Electronic Asset Trading Platform

Major Project CAB 302

Milestone 2

Group 2:

Name	Student ID
Sam Carran	N9469818
Jian Sheng Lee	N10164651
Nicholas Meurant	N10485571
Sophia Politylo	N10489045

Table of Contents

Background

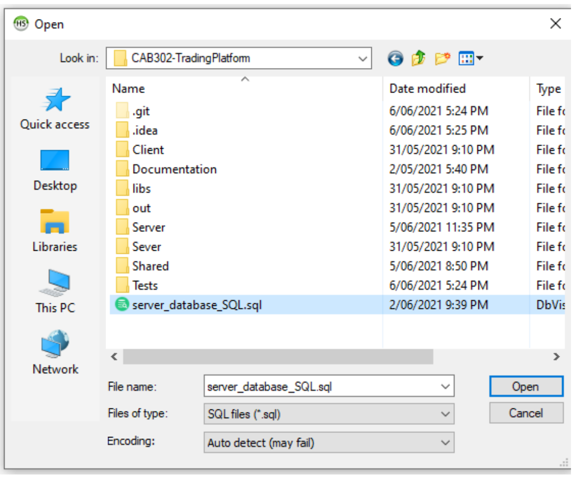
External Libraries used:

To create the Electronic asset trading platform solution 2 external libraries have been used. One is mariadb-java-client-2.72.jar. This fil allowed the server to connect to a database server that was utilizing the MariaDB framework. The other external library that has not been created is by the team is the ByCrypt java file within the shared src directory. This java class was a sourced from Damien Miller, with the team having no contribution to the functioning of this class. The ByCrypt java class enables the created program to hash String's and compare un-hashed and hashed String to each other.

Solution deployment:

Database creation:

To enable the created application to interface with a database server a MariaDB server will either have to be purchased or ran on the local server machine. If you are going the root of running the server on the server application machine the following link: https://downloads.mariadb.org/ can be used to download the MariaDB server application. When installing a MariaDB server there are a few variables which are important to remember to allow other application to connect to the server. As you will be prompted to set a password for the root user on setup. Remember this passwords as it allows full access to the server. Once MariaDB is installed use your preferred database managing software to access the created server, to login type in the url of the server localhost if it is on the machine your using otherwise type in the ip for the computer it is running on. Along with this you can set the user to root and the password to the password you created when installing the server. From here it is assumed you are using HeidiSQl given it comes packed with the MariaDB server. Once you have successfully connected to the server go to the top ribbon bar and press file>load SQl file. From there navigate to the directory that the project has been saved in and select server_database_SQL.sql and click open.

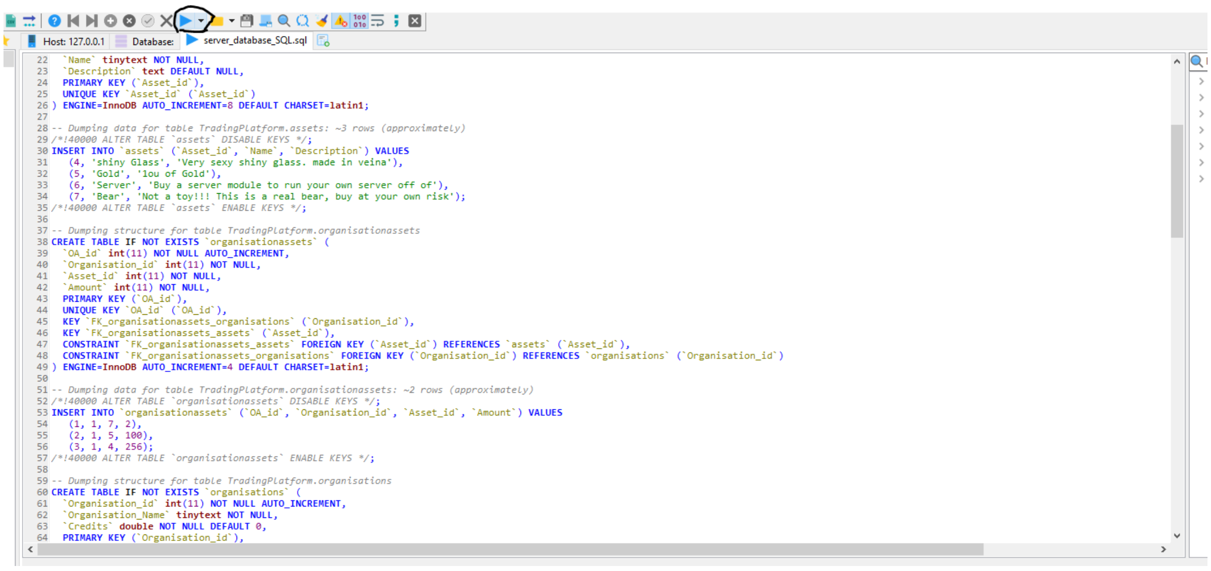


Look in: CAB302-TradingPlatform

Name	Date modified	Type
.git	6/06/2021 5:24 PM	File fc
.idea	6/06/2021 5:25 PM	File fc
Client	31/05/2021 9:10 PM	File fc
Documentation	2/05/2021 5:40 PM	File fc
libs	31/05/2021 9:10 PM	File fc
out	31/05/2021 9:10 PM	File fc
Server	5/06/2021 11:35 PM	File fc
Sever	31/05/2021 9:10 PM	File fc
Shared	5/06/2021 8:50 PM	File fc
Tests	6/06/2021 5:24 PM	File fc
server_database_SQL.sql	2/06/2021 9:39 PM	DbVis

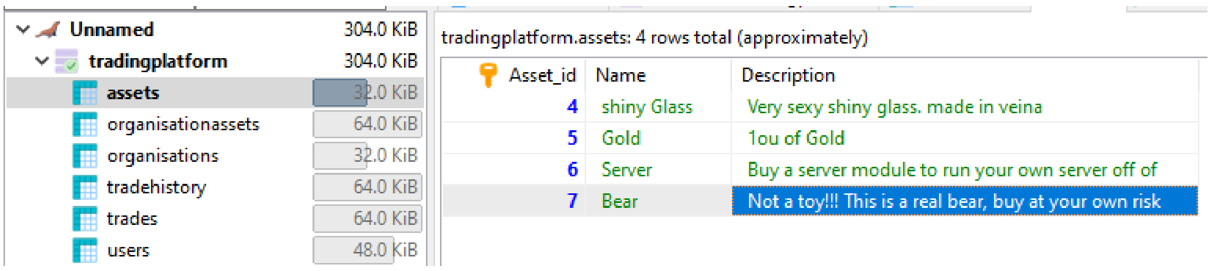
File name: server_database_SQL.sql
Files of type: SQL files (*.sql)
Encoding: Auto detect (may fail)

From there the SQL should be able ran through pressing the blue arrow circled in the figure _. Heildi will than run the SQL code and create the database required for the application. If it doesn't show up refresh the table directory via a right click and then refresh or check that Heildi is connecting to the correct database on start up.



```sql
  `Name` tinytext NOT NULL,
  `Description` text DEFAULT NULL,
  PRIMARY KEY (`Asset_id`),
  UNIQUE KEY `Asset_id` (`Asset_id`)
) ENGINE=InnoDB AUTO_INCREMENT=8 DEFAULT CHARSET=latin1;

-- Dumping data for table TradingPlatform.assets: ~3 rows (approximately)
/*!40000 ALTER TABLE `assets` DISABLE KEYS */;
INSERT INTO `assets` (`Asset_id`, `Name`, `Description`) VALUES
   (4, 'shiny Glass', 'Very sexy shiny glass. made in veina'),
   (5, 'Gold', '1ou of Gold'),
   (6, 'Server', 'Buy a server module to run your own server off of'),
   (7, 'Bear', 'Not a toy!!! This is a real bear, buy at your own risk');
/*!40000 ALTER TABLE `assets` ENABLE KEYS */;

-- Dumping structure for table TradingPlatform.organisationassets
CREATE TABLE IF NOT EXISTS `organisationassets` (
  `OA_id` int(11) NOT NULL AUTO_INCREMENT,
  `Organisation_id` int(11) NOT NULL,
  `Asset_id` int(11) NOT NULL,
  `Amount` int(11) NOT NULL,
  PRIMARY KEY (`OA_id`),
  UNIQUE KEY `OA_id` (`OA_id`),
  KEY `FK_organisationassets_organisations` (`Organisation_id`),
  KEY `FK_organisationassets_assets` (`Asset_id`),
  CONSTRAINT `FK_organisationassets_assets` FOREIGN KEY (`Asset_id`) REFERENCES `assets` (`Asset_id`),
  CONSTRAINT `FK_organisationassets_organisations` FOREIGN KEY (`Organisation_id`) REFERENCES `organisations` (`Organisation_id`)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=latin1;

-- Dumping data for table TradingPlatform.organisationassets: ~2 rows (approximately)
/*!40000 ALTER TABLE `organisationassets` DISABLE KEYS */;
INSERT INTO `organisationassets` (`OA_id`, `Organisation_id`, `Asset_id`, `Amount`) VALUES
   (1, 1, 7, 2),
   (2, 1, 5, 100),
   (3, 1, 4, 256);
/*!40000 ALTER TABLE `organisationassets` ENABLE KEYS */;

-- Dumping structure for table TradingPlatform.organisations
CREATE TABLE IF NOT EXISTS `organisations` (
  `Organisation_id` int(11) NOT NULL AUTO_INCREMENT,
  `Organisation_Name` tinytext NOT NULL,
  `Credits` double NOT NULL DEFAULT 0,
  PRIMARY KEY (`Organisation_id`),
```

Once the SQl has been ran a database called tradingplatform should have been created with 6 tables with a few dummy organisations and assets along with a few users.



Unnamed	304.0 KiB
tradingplatform	304.0 KiB
assets	32.0 KiB
organisationassets	64.0 KiB
organisations	32.0 KiB
tradehistory	64.0 KiB
trades	64.0 KiB
users	48.0 KiB

tradingplatform.assets: 4 rows total (approximately)

Asset_id	Name	Description
4	shiny Glass	Very sexy shiny glass. made in veina
5	Gold	1ou of Gold
6	Server	Buy a server module to run your own server off of
7	Bear	Not a toy!!! This is a real bear, buy at your own risk

To enable the server to connect to this data base edit the DBConfig.cfg file with a text editor as shown like in the following figure:

```
host=jdbc:{your choosen database server}://{YOUR servers IP}/{database
name}?usePipelineAuth=false
username={username to connect to server (default is root)}
password= {password for username}
driver={java connector library to access and understand the database}
```

Where the text in squiggly brackets are variables that will change depending on your set up. Username however can be set to root for most cases (even though this is not best practise), with the password created before being used. Assuming that you have used MariaDB the driver will be set to ‘`org.mariadb.jdbc.Driver`’. With all that done the server should now be able to connect to the server.

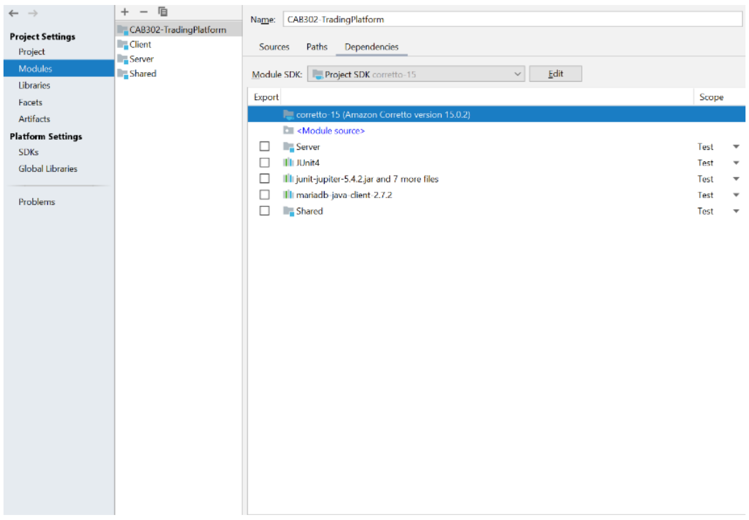
Server connection setup:

Given that this solution utilises both a client and a server two config files have been, created one within the server solution and one within the client solution folder. The server config file outlies the port and IP for the server to use, when opening a connection. While the client-side config is specifying the port and IP for the client to connect to. These files have been split up to allow for the server to be specified using local host if so desired while also allowing the client to connect to the server from another machine using the server's IP. To enable this it means that both the client and the server config file must share the same port address while the client's IP. Currently both files have been set to local host for the IP and 100 for the port.

Intellij project setup:

This project is expected to be run from Intellij with the CAB-302-TradingPlatform folder being expected to be the solutions project directory. The CAB-302-TradingPlatform folder contains the Client directory which houses the classes needed to build and display the GUI for the user to manipulate. The CAB-302-TradingPlatform folder also contains the Server directory which holds the different classes needed to set up a connection to the database. The folder also contains a shared source folder which is does not run by itself but hold parent classes and shared classes between the server and client. Therefore to correctly compile this project shared, server and client all need to be set as source directories with the src from each package being the source where the classes are stored. Within intilij project window you will need to set a dependency on the shared jar package for both the server and client so that it is able to fetch the needed classes.

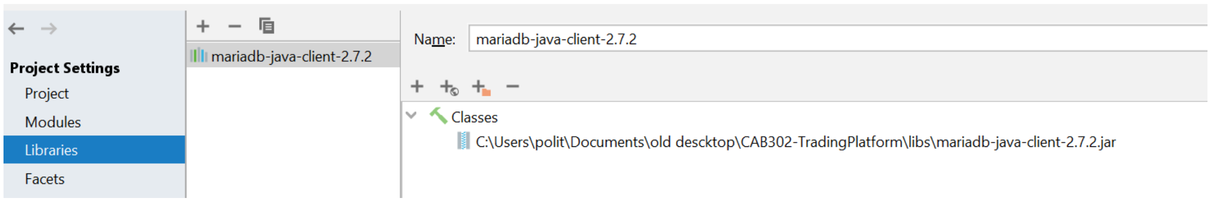
The java version that was used to compile this project was Java Amazon Corretto version 15.0.2. Along with this Junit4, Junit-jupitier 5.42 and mariadb-java-client are all dependent libraries that are need to run this project. The mariadb-java-client jar file can be found within the lib folder of the root file with that needing to be added into the intilij projects library for the project to compile.



Name: CAB302-TradingPlatform

Module SDK: Project SDK corretto-15

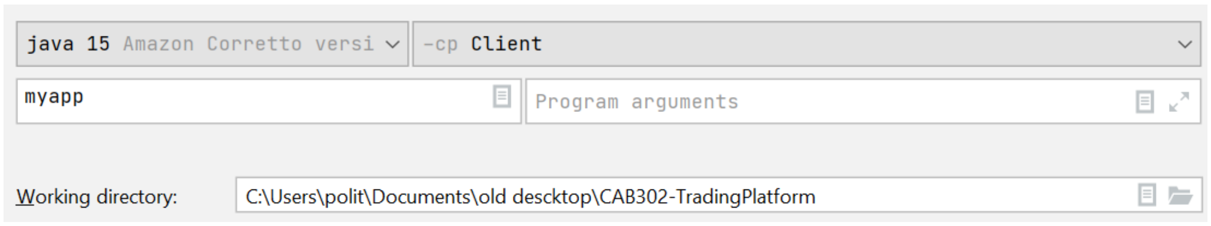
Export		Scope
	corretto-15 (Amazon Corretto version 15.0.2)	
	<Module source>	
☐	Server	Test
☐	JUnit4	Test
☐	junit-jupiter-5.4.2.jar and 7 more files	Test
☐	mariadb-java-client-2.7.2	Test
☐	Shared	Test



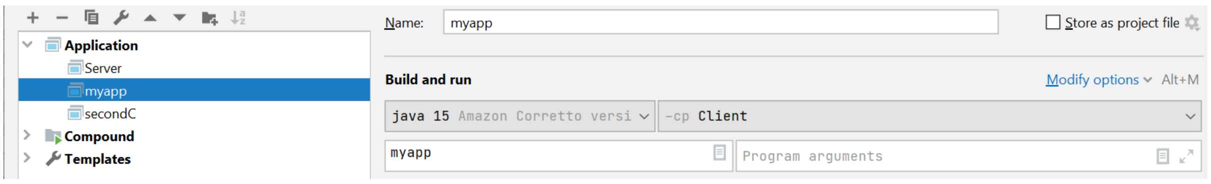
Name: mariadb-java-client-2.7.2

Classes
C:\Users\polit\Documents\old descktop\CAB302-TradingPlatform\libs\mariadb-java-client-2.7.2.jar

Once that has all been set up you can build the project, with the required run command being shown bellow for the client:


java 15 Amazon Corretto versi | -cp Client
myapp | Program arguments
Working directory: C:\Users\polit\Documents\old descktop\CAB302-TradingPlatform

And the run command for the server being shown bellow here:



Name: myapp
java 15 Amazon Corretto versi | -cp Client
myapp | Program arguments

You can than run the server and client get both running at once and click on the server gui open, than refresh button.

Than you can login as

For normal user

Name: user | password: pass

For admin

Name: politylos | password pass

Detailed Design

Java Classes

Class Diagrams

Client-Side Java Class

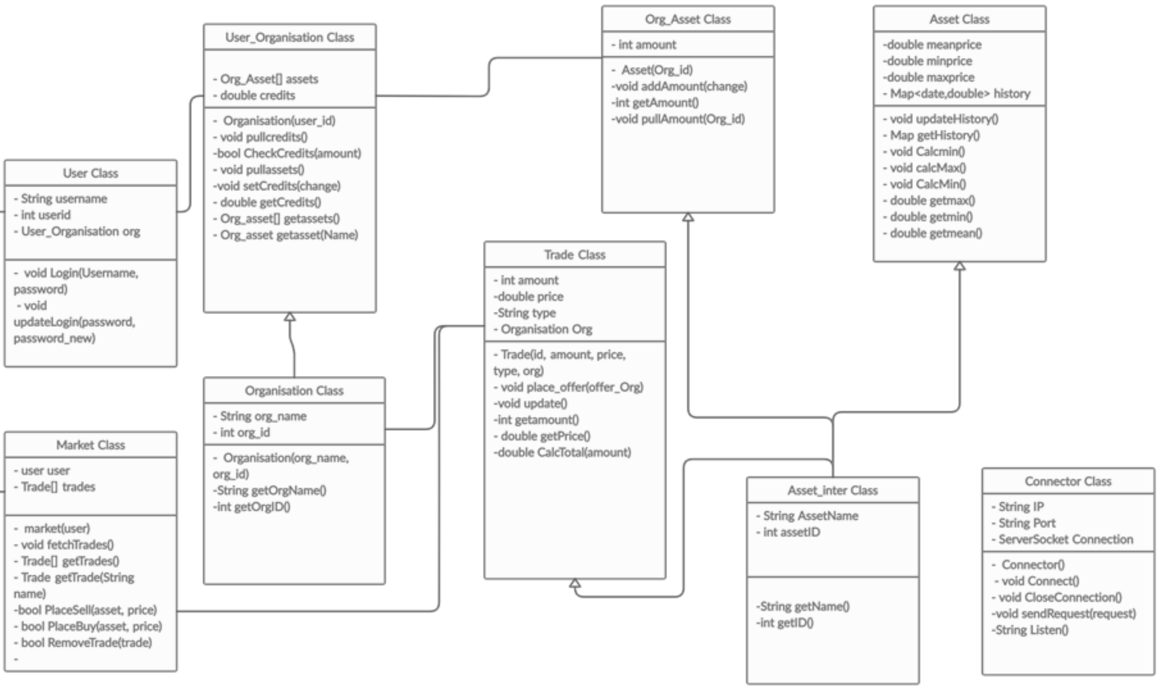


Figure 1 - Class diagram for the client-side class showing inheritance and dependent classe

Server-side Java Class Diagram

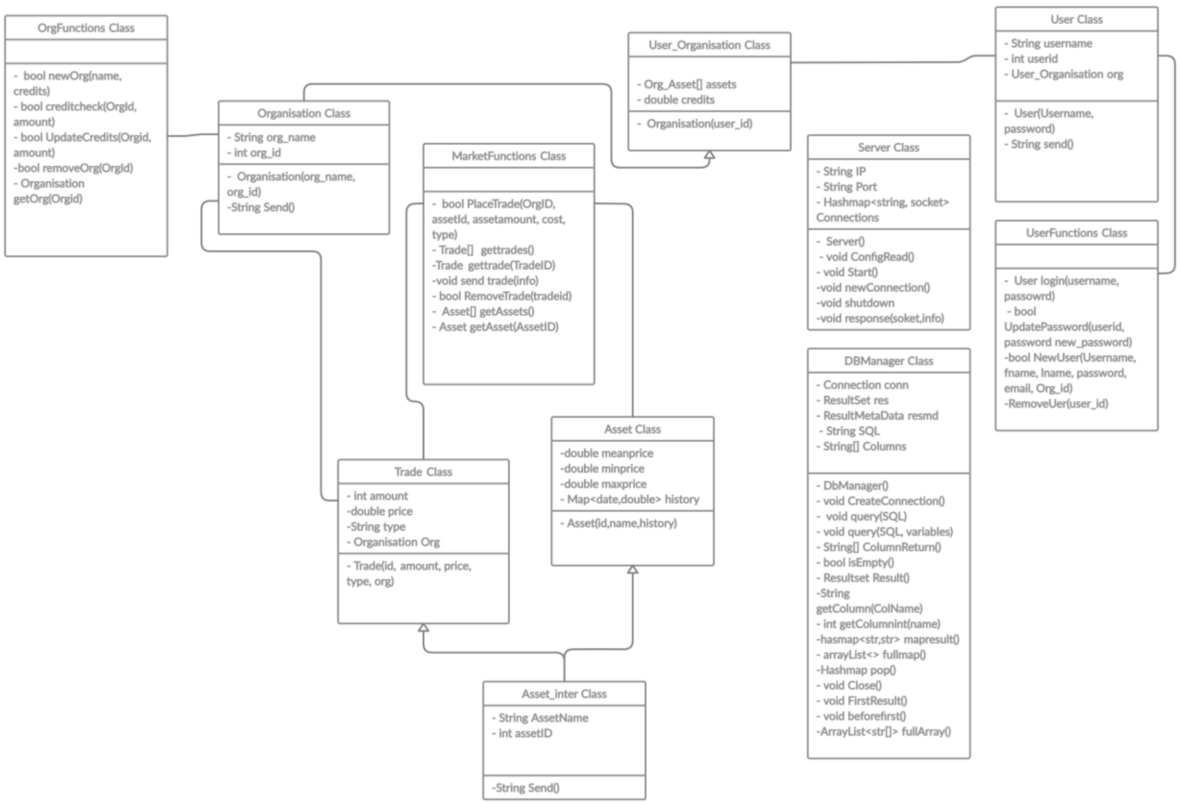


Figure 2 - Class diagram for the server-side class showing inheritance and dependent classes

Class Explanation

Common

User

The user class will be shared between both the server and client side of the application, with the variables stored in it being common across both. However, the client side of this class will not be able to access certain some functions of this class, with the login function being inaccessible on the clients machine. The server side of the user class will be used to construct a user object based on the client's username and login sent, with the class than being returned to the client.

This class is used to store the current logged in user on the client, with the username, id, full name, role and organisation they belong to all being variables present in this class. This class is used mostly for verification of the marketplace, allowing the client to store information on the current user allowing for easy and quick access. One of the user's class variables is another class, the organisation class. This is lumped with in the user class as it allows for a more compressed class structure limiting the need to create separate functions that allow the user to manipulate their organisations, credits and trade request on the marketplace.

Organization

The organisation class is an important class with it holding all the information about a given organisation. This includes variables such as the number of credits they hold, the amount of assets they have, name and id. In order to streamline the application, the amount of assets an organisation owns, a hashmap with the values being of the class asset are used.

The organisation class holds all the functions that are need to manipulate a given organisation on the server, with this class being able to send buy request and sell request to the server for validation.

Assets

The asset class is used to store the variables relating to a given asset type. With the variables of its id and name being stored in it. The asset class is split into two with the organisation_asset extending the generic asset class, with it having the extra field of amount the organisation has of that asset.

This class is mostly used as a way to concisely store all the information about an asset in a simple class, that allows for clutter to be removed from the other classes. Through hiding way the common variables that an asset has.

Trade

The trade class implements the asset class extending it and adding functionality to it. With the variables being expended upon. Adding they type of trade, cost of trade and the organisation that posted the trade to the list of variables within the organisation asset class. This class has basic functionality in being able to pass variables along to the server when a user interacts with the given trade. This means that the trade id is passed to the server, along with the amount and cost of the trade to validate if the user's requested actions is valid for a given trade.

Pack

The pack class is a simple class that allows the client to pass a function id and arguments to the server. This is done through sterilisation of the class. As this class has two variables one being an int specifying the command for the server and a String array called args which specifies the variables to pass

Sever Side

DB Manager

The DB Manager class is a class that allows for the server to connect to the SQL database. The DB Manager class stores a few variables to help make accessing the database quicker, with the current result, passed SQL, Column headings all being stored within this class. This class also provides a wide variety of functions to allow for easy access to the database data. With arrays of the result being able to be returned, while also allowing for a specific column or row to also be returned. This class also has functions stored in it that allow for error handling in the case a bad Query is passed.

Market

The market class holds all the current trades and organisations on the trading platform. With it acting as the main controller for any incoming trade. This is handled through a multitude of functions that check clients trade requests for buying and selling assets, preforming the requested action if valid. This class process the clients trade requests through a FIFO queue ensuring that the trades sent first are preformed, ensuring fairness in case of a future trade conflict.

Server

The server classes store the port and ip of the server. Allowing with a hashmap of the connected sockets on the server. The server class holds the functions that allow the server and client to communicate with it having functions that allow the server to get and send data to clients, along with it being able to pass the relative functions along to the correct function or class constructor.

Client side

Connector

The Connector class is the client's version of the server, with it opening up a socket to the server's given port and ip. This class than stores this socket in a local variable. Thus, allowing for the connector's functions to receive and send information to the server.

GUI

The GUI class holds all the information about constructing the GUI for the client. While also acting as the class where all other classes and functions interact with each other. With the user class being stored as a variable here, along with the market place, with all the current trades.

Testing

A test class was also created. The purpose of this class is to create a dedicated file to handle all the unit testing that the software requires. There is a market class that accompanies this file, the reasoning for this is due to their being two files with the name "Market" and the compiler has a hard time distinguishing between them. The market file in test is exactly the same as the market file in serve.

GUI Designs

The GUI design mock-ups have been split into 3 categories, to help represent the function and flow of the various screens.

GUI Flow between user's home screens

The user's home screens shows the GUI that will greet the user when they boot up the program. With the user being taken to the login screen on boot up. With a correct password and user name input sending them to either the generic user or admin home page depending on their role. Two other screens are also show, these two screens have been included here as they both are pop ups that help the user change personal information or view organisation information that do not directly impact the market place.

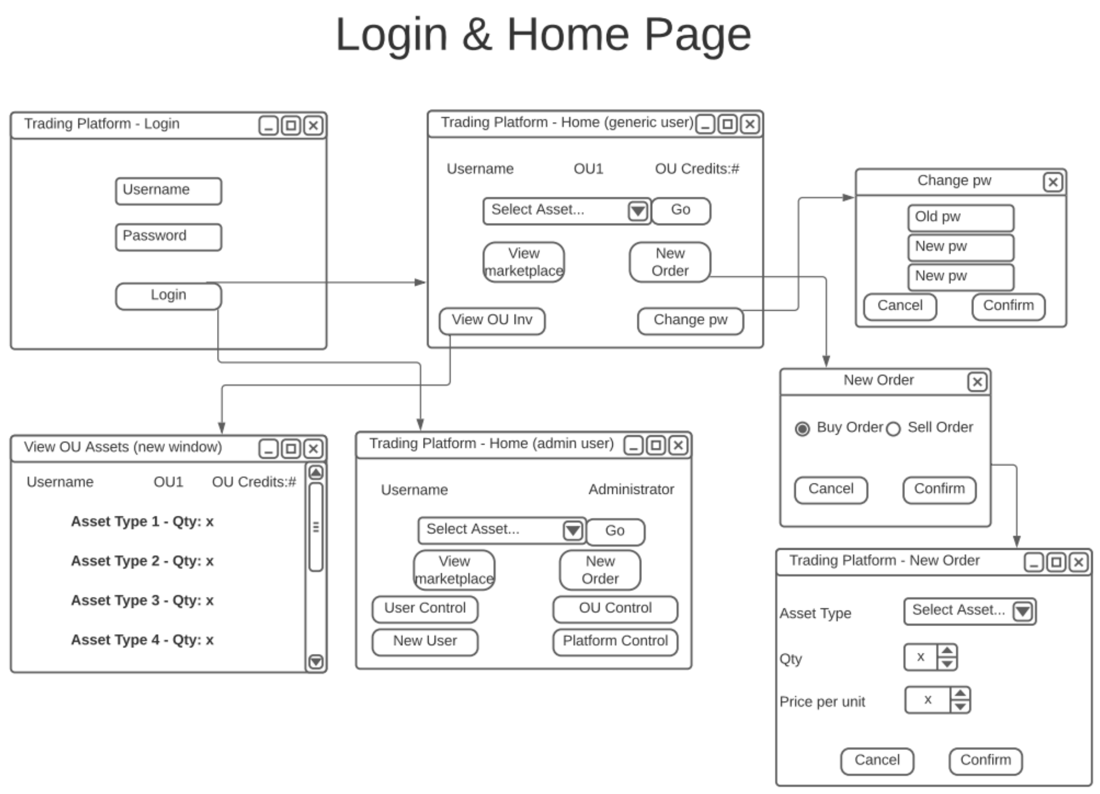
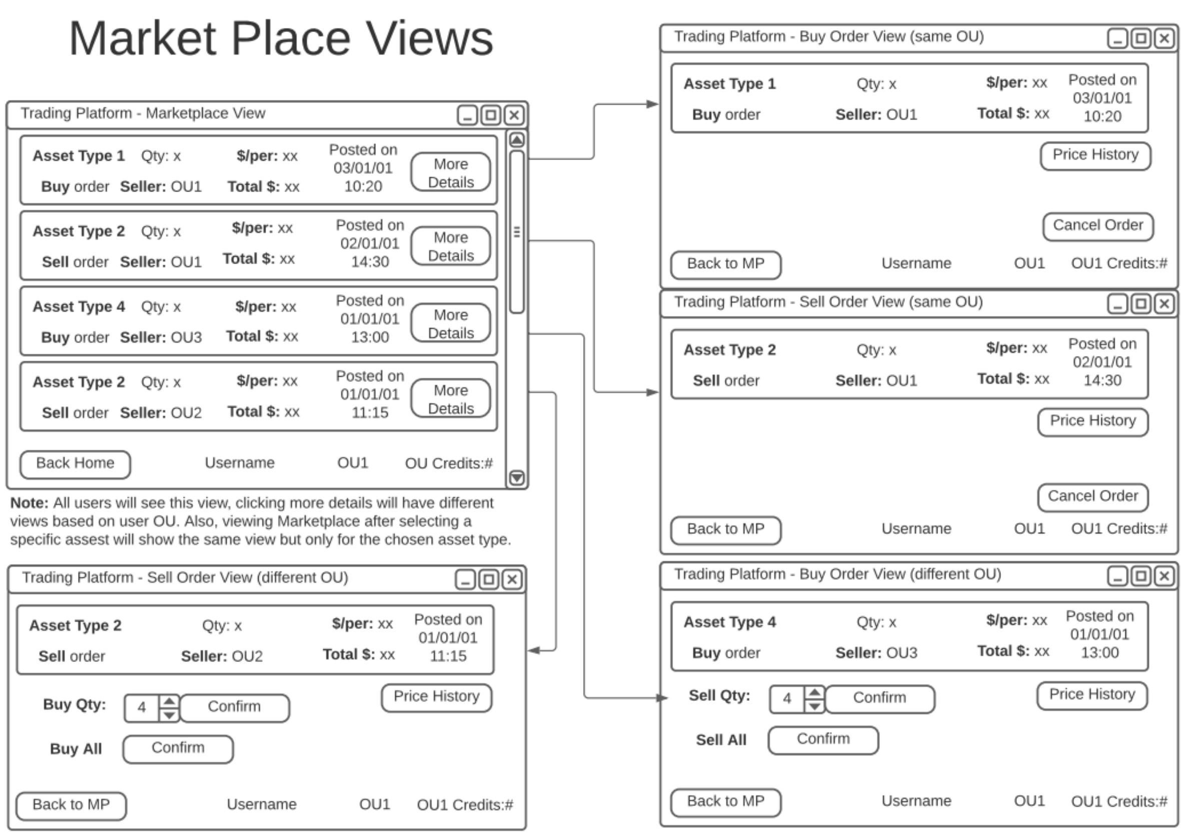


Figure 3 - trading Platform user interface GUI screens

Trading Platform GUI

The trading platform GUI screens represent the main functionality of the GUI. Allowing the user to search and view all current trades, while also allowing them to place new buy or sell request.

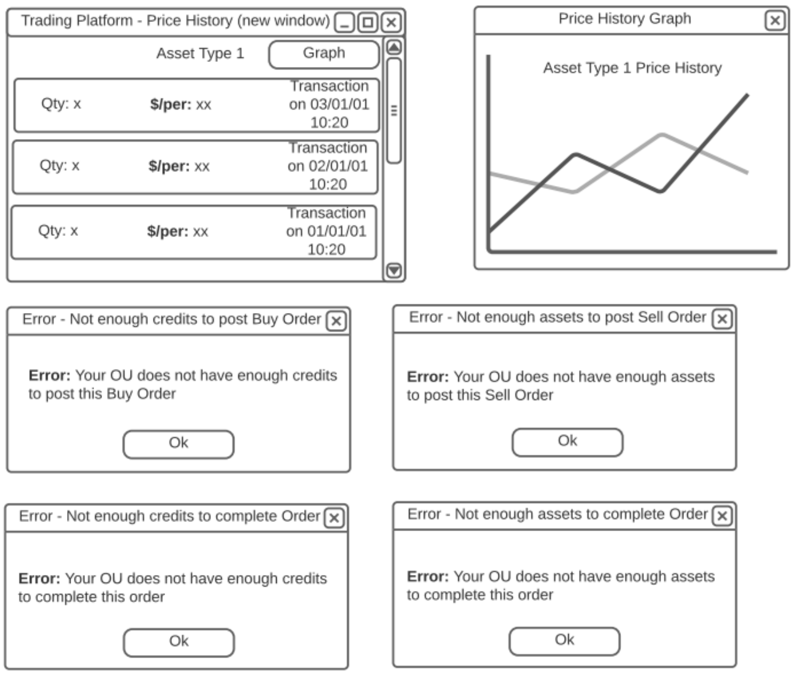


Market Place Views
Trading Platform - Marketplace View
Asset Type 1 Qty: x $/per: xx Posted on 03/01/01 10:20 More Details
Buy order Seller: OU1 Total $: xx
Asset Type 2 Qty: x $/per: xx Posted on 02/01/01 14:30 More Details
Sell order Seller: OU1 Total $: xx
Asset Type 4 Qty: x $/per: xx Posted on 01/01/01 13:00 More Details
Buy order Seller: OU3 Total $: xx
Asset Type 2 Qty: x $/per: xx Posted on 01/01/01 11:15 More Details
Sell order Seller: OU2 Total $: xx
Back Home Username OU1 OU Credits:#
Trading Platform - Buy Order View (same OU)
Asset Type 1 Qty: x $/per: xx Posted on 03/01/01 10:20
Buy order Seller: OU1 Total $: xx
Price History
Cancel Order
Back to MP Username OU1 OU1 Credits:#
Trading Platform - Sell Order View (same OU)
Asset Type 2 Qty: x $/per: xx Posted on 02/01/01 14:30
Sell order Seller: OU1 Total $: xx
Price History
Cancel Order
Back to MP Username OU1 OU1 Credits:#
Trading Platform - Sell Order View (different OU)
Asset Type 2 Qty: x $/per: xx Posted on 01/01/01 11:15
Sell order Seller: OU2 Total $: xx
Buy Qty: 4 Confirm
Buy All Confirm
Price History
Back to MP Username OU1 OU1 Credits:#
Trading Platform - Buy Order View (different OU)
Asset Type 4 Qty: x $/per: xx Posted on 01/01/01 13:00
Buy order Seller: OU3 Total $: xx
Sell Qty: 4 Confirm
Sell All Confirm
Price History
Back to MP Username OU1 OU1 Credits:#

Note: All users will see this view, clicking more details will have different views based on user OU. Also, viewing Marketplace after selecting a specific assest will show the same view but only for the chosen asset type.

Figure 4 - Trading platform marketplace GUI screens

Price History & Errors


Trading Platform - Price History (new window)
Asset Type 1
Graph
Price History Graph
Asset Type 1 Price History
Error - Not enough credits to post Buy Order
Error - Not enough assets to post Sell Order
Error - Not enough credits to complete Order
Error - Not enough assets to complete Order
Ok

Qty: x **$/per:** xx Transaction on 03/01/01 10:20

Qty: x **$/per:** xx Transaction on 02/01/01 10:20

Qty: x **$/per:** xx Transaction on 01/01/01 10:20

Error: Your OU does not have enough credits to post this Buy Order

Error: Your OU does not have enough assets to post this Sell Order

Error: Your OU does not have enough credits to complete this order

Error: Your OU does not have enough assets to complete this order

Figure 5 - miscellaneous GUI elements used to provide the user feedback

Trading Platform Administration GUI

The administrative GUI is a backend interface that is only accessible to users with an admin role. These screens allow admins to add new assets, users and organisation. While also allowing them to delete them or change the credits or assets owned by an organisation.

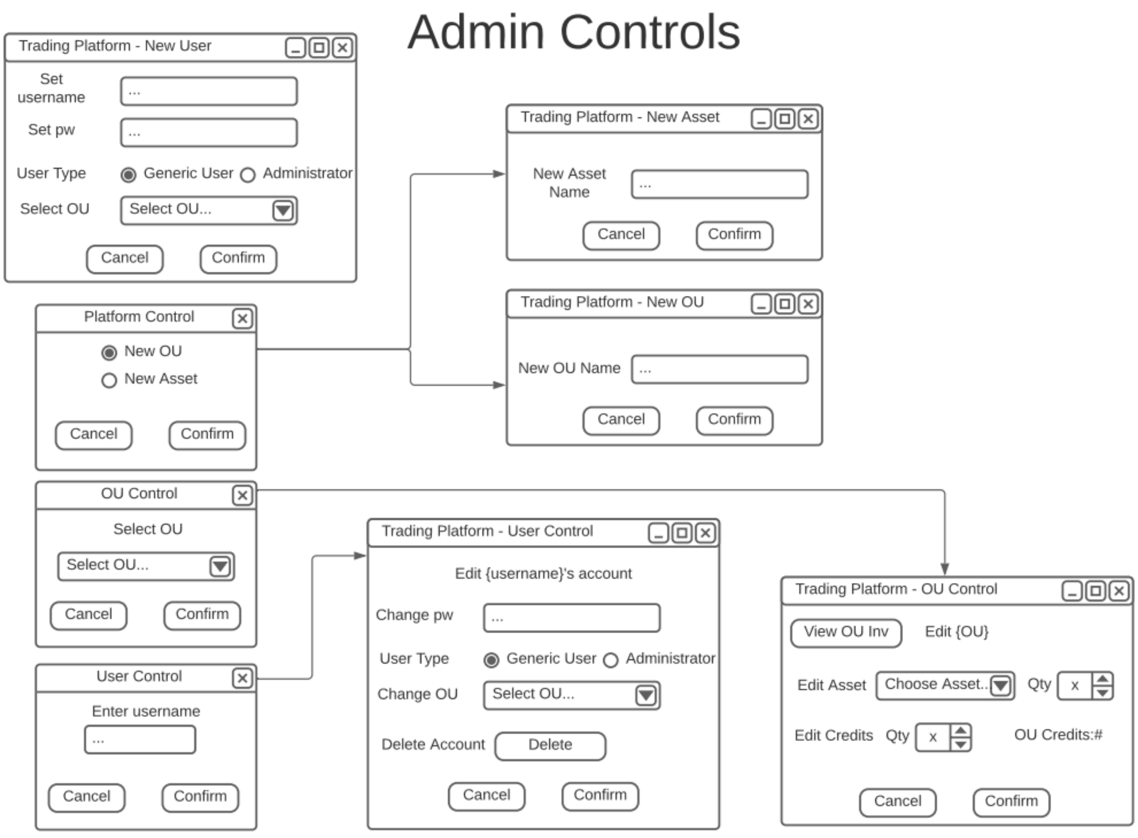


Figure 6 - Trading platform admin GUI screens

Database Schema

The database has been broken down into 7 tables, Users, Organisations, Trade, Asset, OrgainsationAssets, TradeHistoryLined, TradeHistory.

Assumptions:

- Users can only be part of a single organisation.
- An organisation can have many assets.
- An organisation can have many trades.
- A single asset type will be listed in each trade.
- The trade table will contain both sell and buy requests.
- A trade sell request only becomes history once all asset units are sold.
- A trade sell can have many buy requests.

The **Users** table contains all the information pertaining to a unique user such as names, password, and organisation. It also contains fields that can be used at a latter date if the client ever wants to expend the functionality of the trading platform such as an email address and logged in, in case email functionality or interacting with other users become a feature.

The **Organisation** table contains information pertaining to the various unique organisations, such as name and credits.

The **Trade** table contains the information relating to buy and sell request currently active on the marketplace, including the trade type, asset, amount and requested price.

The **Asset** table includes the information about each asset type, name and description.

The **OrganisationAsset** table Contains the information contains the information on what asset each organisation owns, linking the Asset_id and Organisation_id, along with adding the quantity of each asset an organisation owns.

The **TradeHistory** table, contains the information about past trades on the marketplace, including the post data, asset amount, sell price and the trade id.

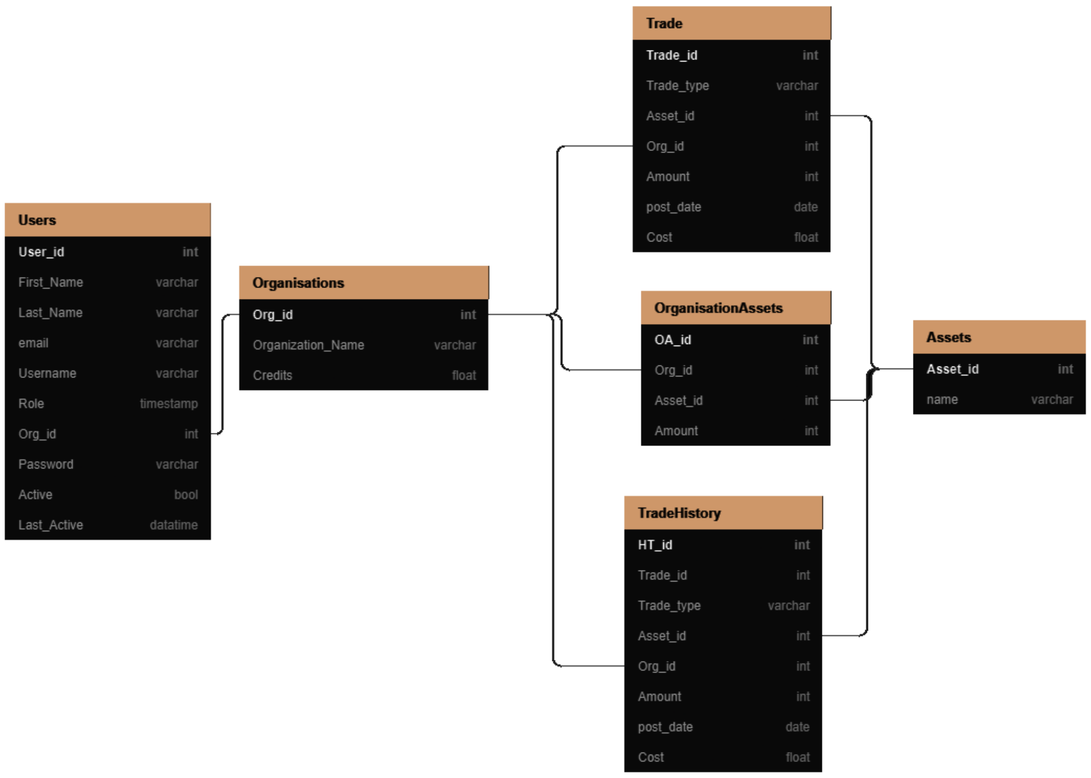


Figure 7 - Database diagram representing the various tables and the connecting keys

Network protocol

To enable communication between the server and the client a thread on the server will constantly listen for new request from clients. When a new request is received the given information from the client will then be transferred over to the correct function, running on another parallel thread.

The Server thread will split out the client's request through a serialised object called pack. This pack object will contain two variables the command id as an int and an array of string that make up the commands to pass to the server. To send information back to the client the port opened to the server will be saved into hash map on the server, with the key being the connection IP, while the value is the socket. The key will then be passed along with the inputs to the function requested, so that any data can be returned to the correct port and return the correct sterilized object or Boolean.

The codes for the server to proves the commands can be seen in table 1. With codes for all the function being split up by function type, so that the function codes look like $'ix'$ where $'i'$ is the function class and $'x'$ is the unique identifier for it within that class. For example, any function pertaining the operation of the market place as an i of 2, while the sell function as an $'x'$ of 1 given it the server code 21, while a buy request has a server code of 22.

When a client has sent a request to the server the client will wait for a response to be given, with it being returned in the same way it was sent. However, on the client side no task id will be sent back as the client already knows how to interpret the data since it has been waiting for the response from the server.

Server code	**Function**	**Sent inputs**
User Functions		
10	Login	(Username, hashed password)
11	Change password	(Username, hashed password, hashed new password)
12	View inventory	(OU_id)
Market Place Functions		
20	View marketplace	
21	Sell Request	(OU_id, asset, amount, price)
22	Buy Request	(OU_id, asset, amount, price)
25	Remove Trade	(trade_id)
Administrative Functions		
30	Add user	(user_id, hashed_password, OU_id, user type)
31	Add OU	(OU_name)
32	Add asset	(asset name, asset description)
33	Add asset to OU	(OU_id, asset_id)
34	Add credits to OU	(OU_id, amount)
35	Remove user	(user_id)
36	Remove OU	(OU_id)
37	Remove asset	(asset_id)
Helper Functions		
40	Update Password (through Admin)	(OU_id, Newpassword)

41	Update User’s Org	(OU_id, New_OU_id, User_id)
42	Update user type	(user_id, New Type)
43	Close Connection	
44	Check Trade	
45	Fetch users	

Table 1 - Task ID and arguments passed to the server from the client

Test Cases

This section contains the test cases that will be used to validate the applications integrity, insuring that all possible cases are accounted for.

Constructors
Removing a user that exists
Removing a user that doesn’t exist
Adding a new user that is unique (no duplicate username)
Adding a new user that isn't unique (duplicate username in database)
Changing password using the wrong old password
Changing a password using the correct password
Changing the password of a user that doesn’t exist
Changing the password of a user (Admin method)
Adding an organisation that is unique (no duplicate Organisation name)
Adding an organisation that isn't unique (duplicate Organisation name)
Adding an asset that is unique (no duplicate asset name)
Adding an asset that isn't unique (duplicate asset name)
Adding credit to an organisation that exists
Adding credit to an organisation that doesn’t exist
Changing the username of a user that exists
Changing the username of a user that doesn’t exist
Deleting an organisation that exists
Deleting an organisation that doesn’t exist
Deleting an asset from an organisation that exists
Deleting an asset from an organisation that they don’t own
Add asset to an organisation
Delete asset from database that exists
Delete asset from database that doesn’t exist
Remove asset from organisation that they are currently holding
Remove asset from organisation that doesn’t exist
Organisation purchasing asset when they have enough money
Organisation purchasing asset when they don’t have enough money
Organisation selling an amount of assets that they own
Organisation selling an amount of assets they don’t own (selling 100, when they only have 10)
Removing a trade that exists in the database
Change a users organisation
Change a users role
Change a users role to its already pre-existing role
Get all users and print to console

Some test cases can be assumed to never occur and therefore don’t need to be tested. For example, don’t need to test for the case of a Changing a user role of a user that doesn’t exist. This is due to the information is pulled from the GUI that is pulled from the server and users that don’t exist can't be pulled so therefore that case is impossible to occur.

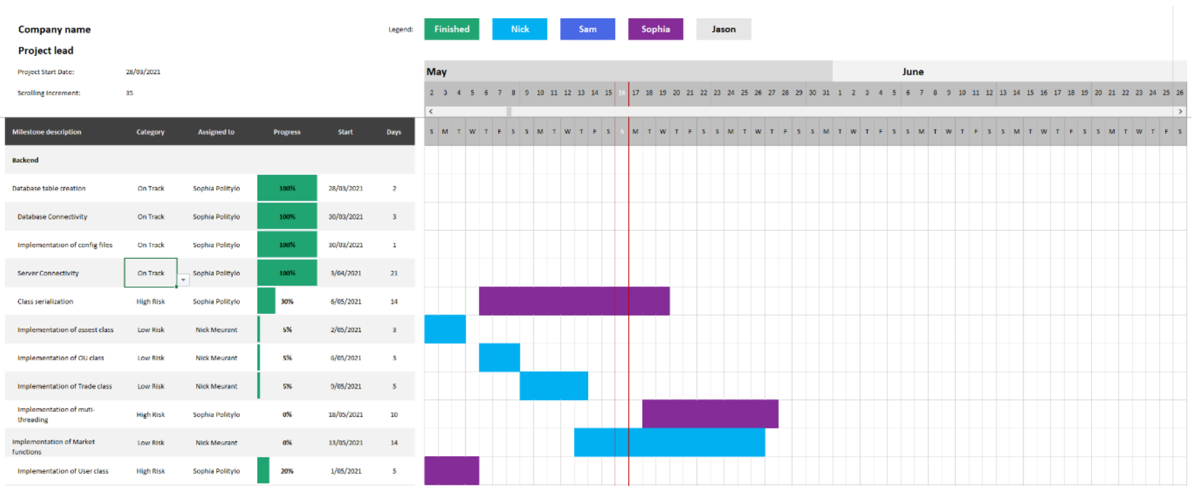
Gannt Chart

Understanding the Gannt chart

The projects Gannt chart has been broken down into 4 major headings, the backend, GUI, test cases and documentation. Within each of these headings there is a list of tasks that need to be completed for the project. Each task has these 5 fields description, category, assigned to, progress, start and days. The assigned to field has the group member tasked with completing the task, the description field is used to give a brief description on what is expected from the task. The progress field is used to show if a task is completed with 100% being done and any other percentage indicating how much of it is completed. This is handy in allowing members to coordinate with each other, when a task needs to be partly completed for another task, allowing for group members to check on each other's progress and know what can be done. The start field is used to show the expected start date of the task, while the days field is used to show the recommend amount of time for that task. These two fields are mostly guiding with small deviations from them being allowed as long as it does not impact any other group members tasks or there is a valid reason, such as a task being more complex than expected or outside factors. The category field is used to colour the Gannt chart's boxes, with each category corresponding to a different key. The purple colouring (High risk) indicates that the task has been assigned to Sophia, while a light blue (low risk) cell indicates that a task is assigned to Nick, a dark blue (med risk) is given to Sam and a grey is given to Jason, the last cell colour of green (on track) is used to indicate that a task is completed. This colour coding system was used as it easily allows the group members to see what task they have been given, while the completed task colour allows for an easy way to filter out done tasks, while keeping them there to show progression.

Gannt Chart for the date after the 16/05/21

Backend Gannt chart



Company name

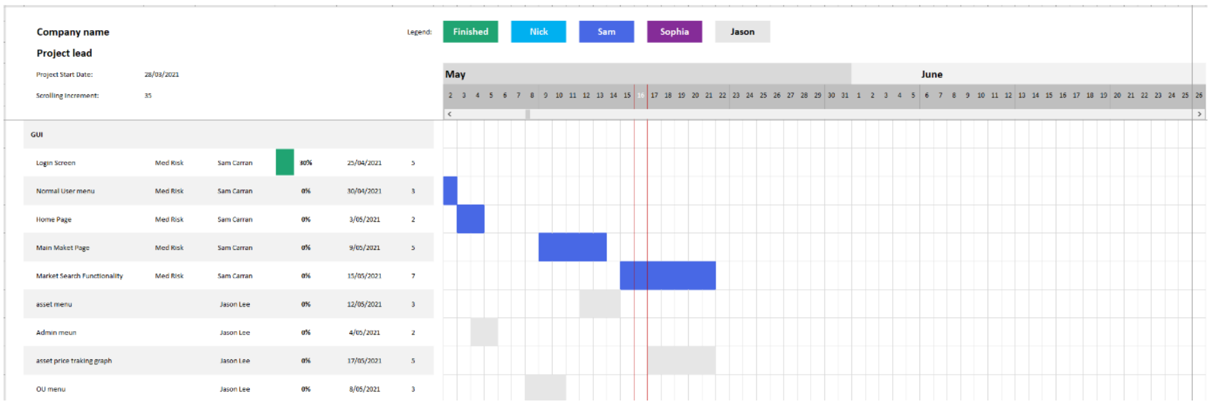
Project lead

Project Start Date: 28/03/2021

Scrolling Increment: 35

Milestone description	Category	Assigned to	Progress	Start	Days
Backend					
Database table creation	On Track	Sophia Politylo	100%	28/03/2021	2
Database Connectivity	On Track	Sophia Politylo	100%	30/03/2021	3
Implementation of config files	On Track	Sophia Politylo	100%	30/03/2021	1
Server Connectivity	On Track	Sophia Politylo	100%	3/04/2021	21
Class serialization	High Risk	Sophia Politylo	30%	6/05/2021	14
Implementation of assest class	Low Risk	Nick Meurant	5%	2/05/2021	3
Implementation of OU class	Low Risk	Nick Meurant	5%	6/05/2021	3
Implementation of Trade class	Low Risk	Nick Meurant	5%	9/05/2021	5
Implementation of muti-threading	High Risk	Sophia Politylo	0%	18/05/2021	10
Implementation of Market functions	Low Risk	Nick Meurant	0%	13/05/2021	14
Implementation of User class	High Risk	Sophia Politylo	20%	1/05/2021	5

GUI Gannt Chart



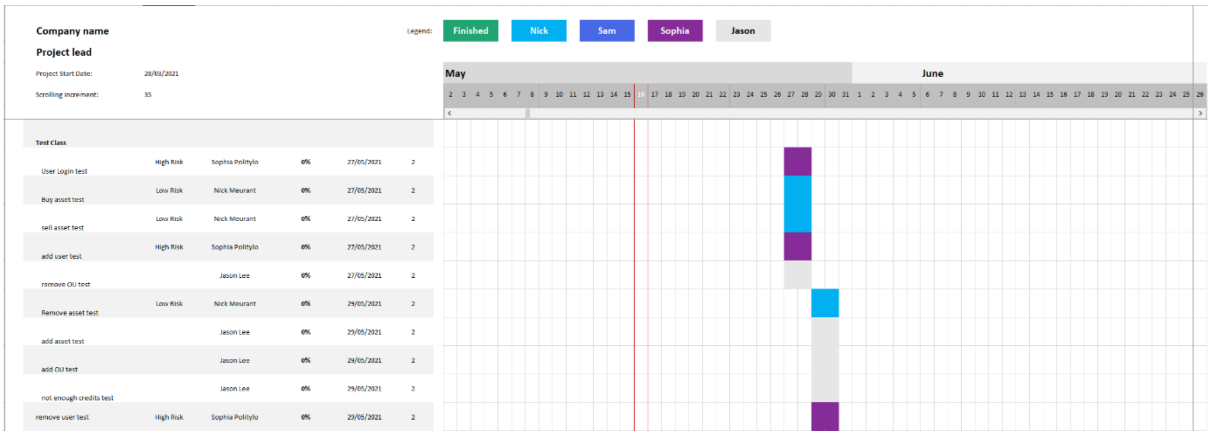
Company name

Project lead

Project Start Date:	28/03/2021
Scrolling Increment:	35

GUI						
Login Screen	Med Risk	Sam Carran	🟩	30%	25/04/2021	5
Normal User menu	Med Risk	Sam Carran		0%	30/04/2021	3
Home Page	Med Risk	Sam Carran		0%	3/05/2021	2
Main Maket Page	Med Risk	Sam Carran		0%	9/05/2021	5
Market Search Functionality	Med Risk	Sam Carran		0%	15/05/2021	7
asset menu		Jason Lee		0%	12/05/2021	3
Admin meun		Jason Lee		0%	4/05/2021	2
asset price traking graph		Jason Lee		0%	17/05/2021	5
OU menu		Jason Lee		0%	8/05/2021	3

Testing cases Gannt Chart



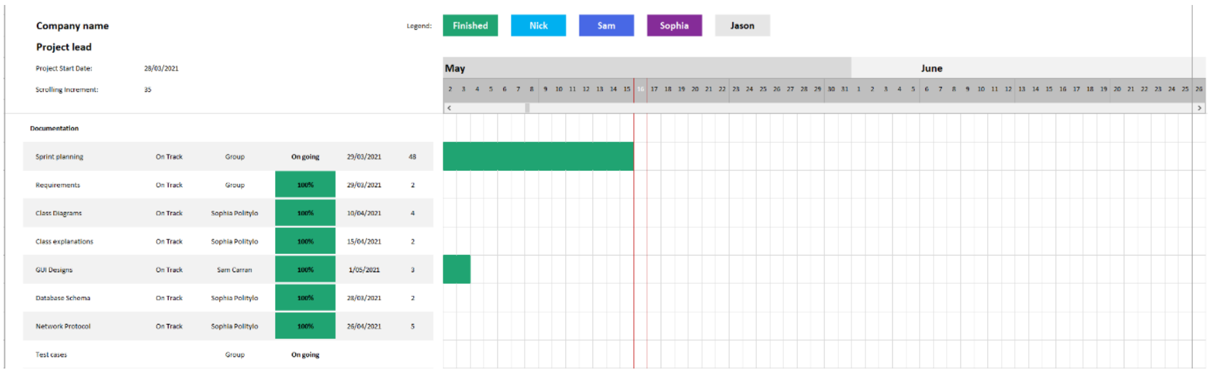
Company name

Project lead

Project Start Date:	28/03/2021
Scrolling Increment:	35

Test Class					
User Login test	High Risk	Sophia Politylo	0%	27/05/2021	2
Buy asset test	Low Risk	Nick Meurant	0%	27/05/2021	2
sell asset test	Low Risk	Nick Meurant	0%	27/05/2021	2
add user test	High Risk	Sophia Politylo	0%	27/05/2021	2
remove OU test		Jason Lee	0%	27/05/2021	2
Remove asset test	Low Risk	Nick Meurant	0%	29/05/2021	2
add asset test		Jason Lee	0%	29/05/2021	2
add OU test		Jason Lee	0%	29/05/2021	2
not enough credits test		Jason Lee	0%	29/05/2021	2
remove user test	High Risk	Sophia Politylo	0%	29/05/2021	2

Documentation Gannt Chart



Company name

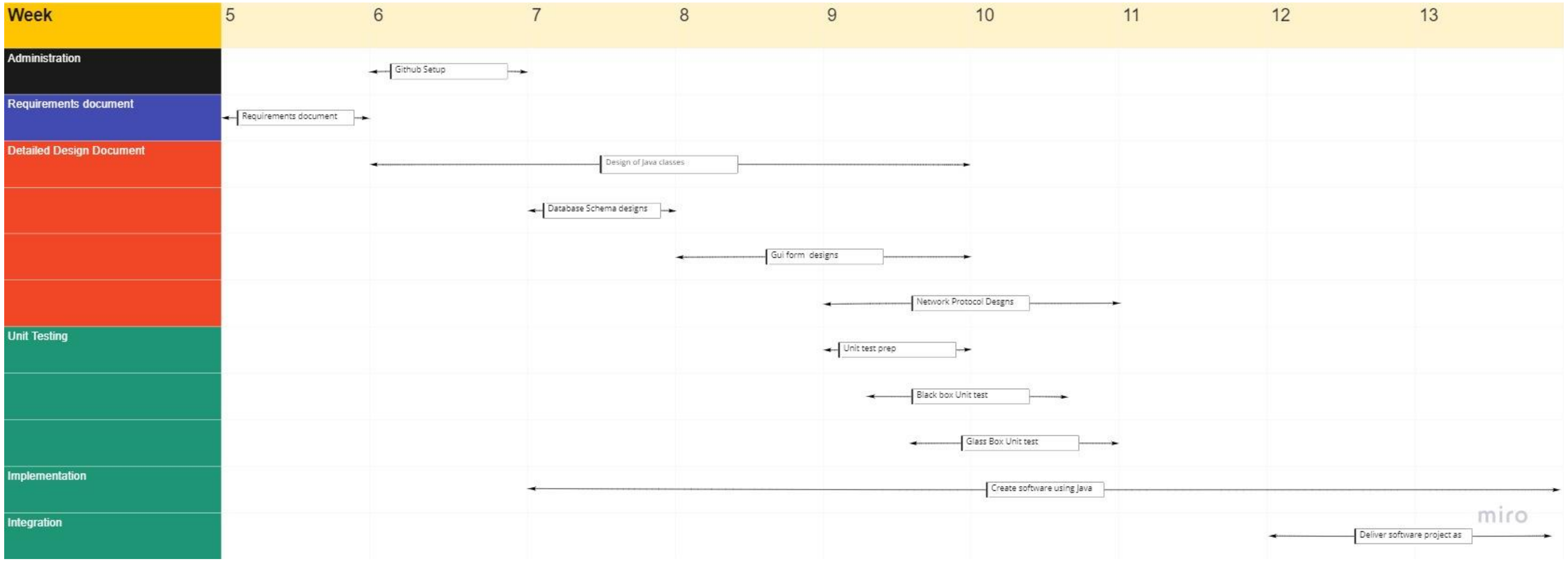
Project lead

Project Start Date:	28/03/2021
Scrolling Increment:	35

Documentation					
Sprint planning	On Track	Group	On going	29/03/2021	48
Requirements	On Track	Group	100%	29/03/2021	2
Class Diagrams	On Track	Sophia Politylo	100%	10/04/2021	4
Class explanations	On Track	Sophia Politylo	100%	15/04/2021	2
GUI Designs	On Track	Sam Carran	100%	1/05/2021	3
Database Schema	On Track	Sophia Politylo	100%	28/03/2021	2
Network Protocol	On Track	Sophia Politylo	100%	26/04/2021	5
Test cases		Group	On going		

Appendix

Sprint Planning for 3/05 – 16/05


Week
Administration
Github Setup
Requirements document
Requirements document
Detailed Design Document
Design of Java classes
Database Schema designs
Gui form designs
Network Protocol Desgns
Unit Testing
Unit test prep
Black box Unit test
Glass Box Unit test
Implementation
Create software using Java
Integration
Deliver software project as
miro