Design Specifications
for
"Capital Games"

Report 2: Part 3
Software Engineering
14:332:452

Team 2:

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

March 17, 2013
Version: 1

Contents

1 System Interaction Diagrams

1.1 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.

1.2 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

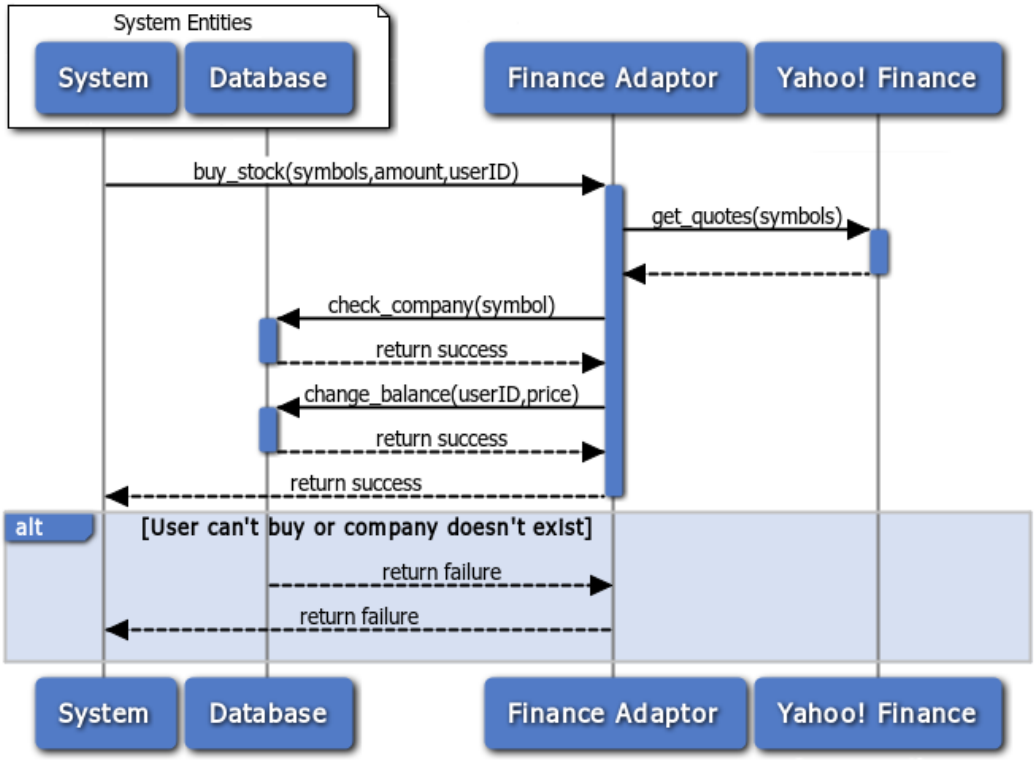


Figure 1.1: When a user buys a stock, the browser will inform the system of the transaction so that it can be approved. The system passes over the process to the finance adaptor who will check if the company is accepting trades, the current price from Yahoo! Finance, and if the user is able to afford 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 marked as a success and the system will be notified. (Related use case: UC-4)

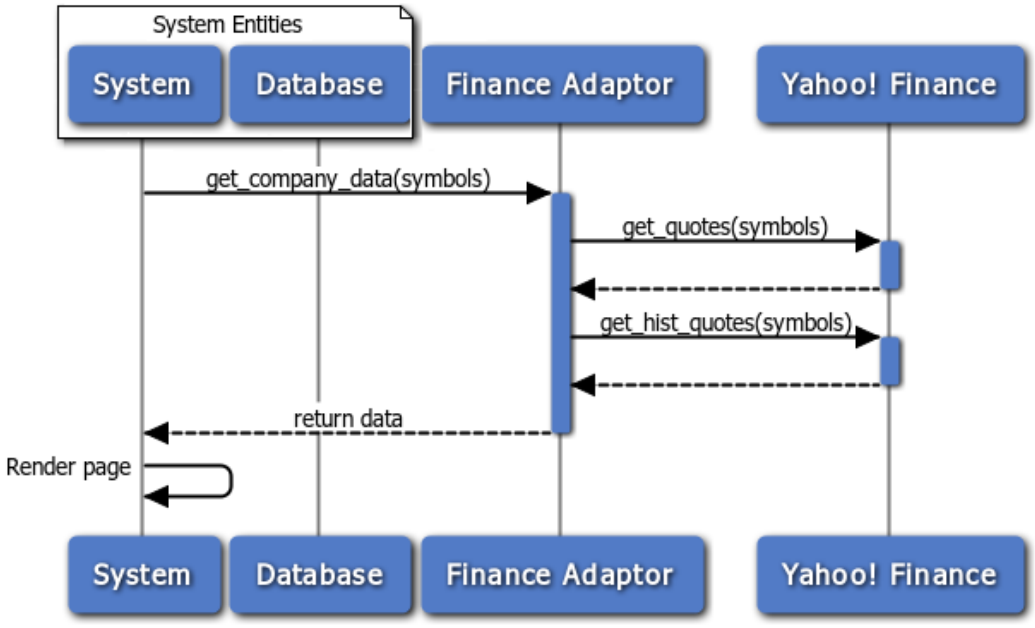


Figure 1.2: When a user wants to view a company page, the company data must be loaded from our finance API. Once the process is passed to the finance adaptor, the quotes and the historical quotes will be pulled from Yahoo! Finance and brought back to the system, who will prepare the page for the user. 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

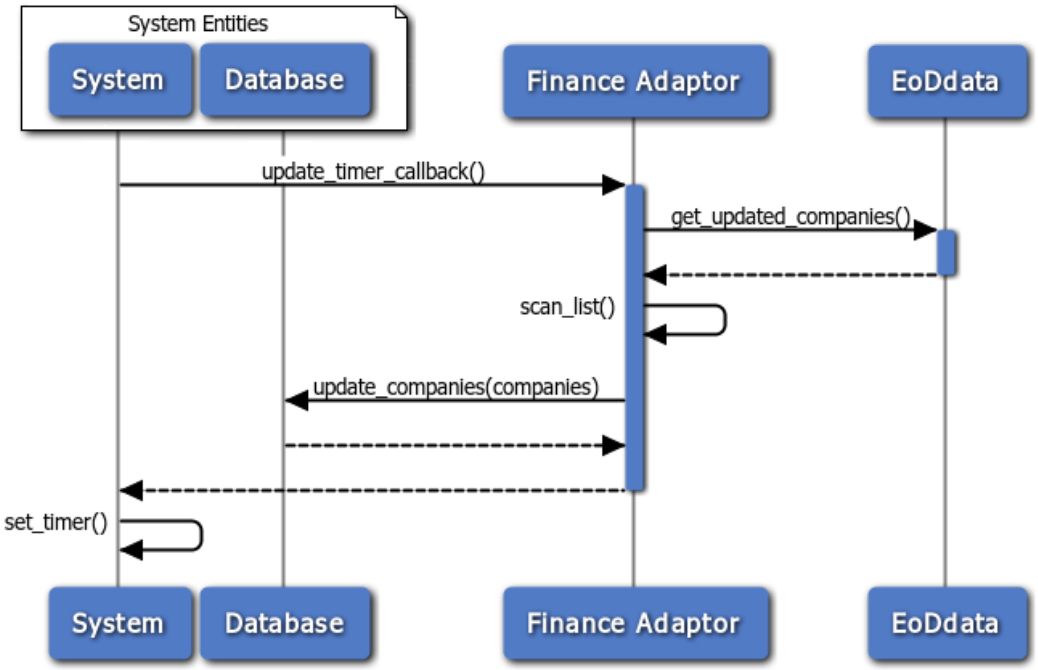


Figure 1.3: We need to keep a local copy of the current companies in our database so we can do rapid processes sing of all of the companies. In order to do this, there will be a timer that is set to update the database every once in a while. When the timer goes off, the system will pass the process onto 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.

1.3 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 posses 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

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.

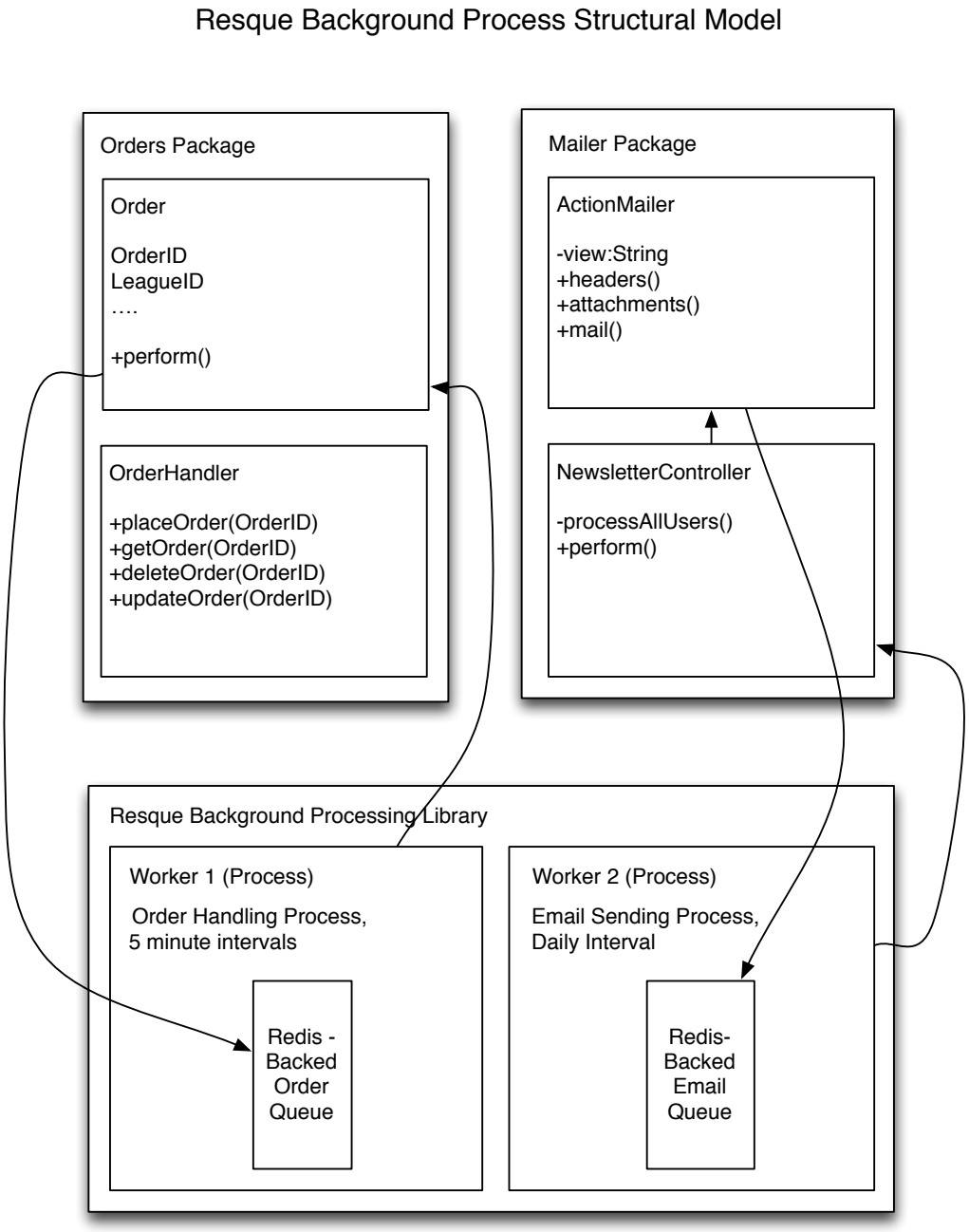


Figure 1.4: The structural model depicts the overall structure of this subsystem. Namely, the Resque Library and two packages or modules which each are responsible for one kind of task. On the left, the orders package displays a relevant subset of all classes that pertain to placing and processing orders. As previously mentioned, the Order object itself implements the perform method. Therefore, it knows how to process its data when it get 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.

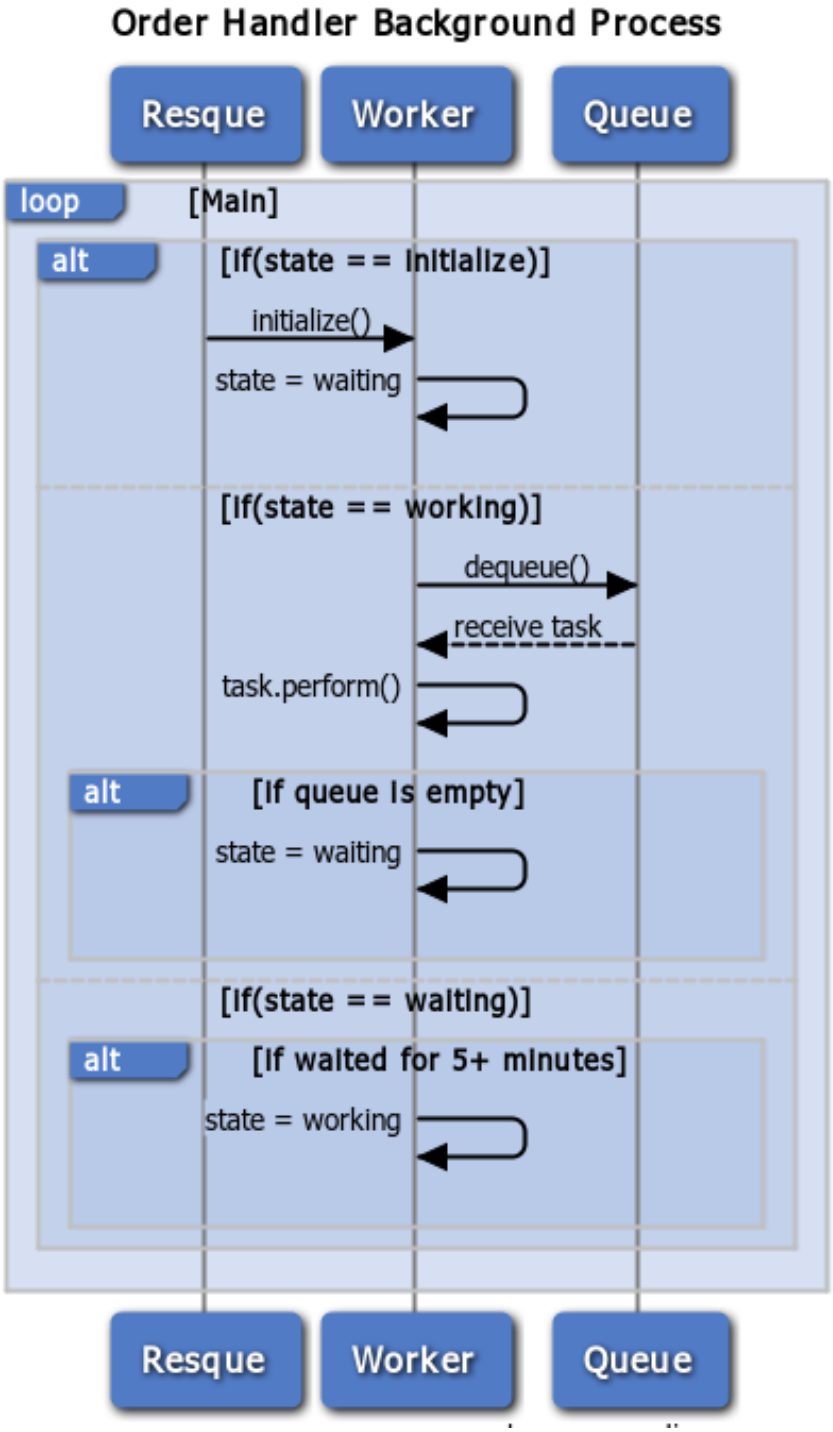


Figure 1.5: The interaction diagram above is roughly divided into two areas, when the process is working and when it is sleeping. This portrays the typical polling behavior of such a background running process. After initialization, when the worker wakes up it attempts to dequeue all objects and call 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.

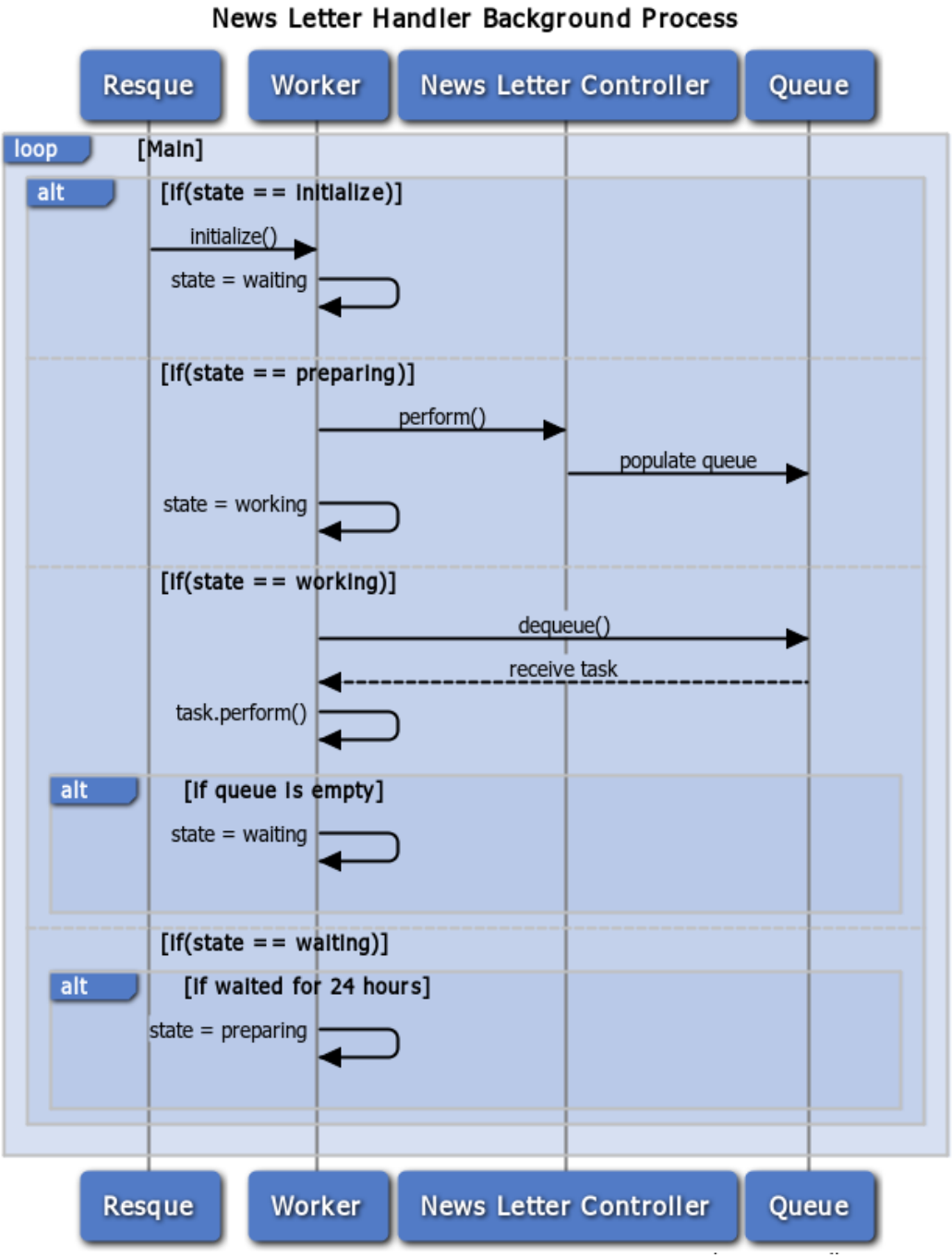
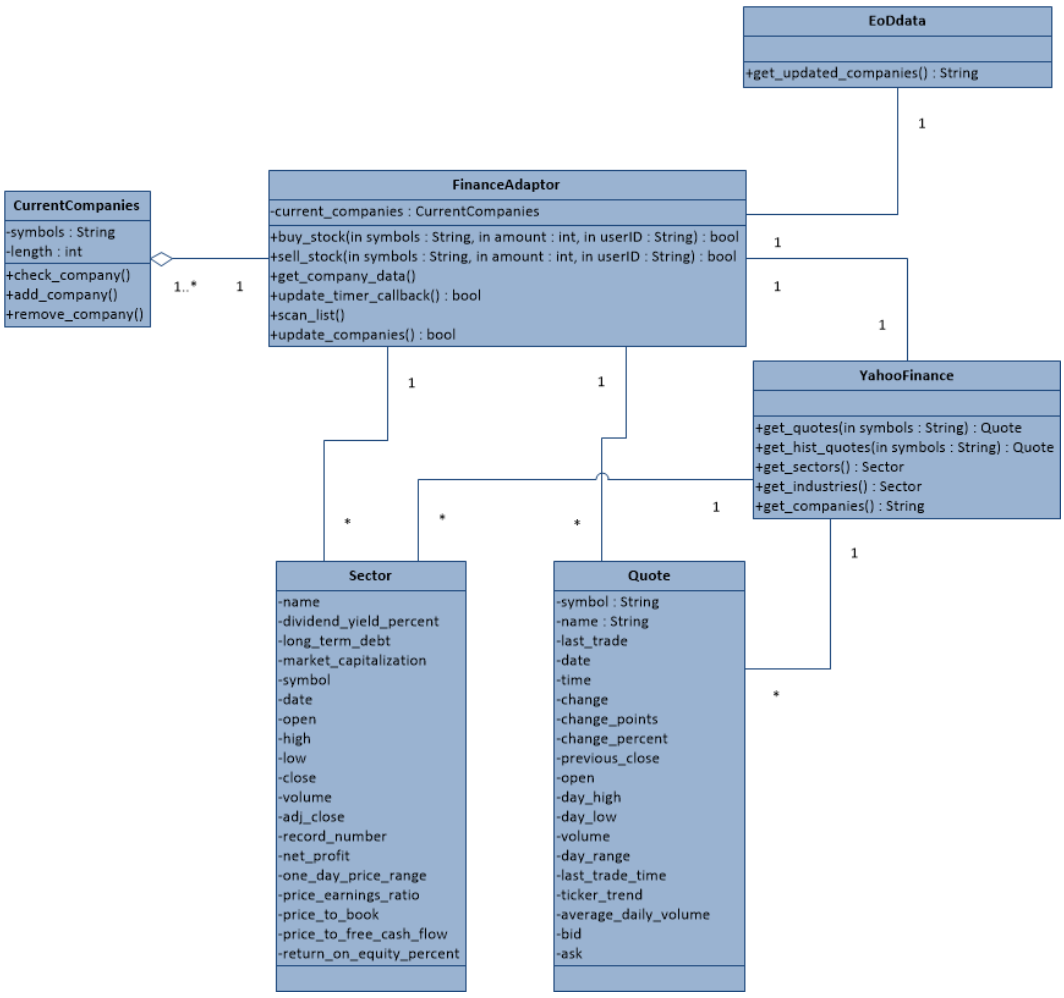


Figure 1.6: Worker 2 behaves a bit differently than worker 1 which results in having an additional state. This "prepare" state is when all the customizing of user-specific emails is done. Afterwards, the process enters the working state where it attempts to fire off all customized emails which were placed 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.

2 Class Diagrams and Interface Specifications

2.1 Financial Adaptor Class Diagram



2.2 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/inddustry names depending on the user query.

Methods
Most methods are boolean, returning either success or failure regardng 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 Caital 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 tabe.

+ 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! Finrance, similar to above.

2.3 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.

2.4 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 dequeueing 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 then 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

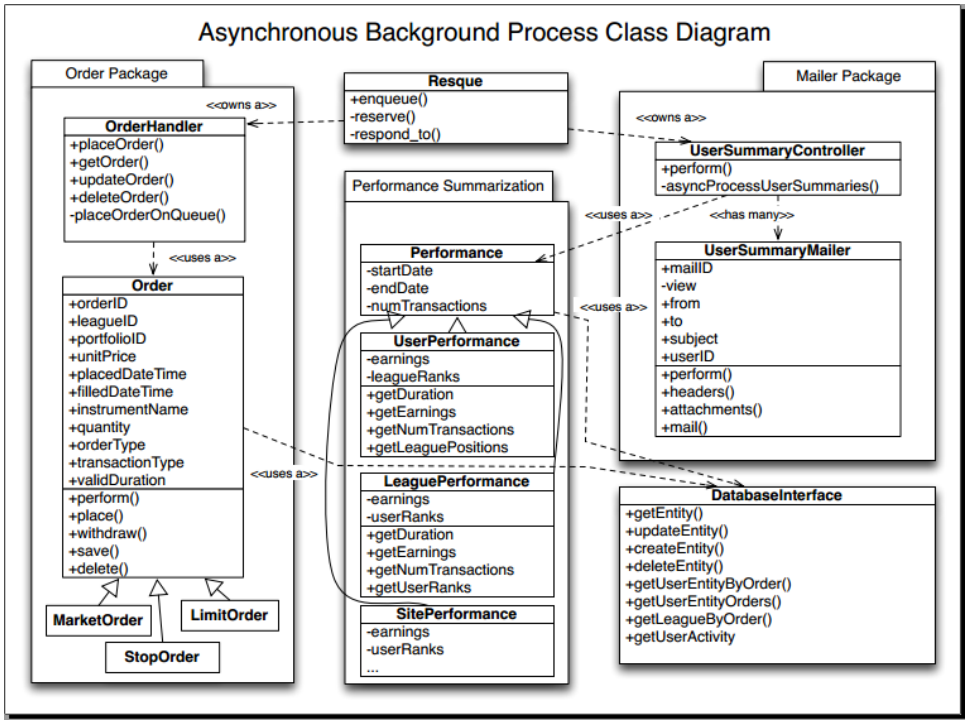


Figure 2.1: Asynchronous Class Diagrams.

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	retreives a user's performance for the User Summary.
League Performance Retriever	LeaguePerformance	retreives a leagues performance for the User Summary.

Table 2.1: Attribute Table.

3 System Architecture and System Design

3.1 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 comformation 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 [1].

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 [2].

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 identificaiton 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.

3.2 Identifying Subsystems

As Capital Games exists as a website, a natural divison 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 queueing 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 retrievial 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.

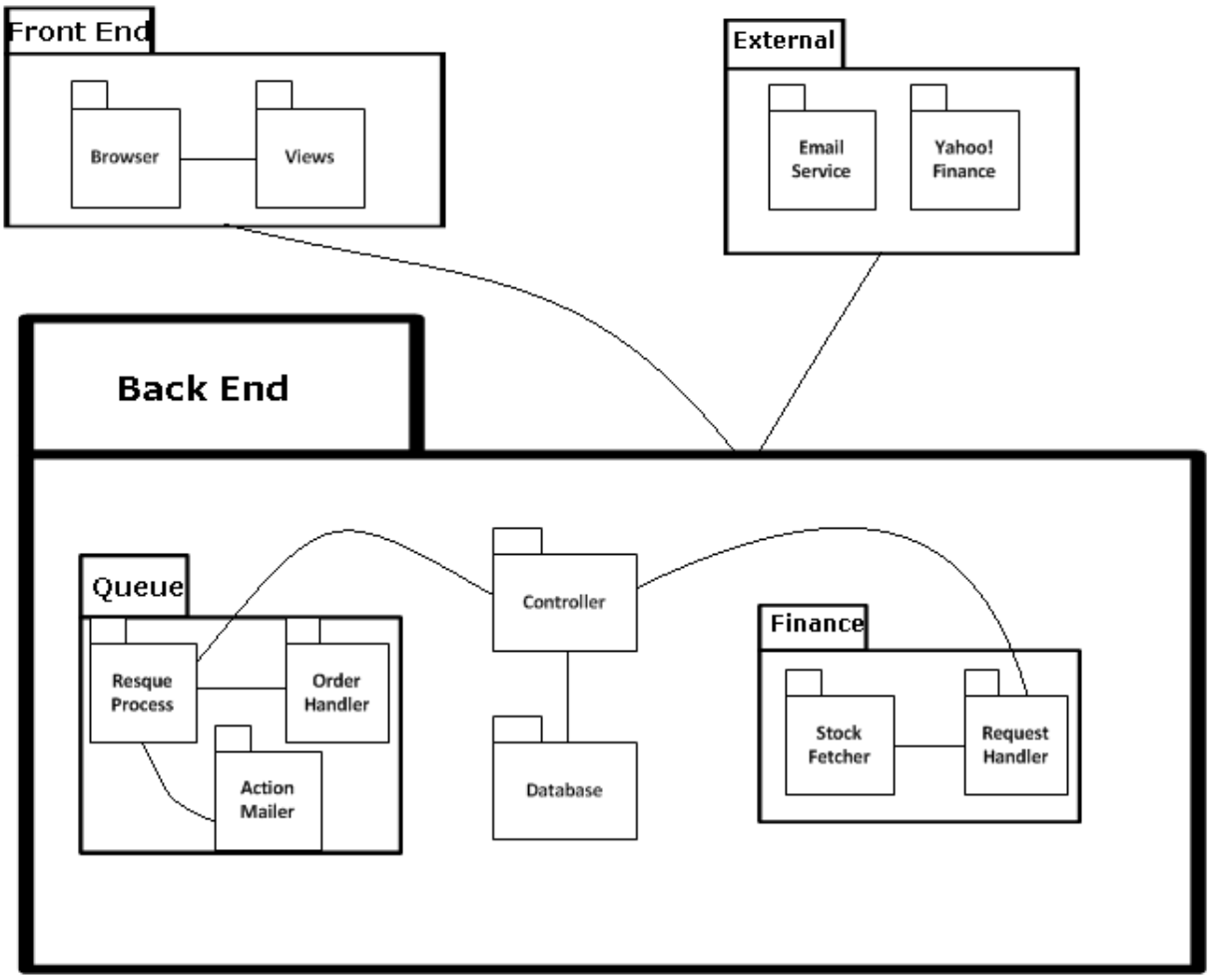


Figure 3.1: The UML package diagram for our system.

3.3 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.

3.4 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.

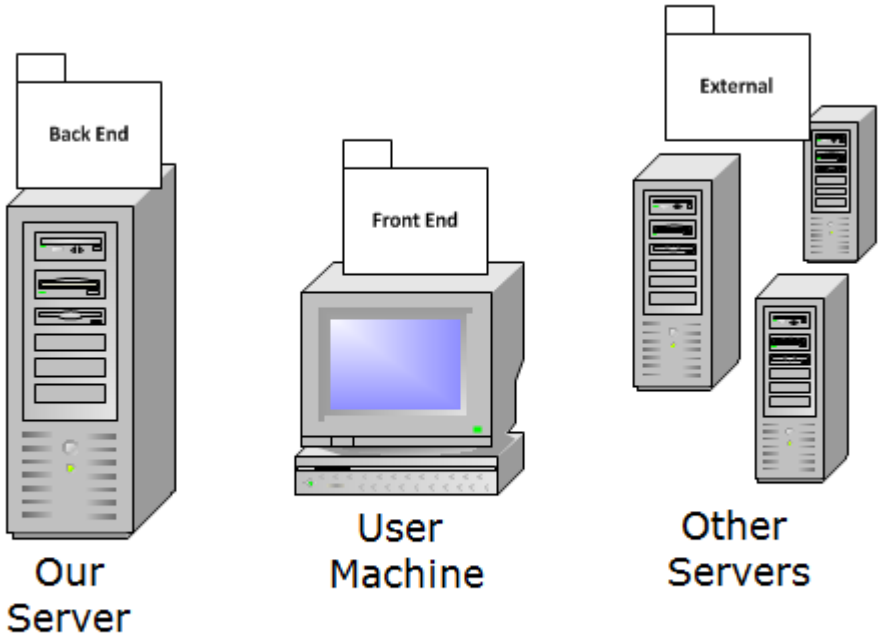


Figure 3.2: The hardware mapping for our system.

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 [3].

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 [4]. 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.

3.5 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.

3.6 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 relativey 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 enacting through Ruby's including protection functions–mainly flock (file lock).

3.7 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 [5]. 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.

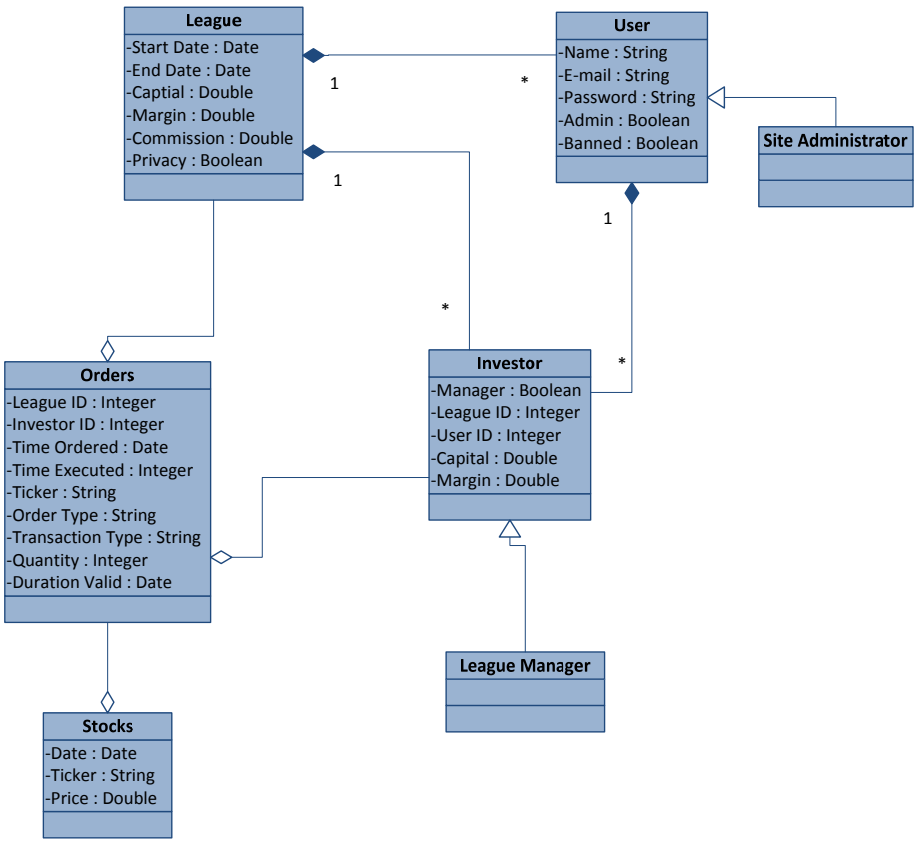


Figure 3.3: The format of the relational database schema implemented by Capital Games for its core features. (Not comprehensive.)

4 Data Structures

4.1 Table

4.2 Queue

4.3 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.

5 User Interface Design and Implementation

5.1 Updated pages

A few changes were brought in addition to our original designs, which are highlighted in this section.

Finalized Header

To bring a consistant theme to the whole website, a header is there to help. We had a header in the last version of the report but we needed navigable tabs to get around the website that will appear on every page, as well as a search bar for quick access to something more specific. Referring to the figure below, one can see that the new header removed the need to have a sidebar and therefore saves space and makes the end result more pleasing to the user.

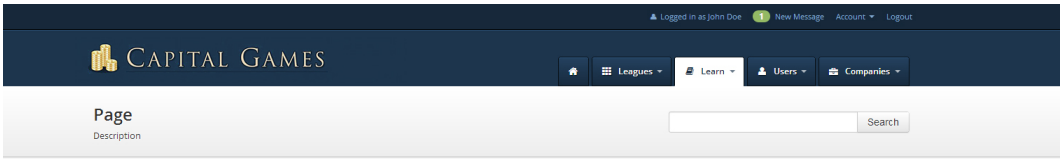


Figure 5.1: The header located on the top of every page.

Finalized League page

The last league page was a bit cluttered with the top three users section, and therefore a change was needed. This change brings about a cleaner representation of the top three users in a league and also an easier design to implement in the end. Bootstrap comes equipped with an accordian widget which has a main section that highlights content and a few buttons that change the content that is being shown, all while providing a clean animation on top of that. We will take advantage of that widget because it looks great and is easy to make.

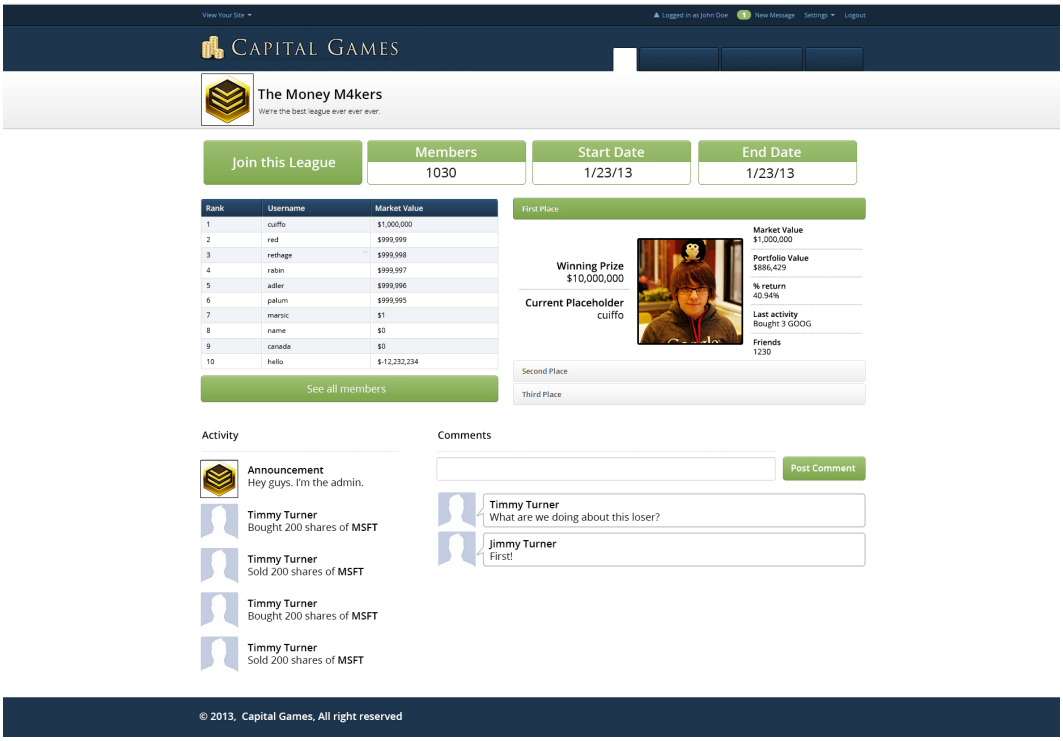


Figure 5.2: Changes to the leagues page.

Initial Front Page

The initial designs also lacked a page that a user is brought to when they first visit out website. A user cannot simply be taken to a login/sign up page without being told anything about the website or being greeted. This new page works as the page a visitor who is not logged in is brought to so they can choose the necessary action. For a user that already has an account, they can choose to log in via the leftmost button or the link up top. For a user that wants to create an account, they can choose to do so by clicking the middle button that will redirect to a simple form. For a user that wants to learn more about us, they can press the rightmost button which will bring them to the interactive tutorials we be implementing.

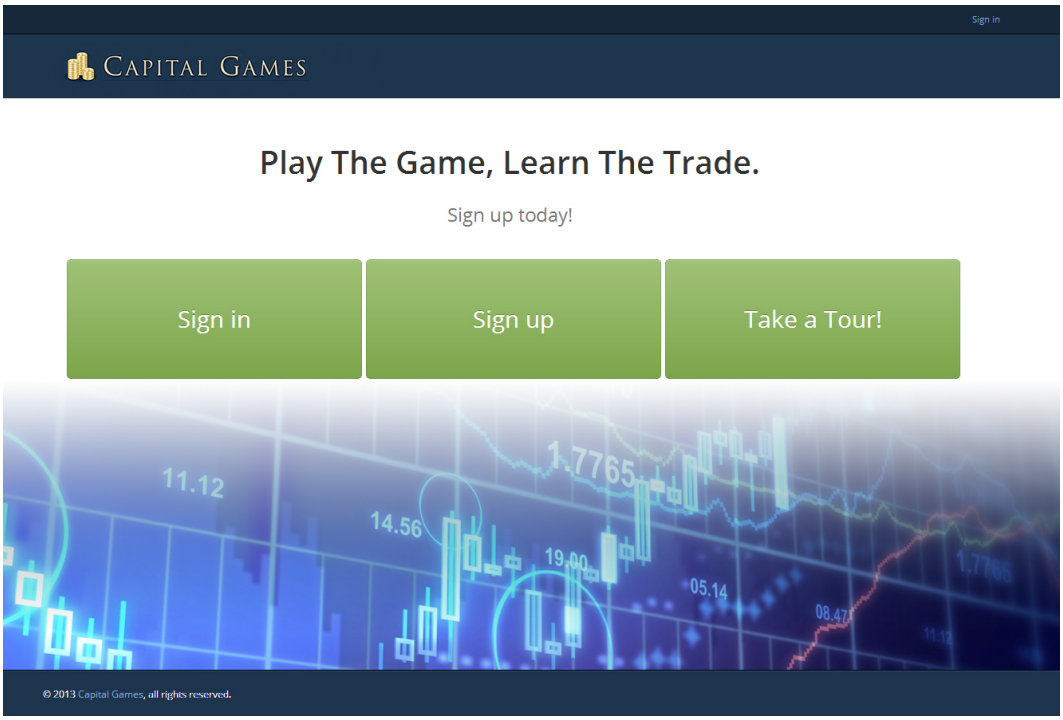


Figure 5.3: The page a user who is not logged in sees when they first visit our website.

5.2 Efficiency of the views

Our website has to be efficient but also not be a terrible pain to the programmers. There are a few ways that this can be dealt with on the view-side of programming.

Separation of the header and content

Ruby on Rails provides a feature that facilitates the creation of pages by having one page that is always loaded and somewhere in the middle it is redirected to the actual content. This means, for us, that we can create the content for each page and then rather than have to make the header each time, we can create that once and then link the content in the header page. On each page load, the header will be loaded and then the code will have a link which will then go find the correct file that contains the content code. Two reasons why this method is great are the ease to the programmer and the fact that the data will be stored in cache because we are loading the same exact file on the second page, therefore making loading much more efficient.

Avoiding long loading times

There are several ways we can reduce the amount of time the end user is going to have to wait for the page to load. One easy thing to do is that any pictures that we include should be scaled down to whatever is the maximum size it will be at. Having a large image and downscaling it is a waste of resources. Also on that topic, whenever a user uploads a picture, we will scale those down to a 100x100 icon. Another way we can avoid long loading times is to rely less on images and create pages with all code unless absolutely necessary. This is just good programming practice because

loading images when you can create something through code is a waste of bandwidth and loading time.

Cross compatibility

We decided to use Bootstrap elements to create our site, which comes with many features that help the programmer by predefining some views that are good-looking and cross compatible with all modern browsers and some older browsers that are still in use. In the case that we make some more features to the site that require non-Bootstrap elements, we can use the tool "Modernizr", which detects any features we are using that will not be supported in older browsers and correct them so that they work there, too.[6] It is an invaluable tool that will keep our pages current and universal.

6 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 [7]. 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.

6.1 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" [8]. 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.

6.2 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.

6.3 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.

6.4 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**
• (pass)Access unrestricted page • (fail)Access restricted page	• **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 6.1: 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
• (pass)enter permissible data • (fail) enter Impermissible data	• **On Pass:** impermissible data is rejected • **On Fail:** impermissible data is

Figure 6.2: 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
• Make order not during asynchronous queueing process (pass) • Make order during asynchronous queueing process (fail)	• **On Pass:** Queue data is not corrupt • **On Fail:** Queue data is corrupt

Figure 6.3: Test Case 3.

Test-case Identifier: TC-4
Function Tested: Queuing 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
• Request user summary for existing user(pass) • Request user summary for inexistent user (fail)	• **On Pass:** User summary is retrieved • **On Fail:** No user summary is generated

Figure 6.4: Test Case 4.

7 Plan of Work & Project Management

7.1 Development and Report Milestones

Illustrated on the next page is a gantt chart reflecting our goals relative to the project deadlines. It incorporates both core development and report items. For our initial stages we focus on environment and platform set-up (i.e. deploying a development webserver) and the initial, core code implementation. At the same time we will finalize the details of our final product via the report milestones.

Development milestones have been spread out following the completion of the first report on 22 February 2013, beginning with deploying our development environment and server through Heroku from which we continue to our next milestone of deploying Ruby on Rails as well as all the Gems and API packages we are incorporating into our project, most notably Yahoo! Finance.

Report milestones are also set concurrently. As we begin to initialize our development environment, we will also build on top of and expand on previous reports to expand upon and fully realize the details of *C*apital Games.

Core goals leading up to Demo 1 include establishing all core functionality for *C*apital Games. This includes the following:

- **Rails framework-deployed core functionality :** This includes a working system for navigating the website, registering a new user account, logging in, and creating as well as participating in leagues.
- **Setting a foundation for the database:** On top of having the aforementioned core functionalities, they also must be able to pass data through a routed database.
- **Implementing the Yahoo! Finance API:**
- **A functional user interface:** Our website should be usable, and having a functional user interface from the start will give us a lot of room to expand and optimize the UI.

7.2 Breakdown of Responsibilities Introduciton

Contributions leading up to the completion of this report are covered in the "Contributions" table on the page following the gantt chart. For the future division of labor, we all plan on subdividing aspects of both the next reports as well as the development of the *C*apital Games Alpha.

7.3 Breakdown of Responsibilities

Responsibilities for server/development environment deployment and set-up will be shared between Val and both Jeffs, as all three equally have great experience in the subject. Meanwhile Nick, and Eric will work of sequence diagrams.

While all other diagrams on the report will be covered by both Jeffs, the User Interface Design and Implementation will be worked on by Val, Dario, and Eric. Nick and Dario will work on both data types and operations while Val and Eric also work on the traceability matrix.

System architecture and design will be covered by Nick and Val. Jeff R. and Dario will begin on the database structure and site routing via the Rails framework. Nick and Jeff A. will work on the implementation of users in the meantime.

From there we will further subdivide work on the final aspects of the website, likely sticking with our initial idea of splitting predominant responsibilities following the model, view, and controller (MVC) design pattern. Jeff R. and Jeff A. will lead work relating to model design, Dario and Eric will lead work relating to views and interface, and Nick and Val will lead work involving controllers. That being said, while the MVC pattern will model sub-component ownership among the team. Individual implementation responsibilities will be distributed a bit more evenly based on the particular strengths of team members.

In summary, Project Ownership will be based on the MVC architecture. To reiterate, Jeff R. and Jeff A. will have ownership over the Model portion, Dario and Eric will have ownership over the Views (user interface, etc.) portion, and Nick and Val will have ownership over the controllers portion. Even beyond Project Ownership, however, responsibility for the whole project will be shared and the success of our MVC architecture requires close coordination between all aspects.

Overall project success will be decided with how well the MVC component teams communicate and work with each other, as *C*apital Games will rely on the interactivity between the Model, Views, and Controller portions of the architecture.

7.5 Report 2 Contributions

		Names					
Category	Points	Jeff A	Eric C	Nick P	Jeff R	Val R	Dario R
UML Diagrams	10 Points	0%	0%	40%	50%	0%	10%
Descr. of Diagrams	10 Points	20%	0%	0%	80%	0%	0%
Alt. Solution Description	10 Points	0%	33%	0%	0%	33%	33%
Class Diagram & Description	5 Points	30%	10%	35%	0%	5%	20%
Signatures	5 Points	0%	33%	0%	0%	33%	33%
Styles	5 Points	25%	25%	10%	20%	20%	0%
Package Diagram	2 Points	0%	0%	0%	0%	50%	50%
Mapping Hardware	2 Points	0%	0%	0%	0%	50%	50%
Database	3 Points	0%	0%	0%	0%	50%	50%
Other	3 Points	0%	0%	0%	0%	50%	50%
Appearance	8 Points	0%	0%	0%	0%	50%	50%
Prose Description	7 Points	0%	0%	0%	0%	50%	50%
Testing Design	12 Points	0%	0%	0%	0%	50%	50%
Document Merging	11 Points	0%	0%	0%	0%	50%	50%
Project Coordination	5 Points	0%	0%	0%	0%	50%	50%
Plan of Work	2 Points	0%	0%	0%	0%	50%	50%

7.4 Gantt Chart of Projected Milestones

Best viewed at 100% or greater:

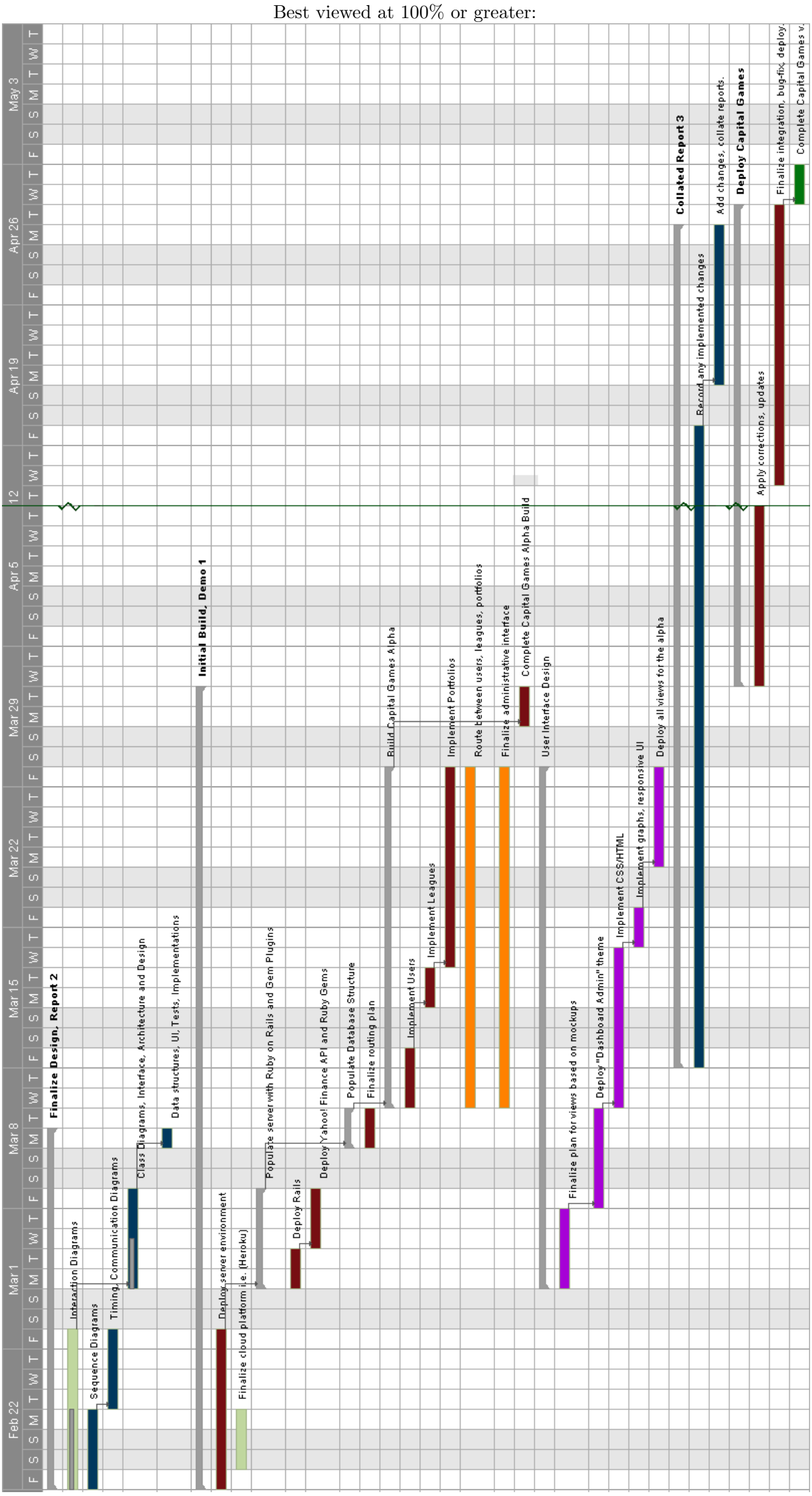


Figure 7.1: This gantt chart projects how we will concurrently work on the project. All blue items are report-related, red and orange relate to the core project development and purple illustrates UI milestones.

References

[1] Wikipedia, "Model-view-controller - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Model-view-controller. [Online; accessed 10 March 2013].

[2] Wikipedia, "Representational state transfer - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Representational_state_transfer. [Online; accessed 3 March 2013].

[3] Wikipedia, "Relational database - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/Relational_database. [Online; accessed 10 March 2013].

[4] Wikipedia, "NoSQL - Wikipedia, the free encyclopedia." http://en.wikipedia.org/wiki/NoSQL. [Online; accessed 10 March 2013].

[5] 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].

[6] Wikipedia, "Modernizr." http://en.wikipedia.org/wiki/Modernizr. [Online; accessed 16 March 2013].

[7] Wikipedia, "Test-driven development - Wikipedia, the free encyclopdia." en.wikipedia.org/wiki/Test_driven_development. [Online; accessed 12 March 2013].

[8] Rails Guides, "Ruby on rails guides: Rails routing from the outside in." guides.rubyonrails.org/routing.html. [Online; accessed 13 March 2013].