Report 2

Part 2

Group 2

Daniel Su
Jason Scatena
Basak Takimci
Thanh Do Huu
Boris Hazanov
Micah Moore
Tam Duong

Table of Contents

2 Table of Contents

3-4 Responsibility Matrix and Allocation Chart

5-11 Interaction Diagrams and Design Principles

12-14 Class Diagrams and Interface Specification

15 Traceability Matrix

16 Architectural Styles

17 Identifying Subsystems

18 Mapping Subsystems to Hardware

19 Persistent Data Storage

20 Network Protocol

21 Global Control Flow

22 Hardware Requirements

23 Algorithms and Data Structures

24-38 User Interface Design

39-44 Test Cases

45 Integration Testing

46-47 Project Management and Plan of Work

	Daniel Su	Micah Moore	Jason Scatena	Boris Hazanov	Tam Duong	Thanh Do Huu	Basak Takimci
Interaction Diagrams(30)	16.7%	16.7%	16.7%	16.7%	16.7%	16.7%	
Classes & Specs (10)		25%			37.5%	37.5%	
Sys Arch & Design (15)	20%	8.3%	20%	26.7%			25%
Algorithms (4)					75%	12.5%	12.5%
User Interface (11)						22.7%	77.3%
Testing (12)		20.8%		50%		29.2%	
Project Management (18)	41.7%	19.4%	38.9%				

Point Allocation Chart

Points:

Daniel Su: 5+3+7.5=15.5

Micah Moore: 5+2.5+1.25+2.5+3.5=14.75

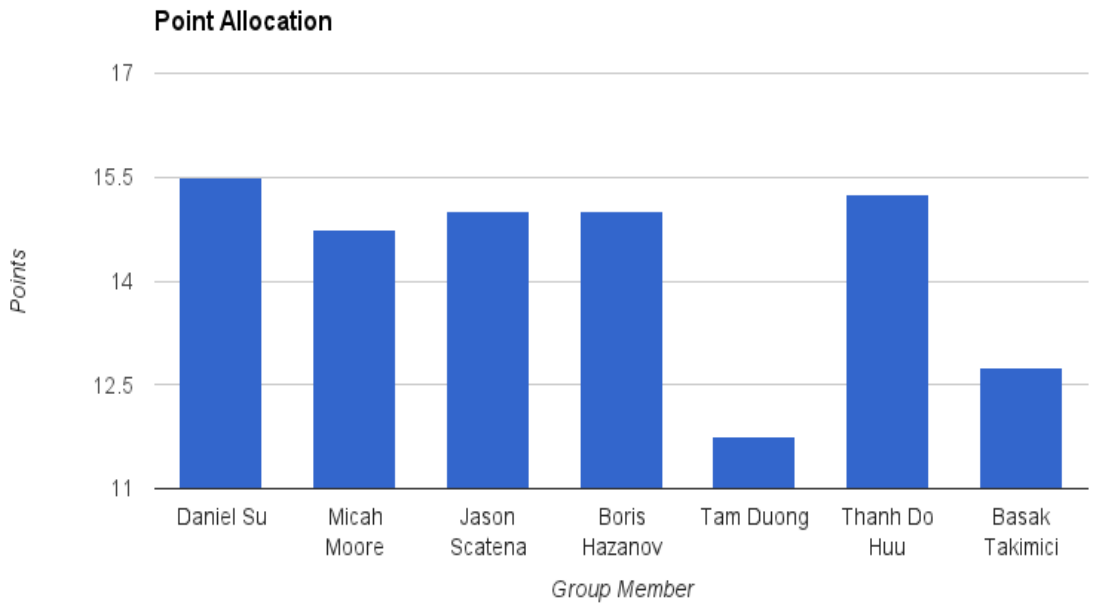
Jason Scatena: 5+3+7=15

Boris Hazanov: 5+4+6=15

Tam Duong: 5+3.75+3=11.75

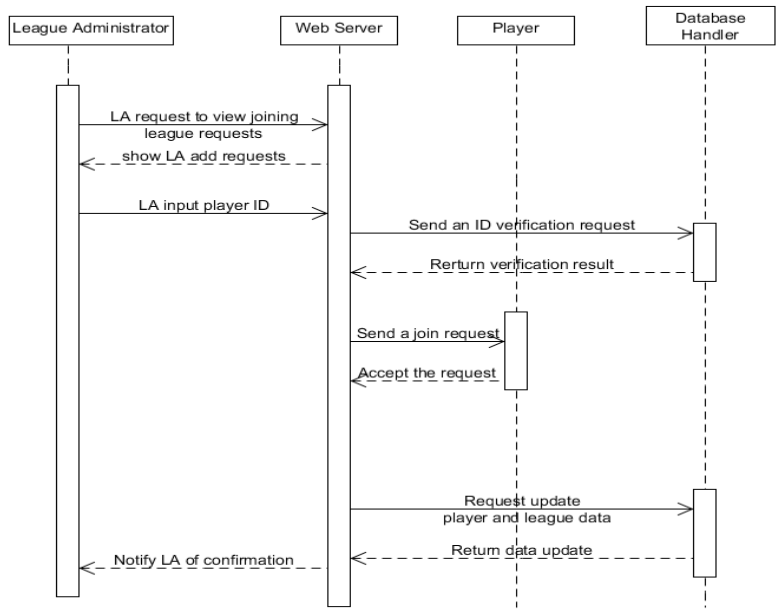
Thanh Do Huu: 5+3.75+.5+2.5+3.5=15.25

Basak Takimici: 3.75+.5+8.5=12.75

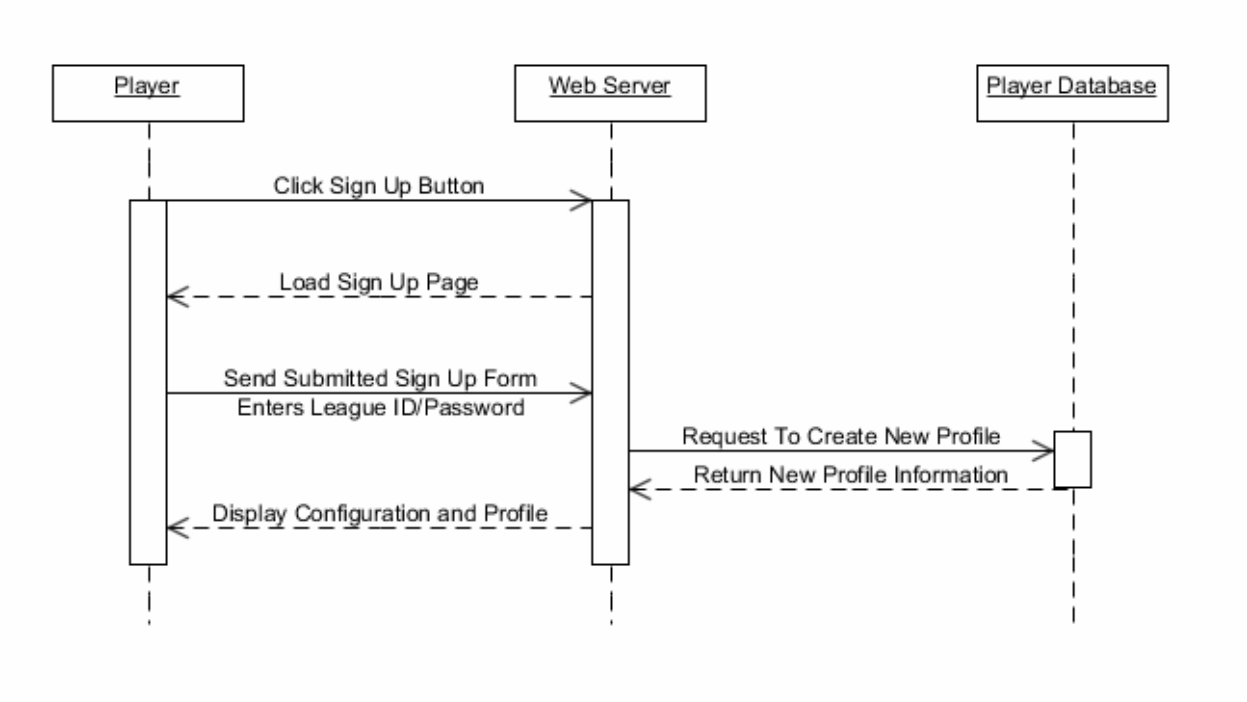


Interaction Diagrams

UC-5: Administration- Manually Add Player to League
To manually add a player to a league, the league administrator first requests the web server to view the player's league joining request. The Web server returns the request list. After viewing the list, the League Administrator inputs the player ID of the players he/she wants to add to the league. The Web Server sends verification requests to the database handler(who deals with the database). The database handler then sends the join requests to players. After each player has accepted a request, the Web Server sends a data update request to the database handler. The database handler then sends the updated info(that the player has been successfully been added to the league) back to the web server to be viewed by the league administrator(and player if they want to). If the league admin fails to input a valid player ID, the web server will notify him/her of the problem and request him/her to input a different player ID until he/she inputs a valid one.
If the player has already joined the league, the web server will notify the league administrator of the error that the player is already in the league.
If the player rejects the request, the Web Server will then send a notification to the league administrator that the player rejected.
UC-5 System Sequence Diagram

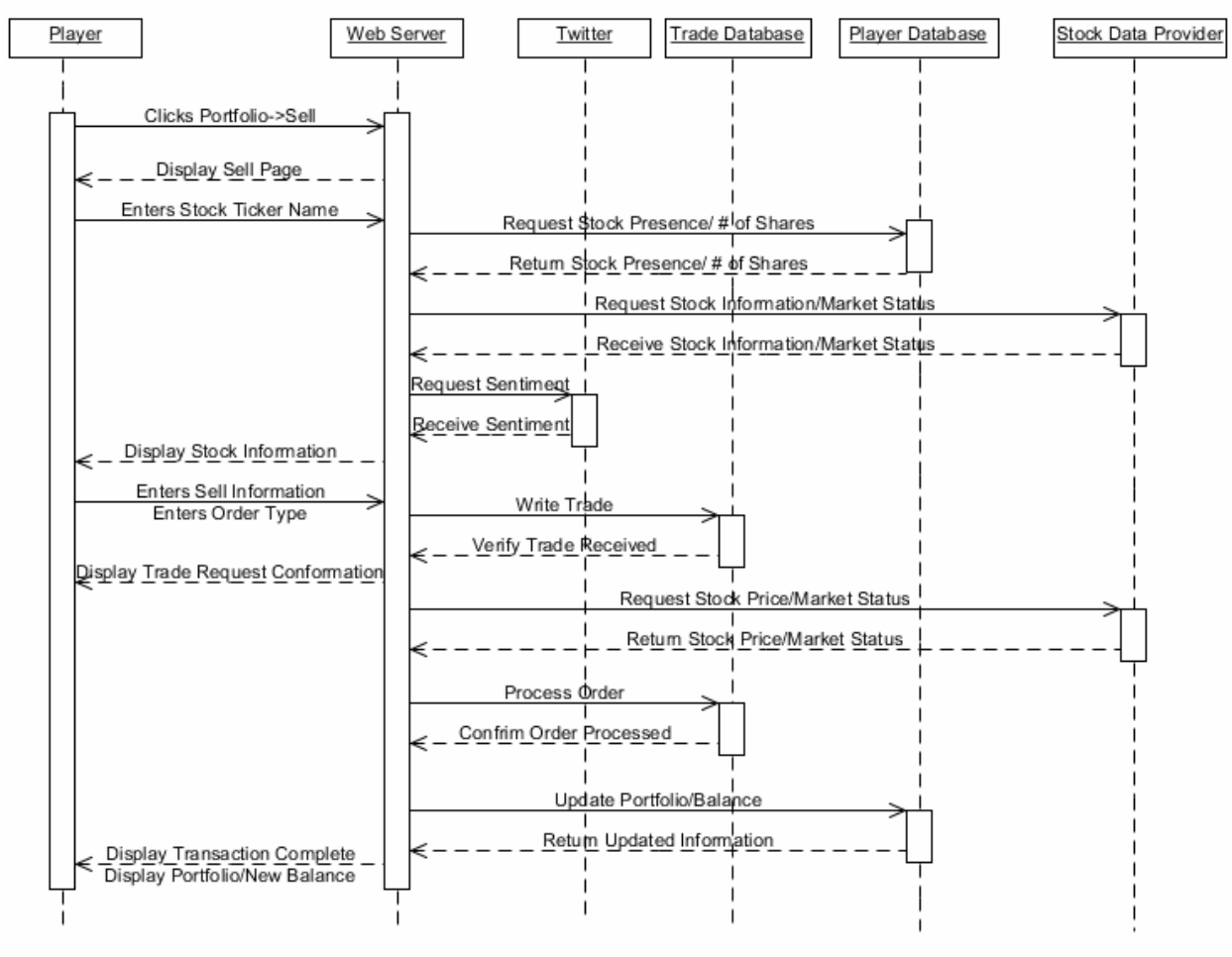


UC-6 Player Requests to Join League:
The league admin will have a unique code/password when they have set the league up. It is assumed that the administrator will email each player in his league this ID/password. The player clicks the sign up button. The web server loads, and sends the sign up page back to the player to fill out. The player enters his information along with the league ID/password given by the administrator. If this league ID/password is entered incorrectly 3 times, a button will appear to email the league administrator to manually add you to the league (which corresponds to the 'Manually Add Player to League' sequence diagram). The web server verifies the player's data. Once verified, the server sends a request for a new player profile to be created in the database. The database then returns this new profile information to the web server, which then in turn displays this information to the player, showing he's logged in.
UC-6 System Sequence Diagram:



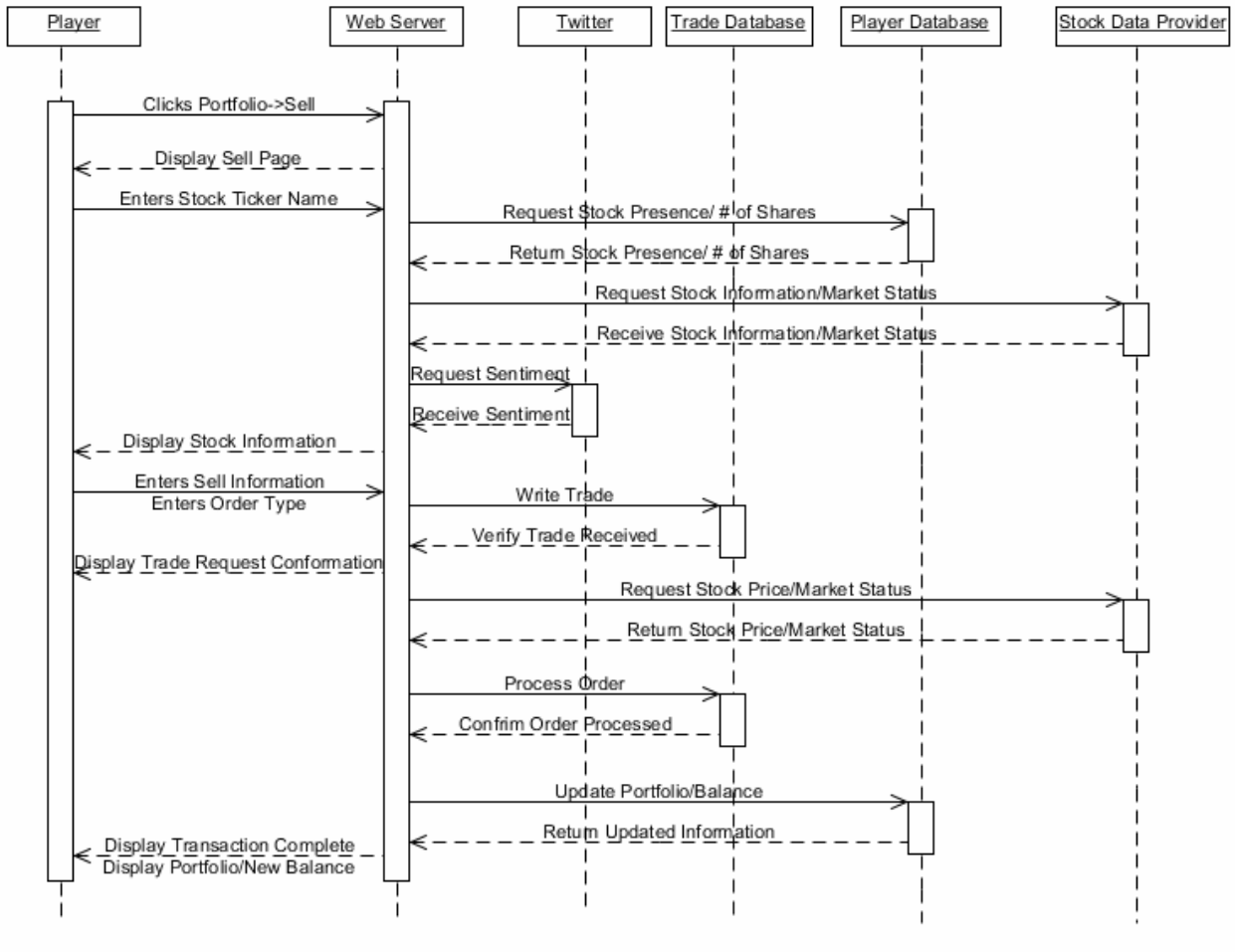
UC-2 Make Trades -- Buy (1 of 2):
The player clicks Portfolio->Buy Stocks. The server displays the buy page(just the stock ticker name field at first). The player enters the stock ticker name. The server requests the stock information, and market status(whether the market is open or closed)from the stock data provider. The stock data provider sends the requested information to the server. The server requests sentiment data from twitter. Twitter sends the raw data to the server which analyzes it, using the NLTK library. The server then displays all the stock's information to the player. The player enters his desired buy share amount, boundaries, and order type. The server takes this information and first checks if the player has sufficient available funds by querying the player database. The player database then returns the player balance to the server. If the balance is enough, the server writes the trade request to the trade database. The trade database verifies the buy request was received and sends the verification to the server, which relays it to display the confirmation of receipt to the player. The server then requests the stock price/market status again from the stock data provider. This is to check whether the stock prices/market status has changed since the order had been submitted. The newly retrieved market status and stock price are returned to the server and as long as the market status hasn't changed and the stock price hasn't changed more than 0.5%(If the stock price has changed more than 0.5% it will send the user back to the Portfolio->Buy page to query for a stock ticker again) the server will send a order request to the trade database to execute the order. The trade database will send a confirmation of the order being processed to the server. The server will then update the player's portfolio/balance in the player database. The player database then sends the update portfolio/balance back to the server, which then, in turn, displays it to the player on his portfolio page.

UC-2 Make Trades -- Buy System Sequence Diagram:



UC-2 Make Trades -- Sell (2 of 2):
The player clicks Portfolio->Sell Stocks. The server displays the Sell page(just the stock ticker name field at first). The player enters the stock ticker name. The server requests verification of the stock presence and number of shares from the player database(portfolio). This information is returned to the server. If the stock is present in the player portfolio the server requests the stock information and market status(whether the market is open or closed) from the stock data provider. The stock data provider sends the requested information to the server. The server then requests the sentiment from Twitter. Twitter sends the raw data to the server, which analyzes it, using NLTK library. The server displays all the stock's information to the player. The player enters his desired sell share amounts, boundaries, and order type. The server then writes the trade to the trade database. The trade database verifies the sell request was received and sends the verification to the server, which relays it to display the confirmation of receipt to the player. The server then requests the stock price/market status again from the stock data provider. This is to check whether the stock prices/market status has changed since the order had been submitted. The newly retrieved market status and stock price is returned to the server and as long as the market status hasn't changed and the stock price hasn't changed more than 0.5%(If the stock price has changed more than 0.5% it will send the user back to the Portfolio->Sell page to query for a stock ticker again) the server will send a sell request to the trade database to execute the order. The trade database will send a confirmation of the sale being processed to the server. The server will then update the player's portfolio/balance in the player database. The player database then sends the update portfolio/balance back to the server, which then, in turn, displays it to the player on his portfolio page.

UC-2 Make Trades -- Sell System Sequence Diagram:



Design Principles

In order best formulate our design we used a combination of deductive and inductive analyses. At the top down global level we wanted to ensure that the overarching architecture of our system was efficient. While our system isn't particularly large, merely using a bottom up approach would not lead to an optimal design. We let the abilities of our team and our time constraint guide our design. For instance we knew from the start that we would be employing the "communication through a common data element" communication pattern. This is because we could use a master database through which all our sub-systems can interact. We determined this to be more appropriate, at least partially, because we have multiple team members that have worked with database design in the past.

This top-down approach, in isolation, is not sufficient for a well designed system. This is why we employed inductive analysis for the design as well. This allowed us to analyze the problem, and come up with a solution that employees good design principles. Our design focuses primarily on incorporating high cohesion with an inclination towards low coupling. We incorporate various classes and objects to perform specialized roles and functions. Then these objects all communicate with the main controller in either returning values, requesting computations from other objects, etc.

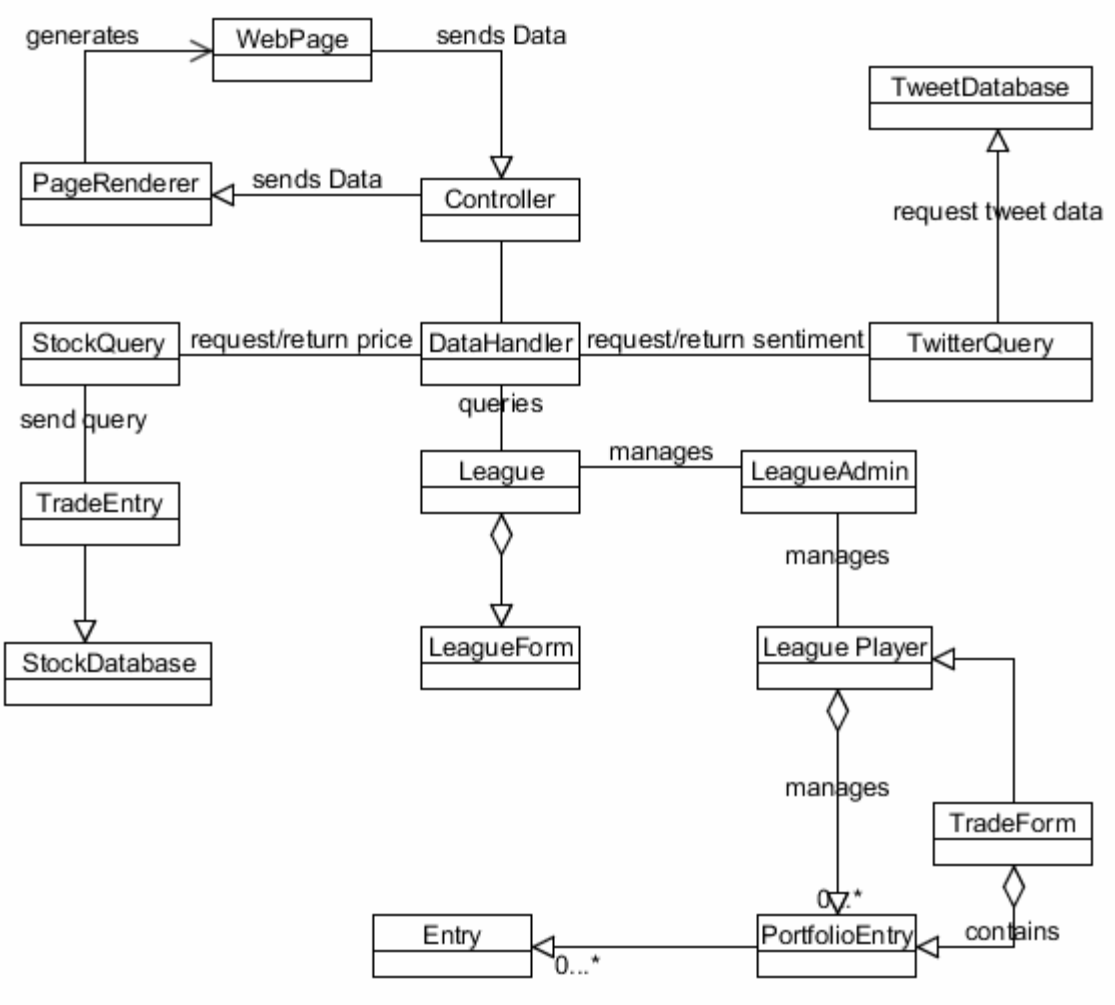
For example, we plan to have an object responsible for data visualization. The main controller would receive the user data from the data request object, then this data would be fed to the data visualization object in order to generate user friendly data images / graphs. We will also have objects responsible for conducting trades and storing the trade results in the database.

So in the case where a user has just conducted a trade and wishes to visualize his/her new portfolio data. The flow would be like this:
Object 1 performs trade and returns trade results to controller.
Controller calls object 2 to store trade results.
(User requests data visualization)
Object 3 fetches portfolio data, and feeds it to object 4.
Object 4 then finishes the task and provides the user with the visualizations.

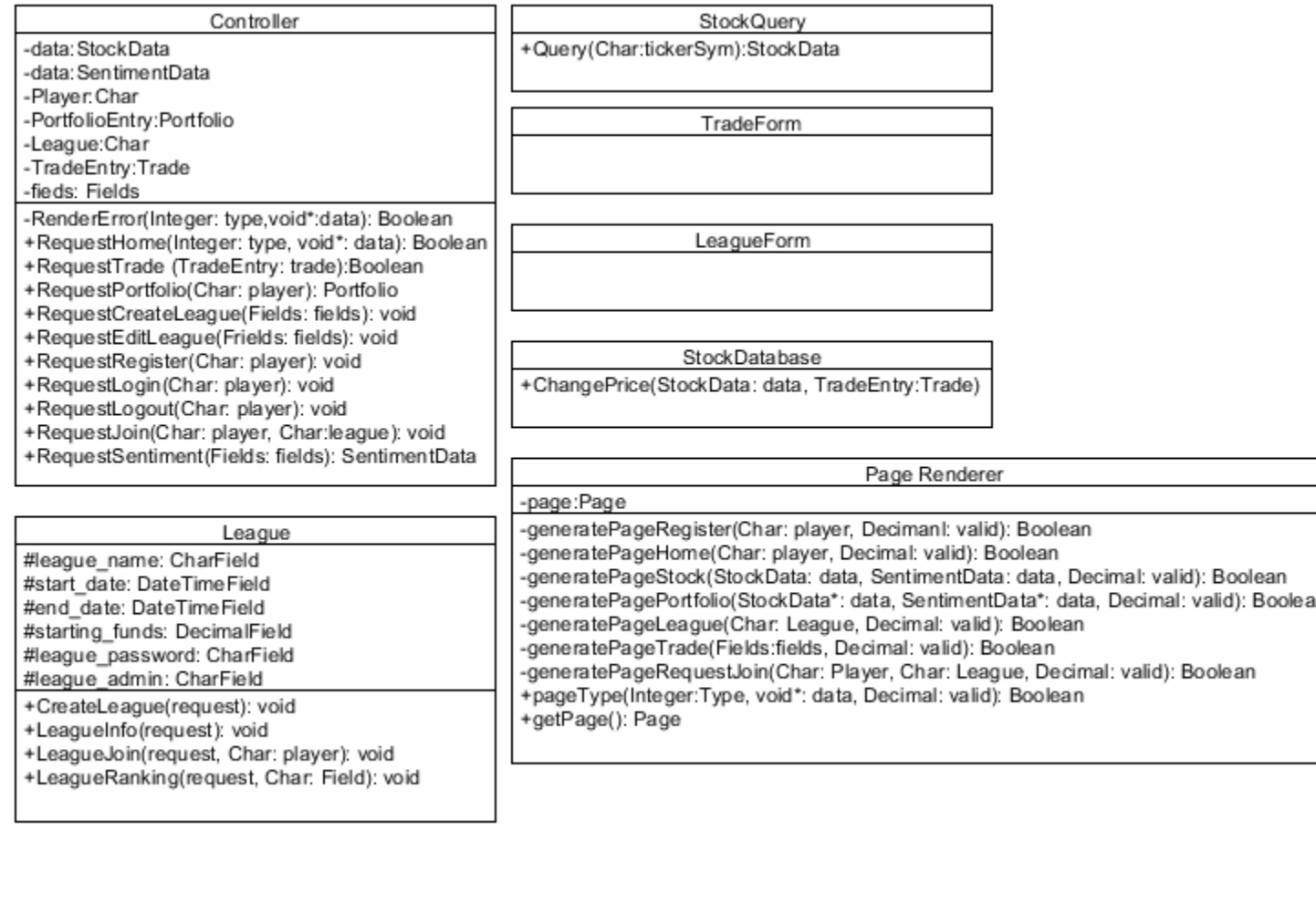
As for coupling, the goal is to have only the main controller communicating between all objects. The individual objects may need to communicate with each other, but it will be through the main controller.

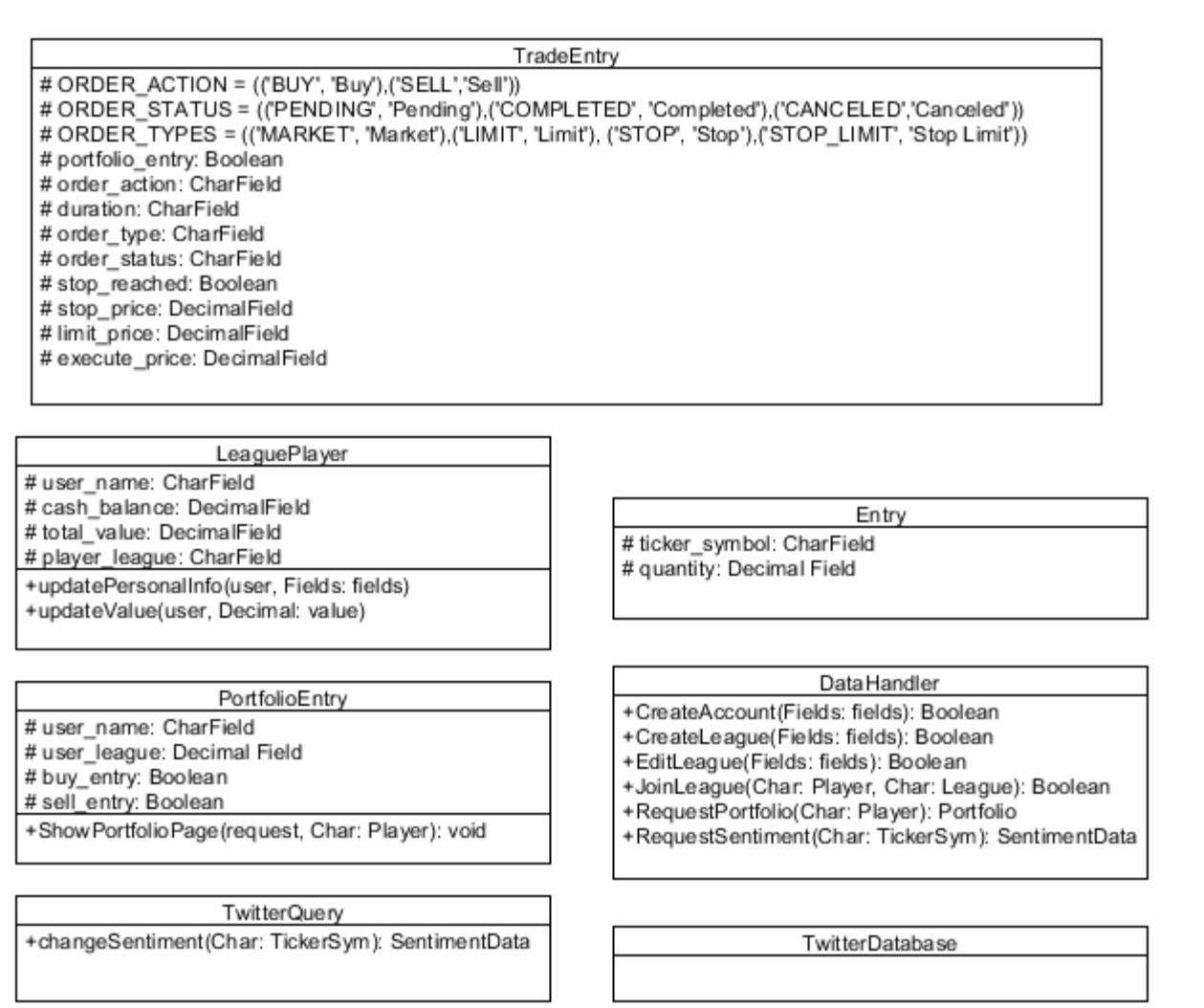
By adhering to these concepts, our product code will be easily read and maintained. Addition of new objects will also be streamlined due to the low coupling concept of having a main controller negotiate on the behalf of individual objects.

Class Diagrams and Interface Specification



Main Diagram


Controller
-data:StockData
-data:SentimentData
-Player:Char
-PortfolioEntry:Portfolio
-League:Char
-TradeEntry:Trade
-fieds: Fields
-RenderError(Integer: type,void*:data): Boolean
+RequestHome(Integer: type, void*: data): Boolean
+RequestTrade (TradeEntry: trade):Boolean
+RequestPortfolio(Char: player): Portfolio
+RequestCreateLeague(Fields: fields): void
+RequestEditLeague(Frields: fields): void
+RequestRegister(Char: player): void
+RequestLogin(Char: player): void
+RequestLogout(Char: player): void
+RequestJoin(Char: player, Char:league): void
+RequestSentiment(Fields: fields): SentimentData
League
#league_name: CharField
#start_date: DateTimeField
#end_date: DateTimeField
#starting_funds: DecimalField
#league_password: CharField
#league_admin: CharField
+CreateLeague(request): void
+LeagueInfo(request): void
+LeagueJoin(request, Char: player): void
+LeagueRanking(request, Char: Field): void
StockQuery
+Query(Char:tickerSym):StockData
TradeForm
LeagueForm
StockDatabase
+ChangePrice(StockData: data, TradeEntry:Trade)
Page Renderer
-page:Page
-generatePageRegister(Char: player, Decimanl: valid): Boolean
-generatePageHome(Char: player, Decimal: valid): Boolean
-generatePageStock(StockData: data, SentimentData: data, Decimal: valid): Boolean
-generatePagePortfolio(StockData*: data, SentimentData*: data, Decimal: valid): Boolea
-generatePageLeague(Char: League, Decimal: valid): Boolean
-generatePageTrade(Fields:fields, Decimal: valid): Boolean
-generatePageRequestJoin(Char: Player, Char: League, Decimal: valid): Boolean
+pageType(Integer:Type, void*: data, Decimal: valid): Boolean
+getPage(): Page


TradeEntry
ORDER_ACTION = (('BUY', 'Buy'),('SELL','Sell'))
ORDER_STATUS = (('PENDING', 'Pending'),('COMPLETED', 'Completed'),('CANCELED','Canceled'))
ORDER_TYPES = (('MARKET', 'Market'),('LIMIT', 'Limit'), ('STOP', 'Stop'),('STOP_LIMIT', 'Stop Limit'))
portfolio_entry: Boolean
order_action: CharField
duration: CharField
order_type: CharField
order_status: CharField
stop_reached: Boolean
stop_price: DecimalField
limit_price: DecimalField
execute_price: DecimalField
LeaguePlayer
user_name: CharField
cash_balance: DecimalField
total_value: DecimalField
player_league: CharField
+updatePersonalInfo(user, Fields: fields)
+updateValue(user, Decimal: value)
Entry
ticker_symbol: CharField
quantity: Decimal Field
PortfolioEntry
user_name: CharField
user_league: Decimal Field
buy_entry: Boolean
sell_entry: Boolean
+ShowPortfolioPage(request, Char: Player): void
DataHandler
+CreateAccount(Fields: fields): Boolean
+CreateLeague(Fields: fields): Boolean
+EditLeague(Fields: fields): Boolean
+JoinLeague(Char: Player, Char: League): Boolean
+RequestPortfolio(Char: Player): Portfolio
+RequestSentiment(Char: TickerSym): SentimentData
TwitterQuery
+changeSentiment(Char: TickerSym): SentimentData
TwitterDatabase

Traceability Matrix

Class/Domain Concept	WebPage	PageRenderer	InputHandler	StockQuery	Stock DataProvidr	PlayerHandler	leagueHandler	ValidityChecker	DataHandler	DataBase
Webpage	X									
PageRender		X								
Controller	X	X								
DataHandler									X	
TwitterQuery									X	
TweetDataBase			X						X	
StockQuery									X	
Trade Entry				X					X	
StockDatabase									X	
League						X			X	
League Form									X	
League Admin						X			X	
League Player									X	
TradeForm									X	X
PortfolioEntry					X		X		X	
Entry									X	

Many of the classes are matching with DataHandler concept. It is because classes are interrogated by the DataHandler. In the domain model it is easy to see how each class is in the single entity. However, in the class diagram everything is more detailed because the class diagram gives more information. It is more understandable why classes are evolved from a single concept.

Architectural Styles

RU Investing uses architectural style with focus on Model/View/Controller approach.

3.1.1 Model/View/Controller

Model contains user information and communicates between several systems over a network. A client may initiate a communication session, while the server waits for requests from any client. The site database will be created using Django and the stock model will be made accessible by API calls to an external stock information provider.
View requests from the model the information it needs to generate an output. The view will be represented by HTML, CSS, and JavaScript.
Controller is the controller logic will be implemented using Python.

3.1.2 Front and Back Ends:

Both the frontend and the backend are using the component-based architectural style by using design and development languages that allows them to be run independent of the platform they are on. And, The front-end component of our system is our Web UI. This is what the public will see. The back end consists of all the behind the scenes business logic for our app.

3.1.3 Event-driven Architecture

A software architecture pattern promoting the production, detection, consumption of, and reaction to events. Any change to the equilibrium of our system by the user is an event. In this way, the user acts as an event emitter (i.e. initiating buy, sells, creating leagues,etc.). The events are handled by the controller logic, which serves as the event consumer for these events. Another type of event that drives our application are changes in stock price. This is used to execute limit, stop, and stop limited orders.

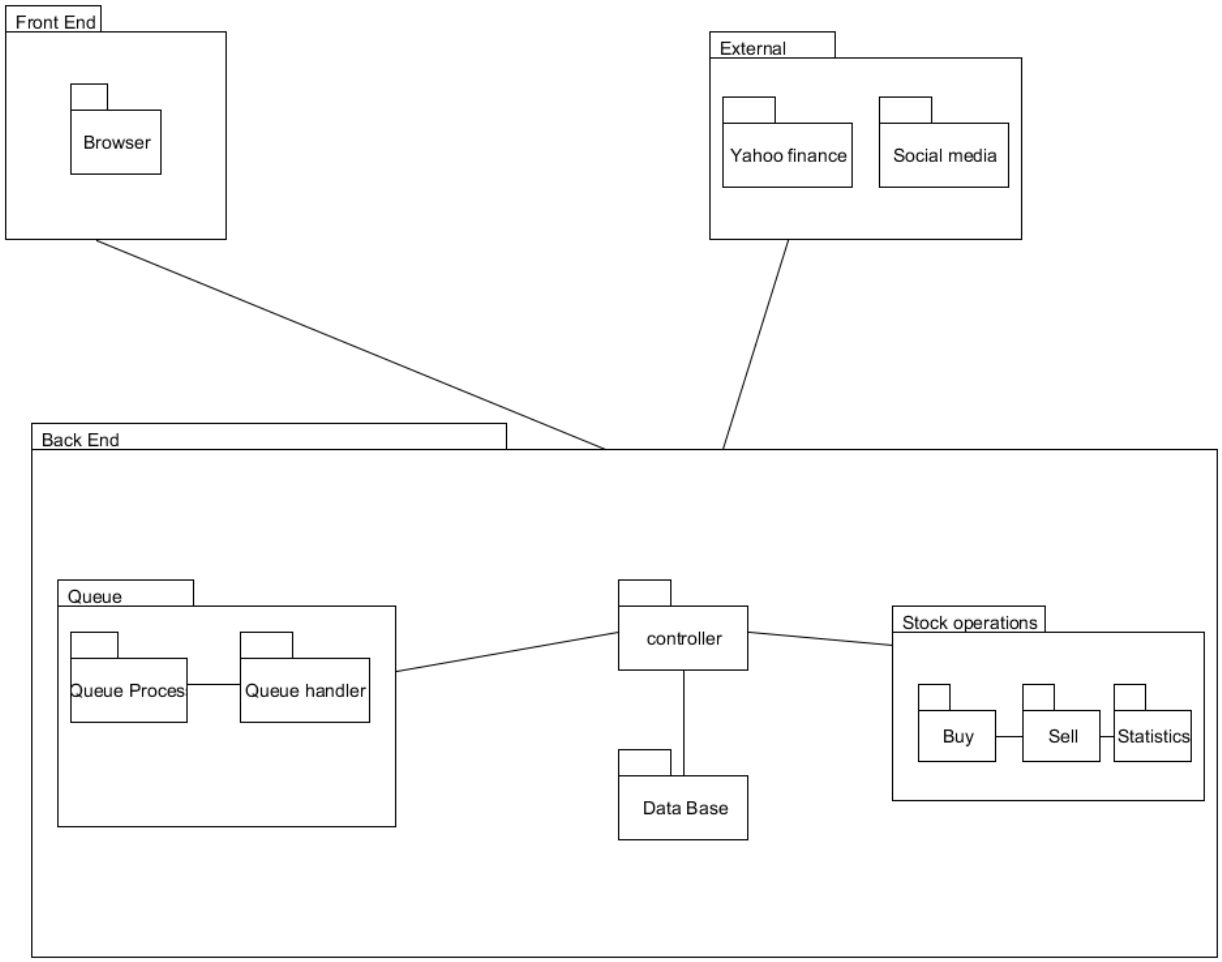
3.1.4 Object-oriented

The responsibilities are divided into different objects, which contain relevant information/data and behavior. In our application, we are planning to use object oriented approach, because it will make our work easier as well as efficient. We can represent Portfolio, Securities, League, and Orders as objects.

Identifying Subsystems

RU investing works as an educational game for finance classes at Rutgers University. The website contains three main subsystems: the visible side to the user which will be the front end of the website, the back end of the system where the technical operations take place and the external end which is used as data source.

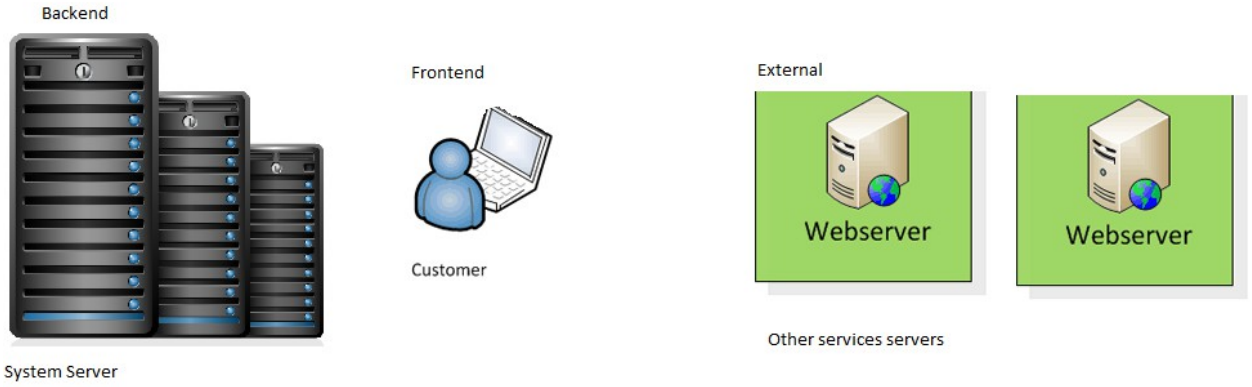
We can see that most of the essential operation occurs in the back end of the system. Essential part of the back end system is the controller which links requests from the queue as buying, selling and show statistics from the user and then assigning those requests to the request handler. The queue will store all requests from the user then transfer them the controller which will assign those operations to the appropriate subsystems. Other subsystems will have the ability to communicate with external databases like yahoo finance and social media networks.



The UML package diagram

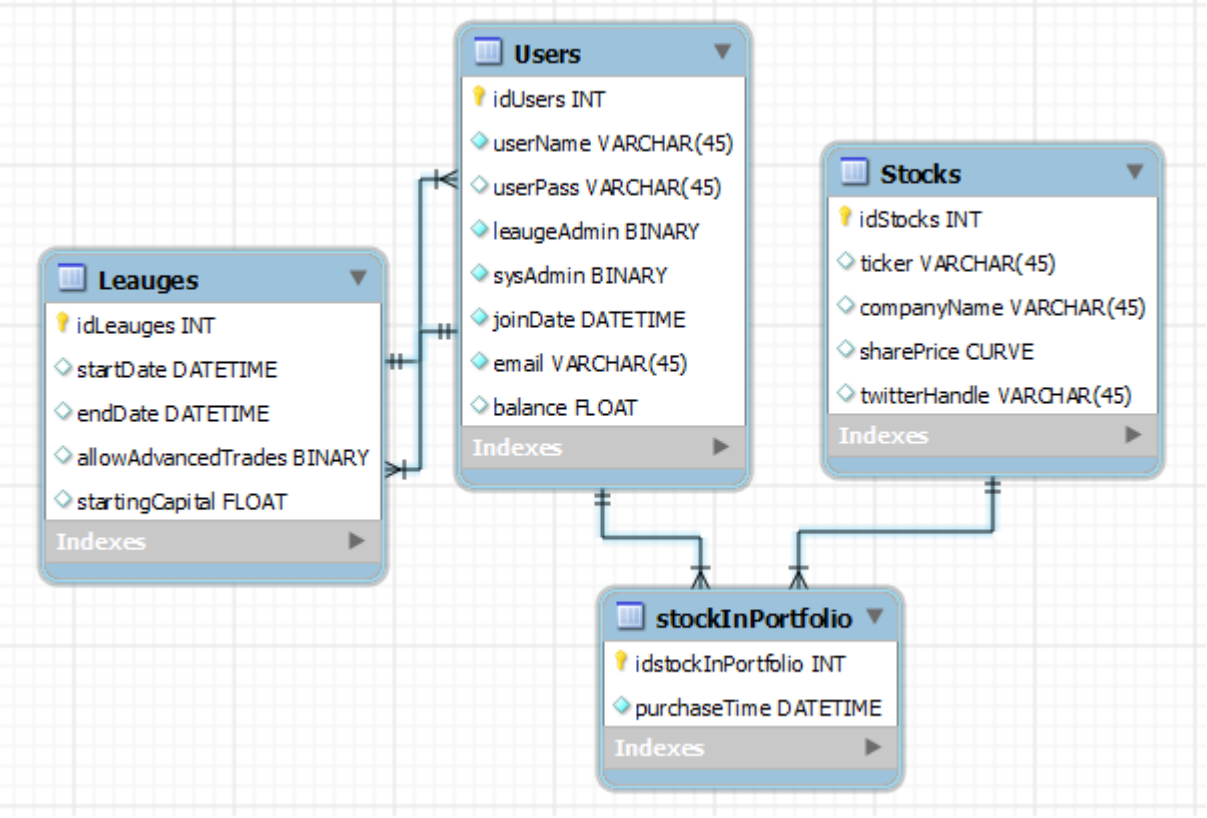
Mapping to Hardware

When running software that is based on a web page there is a standard used on how the hardware is mapped. The front end system will run on the user machine and the back end will run on the server.



Persistent Data Storage

Our project requires quite a bit of persistent storage. We will need to store records of every user and their portfolio, every valid stock, and every league. In order to make sure our system is robust, secure, and easily managed we will be using a time tested industry standard storage scheme. Specifically we will be using MySQL in order to implement a relational database. MySQL which is currently the most popular open-source database software in the world*. After careful analysis of our domain model, we have developed the following model for our planned database. Most of the model should be self-explanatory, to clarify one of the trickier aspects however, the stockInPortfolio table is used to implement a many to many function between stocks and users, while also allowing us to store the purchase time and date, to determine gains and losses over time. The reason we do not keep track of a current net worth, or rank is because all of this is highly variable and can be computed using information already stored with minimal computation.



*source: http://www.mysql.com/

Network Protocols

We will be using two different network protocols to implement our market simulation. We will be using both HTTP and Websockets. We will have a web server that will serve HTML and PHP content to the user through the HTTP protocol. Most basic operations will be through HTTP such as displaying user information and profile data.

However we will implement our trading operations by using the Websocket Protocol. The Websocket Protocol allows for lower overhead as well as lower latency response. Both of which will make our system more efficient and robust in large scale operations.

Global Control Flow

Our system will be both procedure and event driven, depending on which modules are being in use. For example, registering an account will be a procedure driven process. On the other hand, performing real time transactions will be an event driven process where the communication channels are initiated upon the event generated by the user.

Our system will be a time dependent system. For one thing, the stock market opens and closes on a timer. Also a major part of our system will be the event driven real time transactions.

On the topic of threading, our system will be a multi-threaded system. We will need multi-threading to achieve optimal performance during times where there are multiple concurrent connections.

Hardware Requirements

We will be using an online hosting provider that will provide us with a PHP backend and a MYSQL database. The server must include the ZMQ library as we will be using it for passing messages between the client and the server.

The host server also must have a minimum of 1 TB storage space with options to expand upon new users. The network connection must also be extremely stable and at least 50Mbps in order to meet the demand for real time market transactions.

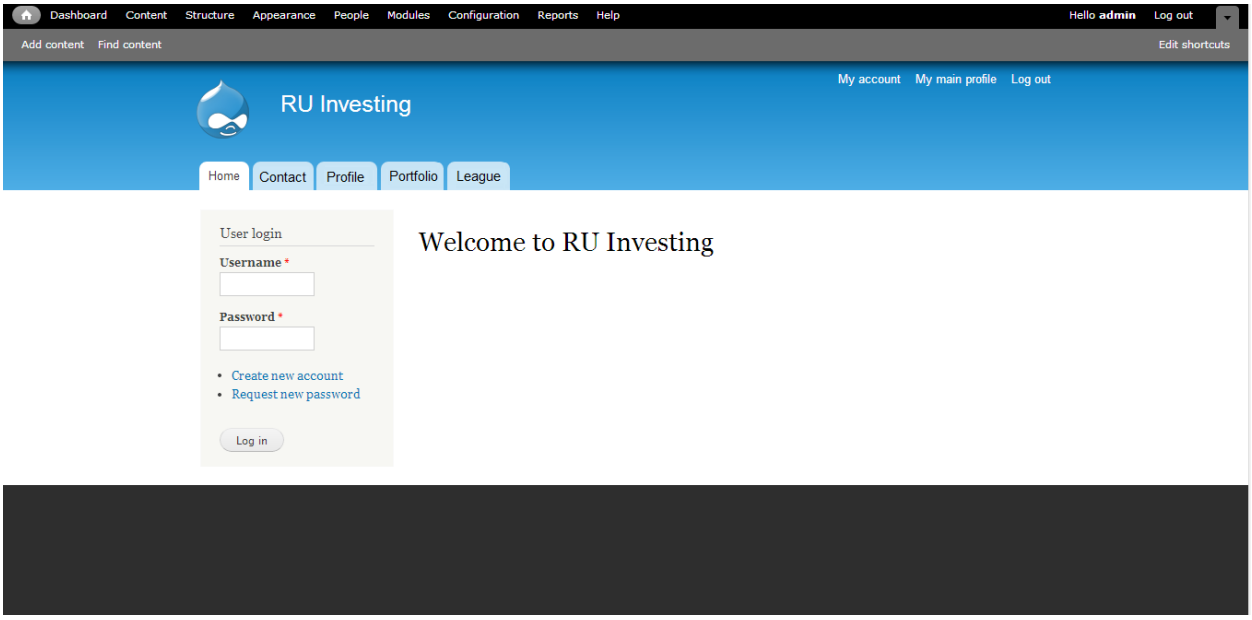
Algorithms and Data Structures

One of the key benefits of the architecture of our design is that it avoids using any sort of complex data structures. By using a database as a central communication hub we can simply use SQL request to pull arbitrary data, based on all sorts of variables. This avoids the need for trees and linked list and other complex data structures. Our php code will merely have to work in primitives, mainly the associative array. Like wise our algorithmic requirements are fairly simple. Most of our requirements such as keeping track of net worth and making trade involve simple addition and subtraction across entries returned from a SQL query.

User Interface Design

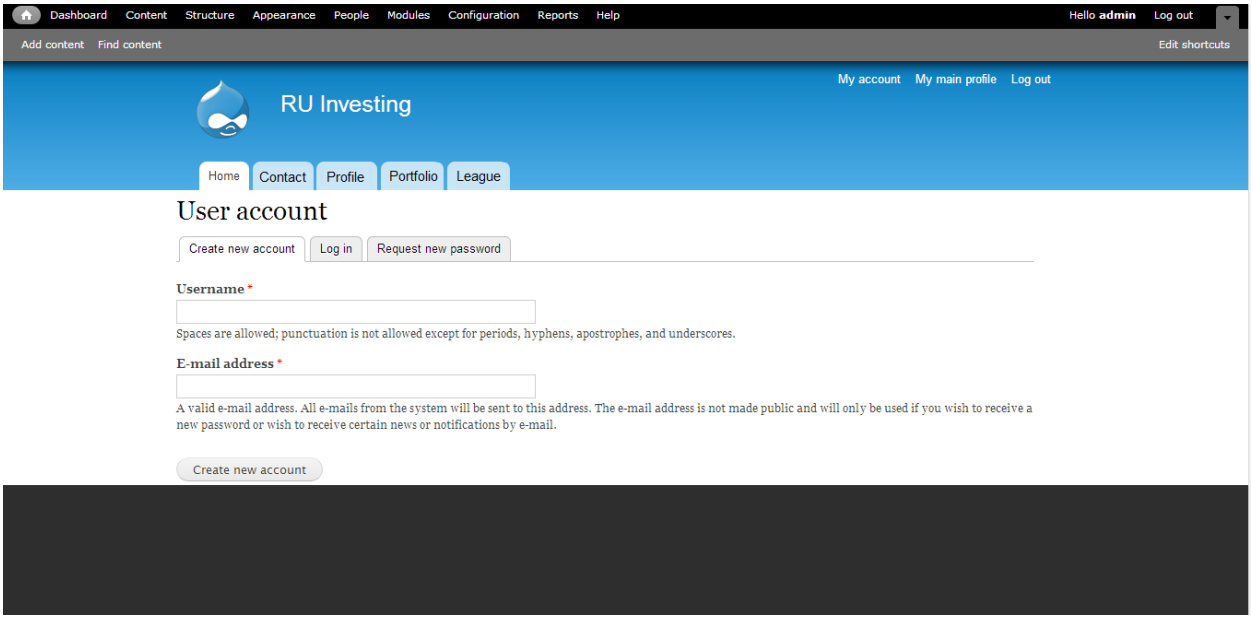
Home/Login Page

When the users first visits the website, This is the homepage that they will be greeted with. The home page allows people to log in or create a new account. The home page also will allow users to request their password if they forget it.



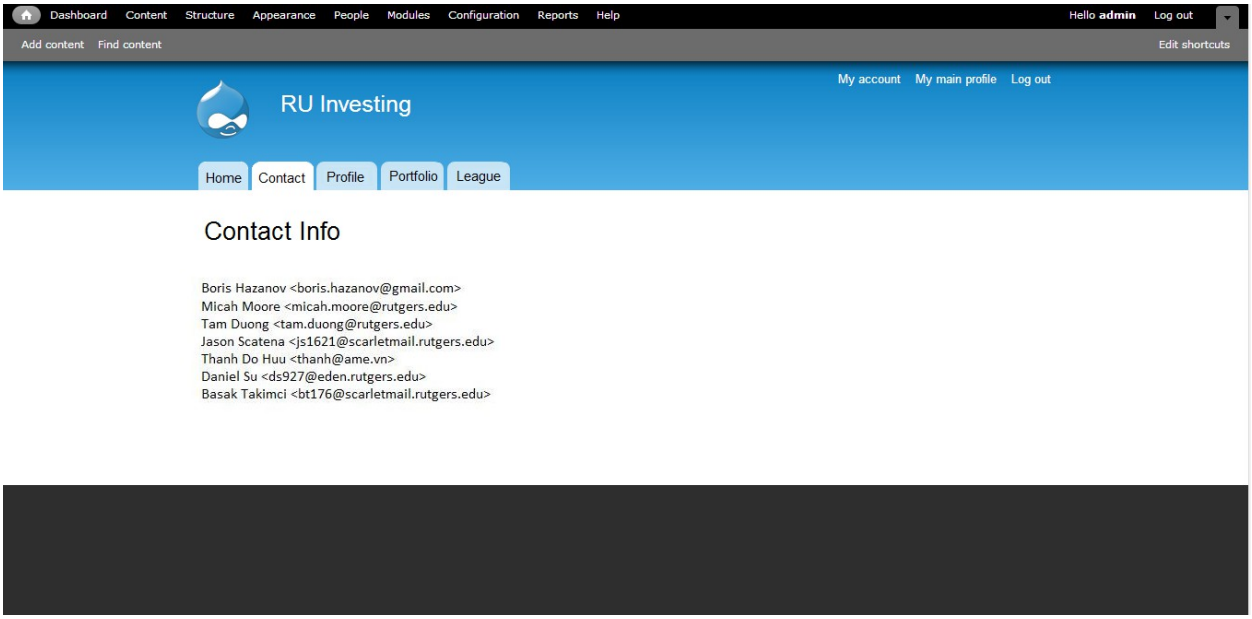
Sign Up Page

The sign up page can be accessed from the home page by clicking “create a new account” link. The users will then be instructed to create a new account. The requirements for the creation are username, email address and password.



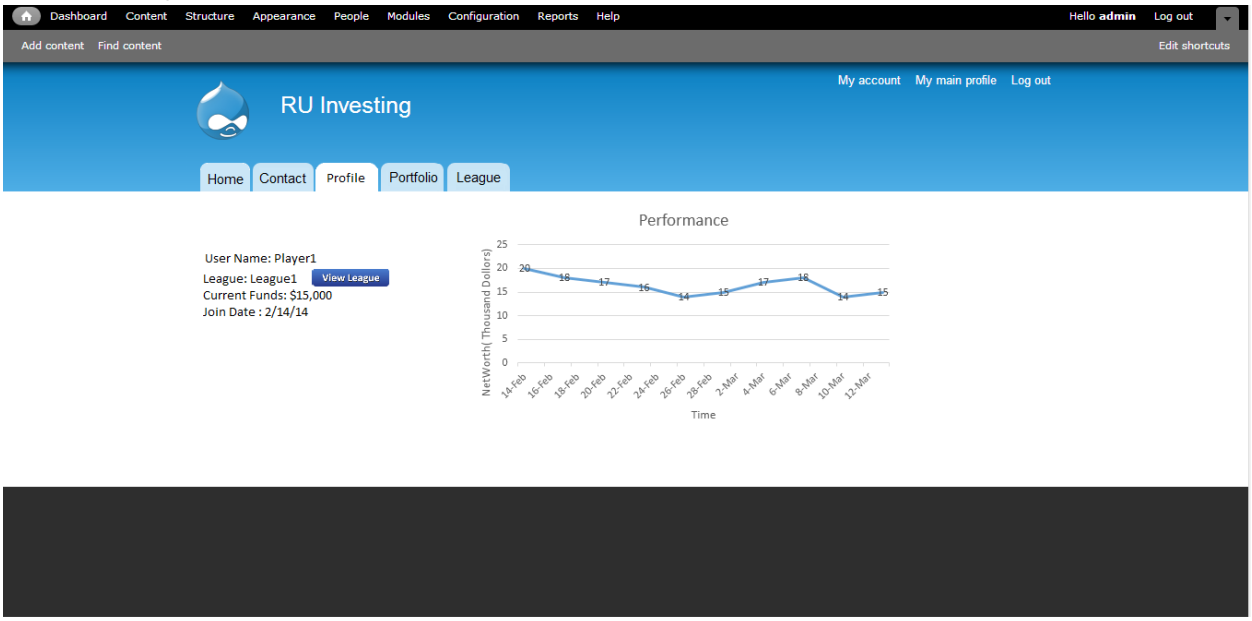
Contact Info

When users click on the contacts page, he or she will get information about the site administrators. The users can acess this page by clicking on the contacts.



Profile Page

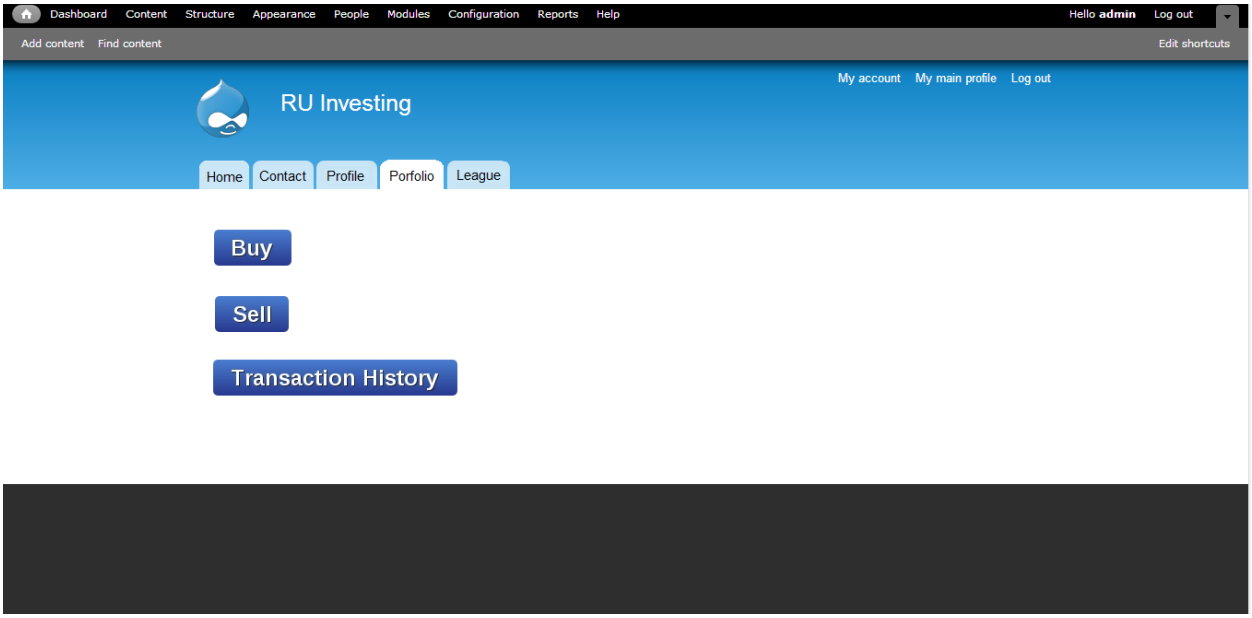
Profile page has general information about the user. User can see his or her balance, the day that the user user joined the website, a performance graph that shows the user's performance in last 30 days.



'

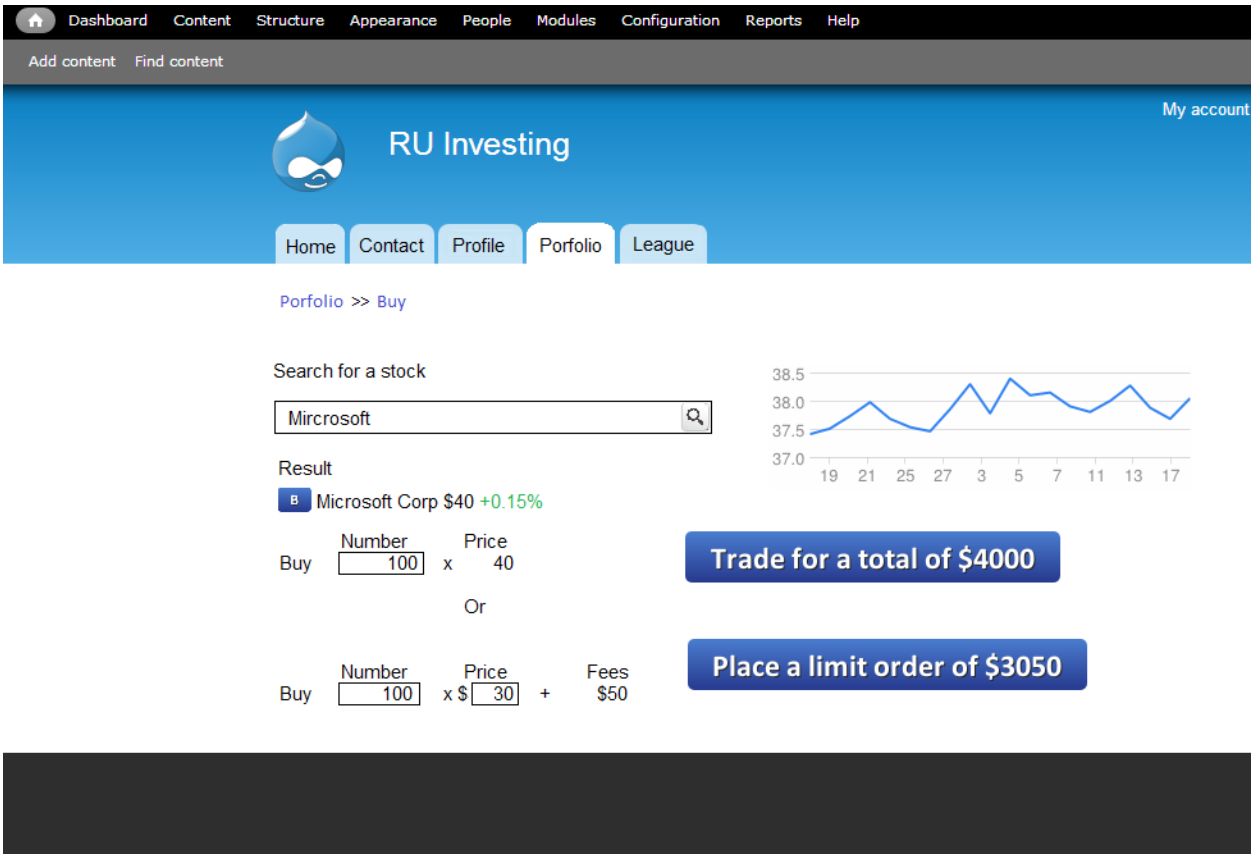
Portfolio Page

In this page the user can buy,or sell stocks. Portfolio page can be accessed after login.



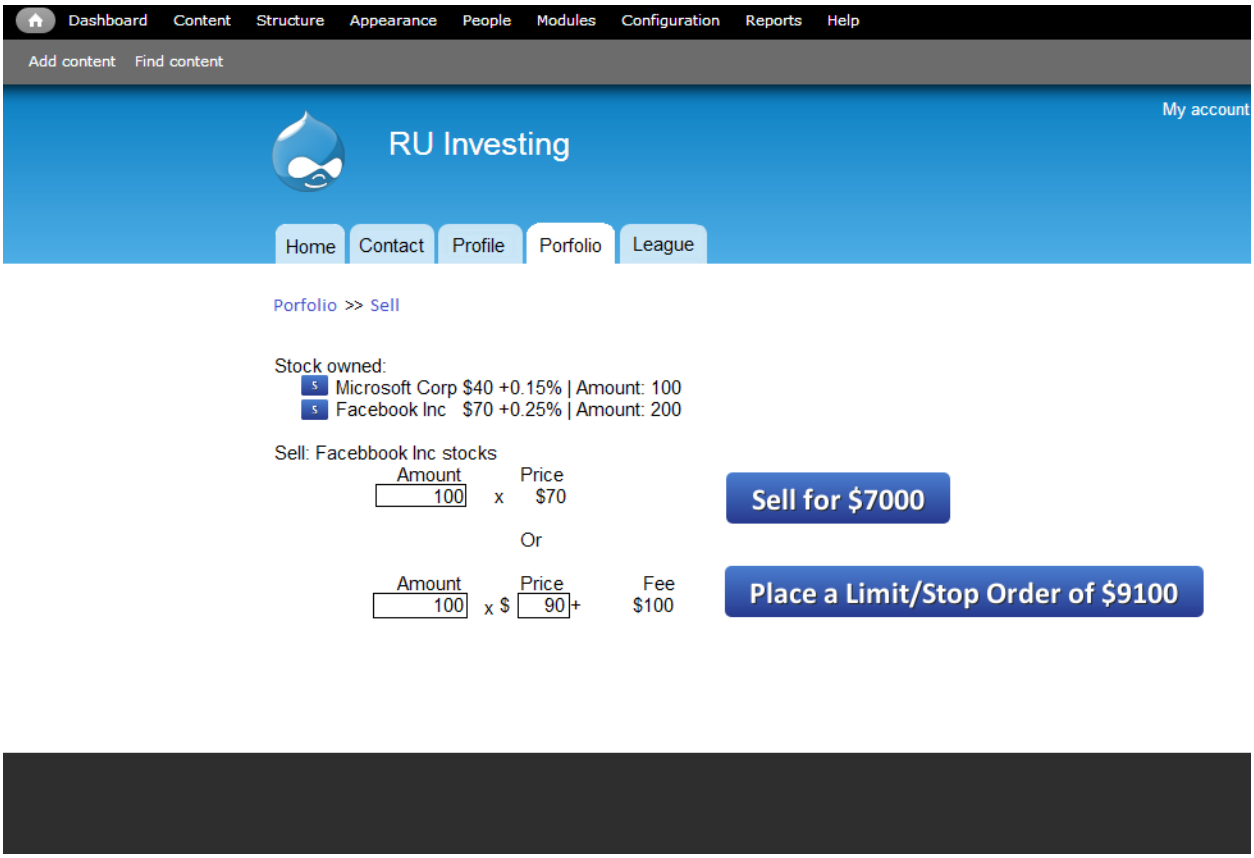
Buy

The user can access the this page by clicking on the buy button from portfolio page. This page main purpose is for the user to search and buy stocks. After selecting the stocks(by clicking a small B button on the left of the resulted stock brands) and type in the trade amount, the users can either make an immediate transaction by clicking “Trade for a total of ….” or place a limit of order for a better profit by fill in the form and selecting the corresponding button.



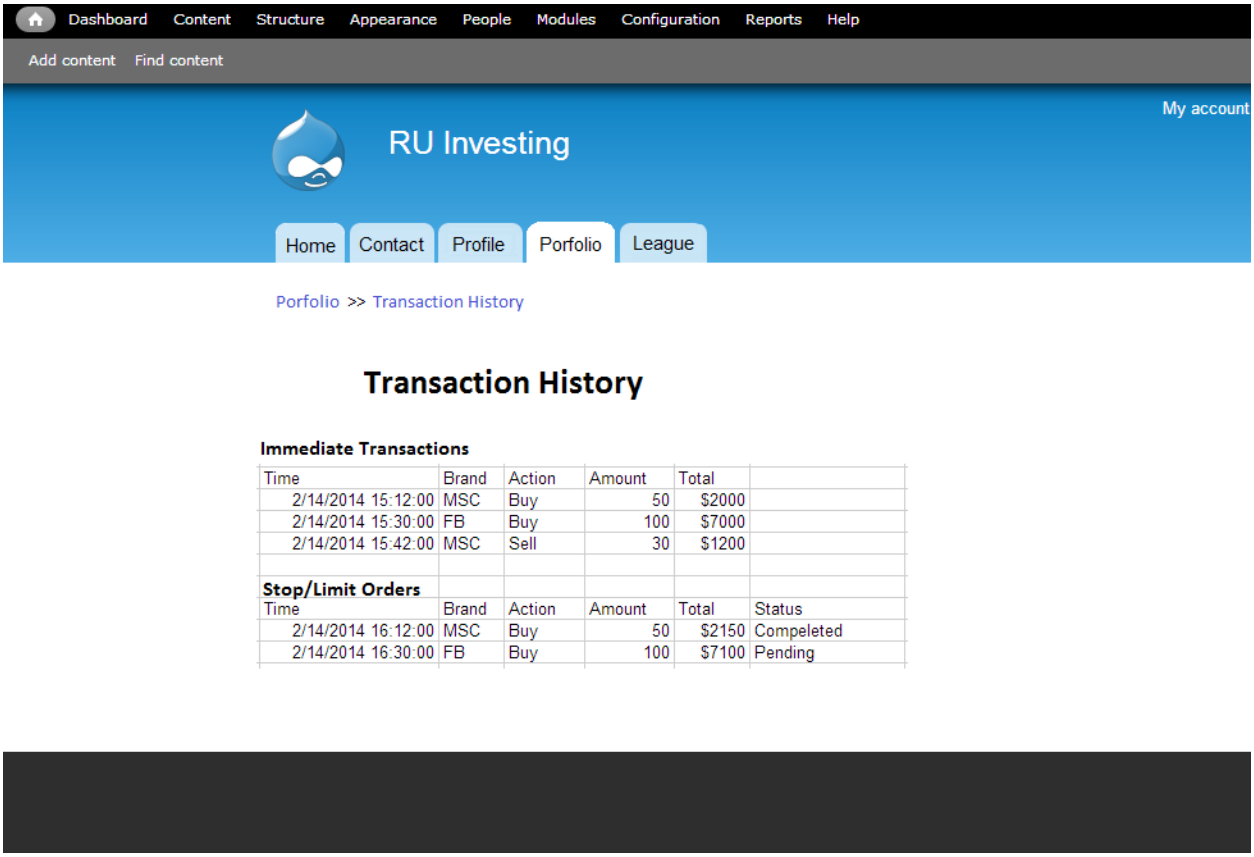
Sell

When the users select the sell button, they will be able to see their owned stocks. Similar to functions in “Buy”, you can make an immediate transaction by clicking “Sell for …” or place a limit/stop order buy fill in the form and selecting the corresponding button.



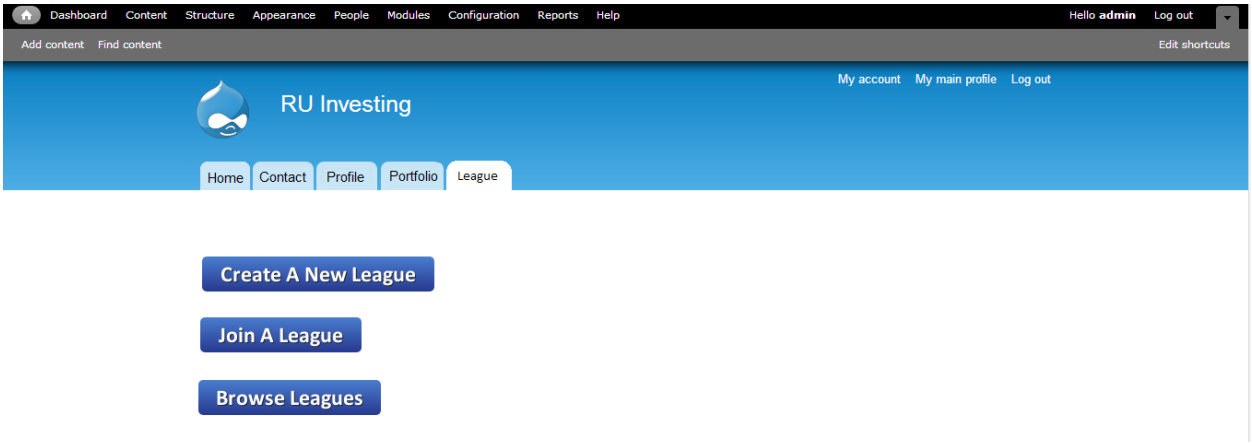
Transaction History

Transaction history is a statement that show all transactions made by the user. The user can see when they buy or sell stocks, the amount, total spent and stock brands. In transaction history the user also can see the status of the stop/limit orders that they placed where those transactions are completed or still pending.



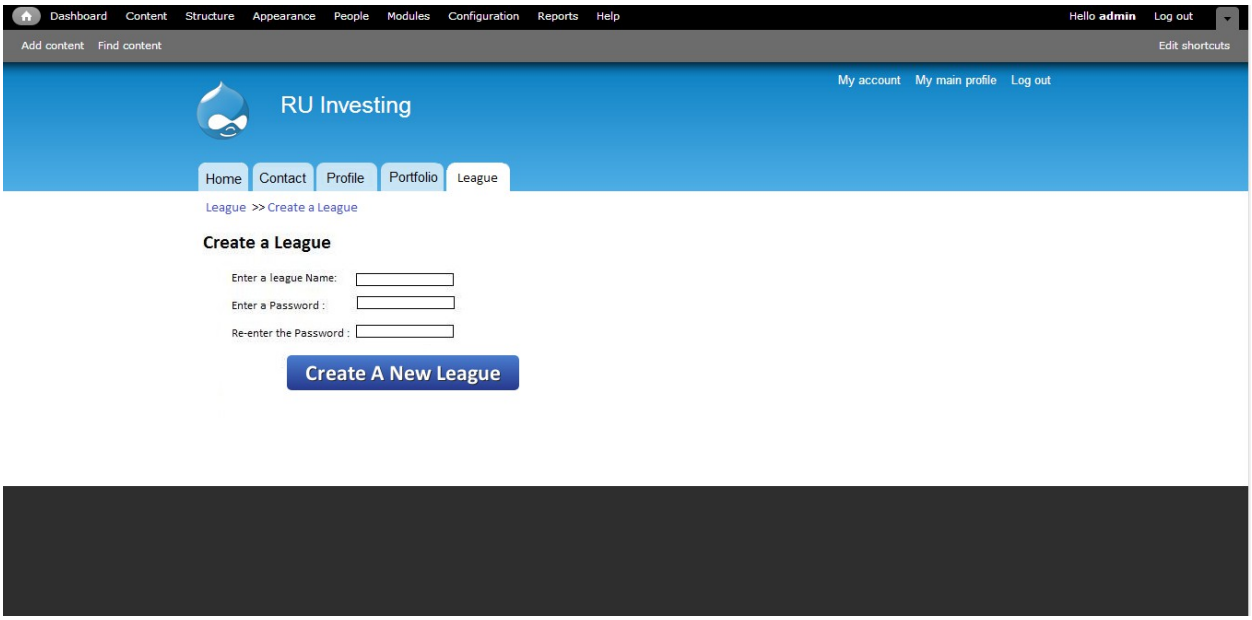
League Page for user without a league

This is the League Page that is view by users who are not in any league at the moment. The user can opt to create a new league, join in a league or browse leagues.



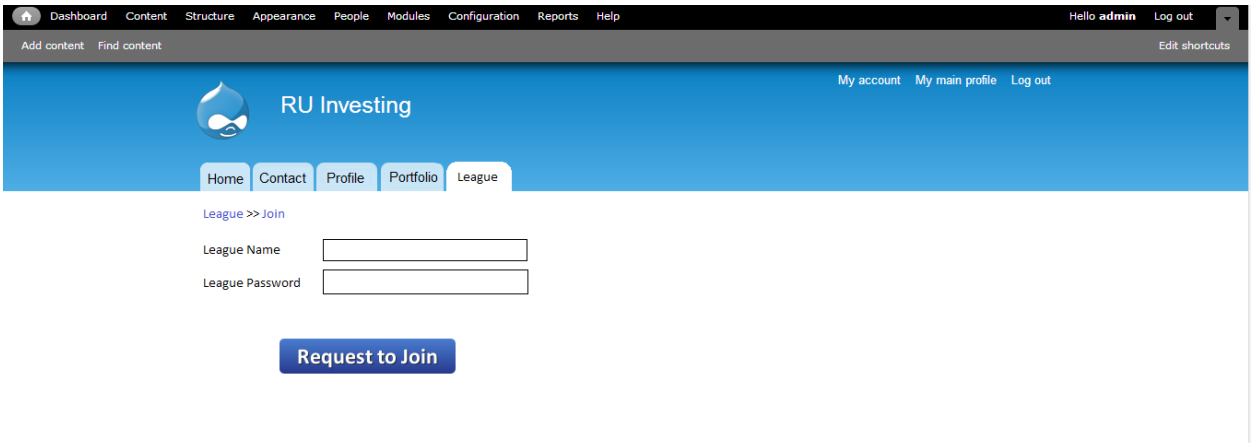
League Creation

The user need to enter a name, password to create a new league. Given that the password is retype correctly and the league name is unique, a new league will be created with the user being the league's the admin and also the first user.



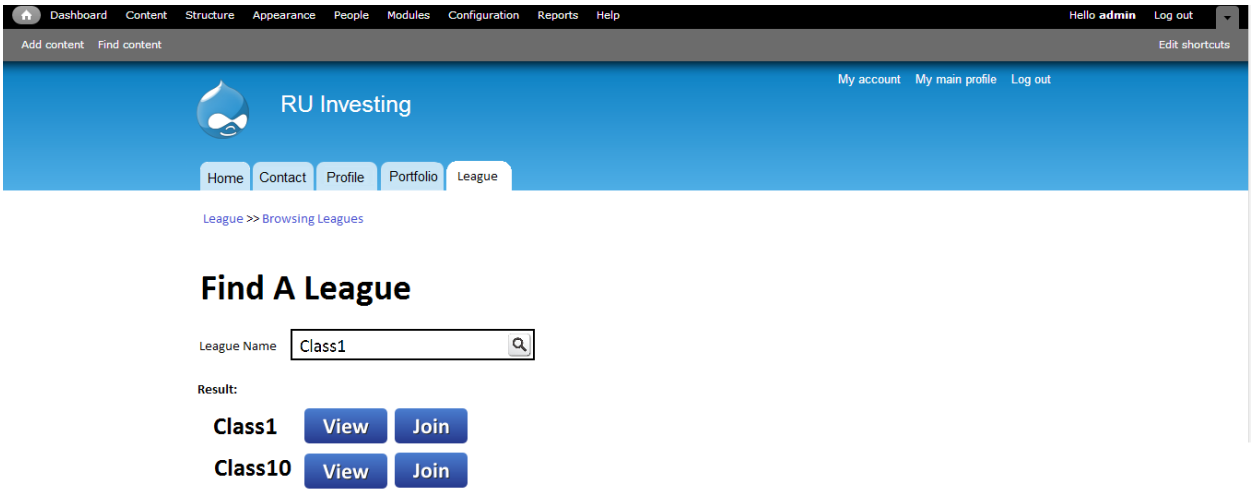
Join A League

In order to join the league, the user has to know league and password. By clicking “Request to join” button after finished filling other information, a request will be sent to the league’s admin, waiting to be processed if the correct data is entered.



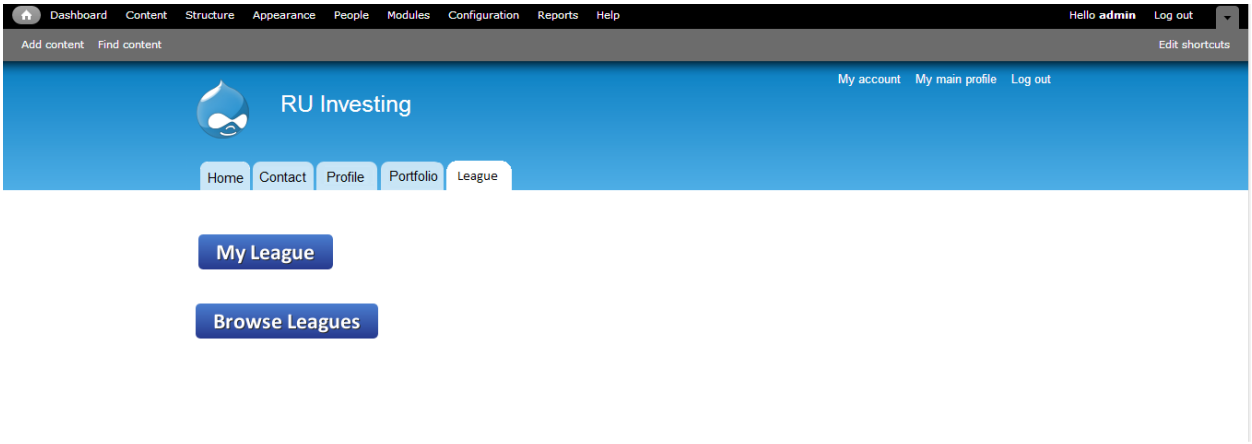
Browsing Leagues

This page can be accessed by clicking “Join a League” button in League page. User can easily search for a league by typing its name in search box then press Enter or click the search button. From the results, clicking View button will direct the user to The League’s Page, clicking Join Button will direct to Join League Page with the league name box auto-filled with the selected league’s name



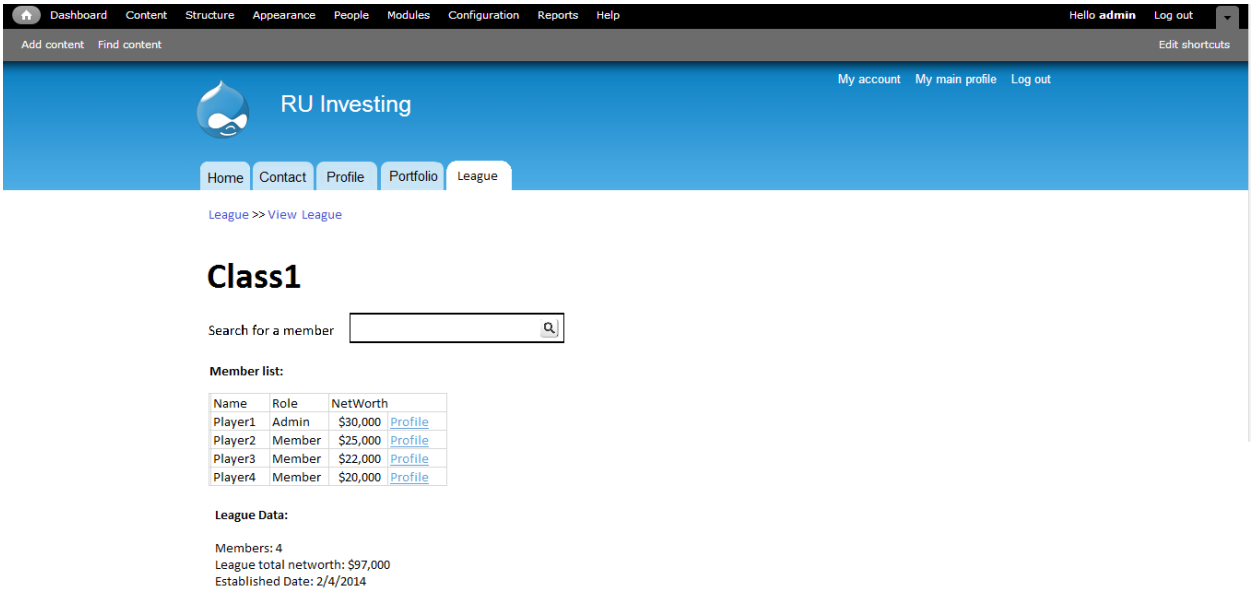
League Page for users already in a league

The users who are already in a league can click on “My League” button to view their league. They can also browse other leagues for information.



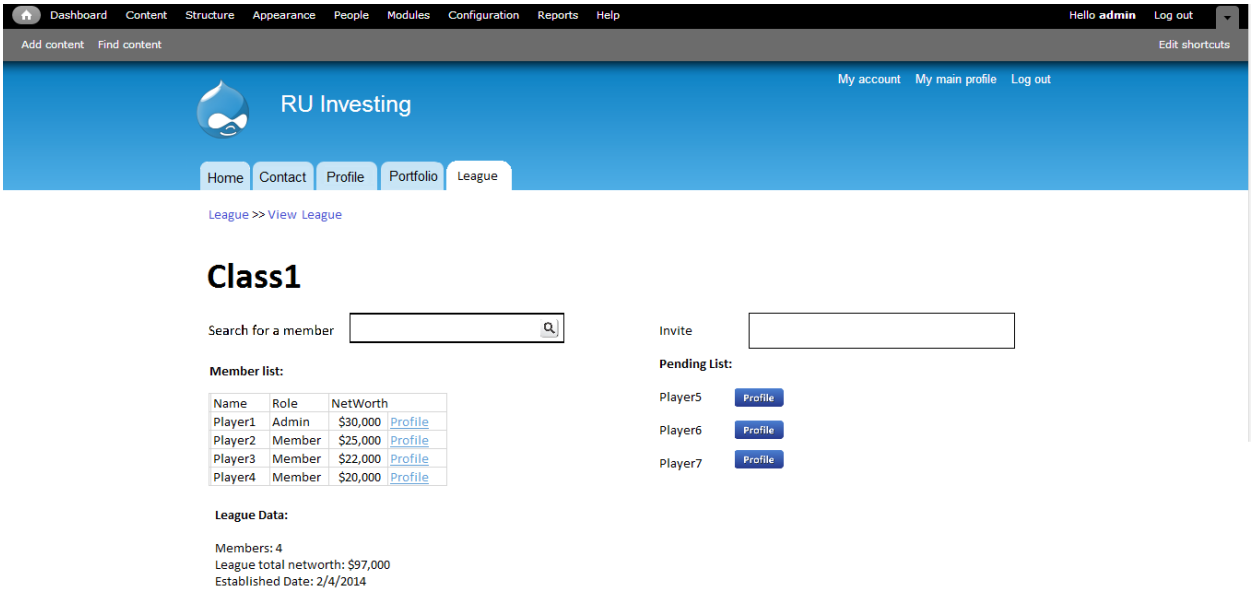
League Page(Members)

The users will be able to check their league's profile. Any member of the league can easily view other's profile by click the "Profile" hyperlink next to their NetWorth.



League Admin's League Page

Similar to the one of league members with an option to invite people to join the league and a list of pending requests of users who want to join in.



Test Cases

Test –case : TC-1 Function tested: Mouseover Pass/Fail criteria: The test passes when the user moves the cursor to a specified region and the region is highlighted and or dialog box pop up. The test fails if criteria above doesn't happen.	
Test Procedure	Expected Results
Test: • The cursor will be moved to a certain location. • Keep cursor on the same location for a certain period of time. Fail: • The cursor will be moved to a certain location.	Pass: • Word is highlighted. • Dialog box pops up and the relevant information displayed. Fail: • Word is not highlighted, script error.

Test Case -1

Test –case : TC-2 Function tested: Send request. Pass/Fail criteria: The test passes when the user send http request and capture the response.
Test Procedure
Pass: • The user clicks a hyper link button and waiting for response. • When there is event caused by a user requesting an http page(refresh). Fail: • User hits the button or the hyper link again.

Test Case 2

Test –case : TC-3 Function tested: Check market time. Pass/Fail criteria: The server will be able to check the market time.
Test Procedure
Pass: • The system will ask from the function to deliver open/close market status. Fail: • The system will ask from the function to deliver open/close market status.

Test Case-3

Test –case : TC-4
Function tested: Form send.
Pass/Fail criteria: The form will be sent and the received data will be checked for invalid entries.

Test Procedure

Pass:

- The system send and receiving a form and accept it information

Fail:

- System cannot send form.
- System cannot accept form.

Test Case-4

Test-case: TC-5 Function Tested: InvalidSymbol Pass/Fail criteria: The test passes if the user enters an invalid ticker symbol and the web page displays “Invalid Ticker Symbol”. The test fails if the user enters an invalid ticker symbol and the web page doesn’t display anything, or the web page displays a stock price.	
Test Procedure	Expected Results
Pass: • The user enters an invalid ticker symbol in the text box in the buy page of the website Fail: • The user enters an invalid ticker symbol in the text box in the buy page of the website	Pass: • The page displays the stock ticker symbol entered followed by “Invalid Ticker Symbol” message. Fail: • The page displays nothing, or it displays the stock ticker symbol entered and an invalid price

Test-case: TC-6 Function tested: FetchStockData Pass/Fail criteria: The test passes if the web page accurately displays the ticker symbol entered and the current stock price of that ticker next to it. The test fails if the web page displays the incorrect price or displays nothing.	
Test Procedure	Expected Results
Pass: • User enters a valid stock ticker symbol in the text box in the buy tab of the web page. Fail: • User enters a valid stock ticker symbol in the text box in the buy tab of the web page.	Pass: • Web page displays the stock ticker symbol followed by the current market price next to it. Fail: • Web page displays nothing, “Invalid Ticker Symbol”, the incorrect ticker symbol, or the incorrect price.

Integration Testing

RUInvesting will be tested using the bottom up strategy. Each part will be tested individually first, then after integration(in case race conditions occur). Each test must also be run in each possible state and time(to make sure it behaves properly whether the market is open or closed). We implemented this strategy in hopes it will allow us to catch bugs at a lower level and pre-emptively fix the bugs at the top level.

mouseOver:
The mouseOver test will verify the mouseOver function is working properly (when the mouse is hovered over a certain field the field should become highlighted). This is important if there are many small fields close to each other, the user could have difficulty distinguishing between them, so the mouseOver functionality will help them differentiate.

sendRequest:
The sendRequest test will verify that our http request handler is working properly. If the user is unable to navigate throughout our website, they will not be able to access information, make trades, or even register for that matter.

checkMarketTime:
The checkMarketTime test will ensure that our system will be able to know when the market is open and when it is closed. This is critical because regular trades cannot occur when the market is closed.

formSend:
The formSend test will verify that the relevant form is sent to the correct address and verify that sent data is received.This test is critical because communication between user-system, system-server and system-system need to work accurately and fast.

InvalidSymbol:
The InvalidSymbol test will verify that if an invalid symbol is entered in the text field of the market transaction tab the web page will display an “Invalid Ticker Symbol” message. This test is important because we only want to display prices of valid stocks.

FetchStockData:
The FetchStockData test will confirm that when a valid stock ticker symbol is entered in the text box of the market transaction tab of the web page, the current market price per share of that stock will be displayed. This is very important because it’s at the heart of our fantasy game. If the user is unable to view current stock prices, he/she will not be able to make informed buying decisions.

Project Coordination and Progress Report

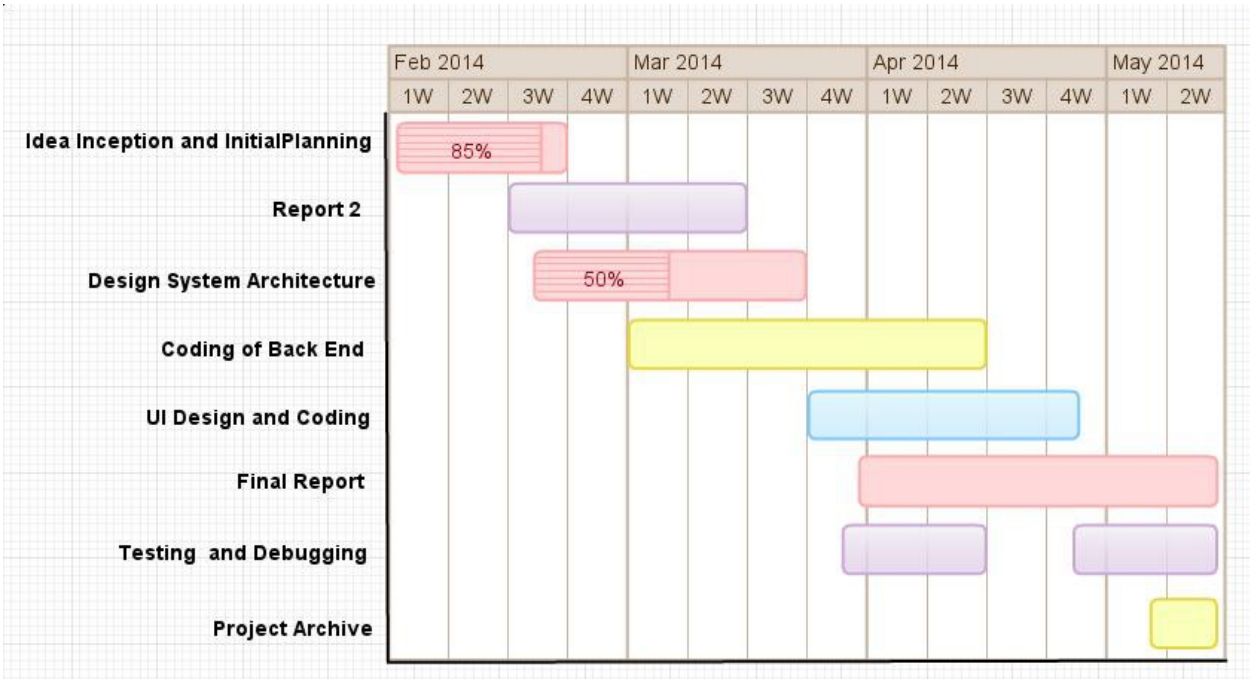
Since beginning to implement our system we have made some great progress. We have started a web server to host our application with a functional UI and SQL server. We decided early on that your first goals should be to create a running website, to add future work to, and to create the complete database. Working on the database in it's entirety rather than implementing use cases one by one means that it will take some time before we can start crossing use cases off completely, but it means that we will have a consistent system, with minimal redundancy of data and code. It also means that after we do a bulk of the "heavy lifting" so to speak, many of the use cases should be quite easy to implement fully.

We have managed to create several key web pages, along with the logic behind them so as to implement multiple of our use cases. Use case UC-1 has been addressed with a functional user registration and login system, accessible through the homepage of our system. We have begun to implement the interface that each user will see when they enter the web service. this is the interface that will allow the user to initiate all the other use cases. We have started to implement a user sorting and classification system that will allow us to implement UC-3, UC-5, UC-6, and UC-7. While we have already done a great deal of the leg-work for this, notably database work and web page design, a fully completed version is contingent on completing the database work. In the process of beginning the database we have also progressed on implementing the profile page(UC-4), which must access relevant data through the appropriate tables. Making trades, UC-2, is another major use case contingent on the completion of the database. Our progress has been steady and well paced and we expect this trend to continue barring unforeseen problems in implementation of future features.

Plan of Work

If we look at the gantt chart laid out in our previous report it is clear that we are on track with our original plan. We have started work on our UI earlier than expected due to how much it is intertwined with the main logic of our web application. As mentioned above we have spent a majority of coding time on the back-end work as shown in the gantt chart. For two weeks of work we are easily ⅓ done with the work as expected from the chart. As we have gotten further in the project we can now elaborate further on how we are breaking up "Coding of

Back-end” as shown below.



ID	Task Name	3 Mar 2014	10 Mar 2014	17 Mar 2014	24 Mar 2014	31 Mar 2014	7 Apr 2014	14
		M T W T F	M T W T F	M T W T F	M T W T F	M T W T F	M T W T F	M
1	Coding of Backend	20%						
1.1	Webserver Implmentation	70%						
1.2	Database Implmentation	33%						
1.3	User Managment Implmentation			25%				
1.4	Stock and Trade System Implmentation			10%				
1.5	Twitter Integration					0%		