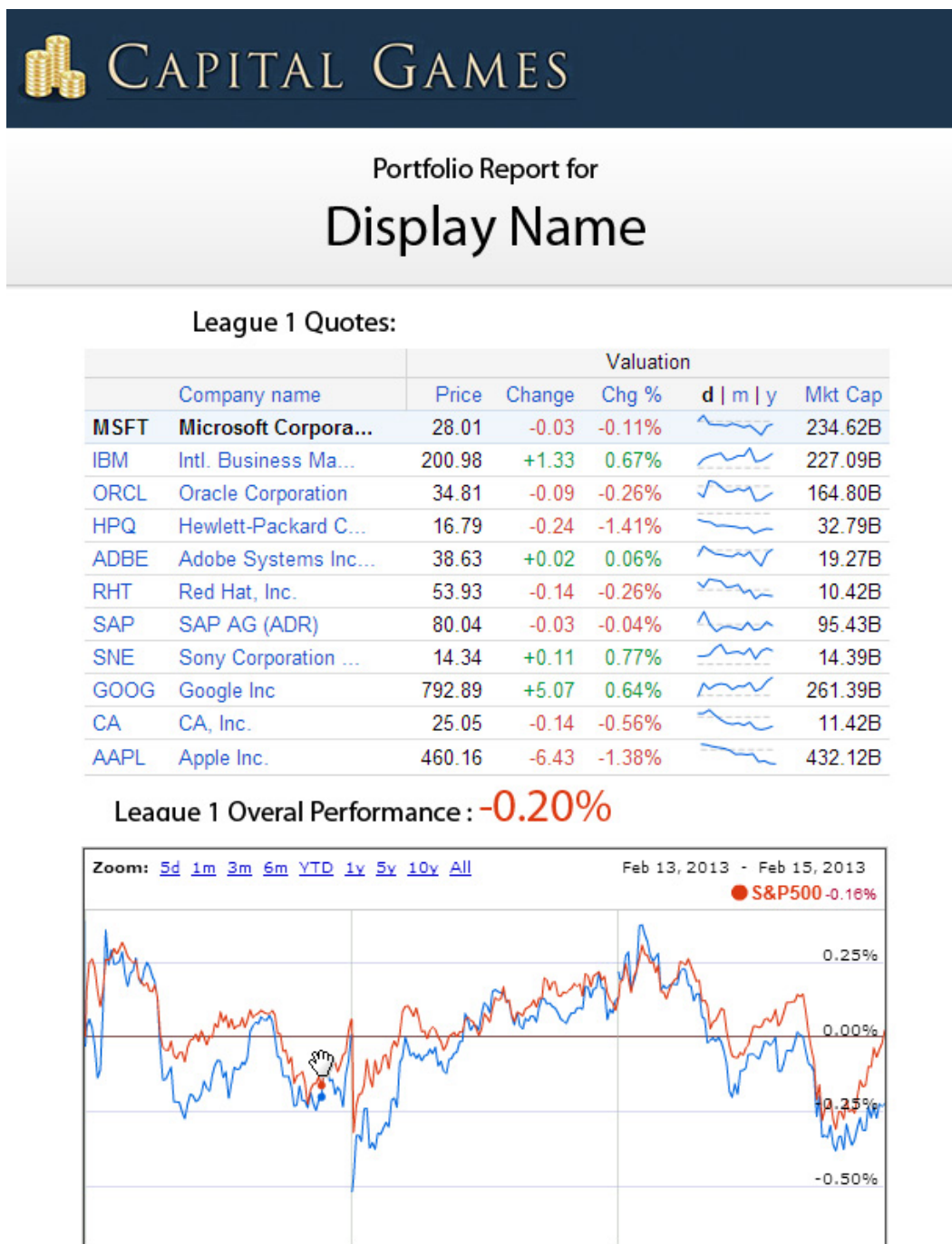


Figure 4.16: Users may choose to view a summary report of a league portfolio, only requiring one click from the portfolio page which would sum to two clicks.

5 Effort Estimation

The “Use Case Points” system of estimating the effort necessary to create the system will be employed. This is motivated by a need to have a metric on the complexity of the design of the system in order to properly motivate resource allocation, with the acceptance that any created metric will be necessarily subjective and arbitrary.[10]

5.1 Background

The estimation of effort is a factor representing the product of sums of various weighting factors. This can also be used to estimate the number of man-hours which will be devoted to completing the project. The factor representing the total weighting factor is:

$$UCP = UUCP \times TCF \times ECF \quad (5.1)$$

$UUCP = UAW + UUCW$ represents the Unadjusted Use Case Weight as a sum of Unadjusted Actor Weight, the weighted complexity of actor involvement, and Unadjusted Use Case Weight, the weighted complexity of the various use cases of the system.

There are two complexity factors: technical, and environmental.

$$CF = C_1 + C_2 \sum_{i=1}^{13} W_i F_i \quad (5.2)$$

TCF , the Technical Complexity Factor, is a heuristic index representing the challenges posed in implementing nonfunctional requirements of a system and is specified by interviews with experienced developers. $C_1 = .6$, $C_2 = .01$, $W_i \in \{.5, 1, 2\}$, and $F_i \in [0, 5]$. ECF , the Environmental Complexity Factor, is another heuristic index representing miscellaneous factors including experience and staffing. $C_1 = 1.4$, $C_2 = -0.03$, $W_i \in \{-1, .5, 1, 1.5, 2\}$, and $F_i \in [0, 5]$.

The UCP can be interpreted as weighted count of the various requirements and specifications needed to implement a system. Therefore, the duration of a project can be estimated by multiplying the UCP by a productivity factor PF representing the average development man-hour needed per use case point.

5.2 Unadjusted Use Case Points

Actor	Description	Complexity	Weight
-------	-------------	------------	--------

Investor	A normal user is interacting with the site through a graphical user interface.	Complex	3 pts
League Manager	League Manager requires a graphical user interface.	Complex	3 pts
Site Administrator	Admin requires private GUI as well.	Complex	3 pts
Database	System interacts with database layer through a predefined framework.	Average	2 pts
Web Browser	Browser interfaces with application through RESTful API over HTTP to navigate and submit forms.	Simple	1 pt
Finance Adaptor	System interacts with Yahoo! Finance through its web API.	Simple	1 pt
Queueing System	System invokes queueing system to schedule events through a predefined API.	Average	2 pts

Use Case	Description	Complexity	Weight
Join (UC-1)	Simple user interface, 4 steps for main success scenario. 2 participating actors (Database, Manager).	Average	10 pts
Change (UC-2)	Average user interface, 4 steps for main success scenario. One participating actor (Database).	Average	10 pts
Browse (UC-3)	Simple user interface, 4 steps for main success scenario, one participating actor (database).	Average	10 pts
Order (UC-4)	Advanced user interface, 5 main steps for main success scenario. Three participating actors (Database, Finance Adaptor, Queueing System).	Complex	15 pts
View (UC-5)	Simple user interface, 3 steps for main success scenario, but complicated display logic involved. One participating actor (Database)	Complex	15 pts
Tutorial (UC-6)	Simple user interface, 4 steps for main success scenario. No participating actors.	Simple	5 pts

Report (UC-7)	Advanced user interface, 5 main steps for success scenario. Up to three participating actors (Database, User, Manager).	Complex	15 pts
---------------	---	---------	--------

5.3 Technical Complexity Factors

Technical Factor	Description	Weight	Perceived Complexity
Distributed System	System is distributed between end users having access through web and main server(s)	2	3
System Performance	Users expect good performance but nothing exceptional	1	3
User Efficiency	End users expect efficiency but there are no exceptional demands	1	3
Complex Internal Processing	System needs to track performance of various user instances both day-to-day and over extended intervals	1	4
Reusability	No requirements for system to be reusable	1	0
Ease of Installation	Ease of installation is low because only one host machine is used in implementation	.5	2
Ease of Use	Ease of use for users is imperative	.5	5
Portability	Portability is only high enough to allow for ease of development on various platforms	2	2
Ease of Change	System will only change marginally, so ease of change is low priority	1	1
Concurrent Use	Concurrency is an issue because users have access to chat, activity, and history feeds, and system needs to poll finance data in approximately real-time	1	4
Security	Security of users is important but Herculean measures are not necessary	1	3
Third Party Access	Because of RESTful interface, limited third party support is possible but not currently supported	1	2

Training Requirements	System is relatively easy to use, but basic tutorials are offered to users	1	1
-----------------------	--	---	---

5.4 Environmental Complexity Factors

Environmental Factor	Description	Weight	Perceived Impact
Development Experience	Beginners with UML-based development and the Construction process	1.5	1.5
Application Experience	Complete novices to the field of finance	.5	0
Paradigm Experience	Beginners to the use of databases and web frameworks	1	1.5
Lead Capabilities	Leads have no prior leadership experience	.5	0
Motivation	Motivation is high but fluctuates over semester	1	3
Stable Requirements	Requirements are well-known but only approximate	2	3
Part Time	All developers are working with very few hours a week	-1	5
Language	Developers are using a modern but unfamiliar language	-1	2

5.5 Calculations

$$UUCP = 3 \times 3 + 2 \times 2 + 1 \times 2 + 10 \times 3 + 15 \times 3 + 5 = 95$$

$$TCF = .6 + .01 \times (34.5) = .945$$

$$ECF = 1.4 - 0.03 \times (5.75) = 1.23$$

$$UCP = 95 \times .945 \times 1.23 = 110.4$$

$$\text{Duration} = UCP \times PF = 110.4 \times 28 = 3091$$

6 Domain Model

At its highest level, Capital Games consists of a few subsystems working together and coordinated by an internal controller. The end user interacts with the application through either a web browser or by directly submitting HTTP requests to the server. These actions are equivalent because user actions are translated into RESTful actions and interpreted equivalently by an appropriate RESTful controller. [11] Once a controller is invoked, it consults the internal subsystems before responding to the request. Each of the subsystems can be identified by the purpose they serve in relation to the application.

6.1 Concept Definitions

Database

By its nature as a data-driven site, data persistence is core to Capital Games. Therefore, a database subsystem is necessary. A challenge often encountered when using databases in an application is the translation of database-native datatypes to the more varied datatypes employed by dynamic applications. [12] To simplify this, Capital Games uses the ActiveRecord object-relational mapper to abstract the logic between the database and the system as a whole. Only privileged portions of the system have access to the database. This maintains the safety of the data while also allowing it to be manipulated more precisely. Per the convention of MVC-style application style architecture (upon which we base our application, described in more detail later), these are known as the Models. The database is explored in more detail in the Data Structures section.

Finance API Adaptor

The data for the application comes from a third party source, Yahoo! Inc. Yahoo! provides both nearly-real-time and historical data on most U.S.-traded stocks. Yahoo! exposes this data through a web API service in which a party can make up to several thousand requests against Yahoo!'s databases daily. The party simply enters arguments into an HTTP request which is interpreted by Yahoo! as a database search, runs the query, and returns the results in CSV format. [13]

In order to interact with the web service, we employ an adaptor plugin which translates between the various syntaxes used by Yahoo! and our own system. Any and all parts of the application which require access to a live data-stream invoke the Finance Adaptor subsystem, which in turn queries Yahoo!. This modularity enables multiple subsystems of the application to have access to live data when necessary.

Queueing (Asynchronous Task) System

Fundamentally, Capital Games is about placing trading orders for various stocks. Though the simplest type, market orders, are executed almost immediately after being placed, stop and limit orders may not be executed for quite some time. [5] [3] [4] This begs the question of how to perform a trade at some undetermined time after the order is placed. Upon further inspection, a few other functions of the site depend on a similar capability. In order to update the user portfolio database regularly or send out newsletters, the system must be able to asynchronously execute certain tasks. Enter a Queueing System.

Whenever a task needs to be performed asynchronously, the task is entered into a designated portion of a Redis database, configured as a queue. Background "workers" (processes) perform tasks as they arrive. Tasks can also be scheduled to occur at specific times or intervals. In this way, everything from polling the datastream for stock updates to performing scheduled updates and e-mails can be coordinated by a single system.

Views Generator

Finally, when all data have been collected and a response needs to be rendered, those data are delivered to a subsystem which dynamically generates the content which are served up to the end-user. The Views Generator contains various modules which simplify translating the data to web-standard HTML and Javascript.

Mailer System

Capital Games is designed to periodically alert users as to their portfolio performance. This is performed by the Queueing System in conjunction with the Mailer system. The framework we employ natively contains a robust mailing system called Action Mailer, which generates content dynamically at runtime. [14] This allows us to perform calculations on leagues and then include that into emails, in addition to raw data.

6.2 Association Definitions

As indicated in Figure 6.2, there are 6 components which are core to our system: Controller, Views, Models, Finance Adaptor, Queueing System, and Mailer.

The Controller acts as the single point-of-entry for all user interactions. It interpretes requests and accordingly accesses the Models, the Finance Adaptor, and the Queueing System, before delivering the necessary data to the Views Generator. By definition, the Controller is the most privileged system component.

The Queueing System is possibly more privileged than the Controller. It has a great deal of autonomy, functioning without the Controller and being able to invoke other systems on its own. Compare this to the Controller, which is only invoked upon requests from a user. The Queueing System communicates with Models, the Finance Adaptor, and the Mailer as necessary to perform its tasks.

Conversely, the Views Generator is the least privileged subsystem. It cannot externally communicate and only responds to to the actor which called it.

The Finance Adaptor, Mailer, and Models are each afforded limited privileges, in that the Models and Mailer need to communicate with the database and Views, respectively, while the

Ticker	Name	Date	Time
Change %	Previous Close	Open	Volume
Day High	Day Low	Day Range	Ticker Trend
Bid	Ask	Average Daily Volume	Price-to-Earnings Ratio

Table 6.1: These are some of the data that Yahoo! Finance provides upon request, and which the adaptor we employ can convert.

Finance Adaptor needs to communicate with external data sources through the Internet. They each respond directly to requests from the components which invoke them.

6.3 Attribute Definitions

Though the application as a whole is not entirely object-oriented, and thus not all parts (ie the Controller) have true attributes, the Models, Queueing System, and Financial Adaptor all do.

Models possess basic attributes for the data they contain, such as user names, email addresses, stocks possessed, etc. These are contained in Figure ???. Similarly, Orders to be performed in the Queue have similar identifiers, as shown in Figure 6.3.

Validation on the data saved by the Models layer is performed automatically by the object-relational mapper, which can enforce data typing rules built into the database. This happens automatically.

Orders data is proxied through the database, and so its data is also validated before being entered. Though a remote edge case is the possibility of an order being valid while placed but being invalidated (for example by a stock no longer being on the market) while in the queue, we do not consider it at this time.

The Queueing System utilizes entities called “background workers”. As the name implies, these are persistent entities which wait for work in the form of queued tasks to hit the Redis database. When this happens, the first available worker pulls the task from the queue.

The Financial Adaptor possesses a set of financial metrics, a brief list of which is tabulated in Table 6.3. When data is retrieved by the adaptor, it is tabulated with some of the parameters shown in the table.

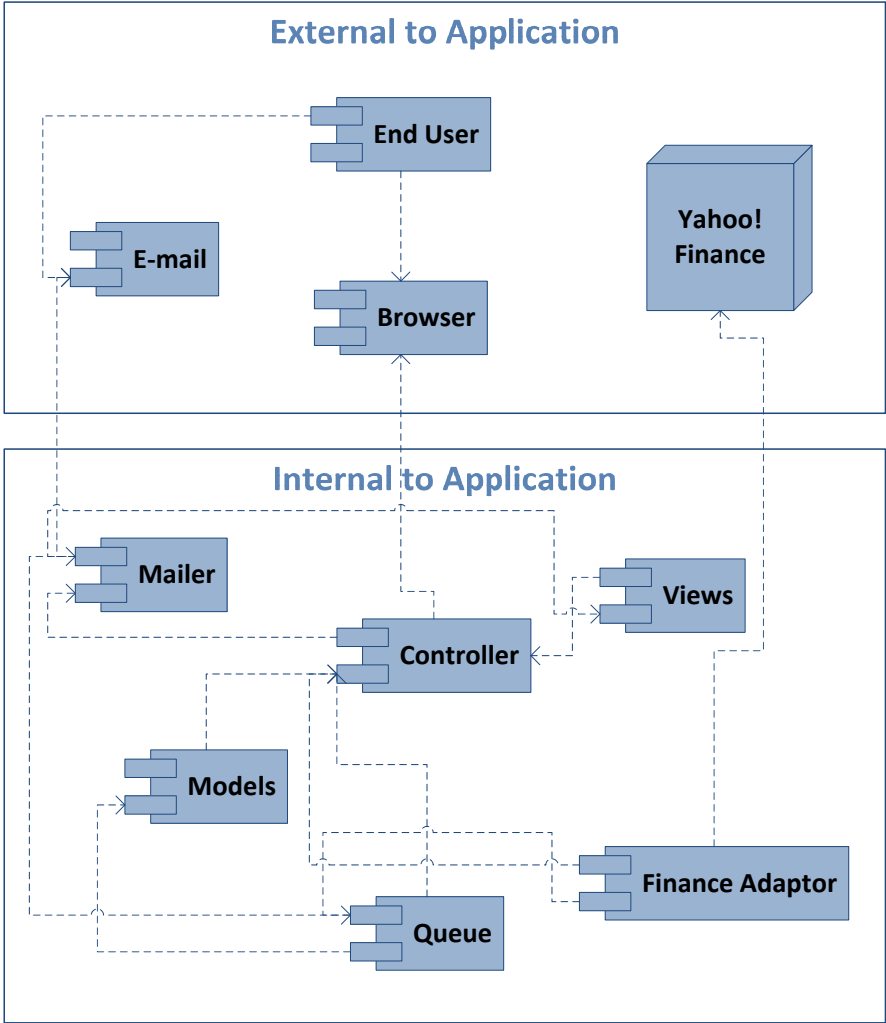


Figure 6.1: This high-level overview of the domain model of our application shows the separation between the external actors User, Browser, and Yahoo! Finance, as well as how the internal component subsystems relate to each other.

Resque Background Process Structural Model

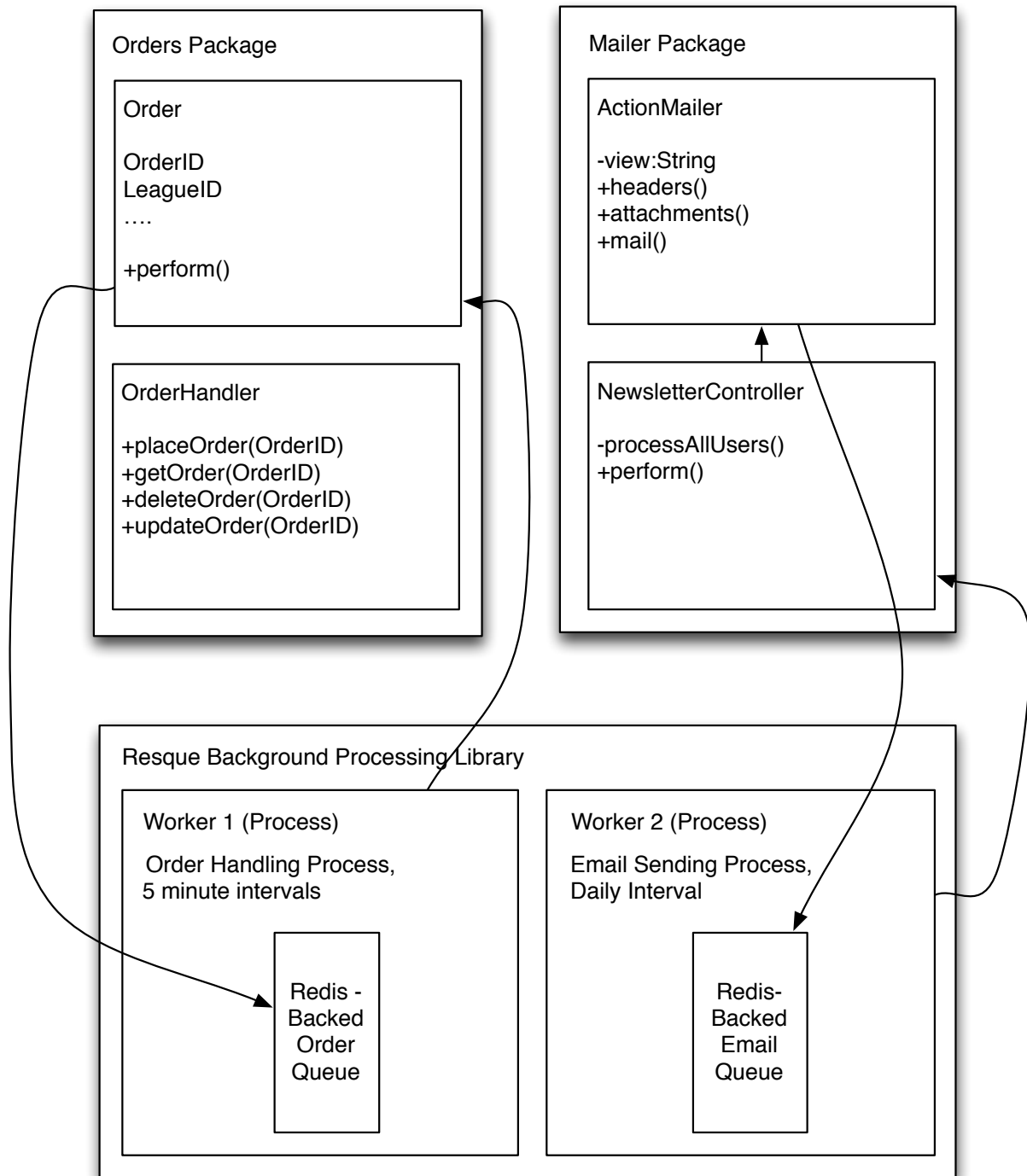


Figure 6.2: The structural model of the Resque Queueing System. Market Orders to be placed are bundled as Orders and served to the queue to be processed every few minutes. Newsletters are performed daily and bundled and served every night. Background workers wait for updates to the Redis database and upon seeing a valid task, pull it and begin processing.

6.4 System Operation Contracts

UC-1 Join or Create League

- *Preconditions*
 - (join) There are open spots available in the league to be joined
 - (join) The league to be joined is public
 - (create) The desired league settings are valid
- *Postconditions*
 - League creation and/or user membership are reflected in database

UC-2 Change League Settings

- *Preconditions*
 - User is league manager for the league to be changed
 - The desired league settings are valid
 - If any users are banned, there is justification
- *Postconditions*
 - League setting changes are reflected in database
 - User status changes are reflected in database
 - League members are notified of all changes made

UC-3 Browse Companies

- *Preconditions*
 - Financial API is currently accepting requests
- *Postconditions*
 - None

UC-4 Place Market Order

- *Preconditions*
 - Financial API is currently accepting requests
 - User is a member of the league in which they are placing order
 - User currently has enough funds or margin to place order
- *Postconditions*
 - User's portfolio reflects changes made to funds, margin, and position
 - Database has been updated with these changes

UC-5 View Portfolio

- *Preconditions*
 - User is member of the league in which the portfolio to be viewed is
- *Postconditions*
 - None

UC-6 Access Tutorials

- *Preconditions*
 - None
- *Postconditions*
 - None

UC-7 Take Disciplinary Action

- *Preconditions*
 - Initiating actor is a site administrator
 - There are outstanding abuse reports
 - Any actions taken against users are justified
- *Postconditions*
 - User's status is updated in the database
 - User is notified of action taken against them

6.5 Economic and Mathematical Models

Perfect Competition

One important concept of the stock market is the idea of perfect competition. Perfect competition is a theory in economics that states that it is not possible for any one participant to have enough resources to control the entire market. In terms of our project, that boils down to the following:

1. One single person cannot control the stock market.[15]
2. Anyone should be able to enter or exit the market with ease.
3. Buyers know the full details of any stock they are to trade.
4. There is no difference in the buying and selling price.[16]

The problem with this model is that it is not entirely perfect or plausible. In reality, exceptionally wealthy individuals can dominate entire sectors of the market; entering or exiting markets is hindered by commission charged by brokers; certain traders may know more about certain stocks than other traders (also known as insider trading); and buying and selling prices differ, according to the bid-ask-spread. Nevertheless, our platform makes simplifying assumptions about the market to avoid most of these issues, and when unavoidable, compromises with the economic reality.

The economic assumption of perfect competition states that one person cannot control an entire market. This assumption is reasonable for normal investors with significantly less capital than the market capitalization of companies or markets, and therefore it is reasonable to assume that in our platform, individuals cannot shift the market price of stocks by their participation. However, in reality, exceptionally wealthy individuals may have more assets than the market capitalization of certain small and even medium size corporations. If such an individual were to attempt to enter or leave a market suddenly, the entire market would experience a shift. To avoid the complication of having to model the effect of such actions by such parties (which is exceptionally unusual and not the intended purpose of this simulation platform), we constrain users' initial seed capital to be below a certain level to prevent them from achieving this level of market domination.

The assumption of being able to freely enter or exit a market is somewhat unrealistic when it comes to the stock market. In general, brokers charge commission to execute any trade on behalf of an investor, which contradicts the stipulation of freedom to enter or exit. Nevertheless, we constrain users' seed capital to be above a certain level, to the point where commission should be nearly negligible.

Though it is impossible to resolve an issue of information disparity (the very nature of which stems from third party sources), we make the assumption it is a non-issue. We assume that all users gain all their information exclusively from the information exposed by our Research tools.

Bid-Ask Spread

Similarly, it is difficult to challenge the bid-ask spread, the difference between the sale and purchase price listed for a stock because the many underlying factors. [17] In certain circumstances, a fairly large bid-ask spread can occur. Although this happens naturally in the stock market, we do not factor it in. We make the simplifying assumption that the bid-ask spread is zero dollars, that being that the bid and the ask are of equal amount, with the ability to program that functionality into future versions of the platform.

Reporting Abuse

There are many algorithmic approaches to the functionality of reporting abuse. We decided to make reporting in a very simple manner, for the sake of keeping this part of the project more lightweight. When someone is reported by a user, it is put on a notification list of the admins. These notifications are listed in a database, listing the users along with the reason for the report. If enough notifications are given, the status of the user's account is at the discretion of the admin. Some possible tracks an admin can take when they run into this problem would be to message the user in order to try to come to an agreement, ban the user forever or take a deduction from the account as a warning to show that they have done wrong. Our reason for this model is that rather than relying on an algorithm is that it will take the pain off of us, the programmers. There are many factors in the process of creating an algorithm to deal with reported users, and leaving it up to an admin to personally solve the problem is a much simpler solution than having to deal with all the different factors that could be taking part in the process. This solution won't always solve the problem. For example, if the site becomes a large success, there will either have to be many admins working on the site or the algorithmic approach will have to be implemented, but it will do for our purposes to keep it simple.

7 System Interaction Diagrams

7.1 Introduction

Following is an analysis of the interactions of the two most important internal subsystems in our system as identified in our domain model, the financial data retrieval subsystem and the asynchronous processing subsystem. The interaction diagrams included clearly describe the interactions that occur within each of these subsystems. They elaborate upon the mechanics behind our use cases, but do not necessarily correspond to them one-to-one. This is because several of our use cases are completely facilitated through the browser and controller to generate views for the users, and as such it would not be interesting or worthwhile to explore the internal interactions. The following analysis clearly describes how market orders are placed and processes, how information is retrieved from Yahoo! Finance, and how we manage asynchronous processes (i.e. a queue) in order to process market orders and enact our mailer system.

7.2 Financial Data Retrieval Subsystem

Enter the Capital Games Financial Adaptor

For the querying and retrieval of real time and historical financial data and stock quotes in a form that is both familiar and friendly to players of Capital Games, we will utilize the *Yahoo! Finance* Application Programming Interface (API), which allows for easy access of *Yahoo! Finance* stock data via data served via URLs that our system can retrieve, parse, and then translate for the use of Capital Games fantasy leagues platform. Since we will be drawing data from *Yahoo! Finance*, it will be represented as external to the system of Capital Games. Internal to our system, however, will be the financial adaptor module that will automatically handle data retrieval from *Yahoo! Finance* based on user queries.

We chose this route over either option of having financial data querying and retrieval built-in to our system or taken from any other API because attempting to construct a built-in, live stock-querying system within Capital Games itself would have been both expensive and impractical — much akin to reinventing the wheel — and because *Yahoo! Finance* has proven itself as stable and reliable versus other available APIs. Thus, this section will explain our intended financial adaptor module for seamlessly delivering *Yahoo! Finance* data for use within Capital Games.

Essentially, by us deploying the a financial adaptor module into Capital Games, users will be able to easily search for stock data within our website and have it near-instantly displayed on the web page they are viewing without the user even being cognizant of all the work being done in the background via our financial adaptor module existing in our server. The financial adaptor module will have all the functionality for making requests for data from *Yahoo! Finance* based on user

input and will actively draw and translate the raw data from *Yahoo! Finance* into a form that can be delivered within our own views ergo the data will be displayed on our web pages.

One consideration we need to take from our end for the building of our financial data is validating user queries for stock symbols. In other words, what would happen in the case that a user attempts to query a stock symbol, company name, industry, or sector that does not exist? To resolve such issues, our adaptor will also draw from our own database built into the website that keeps an updated list of valid stock symbols and names that is drawn from a source similar to *Yahoo! Finance*, *EODData*. We are using *EODData* to supplement our use of the *Yahoo! Finance* API as *EODData* offers easy retrieval of all stock symbols and names in a method that is similar to *Yahoo! Finance*. *Yahoo! Finance* unfortunately does not offer that particular feature, so we will be using *EODData* as a supplement to that, in that respect. We will essentially update our database via *EODData* and our financial adaptor module at each market opening and closure to account for any mergers, acquisitions, or any other major changes involving companies in the stock market.

Once user queries are validated by our financial adaptor module, our financial adaptor module will then parse the user query into a URL format that will allow for the retrieval of data via *Yahoo! Finance*. Upon completing this, the URL will then be passed through our financial adaptor to *Yahoo! Finance*, from which data will be returned to our financial adaptor module via a comma-separated values format (.csv, a container for easily passing volumes of data), which our financial adaptor will then translate into an arrangement that our views can utilize to deliver to the content to the webpage the user made the query from. From there, the user can then view the data and choose whether they would like to interact with the queried stock within Capital Games.

To elaborate on the technical specifications of our financial adaptor, the rest of this section will incorporate and explain interaction diagrams of methods used by our financial adaptor, illustrating the process I summarized regarding how our financial adaptor will go through interacting with *Yahoo! Finance*, *EODData*, and the Capital Games platform.

All interaction diagrams will begin in the following page.

Financial Adaptor Interaction Diagrams

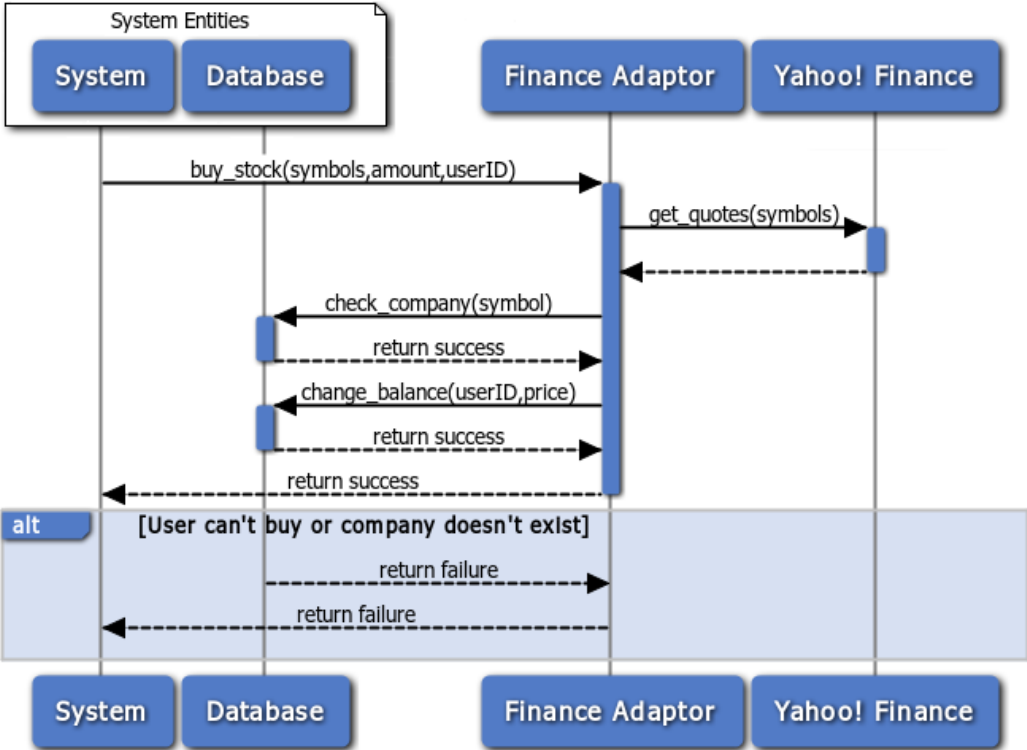


Figure 7.1: When a user buys a stock, the browser will inform the system of the transaction so that it can be approved. The system passes over the process to the finance adaptor who will check if the company is accepting trades, the current price from Yahoo! Finance, and if the user is able to afford the purchase from the database. If all goes well, the transaction will be recorded in the database and the balance will be changed. After all that is complete, the transaction will be marked as a success and the system will be notified. (Related use case: UC-4)

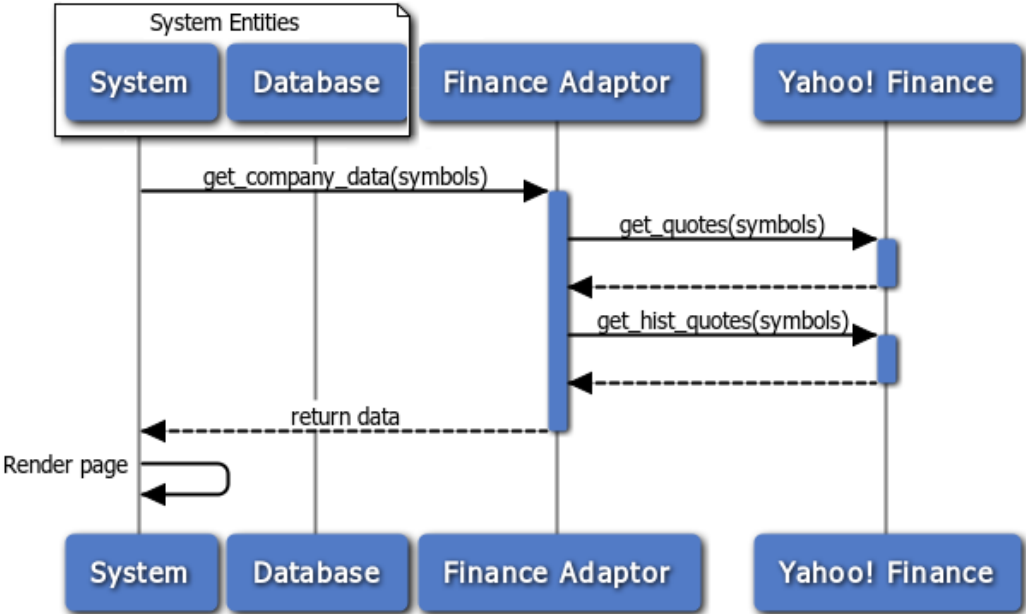


Figure 7.2: When a user wants to view a company page, the company data must be loaded from our finance API. Once the process is passed to the finance adaptor, the quotes and the historical quotes will be pulled from Yahoo! Finance and brought back to the system, who will prepare the page for the user. This is also the process by which user portfolios will be generated, via aggregating the value of all their stocks. (Related use cases: UC-3, UC-5)

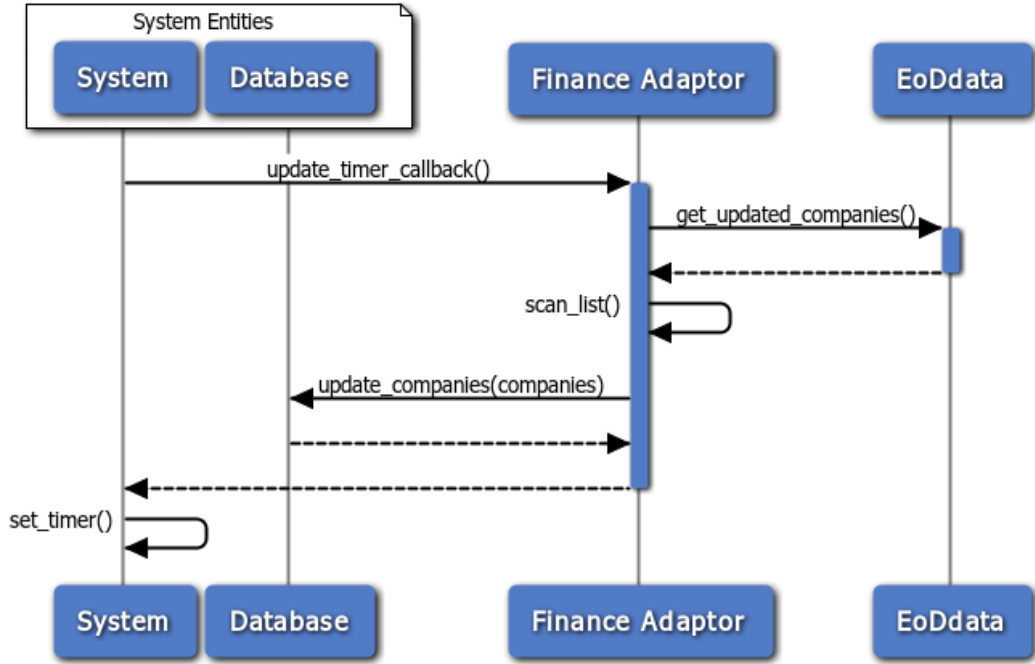


Figure 7.3: We need to keep a local copy of the current companies in our database so we can do rapid processes sing of all of the companies. In order to do this, there will be a timer that is set to update the database every once in a while. When the timer goes off, the system will pass the process onto the finance adaptor. The finance adaptor will then call data from EODData, who knows all of the current companies in the stock market. The finance adaptor will then scan the data for any new/deleted companies and change the database accordingly. After this is complete, the timer will start again so this process can loop.

7.3 Asynchronous Processing Subsystem

Introduction

One of Capital Games' primary requirements is to have an asynchronous processing subsystem. This requirement exists both due to the nature of our system, which involves events conditionally occurring at certain time intervals, and the pursuit to build a scalable product. In an attempt to build a system that most closely represents the real stock market, the decision was made to have a pending order queuing system which processes orders at 5 minute intervals. Many orders are processed directly, however some such as short sales and limit orders have conditions associated with them which determine when exactly they are processed. In addition, as the system involves sending summarized reports of player performance metrics at certain time intervals an asynchronous, non-event driven subsystem is highly necessary.

Nature of the Subsystem

The asynchronous processing subsystem features three primary components. First, the ability to spawn multiple, independent processes to handle the different kinds of asynchronous tasks. Second, the ability to handle arbitrary object types. And finally, the ability to queue tasks that are waiting to be processed. This is why the Resque Background Process Library built in Ruby was an ideal pick. It allows for the creation of customizable background processes known as "workers". Each worker processes a unique queue. Moreover, each queue can have objects of vastly different types, as long as they implement the function "perform". This is very intuitive as it allows each object to possess the code which acts on it. Lastly, it implements a very smart technique of only storing references to objects in the queue as opposed to the objects themselves so that outdated objects are never processed. This forces the worker to request the most recent version of the object from the DB when it starts being processed. Of course this comes at the slight expense of higher load on the DB when a worker is not sleeping. It is possible that this subsystem will be expanded to incorporate caching techniques. However, they are currently not a requirement. Finally, the queues are stored in RAM for the fastest possible performance. Nevertheless, queues are persisted in JSON encoded flat files to ensure redundancy.

Structural Model

The structural model below depicts the overall structure of this subsystem. Namely, the Resque Library and two packages or modules which each are responsible for one kind of task. On the left, the orders package displays a relevant subset of all classes that pertain to placing and processing orders. As previously mentioned, the Order object itself implements the perform method. Therefore, it knows how to process its data when it gets placed in worker 1's queue. While the OrderHandler class isn't directly involved in the asynchronous processing of orders, it is still relevant in this scope and therefore included in the diagram. It is ultimately the class responsible for placing the order object on the queue when an order is placed. Similarly, the mailer package is depicted with a subset of classes which aggregate data about user performance and send out periodic summarizations of performance metrics to all users on the site. Worker 2 is dedicated to processing email related tasks daily. In this case, the architecture is slightly different as the worker doesn't directly call perform on each ActionMailer object, but instead on a NewsletterController which populates the worker's queue with customized ActionMailer Objects.

Interaction Diagrams

There are two interaction diagrams displayed below, each associated with one worker. Due to the inherent background nature of this subsystem, there are relatively few actors involved in this subsystem.

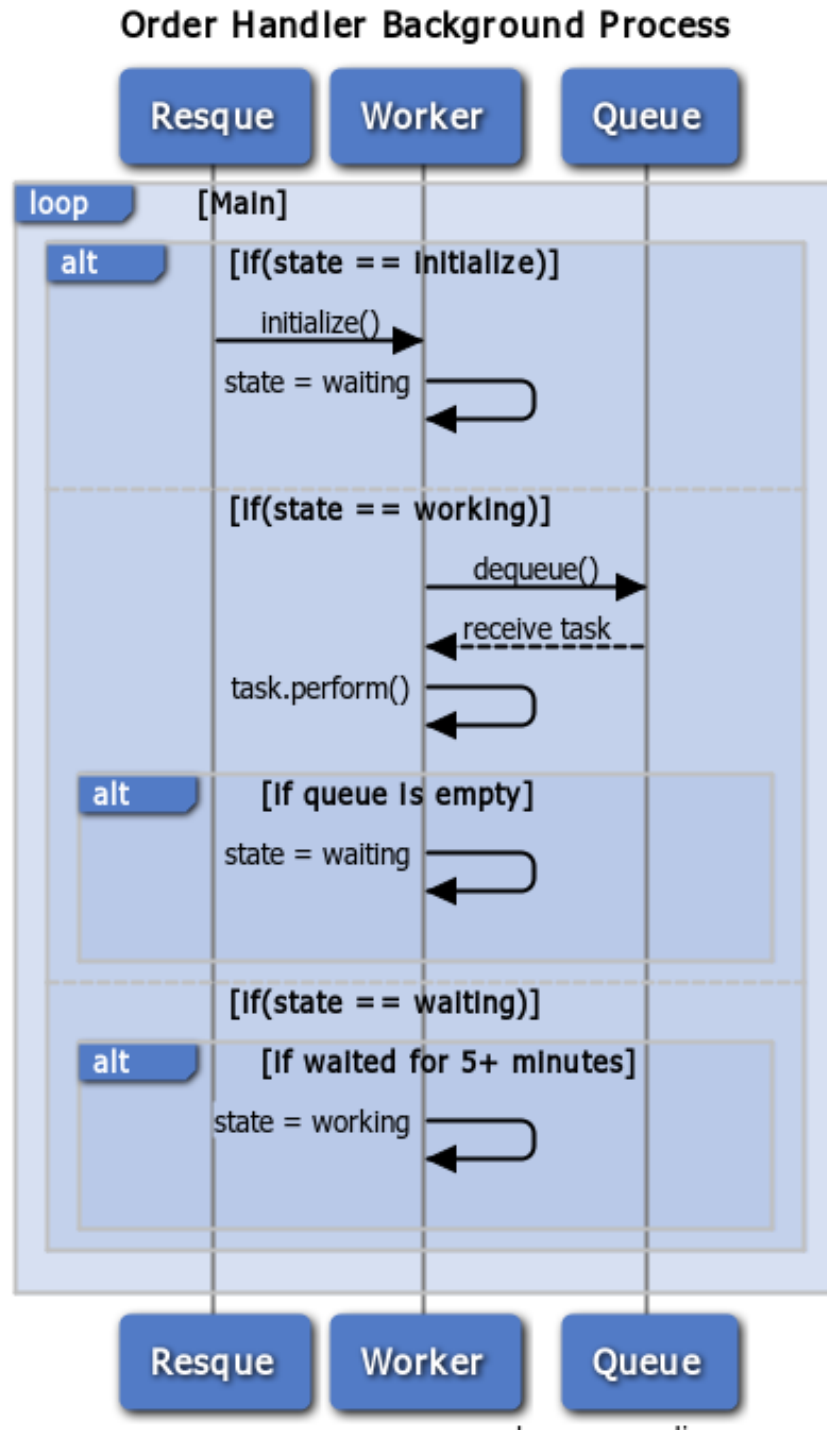


Figure 7.4: The interaction diagram above is roughly divided into two areas, when the process is working and when it is sleeping. This portrays the typical polling behavior of such a background running process. After initialization, when the worker wakes up it attempts to dequeue all objects and call the "perform" method on the object. Since the actual nature of the "perform" method is unique to every object, it is not depicted in this diagram. It is relevant to mention that this individualized execution design allows conditional orders to be processed very easily since the object has all the information needed to make the decision of whether to process at its disposal. Once the queue becomes empty again, the process goes back to sleep. This occurs continually after the spawning of the process.

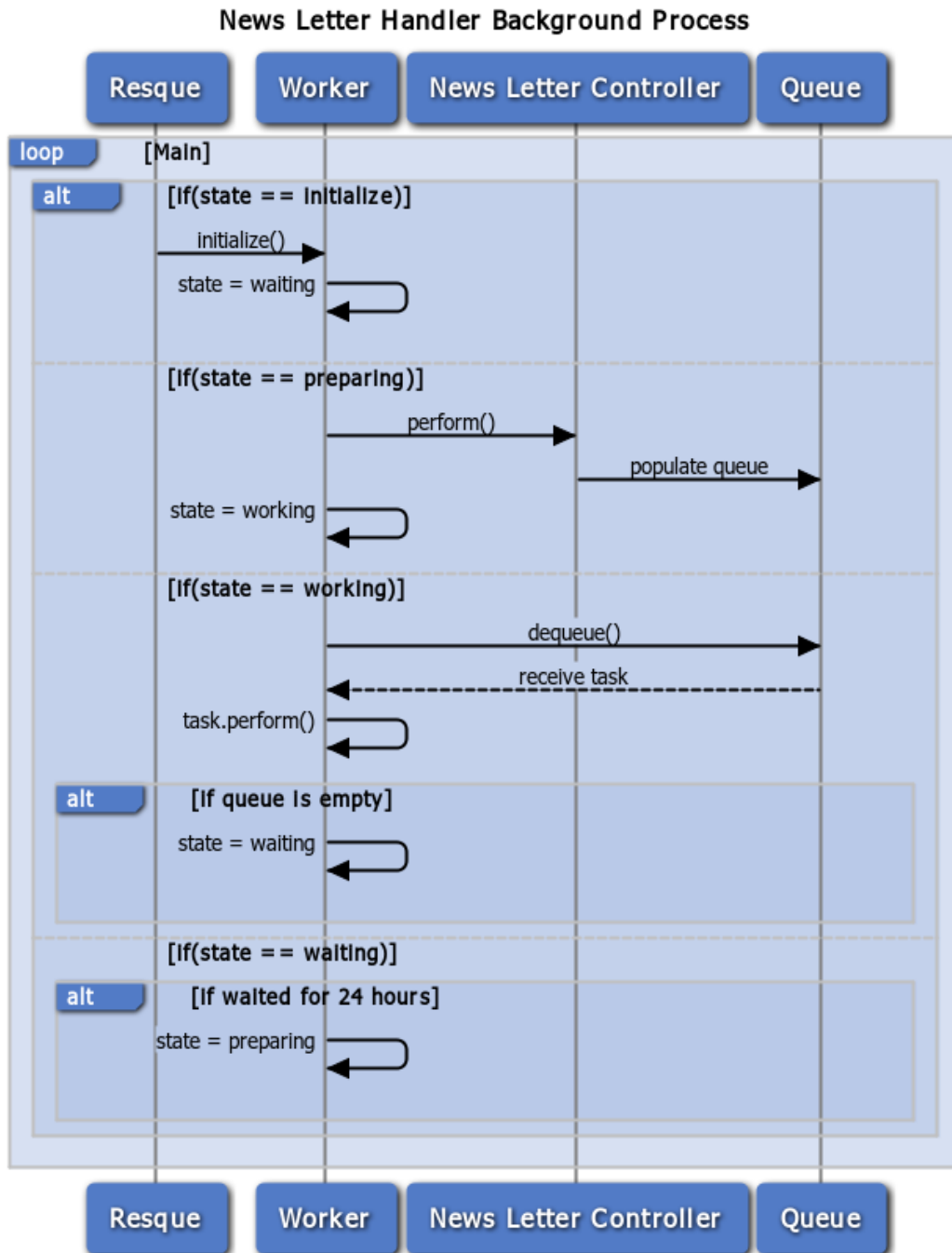


Figure 7.5: Worker 2 behaves a bit differently than worker 1 which results in having an additional state. This "prepare" state is when all the customizing of user-specific emails is done. Afterwards, the process enters the working state where it attempts to fire off all customized emails which were placed onto the queue during the "prepare" state. As in the previous diagram, the diagram incorporates the base case when the worker's queue has been emptied and when the process is sleeping.

7.4 Design Patterns

Various standard design patterns were utilized to provide functionality for things such as authentication, efficient page rendering and object modeling.

Model-View-Controller

The Model-View-Controller (MVC) pattern was heavily used throughout the CapitalGames system to properly organize model logic, business logic and presentation logic. This very intuitive pattern allowed the team to easily delegate work on different levels of the system. Frequently, a selection of team members would develop front-end functionality which required only the views to be altered, while other members implemented backend functionality which was done either in controllers or models. This pattern resulted in a more efficient development lifecycle overall, while also providing some performance gains. Namely, the MVC pattern calls on resources only when they are actually needed which prevents unnecessary overhead. For example, methods developed to be called only programmatically don't attempt to display a view which results in faster responses.

Security Proxy

The security proxy pattern was the core of our secure authentication system. This proxy pattern allowed us to easily protect content based on user role or other variables. The security proxy was implemented very similarly to that described in the textbook. Particularly, it behaved as a transparent filter between an HTTP request and a controllers method. Authentication requirements could easily be chained onto each other making it possible to create custom controller prerequisites. Finally, because the security proxy filtered every request made on a controller instead of just requests made upon login, all sensitive features of the site had a very robust shell which no user could easily bypass. This improved our design by providing solid, system-wide security.

ActiveRecord Pattern

The ActiveRecord pattern, an intelligent implementation of a database access design pattern, was used exclusively to interact with persistent storage technologies used in the CapitalGames system. This pattern offered the major advantage of not needing to hard code any database-specific queries. All requests made to the ActiveRecord Pattern are translated to the currently used DB system's language and data is returned in directly its object form. The lack of need to write direct queries also lead to a great side effect, namely database agnosticism which allowed various database implementations to be used during different stages of development. During development SQLite was used for its lightweight footprint on the developers machine, then for production MySQL was used as it is considerably more efficient when dealing with larger amounts of data. This design certainly improved our development by saving countless hours of development time.

RESTful Design

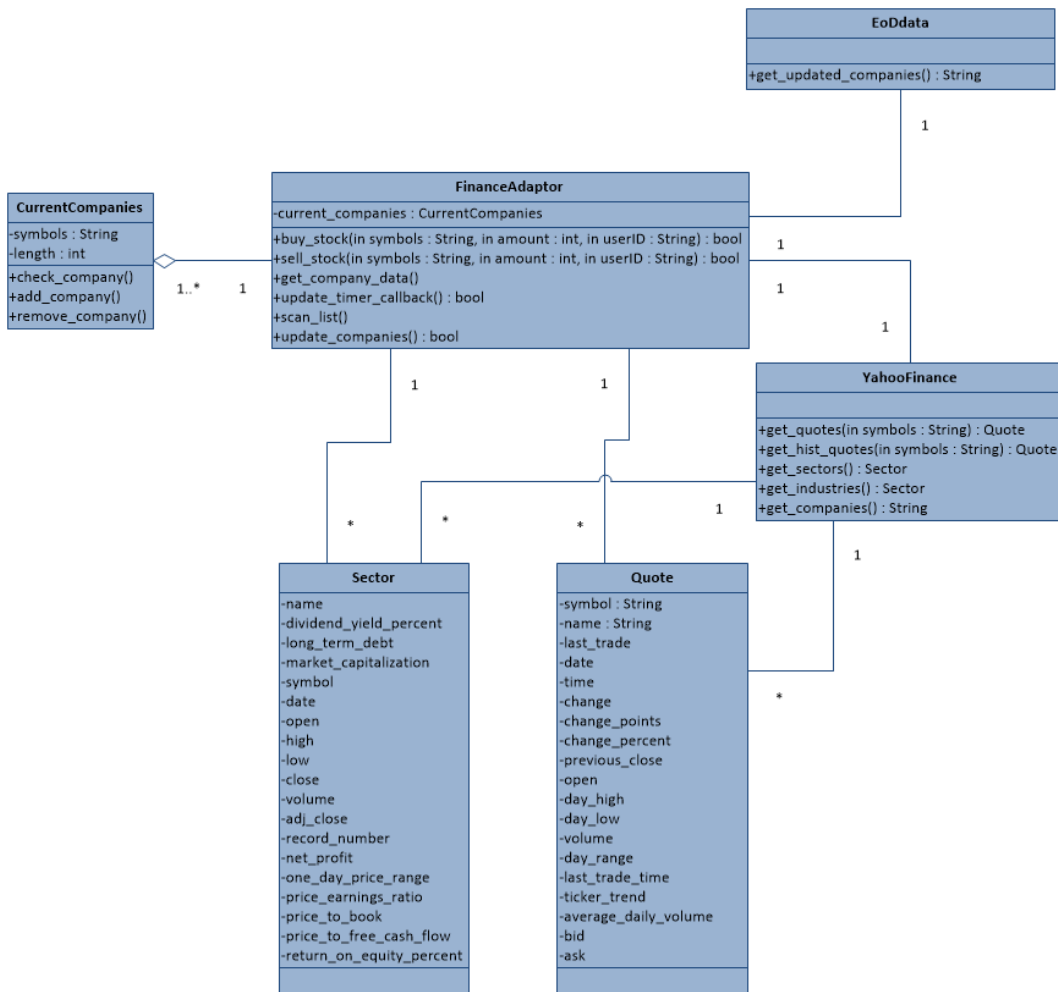
The RESTful design pattern being used more and more now on the web allowed us to implement our asynchronous order processing system. The RESTful design of some internal functionality allowed it to be accessed programmatically and securely through a simple API. As RESTful services are at the heart of Ruby on Rails, it did not require a lot of effort to expose some internal functionality without creating major security holes. Future iterations of CapitalGames will continue to rely on the stateful communication that our RESTful API offers.

Responsive UI Pattern

The Bootstrap UI framework implemented a design pattern completely segregating visual presentation from content and user experience. This provided in a beautiful responsive design which adapted to different client devices ranging from desktops to smartphones. The pattern takes advantage of the flexible markup of HTML5 to customize it on the fly when the page is rendering in the browser using Javascript and CSS. This allowed our team to target the rapidly growing mobile users without much extra implementation effort. It also inherently produced a faster user experience since minimal processing is done during initial page rendering and mostly done asynchronously once the page is already viewable to the user. We actively strived to achieve both of these goals.

8 Class Diagrams and Interface Specifications

8.1 Financial Adaptor Class Diagram



8.2 Financial Adaptor Data Types and Operation Signatures

Finance Adaptor

Attributes

Our Finance Adaptor performs the functions of validating user queries with existing stock symbols, companies, and/or sectors, then mediating between the Capital Games web server and Yahoo! Finance, enabling our fantasy league to be playable in real time. To accomplish data validation, a portion of the Capital Games database is updated regularly to keep our fantasy stock market league up to date based off of EODData, an API allowing for the reference to an up-to-date list of all stock-symbols, company names, and sector/industries.

— **current_companies** : **CurrentCompanies**

This is a reference to a database table updated via an external website, EoDdata, that verifies user queries with actual stock symbols, and/or company/sector/industry names depending on the user query.

Methods

Most methods are boolean, returning either success or failure regarding data retrieval. All other methods are voids, with no arguments, used for executing a specific function.

+ **buy_stock** (in **symbols** : **String**, in **amount** : **int**, in **userID** : **String**) : **bool**

Method called to buy stock; a typical method that would require the user to query up-to-date stock market information via our adaptor.

+ **sell_stock** (in **symbols** : **String**, in **amount** : **int**, in **userID** : **String**) : **bool**

Method called to sell stock; a typical method that would require the user to query up-to-date stock market information via our adaptor.

+ **get_company_data**()

This method returns all information available on Yahoo! Finance regarding a user's queried stock.

+ **update_timer_callback**() : **bool**

An internal timer signaling the stock query from Yahoo! Finance.

+ **scan_list**()

This method checks against the Capital Games' database.

+ **update_companies**() : **bool**

This method updates the information in the Capital Games' database from both Yahoo! Finance and EODData.

Current Companies

Attributes

Current Companies is the database table that our Finance Adaptor actually checks against when validating user queries. At a regular interval (based on method `update_timer_callback` from the Finance Adaptor, the Finance Adaptor retrieves data from EODData to update the Current Companies database table.) This is done to maximize efficiency by minimizing the amount of time the

adaptor must retrieve data from EODData.

— **symbols : String**

This is all stock symbols.

— **length : integer**

This defines how many total stock symbols are on the list.

Methods

All of these methods are invoked after the stock symbol or company name has been validated. All methods perform queries regarding updating the Current Companies table.

+ **check_company ()**

This method returns all informaton regarding a stock, to be parsed by the Finance Adaptor to retrieve what the user is querying for.

+ **add_company ()**

Method called to add a company to the database in cases such as Initial Public Offering of shares.

+ **remove_company()**

Method called to remove a company from the database in case of acquisition.

EODData

Attributes EODData is an external web app, much like Yahoo! Finance, that contains data regarding stocks in bulk. Essentially we are using it to validate stock user queries as it enables us to have a database of all stock symbols and company names.

Methods

+ **get_updated_companies ()**

This method updates the Current Comapnies database table based on the EODData API.

Yahoo! Finance

Attributes

Yahoo! Finance is the main external API we are utilizing for up-to-date stock market information for our fantasy stock market league. It is highly reliable and enables to make several, serparate queries of individual or multiple stocks at once.

Methods

+ **get_quotes(in symbols : String) : Quote**

This method returns quotes from a stock symbol based on Yahoo! Finance. + **get_hist_quotes(in symbols : String) : Quote**

This method returns historical quotes from a stock symbol based on Yahoo! Finance that spans a

larger period of time a user may draw specific information from in a predefined period of time.

+ **get_sectors() : Sector**

Gets information similar to quotes on a financial sector

+ **get_industries() : Sector**

Get information in industries that fall under financial sectors.

+ **get_companies() : String**

Retrieves all company information from Yahoo! Finance.

Sector

Attributes

US Market Sectors are essentially an umbrella category for certain groups of stocks. For example, technology stocks such as Google and Microsoft would belong to the technology sector. These have attributes similar to a stock quote. Essentially all attributes are the stock information one would find searching the sector on Yahoo! Finance.

Quote

Attributes

Quotes will essentially return a list of all data that has been retrieved from Yahoo! Finance, similar to above.

8.3 Financial Adaptor Traceability Matrix

Class	Finance Adaptor	Current Companies	EODData	Yahoo! Finance
Finance Adaptor	X			
Current Companies	X	X	X	
EODData			X	
Yahoo Finance				X
Sector	X			X
Quote	X			X

Our Financial Adaptor practically handles all querying of data. As a result, most classes trace to the Financial Adaptor. While EODData and Yahoo! Finance are external to the database in which all items subordinate to the Financial Adaptor exists, the fact that our Financial Adaptor queries

them for data validation and retrieval makes them essential conceptual entities in our Traceability Matrix.

For example, sectors and Quotes as mapped to the Financial Adaptor exist in their original form inside Yahoo! Finance's respective APIs, hence they map to Yahoo! Finance. Also, Current Companies is also a database table queried by the Financial Adaptor and updated via EODData, hence it maps to both the Financial Adaptor and EODData.

Object Constraint Language

In order to separate ideas in OCL, we will split it up descriptions by each class in the class diagram.

Finanace Adaptor Class

In this class, we have a few constraints that deal with this class being kind of central to all of the other classes that make up our financial adaptor. The function to buy or sell stocks are the same in their constraints. You must have a symbol name which is not equivalent to NULL and is the symbol of a company that exists. The amount must be a positive integer and if you are buying you must have enough to spend on this. Lastly, the userID must be plugged in, so that must also not be NULL and it should hold the username of someone who exists. The function get_company_data must have a symbol passed in that exists and is not NULL. The function scan_list must be called after the list already exists, or there will be an error. Lastly, the update_companies function must have a return from EoDdata that is valid and not corrupted.

Current Companies

All three of the functions in this class have the same constraint, that the symbol passed in must exist. On top of that, the "length" variable must always be equal to the amount of symbols in the "symbols" variable.

Yahoo! Finanace Class

Fortunately for us, this is another simple addition for it is dealt with completely by Yahoo! Finance. Something that we do have to be careful of is making sure that if we are calling the get_quotes or get_hist_quotes, we have to make sure that we are passing in a valid string. For example, if we passed in a NULL string, we would have an error returned.

EoDdata Class

This is the simplest class out of any of the ones we deal with, as there are no variables and only one function that returns a large list of stocks without us having to pass in anything. No constraints in this class.

Sector and Quote Class

The constraints on these classes will work as long as Yahoo! Finanace returned results that are valid. These results are checked in the Finance Adaptor Class, and if it did make it past that point, the data that is passed in is fine, and therefore there are no actual constraints on these classes.

8.4 Design Patterns

Various standard design patterns were utilized to provide functionality for things such as authentication, efficient page rendering and object modeling.

Model-View-Controller

The Model-View-Controller (MVC) pattern was heavily used throughout the CapitalGames system to properly organize model logic, business logic and presentation logic. This very intuitive pattern allowed the team to easily delegate work on different levels of the system. Frequently, a selection of team members would develop front-end functionality which required only the views to be altered, while other members implemented backend functionality which was done either in controllers or models. This pattern resulted in a more efficient development lifecycle overall, while also providing some performance gains. Namely, the MVC pattern calls on resources only when they are actually needed which prevents unnecessary overhead. For example, methods developed to be called only programmatically don't attempt to display a view which results in faster responses.

Security Proxy

The security proxy pattern was the core of our secure authentication system. This proxy pattern allowed us to easily protect content based on user role or other variables. The security proxy was implemented very similarly to that described in the textbook. Particularly, it behaved as a transparent filter between an HTTP request and a controllers method. Authentication requirements could easily be chained onto each other making it possible to create custom controller prerequisites. Finally, because the security proxy filtered every request made on a controller instead of just requests made upon login, all sensitive features of the site had a very robust shell which no user could easily bypass. This improved our design by providing solid, system-wide security.

ActiveRecord Pattern

The ActiveRecord pattern, an intelligent implementation of a database access design pattern, was used exclusively to interact with persistent storage technologies used in the CapitalGames system. This pattern offered the major advantage of not needing to hard code any database-specific queries. All requests made to the ActiveRecord Pattern are translated to the currently used DB system's language and data is returned in directly its object form. The lack of need to write direct queries also lead to a great side effect, namely database agnosticism which allowed various database implementations to be used during different stages of development. During development SQLite was used for its lightweight footprint on the developers machine, then for production MySQL was used as it is considerably more efficient when dealing with larger amounts of data. This design certainly improved our development by saving countless hours of development time.

RESTful Design

The RESTful design pattern being used more and more now on the web allowed us to implement our asynchronous order processing system. The RESTful design of some internal functionality allowed it to be accessed programmatically and securely through a simple API. As RESTful services are at the heart of Ruby on Rails, it did not require a lot of effort to expose some internal functionality without creating major security holes. Future iterations of CapitalGames will continue to rely on the stateful communication that our RESTful API offers.

Responsive UI Pattern

The Bootstrap UI framework implemented a design pattern completely segregating visual presentation from content and user experience. This provided in a beautiful responsive design which adapted to different client devices ranging from desktops to smartphones. The pattern takes advantage of the flexible markup of HTML5 to customize it on the fly when the page is rendering in the browser using Javascript and CSS. This allowed our team to target the rapidly growing mobile users without much extra implementation effort. It also inherently produced a faster user experience since minimal processing is done during initial page rendering and mostly done asynchronously once the page is already viewable to the user. We actively strived to achieve both of these goals.

8.5 Asynchronous Subsystems

The nature of Order's requires a vary particular type of asynchronous handling. Lucky for us we were able to find a ruby gem that makes this messy process quite elegant. Resque allows one to queue up tasks and execute them in "first in first out" (FIFO) order by dequeuing the next enabled task in-line and performing it. For our application we need to be able to wait before processing certain orders based on their dependencies and characteristics. Rather than have a different data-type and handler for every type of order, we took the approach to consolidate all order types into a single order data-type that has a field that specifies the transactionType. The orderHandler can be considered more of a wrapper function as it checks the transactionType of the order it is to perform and send it off to be handled uniquely based on the checked value. While market orders, are executed almost immediately after being placed, stop and limit orders may not be executed for quite some time. Whenever a task needs to be performed asynchronously, the task is entered into a designated portion of a Redis database, configured as a queue. Background "workers" (processes) perform tasks as they arrive. There are specifically two dedicated processes named Worker1 and Worker2, dedicated to order processing and UserSummary sending respectfully. Worker two runs every 24 hours and is responsible for populating a list of one UserSummary task for each user. In order for the UserSummaryController to obtain all necessary information on user and league performance information. The Performance Summarization objects are invoked to handle the retrieval of those specific stats, with the retrieval being handled by the DatabaseInterface object.

Asynchronous Subsystem Diagram

Object Constraint Language

To separate topics in this section, we will talk about each grouping of classes one at a time.

Order Package

The first section of the "Order Package" stub is a class that deals with handling orders. The constraints for this class are the usual – the function will work as long as they are given a correct order. That means that it has to be from a league that is in the boundary of it's date, if the order is a buy it must be under the constraint of how much money the user has and if it is a sell they must already own that much stock.

As of the actual order class, most of the variables in the class are but integers. The constraint on these integers is that all of them are positive, and league_id and portfolio_id must already exist. Also, once both variables are filled, placedDateTime must be before filledDateTime.

Resque

Resque is the simplest class in the asynchronous system as its only constraint is that it must have a valid order passed in.

Performance Summarization

The performance class has constraints on the variables startDate and endDate. These two variables only have the constraint that startDate must be earlier than endDate. It also has the constraint that the numTransactions variable must be a non-negative integer. The rest of the variables related to Performance Summarization are integers that must also be non-negative.

Mailer Package

The mailer package class is a simple class that will succeed as long as the orders constraints went well. All of the variables in UserSummaryMailer are simple to check if they are correct, they are all strings that should not be NULL or they are integers that should represent an ID so they must be non-negative.

Database Interface

Every function in this class can do a simple check to see if the object that was passed in exists. All of the functions deal with operations on an object so checking for existence is all they will need as constraints.

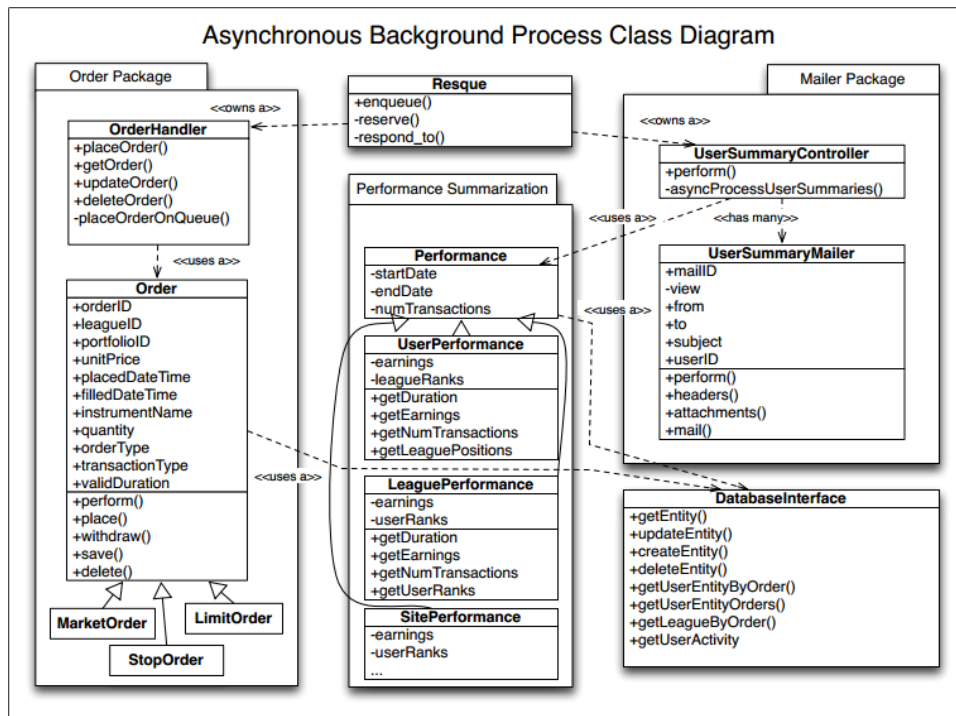


Figure 8.1: Asynchronous Class Diagrams.

Attribute Table

Concept	Attribute	Meaning
Task Queue	resque	resque is a ruby gem that queue's Order Handling jobs and processes them first in first out.
Order Handler	orderHandler	function responsible for processing orders.
Order	order	data type that contains order details to be processed by the orderHandler.
Mail Controller	UserSummaryController	handles queueing userSummaryMailer tasks and then executing them.
Mail Sender	UserSummaryMailer	handles the generation and sending of a single User Summary.
User Performance Retriever	Performance	retrieves a user's performance for the User Summary.
League Performance Retriever	LeaguePerformance	retrieves a leagues performance for the User Summary.

Table 8.1: Attribute Table.

9 System Architecture and System Design

9.1 Architectural Styles

Capital Games was designed to conform with several well-established software design principles. Some were chosen because of the software technologies employed (ie an MVC-based web framework), others represent a natural evolution of the needs of the system.

Model-View-Controller

Our philosophy in designing our website is to maintain a separation between the subsystems responsible for maintaining user information and those responsible for presenting it, in conformation with modern software engineering practice.

Therefore, we employ the Model-View-Controller (MVC) architecture pattern. In MVC, a View requests from the model the information it needs to generate an output; the Model contains user information; and the Controller can send commands to both the views and the models [18].

This approach has made site design easier, by abstracting the interface specifications from the system responsibilities. The Views and Models each know only what they need, while the Controller and associated subsystems perform all the “business logic”. The only complexity added by the decision to employ MVC is that updates to system components often have a ripple effect and require numerous modifications elsewhere in the system.

Representational State Transfer

As a well-designed web application, Capital Games conforms with the universal practice of employing RESTful design principles. RESTful design dictates, amongst other constraints, that a platform have a client-server relationship with the user (see below), that the interface is uniform, and that all information necessary for a request can be understood from the request sent to the server [11].

We strive to keep the interface as uniform as possible so that it is clear to the user how he is interacting with Capital Games, on a multitude of levels. For example, when purchasing a group of stocks, a user may graphically “click on” a submit button for a certain order, but in effect he is also submitting an HTTP POST request with appropriate form data to the Orders resource.

This identification of resources creates a tradeoff. On the one hand, all RESTful architecture must be designed at once, so that all resources are identified simultaneously, and the state transfers are possible to each of them. On the other hand, once resources are properly identified, the distribution of responsibilities is trivial for every possible interaction.

Data-centric

As a financial trading platform, Capital Games revolves around user data. To simplify access to that information from a variety of systems and to organize the data coherently and with the possibility of rapid retrieval, we eventually store all user data in a relational database. In this way, advanced queries can be performed on sets of data, both in application layer logic as well as by database administrators. Additionally, storing user data outside of a particular program's memory space enables subsystems which exist outside of the current application layer to also have access to the data. This additionally presents greater flexibility in terms of scaling site infrastructure.

Client-Server

By its definition as a web application, Capital Games follows a client-server model. The client, a user, interacts with the server, the various systems encapsulated by Capital Games.

9.2 Identifying Subsystems

As Capital Games exists as a website, a natural division of subsystems arises: front end and back end. Front end essentially describes all the computations and objects that exist on the user's side of interaction with our application, and back end describes all the computations and objects that exist on the server's side. It is exceedingly simple to determine which parts of our system belong in the front end in the back end. We will also define another subsystem called "External" which will contain all the pieces necessary to our application but not technically a part of it. A high-level view of our system in the form of "packages" or subsystems follows on one of the next few pages.

As it turns out, we can go deeper into our system to define subsystems within the back end. Though the front end is relatively simple, the back end of our system is where most of the computation and interesting events occur. There are two major subsystems as have been described in previous sections of this report: financial data retrieval and the queuing subsystems. In addition to these two subsystems, the database and the controller exist within the back end, but it does not seem appropriate to further include them in another subsystem, as they are essentially separate, stand-alone packages that interact with or call upon the other packages within the system.

The financial data retrieval subsystem is the simpler of our two subsystems. It only requires the ability to handle requests given to it by the controller (requests ultimately generated by a user) and the ability to fetch data from Yahoo! Finance in response to a valid request. The queuing system is only somewhat more complicated, needing background processes to monitor outstanding tasks, an Action Mailer object to handle sending e-mails to users, and an order handler that can understand and process orders. Though the controller facilitates all their interactions with the rest of the system, these two packages dominate most of our application design and are the backbone of its functionality.

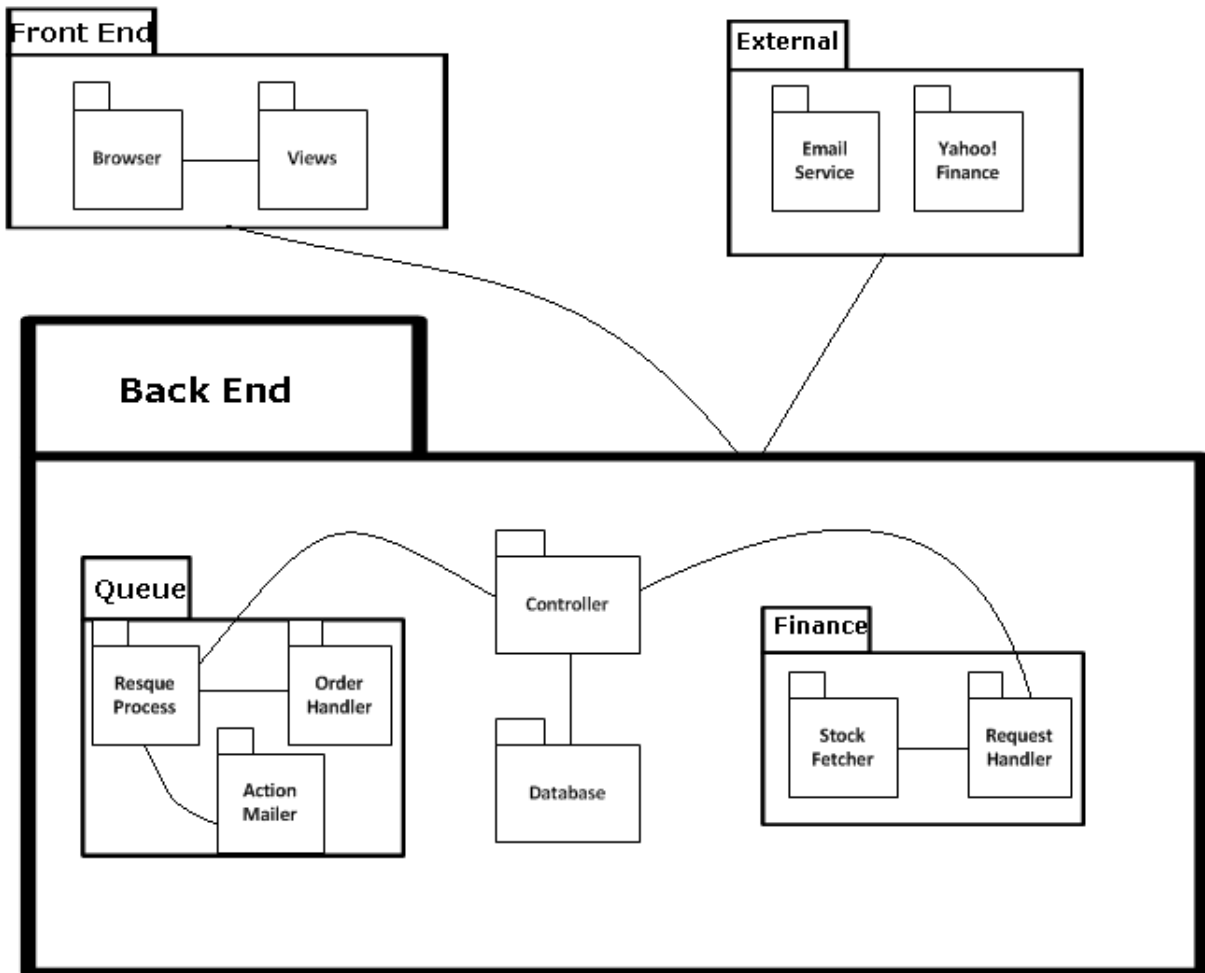


Figure 9.1: The UML package diagram for our system.

9.3 Mapping To Hardware

When it comes to web design, there is a standard on how hardware is mapped. All front end parts of the system run on the user’s machine (be it a computer, tablet, or smart phone), and all back end parts of the system will run on a server owned by the developer or the developer’s company. This follows from the architecture of the web, and there is really no way to deviate from it. To clarify the hardware mapping of our system, a diagram is included within the next few pages.

9.4 Persistent Data Storage

As described previously, Capital Games is data-centric and therefore cannot exist without a robust mechanism for persisting user data between “uses” of the system.

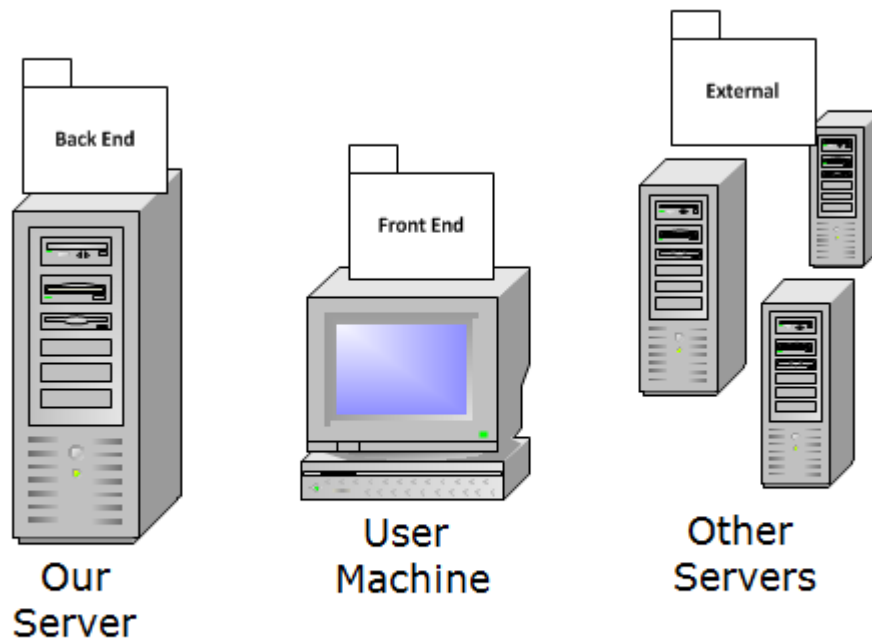


Figure 9.2: The hardware mapping for our system.

Various methods exist for the persisting of data. They range from serialization of system state variables into files to storing all data in the form of elaborate relational database systems, and anything in between. Capital Games primarily makes use of relational and non-relational databases.

In a relational database, stand-alone sets of data are placed into indexed tables stored in computer memory, with each row in a table representing a single set and each column representing a single attribute. A table can have a very large (sometimes even infinite) number of columns and rows. A database possesses many interrelated tables which are cross referenced by their indices (called primary and foreign keys). These relations allow complex queries against tabulated data [19].

For example, consider the figure shown previously, repeated here. A league is the most fundamental data structure of Capital games, yet it is not aware of its users, their portfolios, or of any orders associated with them. To retrieve these data, a query can be performed which pivots around the indices relating the tables. To find out a list of users participating in a league, one could take that league's index (not shown in figure for brevity) and search for all investors with that index; then take the user indices contained in the resulting query and dereference them to identify the original users.

Additionally, Capital Games makes use of a so-called “NoSQL”, or non-relational database. The format of such a database is practically unrestricted, and data need not belong in tabular format. This approach exchanges speed and scalability for querying power [20]. Capital Games utilizes the Redis NoSQL database to store outstanding queued jobs. This structure was chosen so that the database store can be seamlessly scaled across many machines if need be, and because of

its light weight.

9.5 Network Protocol

Though it is not necessarily an interesting topic to discuss for our project, it is none the less important to take note of. Because Hypertext Transfer Protocol (HTTP) is the predominate communication protocol distributed throughout the internet, it is critical that our website relies on it to make requests and send information between our user and system. Really, there is no other option if we desire Capital Games to be successful. HTTP is already a strictly and well defined protocol; for a description, see [this reference](#).

9.6 Global Control Flow

Execution Order

In general, our system is event-driven in terms of execution. As far as the user is concerned, our server sits and waits for a request to be made by a user accessing some part of our website. Though this is a simplification of the actual model, it is a good description of the general order of events within our system. The users can, nearly in any order, access different parts of our websites, search different companies, place different orders, etc., at their will. Any of these actions generate a request to our server, which then creates the necessary views, enacts the necessary computations, and takes any other necessary actions to facilitate the request.

To some degree, however, there are some procedures that drive our system as well, which force users to experience certain things in a predefined order. I will identify a few of these procedures hence:

- Registration: Before any user can begin browsing our site and joining leagues, they need to make an account.
- Order placement: Before a user can place an order, they need to join a league.
- Tutorials: When a tutorial is initiated, each user will experience the tutorial in the same order as all other users, excepting them terminating the tutorial prematurely.

However, on the whole, our system is still definitively an event-driven one.

Time Dependency

Real-time is very important to our system, though it does not entirely define it. While the user browsing our website is a real-time experience, there are a lot of back-end computation and processing that occur on our server based on real-time timers. In addition, as our system is strongly reliant on the stock market, which has certain times of operation, real-time matters quite a bit. I shall identify the timers present in our system:

- E-mail Timer: Based on the user's set preferences, they can receive periodic e-mails from our system describing their portfolios' progress over the last period, which can be set to daily or weekly.

- Market Open and Close: The stock market is only open and closed during certain times of the day, so our system must rely on these times to limit the placement of orders by users.
- Resque Process Check: As described earlier in our report, many of our system’s tasks are carried out by a queueing subsystem. In short periods, this queueing process must check if there are any outstanding tasks to operate upon. The period is as yet defined, but will be chosen for a balance between ensuring quick execution and reasonable server load.

Concurrency

There is a bit of concurrency within our system. Outside the main stream of execution with potentially parallel gets and posts from users’ browsers, this concurrency occurs mostly within the queueing system earlier mentioned. It is relatively simple; there are persistent processes that handle order processing and e-mail updates. As these are entirely separate functions, there is no need for synchronization between these two threads of control. Synchronization between these threads and the rest of our system (i.e. the user interactions with the browser and the browser’s interactions with the controller) to ensure that no data is being altered by separate entities at the same time is enacted through Ruby’s including protection functions—mainly flock (file lock).

9.7 Hardware Requirements

The hardware requirements for Capital Games are minimal on the client side, and moderate on the server side.

Internet Connection

The server needs to have an internet connection. Because all data are transmitted as text, it is technically possible for the server to function on even a low-bandwidth connection. Obviously this is not ideal and low bandwidth can increase server latency during peak use hours.

Disk Space

Under the current configuration, Capital Games does not commit any additional resources to the server’s disk storage during runtime. Rather, all data are stored to memory, and only backed up to the disk. Therefore, the disk requirements for Capital Games is simply the sum of the storage occupied by all program instructions for the system, or approximately 1GB at the time of this writing.

System Memory

Because all runtime data are stored to the server’s memory, as well as the space in memory occupied by the actual system runtime, having a large amount of “headroom” is vital to the performance of the application. Although it is hard to analyze performance requirements of an application that is still in active development, empirical evidence from users of similar technology make a few key observations. First, the amount of memory consumed by an idle application can work out to be over 100MB. Next, the active application will load copies of its database-stored information into memory in order to operate over it, which can result in large spikes in memory usage. Finally, operating over the loaded data itself can consume a large amount of memory. This is in addition to any memory occupied by the databases and worker processes [21]. Therefore, having at least 200MB

should be the minimum required for internal testing of our application. Obviously, increasing user base will exponentially increase the memory requirements of our application.

Client-side Hardware Requirements

The user needs to have an internet connection in order to interact with the server remotely. Although the intended use of the system entailing the use of a graphical web browser strongly encourages the use of a monitor (as mentioned previously, the responsive nature of the application means that screen resolution is not a limiting factor), it is also possible for technically proficient users to interact with the server through its RESTful resources. At some future date, we may publish the official RESTful API for Capital Games, but at this point, interacting purely through a command line interface is discouraged.

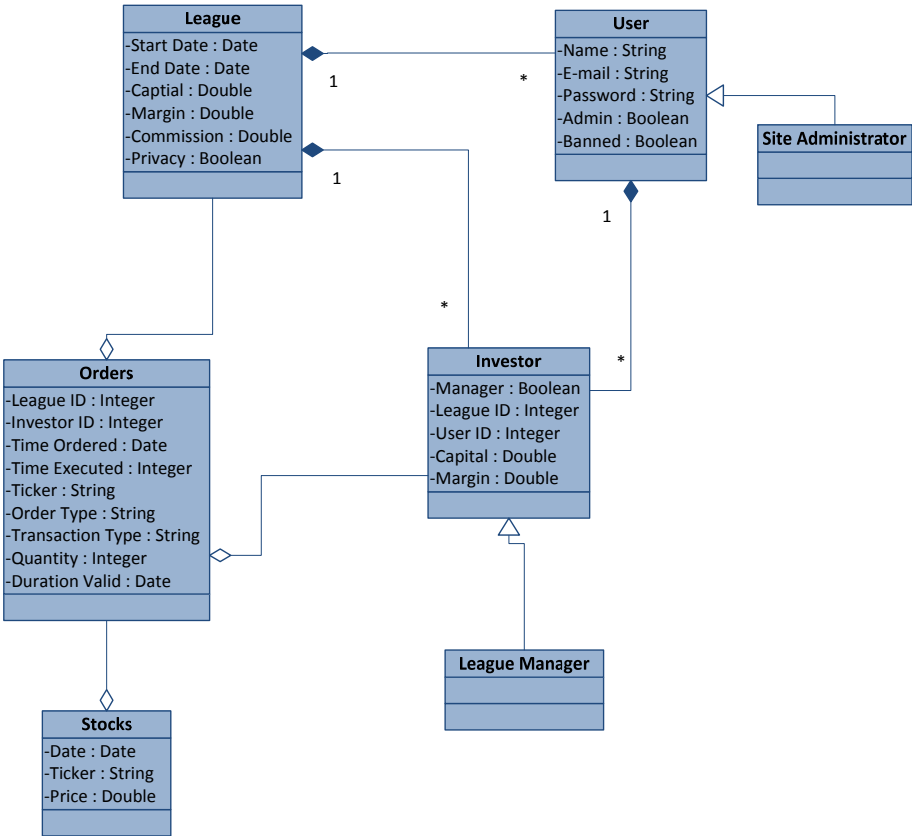


Figure 9.3: The old design for the database for Capital Games.)

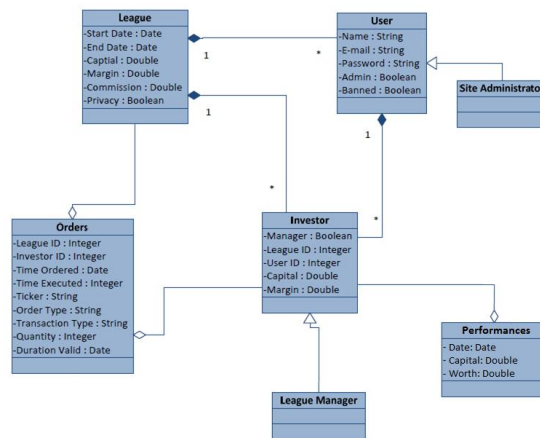


Figure 9.4: The format of the relational database schema implemented by Capital Games for its core features. Notice that the "Stocks" table has been removed and a "Performances" table has been added. Originally, the plan had been to store the historical performance of stocks as a way of caching data to later perform quick calculations of a portfolio's historical worth for the purpose of graphing and displaying it to users. However, we decided instead to cache the daily performance of portfolios directly in the Performance table as it was both more efficient and simplified calculations.

10 Data Structures

10.1 Table

By design, databases used for web apps are stored in tables. Via SQL querying, tables are relatively efficient for both time and space. For each object “type”, there a single table used with relationships as depicted in the diagram above. Each row in a table represents an instance of that object and each column represents an attribute of that object. For storing and retrieving data over the Internet, a table is pretty much the only way to go when integrating with a website.

10.2 Queue

The nature and description of our queue is discussed in the asynchronous processing section described in an earlier section.

10.3 Tree

One more data structure that will be implemented for our system is a tree. As described earlier, the finance adaptor will need to make use of the information on EoDdata so that companies can be validated for existance before going through a trade. EoDdata does not come with a simple solution to find out if a single company is in existance and neither does Yahoo! Finance, therefore we must build a function that will do this for us. We could scan through every company on EoDdata everytime we need to validate, but that would waste too many resources. Instead, we decided to keep a local copy that will have very fast lookup of companies. The way that this will be implemented is to keep a tree in which the nth level of the tree represents the nth letter of the company symbol. For example, if the company with symbol “GOOG” exists, the head will point to G, which will point to O and so on. The last letter in the symbol will also have a boolean value to denote that this is the end of a symbol so that there could be companies with the same letters but one with an extra letter at the end. The reason for using a tree is because it will have a time complexity equal to the length of the symbol, which is a very small value, and a space complexity much smaller than if we used a structure such as a hash table. All we need for this tree is the ability to add a symbol, remove a symbol and check if a symbol exists. With these three simple commands, we can create our tree and maintain it to stay up-to-date.

11 User Interface Design and Implementation

11.1 Finalized Product

Here, we reflect on the changes that were ultimately made to simplify our design.

Pages for a visitor

When a user first comes to see our website, they will be greeted with this page:

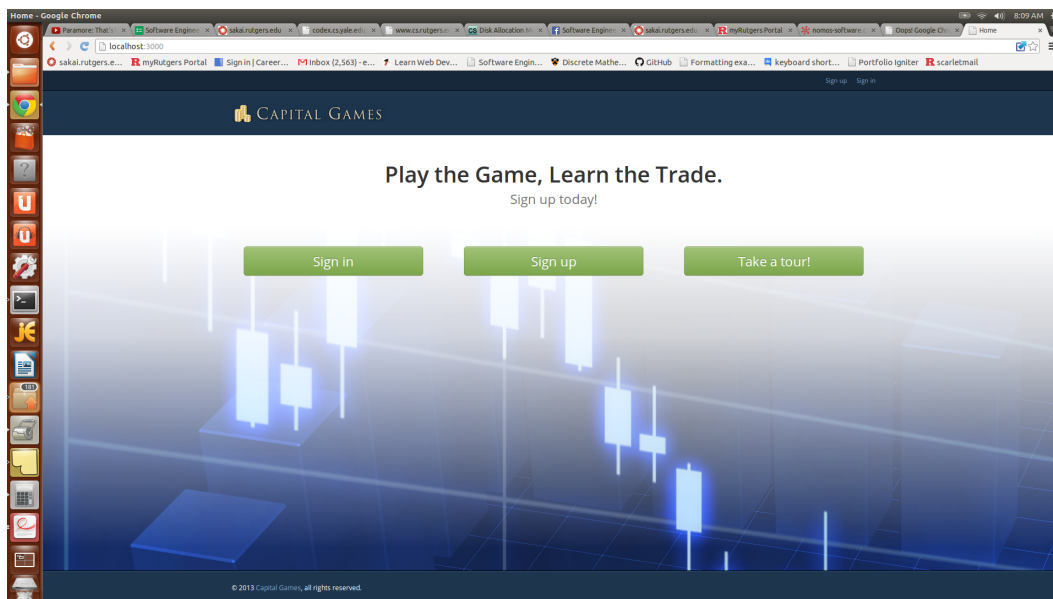


Figure 11.1: Homepage of the website

If they click on the sign-in option, they will be taken to this page:

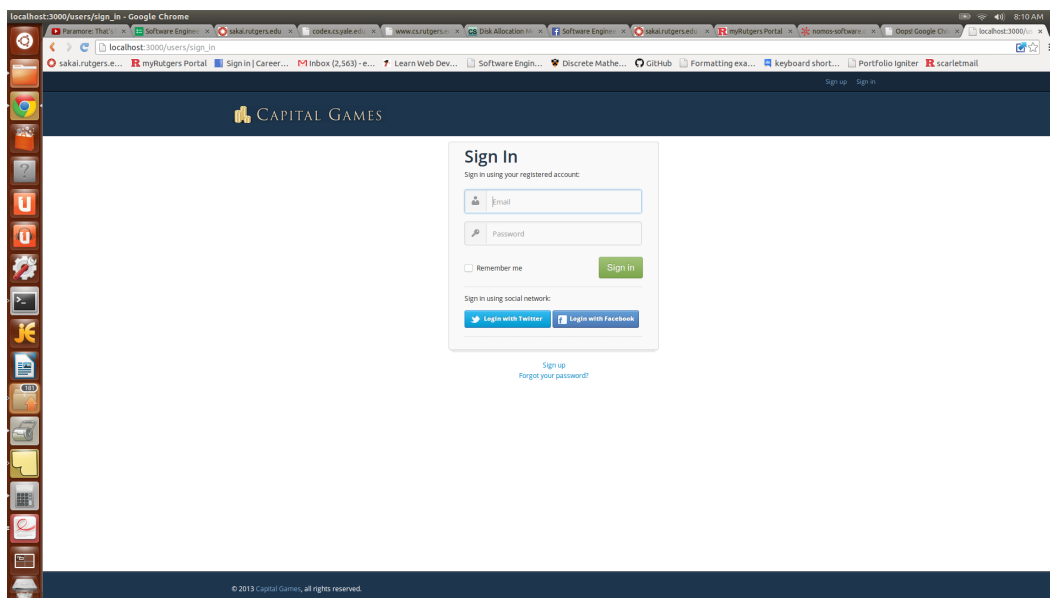


Figure 11.2: The login page

Alternatively, if they click on the sign-up page, they will be greeted with this page:

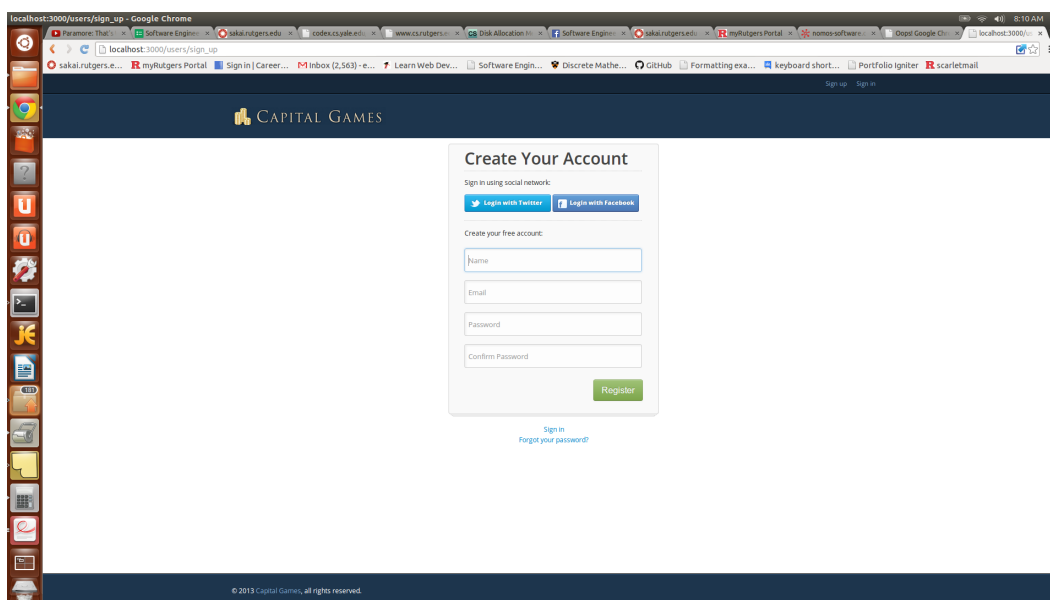


Figure 11.3: The sign-up page

Lastly, if they click on the "Take a Tour" button, they will be taken to our "Learn" section of the website. It's a simple text guided tour through our website that lets the user learn a little bit about the way our site works before (or after) signing up.

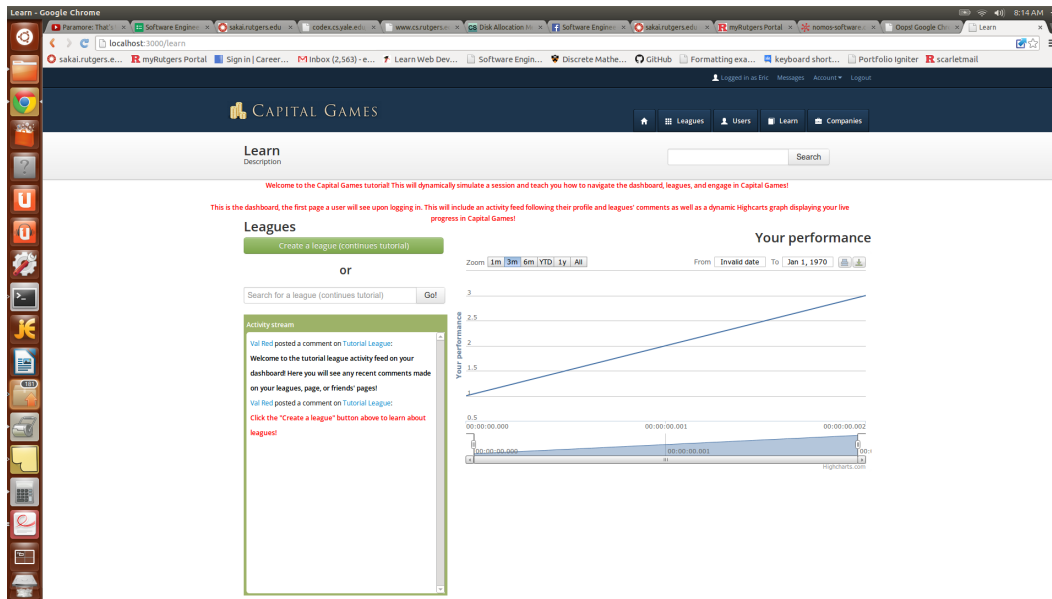


Figure 11.4: The first tutorial

The Dashboard

Once a user is logged in, they will be greeted with their dashboard. This is a customized page that tells them information about some events that have been going on in the site as well as the progress in all of their leagues.

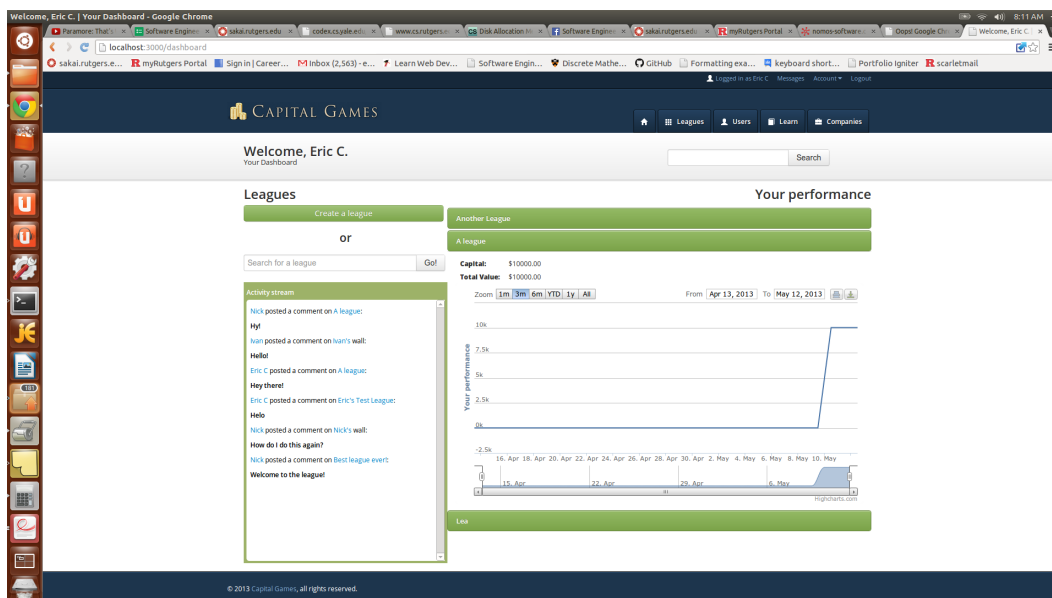


Figure 11.5: The Dashboard

Leagues

The final design for leagues was simplified the most. If you want to find a league, you visit the leagues page. There you can search, filter through, or create a league.

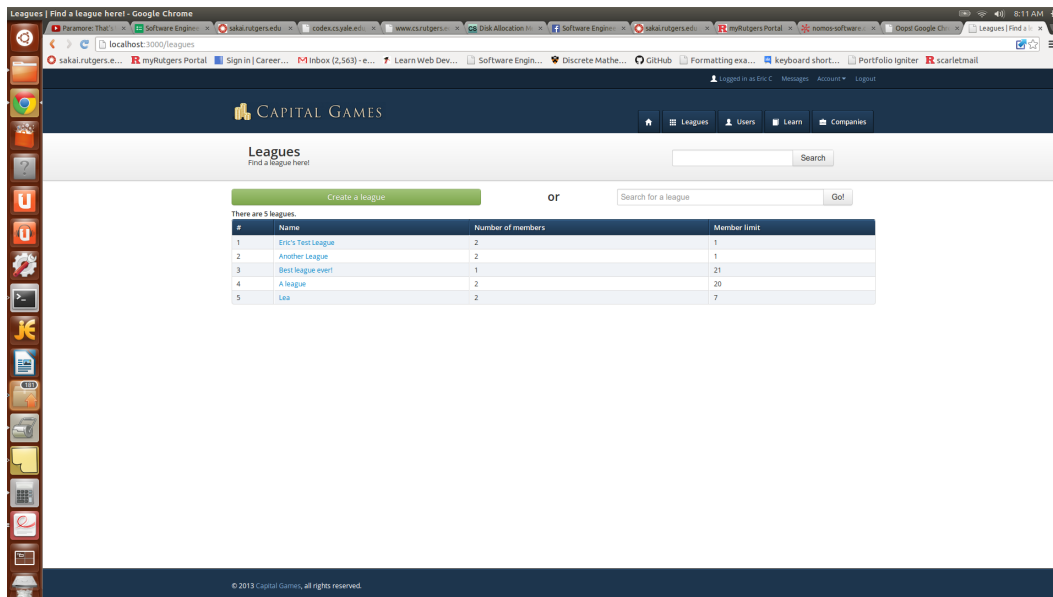


Figure 11.6: The leagues page

If you want to create a league, you will be greeted with a very simple page for making a league.

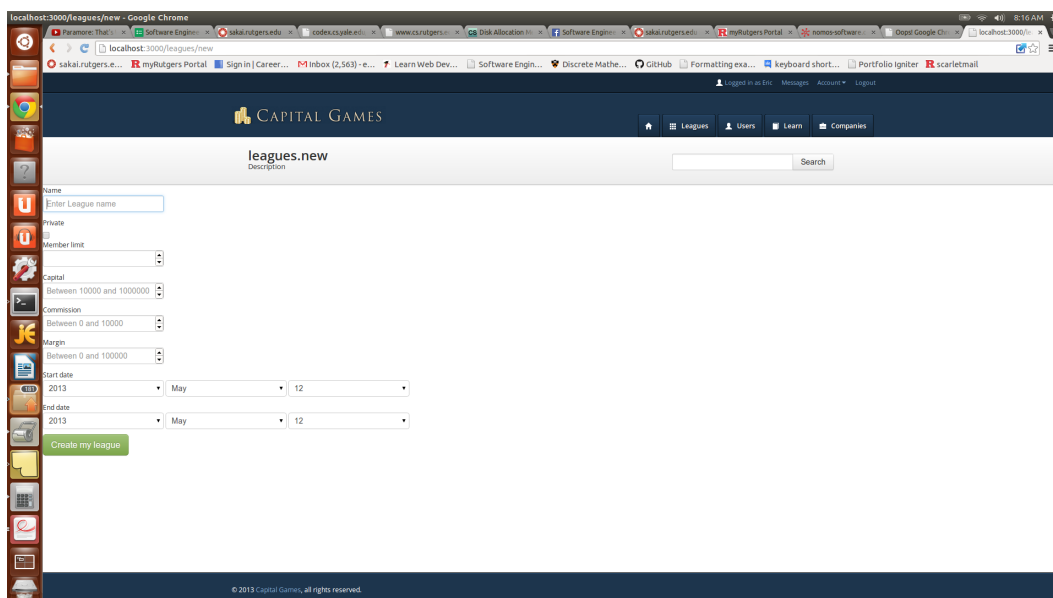


Figure 11.7: The create a league page

Once you have a league or are in it, you can check out how users are doing by checking out the table with users sorted by their rank or see comments that other users have posted.

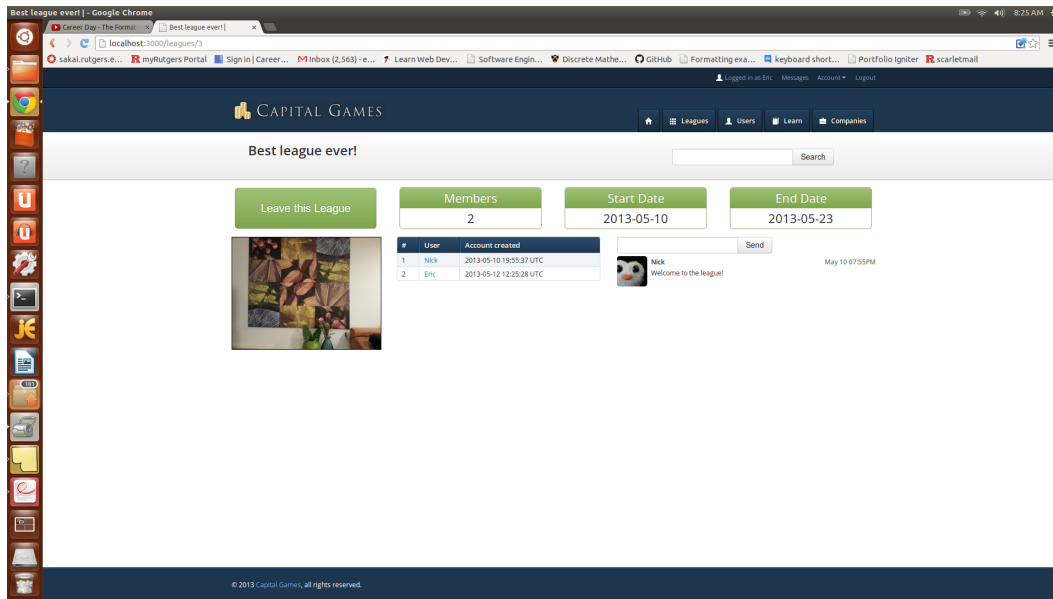


Figure 11.8: A league page

If you're an admin of the page, you can check out the league settings, where you can change the name, description and league picture in the "Basic Settings" tab, and you can delete the league in the "Delete League" tab. The others were not implemented yet.

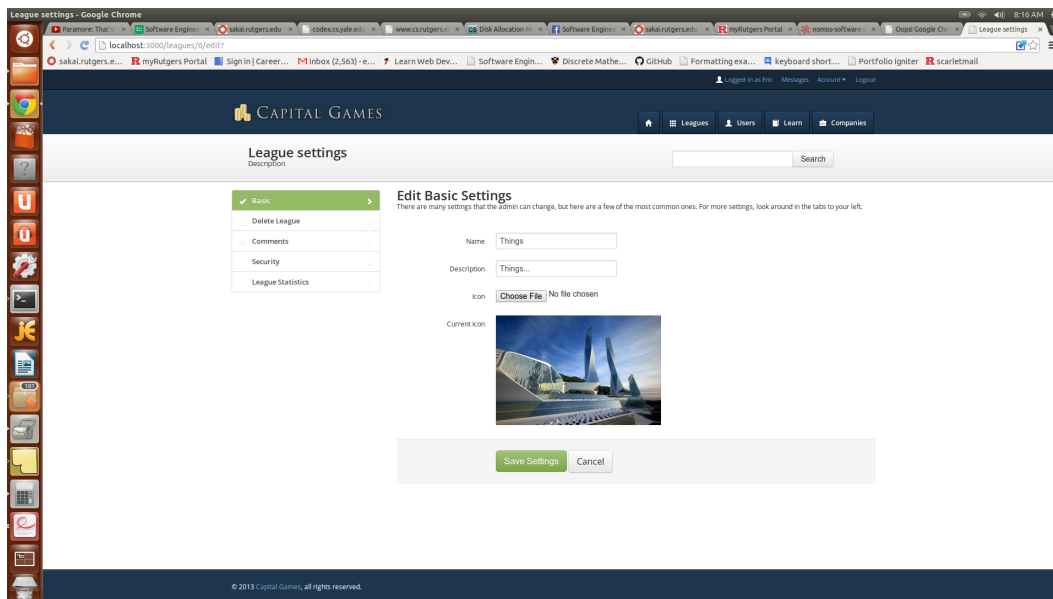


Figure 11.9: The league settings page

If you want to check out a user's performance in the league, you just have to click on their name on the table in the league page. Here, you can see their rank, worth, and recent orders they have made.

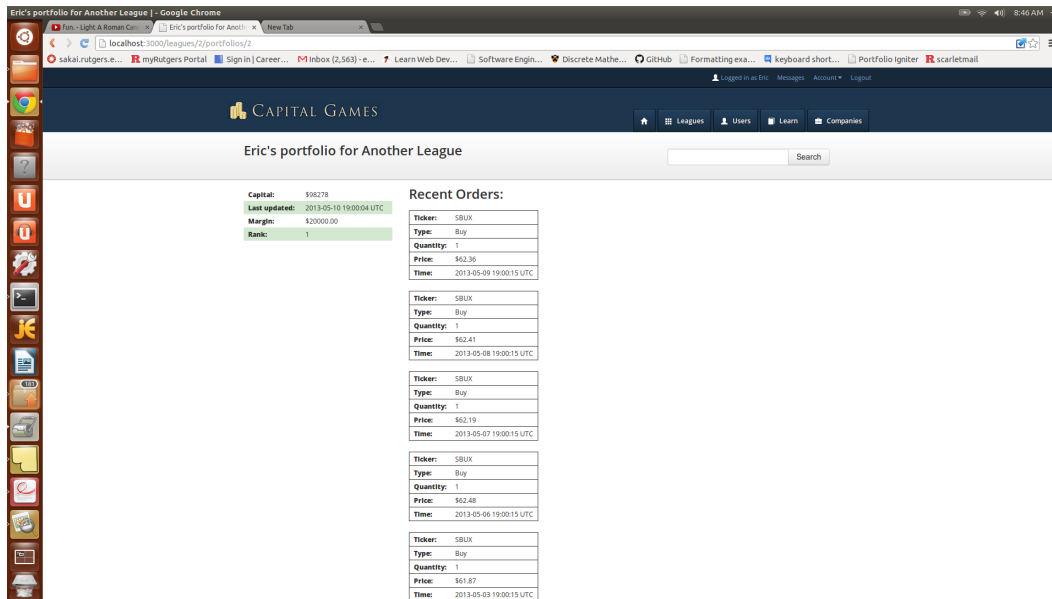


Figure 11.10: The portfolio page

Users

With our specialized social system, it's easy to keep in touch with other users. One way you can do that is by dropping a comment on their wall. Their wall is a place that displays their profile picture, their leagues and their comments.

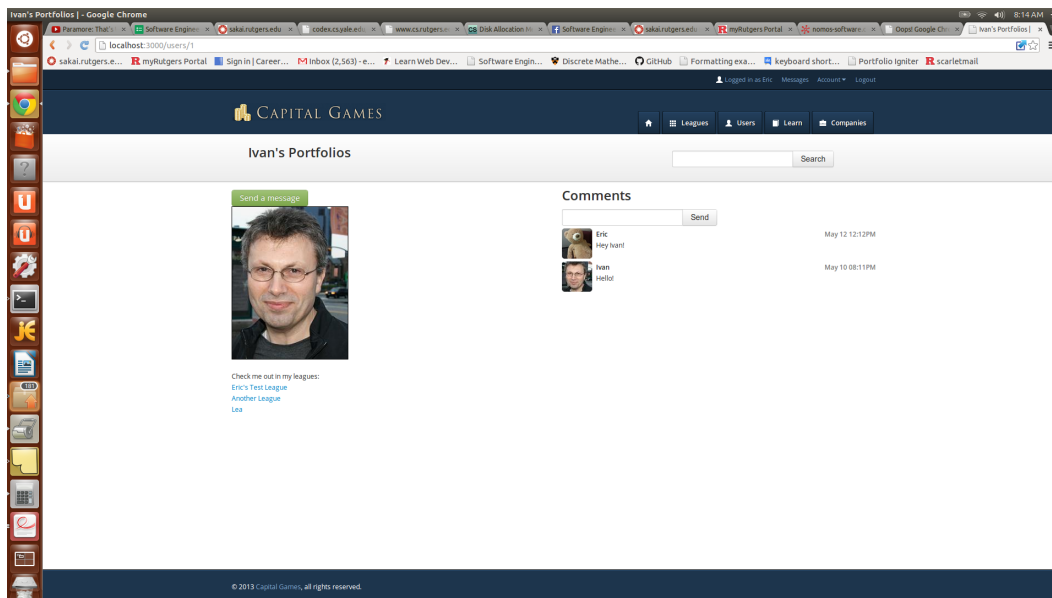


Figure 11.11: The user page

If one wants to change their settings, they can look up at the top bar in the header and click on "Account" and then "Settings". This will allow you to change your name and your profile picture.

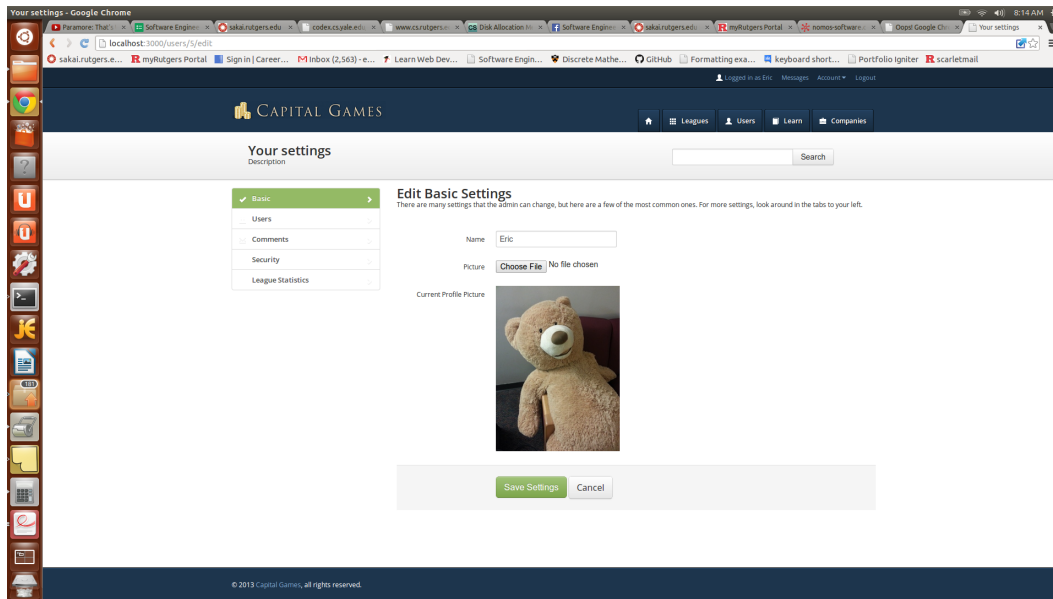


Figure 11.12: The user settings page

Companies

The last part of the website that we will highlight is the companies pages. You can go to the "Companies" page of the website and you will be greeted with recent news about the market and the ability to search for a company by symbol.

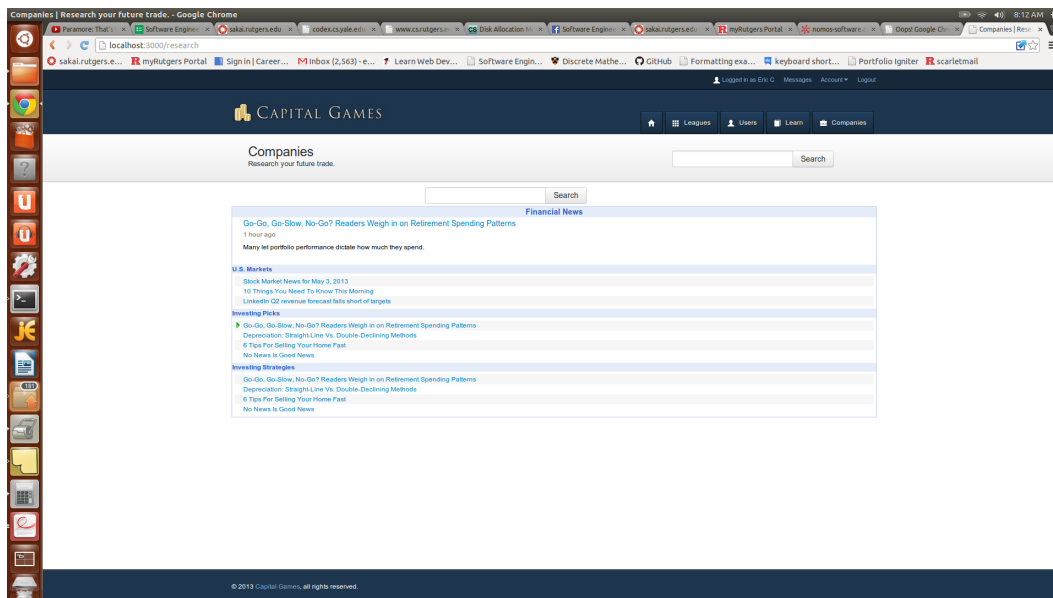


Figure 11.13: The companies page

Once you search for a company, you will find a lot of relevant information about that company to help you in your purchase of stocks.

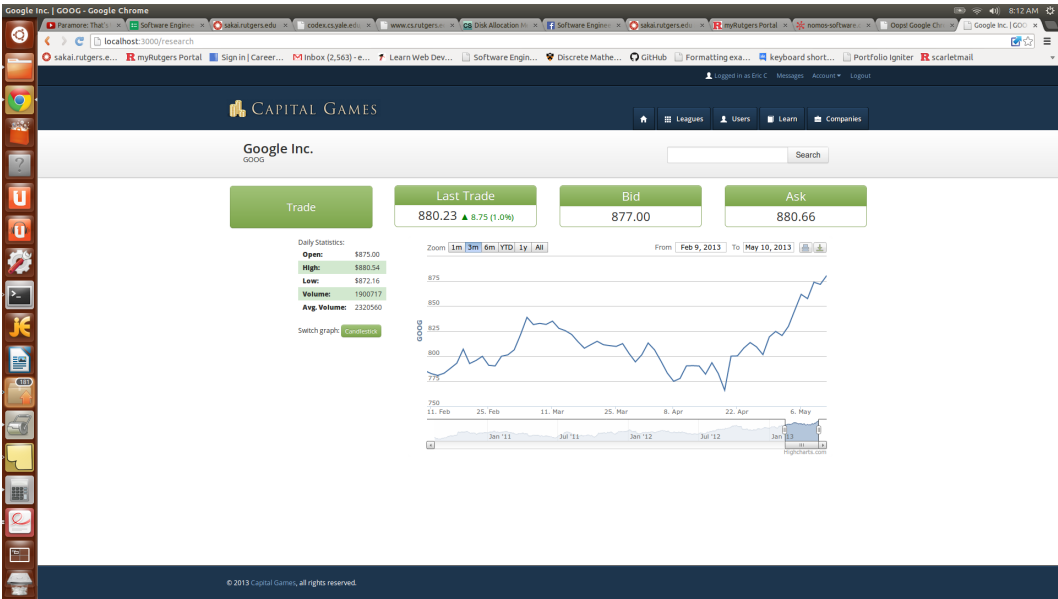


Figure 11.14: A company page

If you want to buy/sell a stock, you can press on the "Trade" button, which bring up this dialogue:

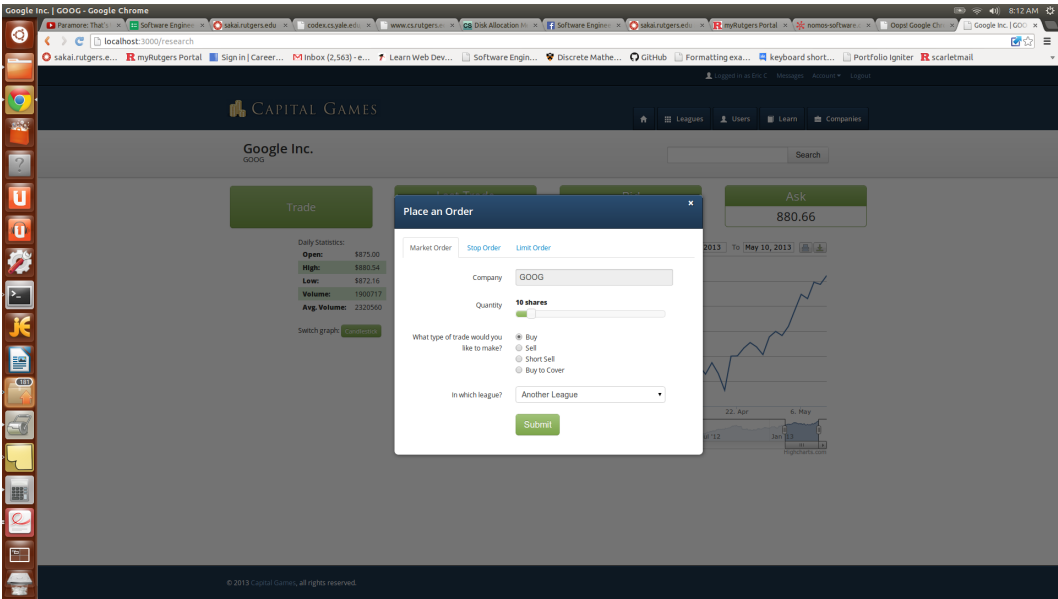


Figure 11.15: Buying or selling stocks

There are a lot of options to choose from there and they all work as the usual market does.

12 Design of Tests

No application is complete until it has been tested as thoroughly as possible for security holes, broken functionality, and any other lacking features. Shipping without testing is a guarantee to have all manner of bugs and security holes. However, even with thorough testing, it is not usually possible to find and resolve every flaw before shipment. To this end, developers utilize *testing suites* to try and test programs efficiently and effectively. Tests can be designed for individual units and components as well as the broader system and the integration of the units. While not perfect for finding all flaws in a program (usually errors are discovered by looking for them, which generally requires either knowledge of an existing error or “luck” in an error making itself apparent during development), testing can serve to find almost all errors and flaws in an application.

However, developers face a dilemma. Developing an evolving application can cause existing tests to become outdated, while designing and running tests is time taken away from actually building the application.

A modern approach to this tradeoff is to build the feature set of an application around measurable, predefined tests [22]. In this technique, known as Test-driven Development, developers iteratively define tests for intended future features, confirm that those features are not yet implemented (by running those tests), and then implementing the solutions. Though this approach does not (generally) test for all possible interplay between components, it is usually employed in high-paced development environments such as ours, where the coverage provided is usually respectable enough to prevent most problems.

Accordingly, we first define the features and tests we plan on developing around, proceed to analyze the coverage offered by these tests, and then briefly discuss how we intend to test the integration of the components.

12.1 Test Cases

Due to project constraints, we cannot afford to thoroughly test existing packages for functionality we incorporated to streamline the design process. These packages include Ruby on Rails as well as Ruby gems (packages) for interfacing with Yahoo! Finance, various databases (ie MySQL, SQLite), the Resque queueing system, and other auxilliary package for Rails. Likewise, we cannot unit test the HTTP server we are using (Apache) and its Ruby extension (Phusion Passenger) or any of the databases. Rather we will focus on testing just the units of our application and their integration with each other.

Routing

As described earlier, Capital Games contains models for users, managers, administrators, trades, etc. Intrinsic to the Rails web framework we employ, most of these models are represented internally to the controller as “resources” [23]. At any point in time, any user (even a non-user!) could attempt to gain access to a resource to which they are not privileged, such as an administrator panel. Routing unit tests will confirm that only authorized users will be able to access restricted pages. As an extension of this premise, Routing unit tests will also confirm that pages with low privileges are accessible to all users and front-facing pages can be seen even without being logged in.

Database Models

Because of the data-centric design of Capital Games, protecting the integrity of the database entries is of the utmost importance. The Ruby on Rails framework has safeguards and validation for this purpose, but we still need to thoroughly unit test each of the models to ensure that only permissible combinations of attributes are able to be entered, and that proper error handling occurs to resolve attempts at improper attribute definition.

Queueing System

Capital Games heavily relies upon the queueing system to act as a computational highway for all asynchronous tasks. Due to the nature of this system we must prepare for race conditions; the different ways our data can be effected based upon the order of executing processes that are acting upon the queue. We will need to prepare a set of tests to express how the queue performs when open orders are altered by other processes during different phases of the queueing system. Based on our test results it might be necessary to implement data locking.

Finance Adapter

Whenever using external resources it is vital to understand the different ways in which they communicate not just when functioning as expected, but also when failing to perform properly. Since we do not have the ability to shut down the external Financial Adapter’s Servers we can not run tests that give us feedback on what functionality to expect on failure. This leaves us without the ability to test the Financial Adapter and instead pro-actively safeguard against failure. Due to this we must build a wrapper that anticipates all perceivable failures coming from the Financial Adapter.

12.2 Test Coverage

In order to attain full functionality of Capital Games without bugs, we must be sure that none of its parts have errors themselves. Due to many dependencies such as other running processes, system states, and transitions, the same test will need to be preformed for each possible configuration to make sure that each part works in every possible scenario that it can be ran. This will require extending certain tests to run at the same time as background processes, and having parts called from all possible initiating parts. When working with integrated parts it is not simply enough to assume that parts will work once integrated just because they work independently. By extensively testing each possible run case we ensure that there are no points of failure once the system is launched.

12.3 Integration Testing

In order to achieve the most thorough testing, Capital Games will be tested using the bottom-up strategy. Each part of Capital Games that we wrote will be extensively tested individually first. However simply testing each part individually is not enough due to race conditions and other integration issues that may exist in the systems described above. Because of this, parts must be tested after integration as well. Knowing that functionality is state specific and transition specific for any state machine, each test must also be ran in all possible states. In addition to all previously listed conditions, tests need to be preformed at different times to make sure that functionality during backend asynchronous tasks do not have any bugs. We have chosen the bottom-up testing strategy based on the principle that bugs at the bottom level will dictate bugs at the top level, while bugs at the top level may very well be independent of bottom level performance. By carefully analyzing every part to part integration we can work our way up to a flawless design.

12.4 Test Cases

Test-case Identifier: TC-1 Function Tested: Routing Pass/Fail Criteria: The test passes when a user is rejected from a page that they should be restricted from viewing. The test fails if a user can access a page that they should not have access to	
Test Procedure	Expected Results
<ul style="list-style-type: none"> • (pass)Access unrestricted page • (fail)Access restricted page 	<ul style="list-style-type: none"> • On Pass: The page being accessed is sent to the web browser • On Fail: An access restriction message is sent to the web browser

Figure 12.1: Test Case 1.

Test-case Identifier: TC-2 Function Tested: Database Models Pass/Fail Criteria: The test passes if Database Modes reject impermissible combinations of attributes, the test fails if it allows them.	
Test Procedure	Expected Results
<ul style="list-style-type: none"> • (pass)enter permissible data • (fail) enter Impermissible data 	<ul style="list-style-type: none"> • On Pass: impermissible data is rejected • On Fail: impermissible data is

Figure 12.2: Test Case 2.

Test-case Identifier: TC-3 Function Tested: Queueing System Pass/Fail Criteria: The test passes if data is not corrupt during asynchronous processes and fails if it is corrupt	
Test Procedure	Expected Results
<ul style="list-style-type: none"> • Make order not during asynchronous queueing process (pass) • Make order during asynchronous queueing process (fail) 	<ul style="list-style-type: none"> • On Pass: Queue data is not corrupt • On Fail: Queue data is corrupt

Figure 12.3: Test Case 3.

Test-case Identifier: TC-4 Function Tested: Queueing System Pass/Fail Criteria: The test passes if a User Summary is not sent to an inactive user, and fails if a NULL user summary is generated.	
Test Procedure	Expected Results
<ul style="list-style-type: none"> • Request user summary for existing user(pass) • Request user summary for inexistent user (fail) 	<ul style="list-style-type: none"> • On Pass: User summary is retrieved • On Fail: No user summary is generated

Figure 12.4: Test Case 4.

13 History of Work, Current Status, & Future Work

13.1 History of Work

Throughout the semester we completed several of our planned milestones in a punctual, thorough, and consistent manner.

Our first planned milestone, completing the Report 1 Part 1 prior to 12 February 2013, was met on time. We continued to meet our report deadlines for Report 1 Part 2 and the full, compiled Report 1 by 18 February 2013 and 22 February 2013, respectively. For the second report, we successfully met our deadlines for Report 2 Part 1 (consisting of Sequence Diagrams, timing and communication diagrams) and Report 2 Part 2 (consisting of Class Diagrams, Interface, Architecture design, data structures, UI, tests and implementations) on 1 March 2013 and 8 March 2013, respectively.

As we met our initial Report deadlines, we also simultaneously began work on the initial build of *Capital Games*. We began with deploying our server environment, which took place between 22 February and 2 March 2013. While we were initially weighing our options between dedicated virtual private server (VPS) and Heroku, we determined that a dedicated virtual private server would better suit our needs and we successfully deployed it by 2 March 2013. In this time, we also finalized our plan for mockup-based views and the CSS/HTML plan.

Populating the server with Ruby on Rails and Gem plugins, however, took a little longer than anticipated. This is because we needed to confirm and test for full compatibility between our server's Ubuntu operating system, Debian 6 "Squeeze" before deploying Ruby on Rails and begin programming on it. This extended beyond our initial range of 1 March 2013 to 8 March 2013 and was completed by 13 March 2013.

We completed our full Report 2 deadline by 17 March 2013. After this point we shifted gears and committed to having *Capital Games* Alpha build prepared for our demo. In the weeks between 17 March 2013 and 2 April 2013 we met several of our objectives. The Yahoo! Finance API was implemented, the MySQL database structure was deployed and populated. Our routing plan was completed, our views (webpage layouts via the "Dashboard Admin" theme) were implemented, and we successfully got users, leagues, and portfolios at a working state in which data could be utilized between them in our routing structure. We even had our graphs via Highcharts and responsive UI at a working state! Thus we met our 2 April 2013 deadline and were able to present Demo 1 with a stable, presentable, and functional Alpha build.

Our biggest strength was establishing manageable goals that still built on the strengths and failures of our predecessors. We chose a superior UI but were not overly-ambitious with functionality, otherwise; this enabled us to meet our goals in a punctual manner. We almost fully implemented our orders system, in addition, so we were actually ahead of our initial goals for our *Capital Games* Alpha build!

13.2 Current Status

Currently, *Capital Games* is a functioning Rails application. We have an online version of our latest builds on the domain <http://capitalgam.es/> based on our Virtual Private server. It features working orders, leagues, portfolios, and user systems with a fully-functional and responsive UI that can be used in tablets and phones in addition to personal computer to be user friendly for anyone on a smart device. Most core functionalities have been deployed and the current status is debugging and optimizing our website to address orders in an asynchronous fashion to maximize the efficiency of our application.

13.3 Key Accomplishments

Capital Games unlocked the following achievements:

- A Ruby on Rails-based Framework for core Web Application functionality
- A responsive UI usable on smart devices based on the “Dashboard Admin” theme.
- A minimalistic and easy-to-use system for users, portfolios, and leagues
- A research page that draws news feeds pertinent to investment options
- An implementation of highcharts that presentably showed candlestick and line-graph data

Capital Games also deployed the following use cases:

- Buy Stock
- Sell Stock
- Query Stock
- View History
- View Portfolio
- Register
- Create League
- Submit Comment
- Join League
- Change League Settings
- Browse Companies

13.4 Future Work

The final stretch for a release-build *Capital Games* would be an immersive tutorial system activating upon user registration. This would have extensively took advantage of AJAX and our Rails framework but also is the most complex type of functionality to deploy on a live version of *Capital Games*, hence it would have required careful, slow, and surgical-level development. This was one of our major post-demonstration goals, but the level of sophistication for a fully-functional live tutorial upon user registration was highly impractical for the remaining time we had.

We also would like to perfect our asynchronous processing system for orders so that the stop and short options would work as true-to-life as possible. This was also one of our major post-demo goals, and while this is one of the major portions of our final demo updates, there are many more efficient practices in the field of asynchronous processing systems that would be able to expand our *Capital Games* long after we complete our final demonstration build.

While we intended to implement a full Reporting and Disciplinary functionality within a Site Administration suite, development on this fell in favor of the core functionality and orders, which required a lot more dedicated manpower and hours than anticipated with the problems of asynchronous processing as well as communicating with the Yahoo! Finance API and the Highcharts implementation.

Finally, to maximize user retention, another future work goal would have been to implement social media and e-mail notification options so users can be reminded to check and update their portfolios and leagues on a regular basis. A mailer system was intended, however, development on that particular item fell in favor of on-site notification and chat options, which were fully implemented by the Alpha into final demo release.

Our Rails framework for *Capital Games* was deployed so well, however, that it's expandability lends itself to a swift transition to these Future Work goals!

13.5 Project Management

Category	Points	Names					
		Jeff A	Eric C	Nick P	Jeff R	Val R	Dario R
Customer Statement	6 Points	20%	0%	0%	80%	0%	0%
Glossary of Terms	4 Points	50%	0%	0%	50%	0%	0%
System Requirements	6 Points	0%	33%	0%	0%	33%	33%
Functional Req. Spec.	30 Points	30%	10%	35%	0%	5%	20%
Effort Estimation	4 Points	0%	25%	0%	75%	0%	0%
Domain Analysis	25 Points	25%	25%	10%	20%	20%	0%
Interaction Diagrams	40 Points	10%	15%	15%	20%	10%	30%
System Arch. and Des.	15 Points	16.67%	16.67%	16.67%	16.67%	16.67%	16.67%
Data Structures	25 Points	0%	25%	0%	0%	75%	0%
History of Work	5 Points	0%	0%	5%	5%	90%	0%
Project Management	13 Points	3.85%	26.92%	30.77%	0%	19.23%	19.23%

References

- [1] Investopedia, “What are the minimum margin requirements for a short sale account?.” <http://www.investopedia.com/ask/answers/05/shortmarginrequirements.asp>. [Online; accessed 22 February 2013].
- [2] Investopedia, “Margin definition — Investopedia.” <http://www.investopedia.com/terms/m/margin.asp>. [Online; accessed 22 February 2013].
- [3] Investopedia, “Stop order definition — Investopedia.” <http://www.investopedia.com/terms/s/stoporder.asp>. [Online; accessed 22 February 2013].
- [4] Investopedia, “Limit order definition — Investopedia.” <http://www.investopedia.com/terms/l/limitorder.asp>. [Online; accessed 22 February 2013].
- [5] Investopedia, “Limit order definition — Investopedia.” <http://www.investopedia.com/terms/m/marketorder.asp>. [Online; accessed 22 February 2013].
- [6] Investopedia, “Short (or Short Position) definition — Investopedia.” <http://www.investopedia.com/terms/s/short.asp>. [Online; accessed 18 February 2013].
- [7] Investopedia, “Buy to cover definition — Investopedia.” <http://www.investopedia.com/terms/b/buytocover.asp>. [Online; accessed 22 February 2013].
- [8] Investopedia, “Bid-ask spread — Investopedia.” <http://www.investopedia.com/terms/b/bid-askspread.asp>. [Online; accessed 23 February 2013].
- [9] Wikipedia, “Responsive web design - Wikipedia, the free encyclopedia.” <http://www.investopedia.com/terms/m/marketorder.asp>. [Online; accessed 22 February 2013].
- [10] I. Marsic, *Software Engineering*. New Brunswick, USA: Ivan Marsic, 2012.
- [11] Wikipedia, “Representational state transfer - Wikipedia, the free encyclopedia.” http://en.wikipedia.org/wiki/Representational_state_transfer. [Online; accessed 3 March 2013].
- [12] Wikipedia, “Object relational mapper - Wikipedia, the free encyclopedia.” http://en.wikipedia.org/wiki/Object_relational_mapper. [Online; accessed 3 March 2013].
- [13] P. Ponzio, “Yahoo data download.” www.gummy-stuff.org/Yahoo-data.htm, 2004. [Online; accessed 3 March 2013].
- [14] Rails Guides, “Ruby on rails guides: Action mailer basics.” guides.rubyonrails.org/action_mailer_basics.html. [Online; accessed 3 March 2013].

- [15] Investopedia, “Perfect competition — Investopedia.” <http://www.investopedia.com/terms/p/perfectcompetition.asp>. [Online; accessed 23 February 2013].
- [16] Wikipedia, “Perfect competition - Wikipedia, the free encyclopedia.” http://en.wikipedia.org/wiki/Perfect_competition. [Online; accessed 23 February 2013].
- [17] Investopedia, “What are determinants of the bid-ask spread? — Investopedia.” <http://www.investopedia.com/ask/answers/06/bidaskspread.asp>. [Online; accessed 23 February 2013].
- [18] Wikipedia, “Model-view-controller - Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/wiki/Model-view-controller>. [Online; accessed 10 March 2013].
- [19] Wikipedia, “Relational database - Wikipedia, the free encyclopedia.” http://en.wikipedia.org/wiki/Relational_database. [Online; accessed 10 March 2013].
- [20] Wikipedia, “NoSQL - Wikipedia, the free encyclopedia.” <http://en.wikipedia.org/wiki/NoSQL>. [Online; accessed 10 March 2013].
- [21] D. Collective, “How much memory should a ruby on rails application consume? - Stack Overflow.” <http://stackoverflow.com/questions/2971812/how-much-memory-should-a-ruby-on-rails-application-consume>. [Online; accessed 10 March 2013].
- [22] Wikipedia, “Test-driven development - Wikipedia, the free encyclopedia.” en.wikipedia.org/wiki/Test_driven_development. [Online; accessed 12 March 2013].
- [23] Rails Guides, “Ruby on rails guides: Rails routing from the outside in.” guides.rubyonrails.org/routing.html. [Online; accessed 13 March 2013].