By Group of 4, #20

AKA “G420 Developers”

Robyn Bullemor N7133138

James Crawford N10200631

Nathan Donald N10472282

David Bulyaki N5882184

2020-05-02

CAB302

Detailed Design Document

Online Trading Platform

Contents

[0 - Overview 1](#_Toc70789122)

[1 - Javadoc 2](#_Toc70789123)

[2 - Backend Classes (Brief Overview) 3](#_Toc70789124)

[2.1 - Asset 3](#_Toc70789125)

[2.2 - Department 3](#_Toc70789126)

[2.3 - ITAdminUser 3](#_Toc70789127)

[2.4 - Online Trade 3](#_Toc70789128)

[2.5 - User 3](#_Toc70789129)

[3 - Graphical User Interface 4](#_Toc70789130)

[3.1 - GUI Design 4](#_Toc70789131)

[3.2 - GUI Classes / Package Overview 5](#_Toc70789132)

[3.2.1 - MenuBar 5](#_Toc70789133)

[3.2.2 - Reference 5](#_Toc70789134)

[3.2.3 - Tools 5](#_Toc70789135)

[3.2.4 - Views 6](#_Toc70789136)

[3.3 - General Functionality 6](#_Toc70789137)

[4 - Database Schema 7](#_Toc70789138)

[4.1 – Component design description 7](#_Toc70789139)

[4.2 – Component interfaces 7](#_Toc70789140)

[4.3 – Permissions 8](#_Toc70789141)

[4.4 – Workflows and algorithms 8](#_Toc70789142)

[4.5 Triggers 8](#_Toc70789143)

[4.6 – Software requirements mapping 8](#_Toc70789144)

[5 - Networking Protocols 9](#_Toc70789145)

[5.1 - Networking Design 9](#_Toc70789146)

[5.1.1 – Server Classes 9](#_Toc70789147)

[6 - Appendix 10](#_Toc70789148)

[Appendix 1: Database Object Role Model 10](#_Toc70789149)

[Appendix 2: Database Script 11](#_Toc70789150)

[7 - Bibliography 20](#_Toc70789151)

# 0 - Overview

Legitimate Corporation Pty Ltd approached G420 Developers to design an online trading platform. The objective was to facilitate an efficient exchange of internal resources, known as commodities (or an 2.1 - Asset). The proposition was that each company department was allocated a certain budget of universally-recognised credits, which they could then use to barter for commodities with anyone who was active on the online trade platform.

The idea is that users of this system do not tediously micromanage their transactions. Instead, they create buy or sell orders, where they nominate the type of commodity they wish to buy or sell, specify their desired unit price (to buy or sell), and finally the quantity thereof. Once a Buy or Sell order has been created, autonomous processes cross examine the compatibility of the existing Buy & Sell orders.

Transactions whose Buy & Sell terms are evenly matched will process automatically without the need for a user to intervene. To help avoid human users needing to micromanage Buy & Sell orders, trades will also occur automatically when a buy order is willing to purchase commodities for a greater unit price, than what existing Sell orders are listing. In this case, the program will protect the interest of the buyer by only paying for the nominated Sell price (the lower value), and not the original, higher Buy price. The whole concept for this online trade platform is to help streamline interdepartmental resource-sharing. It then makes sense to create a program that has the convenience of inbuilt, automated functionality, to free employees up to do their regular, daily duties.

Equally important is the user interface. The client has expressly conveyed that they refuse to accept a terminal-based application, and would much rather interact with a GUI. Special considerations for the GUI design include for its’ need to be intuitive, pleasant to use and appealing to the user (as well as maintaining the scope of the features requested). There has been some inspiration drawn from existing online trading platforms.

A database schema has been prepared to help manage the relationships between the data objects. Laying a sound foundation for the database will significantly aid in the effectiveness of the online trading platform to retrieve and store relevant data. The online trading platform must be flexible enough to not limit the number of commodity types (2.1 - Asset), users (2.5 - User) and admin members (2.3 - ITAdminUser). The online trading platform also incorporates network connectivity. It can host multiple users, ensuring that their login credentials are hash encrypted to protect their security.

The following sections provide a more detailed overview of the application’s programming logic and structure.

# 1 - Javadoc

G420 Developers has taken the time to describe the online trade platform’s methods and classes within an exported Javadoc. “overview-summary.html” (Figure 1) Is a good starting point for examining the Javadoc content, as it lists the main packages that contain the application’s code content.

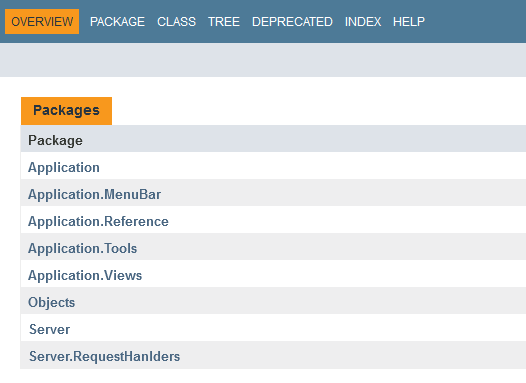


Figure : Javadoc "overview-summary.html".

# 2 - Backend Classes (Brief Overview)

Whilst the Javadoc does contain adequate descriptions for the backend classes (Figure 2) relating to the application’s use of data objects, some overview for the backend functionality is provided here in section 2 for reader convenience.

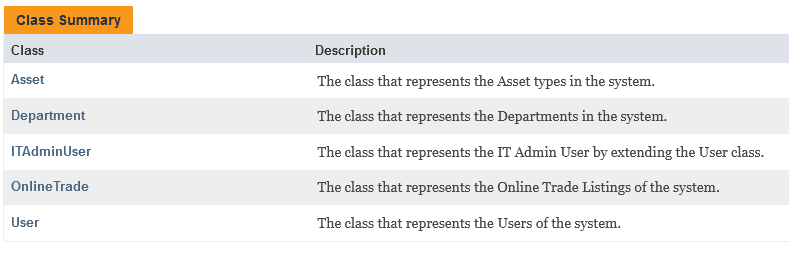


Figure : Backend Classes.

## 2.1 - Asset

The class used for representing the “Asset” or the types of corporate resources or commodities to trade for credits.

## 2.2 - Department

The class used for representing the corporate department.  
Also contains the number of credits currently available for the *Department* to spend.

## 2.3 - ITAdminUser

The class used for representing the users who are enabled with special privileges to manage the existing types of *Asset*s, as well as the delegation of credits to corporations, and managing users. An *ITAdminUser* can also do everything a normal *User* can.

## 2.4 - Online Trade

The *OnlineTrade* objects are the primary means of representing the necessary data relating to the Buy and Sell orders. The database relationships are used to associate the relatively simple *Department* objects with the more, comprehensive *OnlineTrade* objects. The *OnlineTrade* objects are designed to be flexible, where appropriately setting their *tradeType* will determine whether they’re representing a Buy or Sell order.

## 2.5 - User

The class used for representing the users who are enabled with regular privileges to create Buy or Sell orders. A User cannot add more credits to a Department, nor manage commodities (*Asset*) or other *User*s.

# 3 - Graphical User Interface

## 3.1 - GUI Design

The GUI draws inspiration from an existing and popular online trading platform, the Steam Community Market (Figure 3). This example is relatively simple, but provides some good insight regarding the appeal of intuitive icons representing the resources being traded, along with other stats such as quantity and unit price.

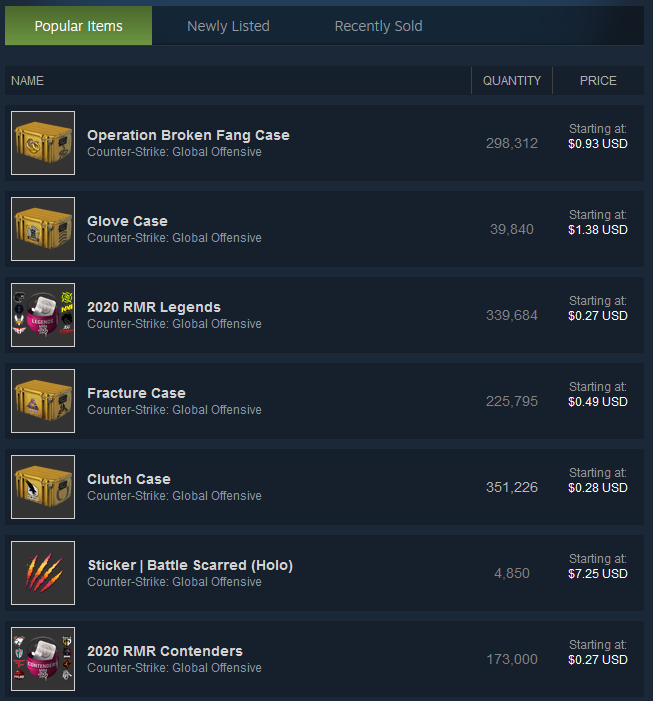


Figure : Steam Community Market

An initial GUI concept depicted how a similar concept can be implemented with the notion of trading corporate resource commodities for a given unit price in credits. A central Box component is loaded with the content of all the active OnlineTrade objects being referenced from the database. The idea here is that the database contents can be converted into some sort of *Java ArrayList<>*, and then be displayed in a user-intuitive format.

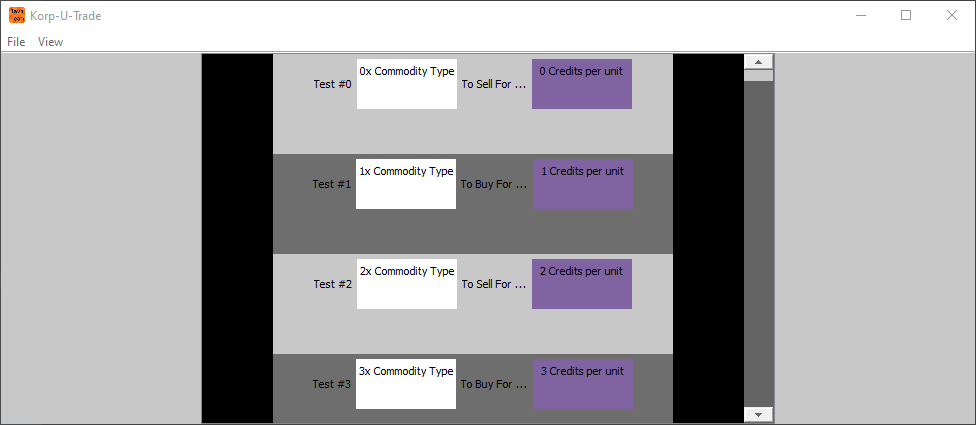


Figure : GUI Initial Prototype showing active OnlineTrades

## 3.2 - GUI Classes / Package Overview

GUI design can become very complex very quickly, and easily runs the risk of losing maintainability as Java components are incorporated into an ever-increasing feature set. It was therefore important to determine a sound foundation as to the methodology for organising the GUI classes in a way that was intuitive, and maintainable from the start. This way the code will tend to progressively accumulate in content without accumulating technical debt [3].

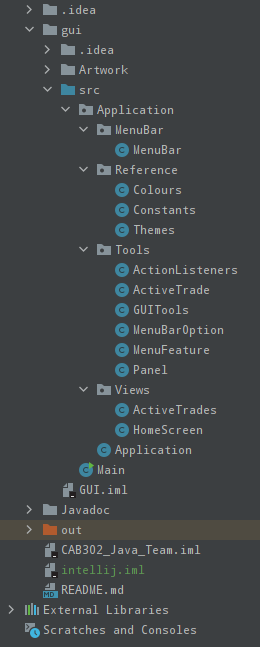
Figure 5 demonstrates the compartmentalisation of each GUI class. The overarching package Application incorporates a number of sub-packages *MenuBar*, *Reference*, *Tools* and *Views*.

Figure : GUI Classes

### 3.2.1 - MenuBar

As per its’ name, *MenuBar* contains the necessary logic to arrange the Java Components that constitute the Menu bar that appears at the top of the GUI. This specifically relates to the menu buttons such as “File”, “View” and “Help”, along with all of their options (See Javadoc).

### 3.2.2 - Reference

Contains a number of classes, each of which primarily contain constants IE Java variables initialised with the *final* keyword. They contain the programmer-defined variables that determine the desired GUI characteristics (See Javadoc).

### 3.2.3 - Tools

Is a very foundational package that serves as the backend workhorse for the GUI. It houses the object classes and methods necessary for the intuitive and maintainable composition of the GUI code, particularly to do with features and parsing of data (See Javadoc).

### 3.2.4 - Views

Contains classes which mainly combine the functionality of the existing classes in order to produce the visual representation of a particular user view. Classes within the *Views* package do not bring any novel functionality into the program, in terms of new object types or methods, but rather they possess substantial methods which in turn call a combination of smaller methods to create the new View for the GUI to display (See Javadoc).

## 3.3 - General Functionality

Initial concepts depict the home screen (Figure 6) having large user icons to conveniently access the features they want without wasting time. Again, this program is being developed for workers in a corporate environment who might be busy, and engaged in multi-tasking duties. Having large, coloured buttons to navigate to different parts of the program may help make mental associations to features, and should help enable users to become more efficient over time.

Whilst this is only a crude concept, the button captions do reveal the core set of features that G420 Developers aim to deliver for the client. These include a provision to create new Buy or Sell orders, manage existing Buy or Sell orders, create new commodity types, view active orders (*OnlineTrade* objects) that exist the market (server),

Figure : Home Screen Concept

inspect commodity trends (to determine competitive unit prices for Buy or Sell orders), and for *ITAdminUser*s only, managing *User*s, Credits and Commodities (*Asset*s).

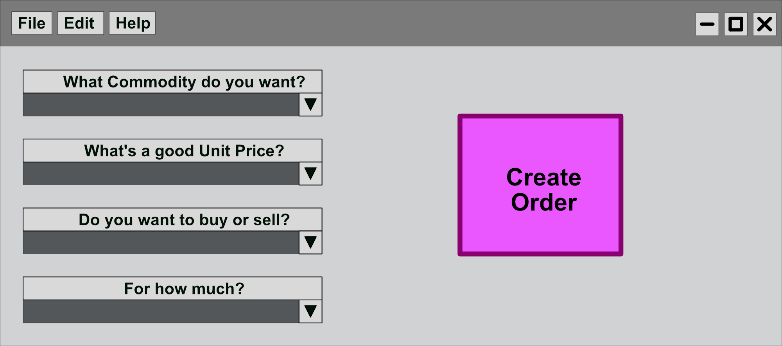
Figure 7 depicts an insight as to what one of these features “Create new Commodity” might look like. Here we see 4 x drop-downs that intuitively prompt the user regarding the specifics of their Buy or Sell order. Although this depiction is rather crude, and it’s highly likely a better   
solution will be developed for the final version of this feature.

Figure : Create new Order

# 4 - Database Schema

## 4.1 – Component design description

Data is split into 7 tables: *Orders*, *Assets*, *Inventories*, *Trades*, *Users*, *OrganisationalUnits* and *Notifications*.

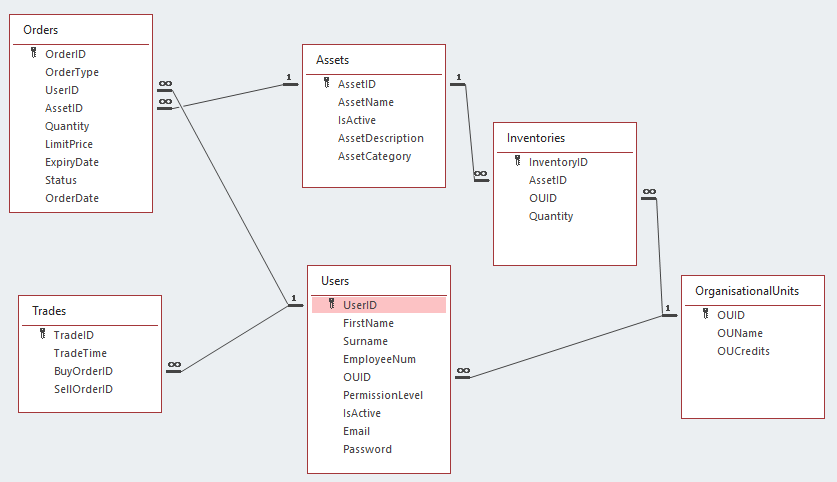


Figure : Database Relationships

Further details to be added once plan is more certain.

See Appendix 1 (Figure 10) for a full Object Role Model.  
See Appendix 2: Database Script for full database script.

## 4.2 – Component interfaces

Information can be entered into the database either by the user, via the GUI or programmatically, as triggered by certain events.

An initial database has been populated based on information provided by Legitimate Corporation Pty Ltd including *Asset*s, *User*s, *OrganisationalUnits* and *Inventories*. To maintain system stability, the *Trades* table is only modifiable via the built-in algorithms or manually by CAB302 Java Team.

## 4.3 – Permissions

Excluding primary keys, *Assets*, *Users*, *OrganisationalUnits* and *Inventories* tables can be created and modified manually by Database Administrators (2.3 - ITAdminUser). Records can be made inactive but cannot deleted to maintain system integrity.

*User*s (2.5 - User) will be able to edit their own password and will have read only access to their user information as well as active trade, order and asset information, inventory and details of their own Organisational Unit. Users can also edit and cancel orders which they have placed up until the time that the order is approved.

Organisational Unit Managers will have User level permissions, as well as the ability to modify their own OU Inventories, approve or cancel pending orders from within their OU.

## 4.4 – Workflows and algorithms

To be confirmed once coding begins

## 4.5 Triggers

When a new record is added to the Orders table a notification is sent to the originating OUs Manager for the order to be approve. Upon approval/non-approval, a notification will be sent to the originating User, advising them of the order status.

If an order has been approved, a search is performed on active orders to find a matching buy/sell order. If a matching order is found, a check is performed to ensure that the buying OU still has enough credits to complete the trade. Another check is performed to ensure that the selling OU has a sufficient Inventory. If either of these conditions cannot be met, the failing Order is skipped and the search will continue until a suitable Order is found. If there are no suitable Orders, no further action is taken.

Once a suitable match is found, a new record is added to the Trades table and both Order records are marked as complete. A notification is sent to both order users to advise that the trade has been completed. The selling OUs Inventory record Quantity is decreased by the required amount while the buying is increased. Similarly, the selling OUs *OUCredits* is increased by the required amount while the buying OUs is decreased.

At 12:01am, a search will be run on the Orders table to check the *ExpiryDate* of each active record. If the *ExpiryDate* has passed, the record will be marked as inactive. If the *ExpiryDate* is within the next two days, a notification is sent to the Order User advising them that their order is due to expire.

TBC - Partial fill orders ….. to be confirmed.  
TBC - How will we handle linked records being made inactive? E.g. If an OU is made inactive, is a user still able to trade?

## 4.6 – Software requirements mapping

MariaDB has been used for the database server as it provides “performance and stability, especially in high load environments” 1 in an open-source platform [1].

# 5 - Networking Protocols

## 5.1 - Networking Design

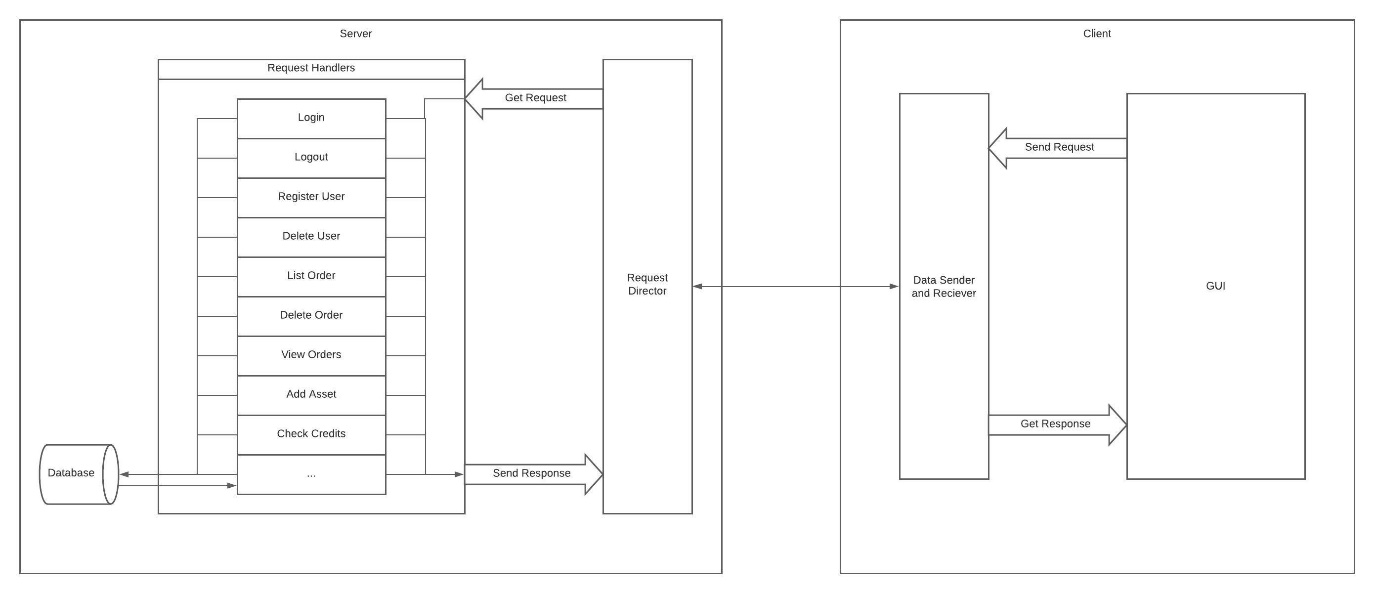


Figure : Network Design Diagram

On start up the client’s program will read a simple configuration file which will give the servers IP and Port (the project will use a single server). Once connected, the user will be able to interact with the GUI to view, edit, and delete a multitude of data stored on the server's database. When the program requires data from the server, an appropriate request will be sent to the server. The server will use a request director to send the request to the appropriate request handler. The request handler will decide if the request is eligible to be fulfilled, if so then the server will complete the request and send an appropriate response. This flow is visible in the network diagram (Figure 9)

### 5.1.1 – Server Classes

A detailed description of each class and method can also be seen in the Javadoc.

**Main Server Classes:**

Server

The Server class contains the methods that initially launch the server and open a port. Once a port has been opened, each client that connects via a socket will be handled in a ‘ClientHandler’ thread. The ClientHandler method manages the sending and receiving of String ArrayLists. When a request is sent by a client, the ClientHandler receives it and sends it to a new instance of ‘RequestDirector’. When an ArrayList is returned, it sends it over the socket back to the client that made the request.

Client

Th Client class is the class that should initially be run by any user. When running this class the main will initiate the creation of the GUI. From there the GUI will be able to make requests to the server through the ‘sendData’ method in Client. This method will take the data and send it through the object output stream created by the ‘start’ method. Once data has been sent, the ‘getList’ method can be called to listen for the server’s return. This method will then return the ArrayList as soon as it has a response from the server. If the server doesn’t respond in 9 seconds, the connection will time out.

RequestDirector

The ‘request’ method within this class takes the String in position 0 of any request sent and directs it to the method that can handle that request. If no request matches it will return that the request was invalid. However, if it is valid, once it forwards the request to the appropriate request handler, it then returns the ArrayList that the request handler returns. This is then sent back to the Client through the ClientHandler.

GetPropertyValues

This class contains multiple methods which are able to return the following values from the config.properties file: Host, Server Port, Database Port, Database Name, Database User, and Database Password. This will be accessed by both the server and the client to retrieve these values. This allows the server and database to be easily moved with each client only requiring an updated config.properties file.

Encryption

This class is used for simple encryptions that will only need to be performed server side. This allows the users permissions and token to be encrypted and returned to the user on login, dictating what content is available to them. This encryption is simply to make the login slightly more secure.

**Request Handlers:**

AddOrder

The ‘addOrder’ method in this class takes an ArrayList of strings containing all the data needed to create an *OnlineTrade* object. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Trade ID, Trade Type, User ID, Asset ID, Department ID, Quantity, Uni Price, Order Status, Expiry Date and Time]. The OnlineTrade will then be added to the database. Will then return success or failure.

CreateAsset

The ‘createAsset’ method in this class takes an ArrayList of strings containing all the data needed to create an *Asset* object. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Asset ID, Asset Name, Asset Description, A Boolean value of whether the Asset is active]. The Asset will then be added to the database. Will then return success or failure.

CreateOrganisation

The ‘createOrganisation’ method in this class takes an ArrayList of strings containing all the data needed to create a *Department* object. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Asset ID, Asset Name, Asset Description, A Boolean value of whether the Asset is active]. The Department will then be added to the database. Will then return success or failure.

DeleteAsset

The ‘deleteAsset’ method in this class takes an ArrayList of strings containing the ID of the *Asset* to be deleted. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Asset ID]. The Asset will then be deleted from the database. Will then return success or failure.

DeleteOrder

The ‘deleteOrder’ method in this class takes an ArrayList of strings containing the ID of the *OnlineTrade* to be deleted. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Trade ID]. The Trade will then be deleted from the database. Will then return success or failure.

DeleteOrganisation

The ‘deleteOrganisation’ method in this class takes an ArrayList of strings containing the ID of the *Department* to be deleted. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Department ID]. The Department will then be deleted from the database. Will then return success or failure.

DeleteUser

The ‘deleteUser’ method in this class takes an ArrayList of strings containing the ID of the *User* to be deleted. The data in the ArrayList is ordered as follows: [ Request type, Permissions, User ID]. The User will then be deleted from the database. Will then return success or failure.

GetAssets

The ‘getAssets’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve a Tree Set of all the assets names in the database. From there each asset name will be added to an ArrayList to be returned to the Client.

GetAssetSize

The ‘getAssetSize’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve the number of *Assets* in the database and add this number to an ArrayList to be returned to the Client.

GetDepartments

The ‘getDepartments’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve a Tree Set of all the Department IDs in the database. From there each department ID will be added to an ArrayList to be returned to the Client.

GetDepartmentSize

The ‘getDepartmentSize’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve the number of *Departments* in the database and add this number to an ArrayList to be returned to the Client.

GetSingleAsset

The ‘getSingleAsset’ method in this class takes an ArrayList of strings containing the name of the *Asset* to be returned. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Asset Name]. The Asset will then be retrieved from the database. The asset will then be split and added to an ArrayList to be returned to the Client.

GetSingleAssetID

The ‘getSingleAssetID’ method in this class takes an ArrayList of strings containing the ID of the *Asset* to be returned. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Asset ID]. The Asset will then be retrieved from the database. The asset will then be split and added to an ArrayList to be returned to the Client.

GetSingleOrder

The ‘getSingleOrder’ method in this class takes an ArrayList of strings containing the ID of the *OnlineTrade* to be returned. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Trade ID]. The OnlineTrade will then be retrieved from the database. The OnlineTrade will then be split and added to an ArrayList to be returned to the Client.

GetSingleOrganisation

The ‘getSingleOrganisation’ method in this class takes an ArrayList of strings containing the ID of the *Department* to be returned. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Department ID]. The Department will then be retrieved from the database. The Department will then be split and added to an ArrayList to be returned to the Client.

GetSingleUser

The ‘getSingleUser’ method in this class takes an ArrayList of strings containing the ID of the *User* to be returned. The data in the ArrayList is ordered as follows: [ Request type, Permissions, User ID]. The User will then be retrieved from the database. The User will then be split and added to an ArrayList to be returned to the Client.

GetTrades

The ‘getTrades’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve a Tree Set of all the OnlineTrade names in the database. From there each OnlineTrade name will be added to an ArrayList to be returned to the Client.

GetTradeSize

The ‘getTradeSize’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve the number of *OnlineTrades* in the database and add this number to an ArrayList to be returned to the Client.

GetUsers

The ‘getUsers’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve a Tree Set of all the User names in the database. From there each User names will be added to an ArrayList to be returned to the Client.

GetUserSize

The ‘getUserSize’ method in this class takes an ArrayList of strings containing just the request type at position 0. The method will then retrieve the number of *Users* in the database and add this number to an ArrayList to be returned to the Client.

Login

The ‘login’ method in this class takes an ArrayList of strings containing the login details entered by the user in the GUI. The data in the ArrayList is ordered as follows: [ Request type, Login request type, User ID, Hashed password]. The stored hashed password from the database for the user with the sent ID will then be compared with the hashed password sent. If they are equal the method will return success, if they are not, it will return failure.

ModifyAsset

The ‘modifyAsset’ method in this class takes an ArrayList of strings containing the details of the *Asset* to be modified. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Asset name, Asset Description, Asset Category, A Boolean value of whether the Asset is active]. This data will then be used to create an *Asset* object. The matching *Asset* in the database will then be updated to hold the new details. The method will return success or failure.

ModifyTrade

The ‘modifyTrade’ method in this class takes an ArrayList of strings containing the details of the *OnlineTrade* to be modified. The data in the ArrayList is ordered as follows: [ Request type, Permissions, Trade ID, Trade Type, User ID, Asset ID, Department ID, Quantity, Uni Price, Order Status, Expiry Date and Time]. This data will then be used to create an *OnlineTrade* object. The matching *OnlineTrade* in the database will then be updated to hold the new details. The method will return success or failure.

ModifyUser

The ‘modifyUser’ method in this class takes an ArrayList of strings containing the details of the *User* to be modified. The data in the ArrayList is ordered as follows: [ Request type, Permissions, User ID, First name, Surname, Email, Department ID, Permission level, Hashed Password, Salt, Boolean value whether the user is active]. This data will then be used to create an *User* object. The matching *User* in the database will then be updated to hold the new details. The method will return success or failure.

RegisterUser

The ‘RegisterUser’ method in this class takes an ArrayList of strings containing all the data needed to create an *User* object. The data in the ArrayList is ordered as follows: [ [ Request type, Permissions, User ID, First name, Surname, Email, Department ID, Permission level, Hashed Password, Salt, Boolean value whether the user is active]. The User will then be added to the database. Will then return success or failure.

# 6 - Appendix

## Appendix 1: Database Object Role Model

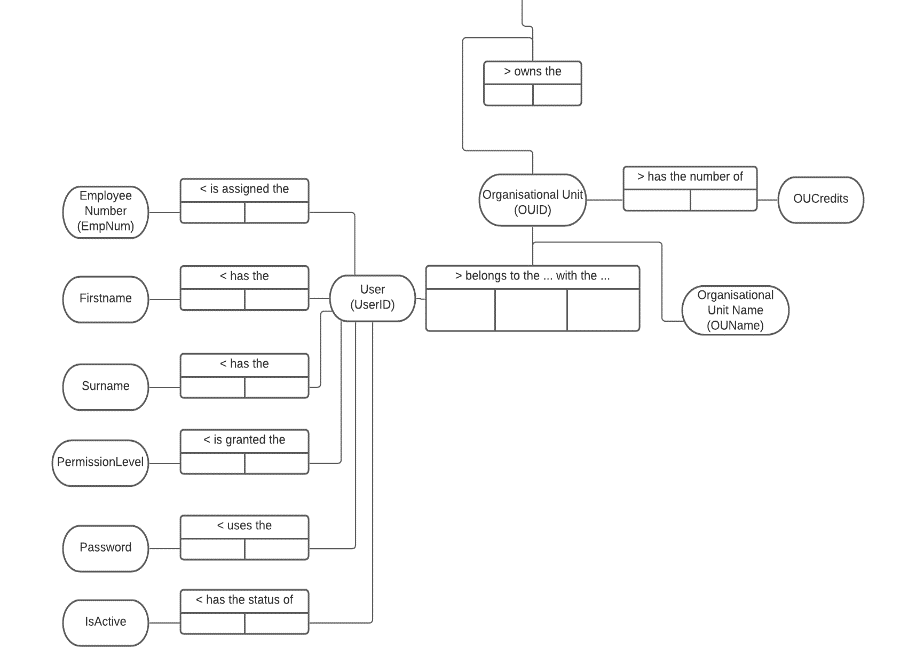
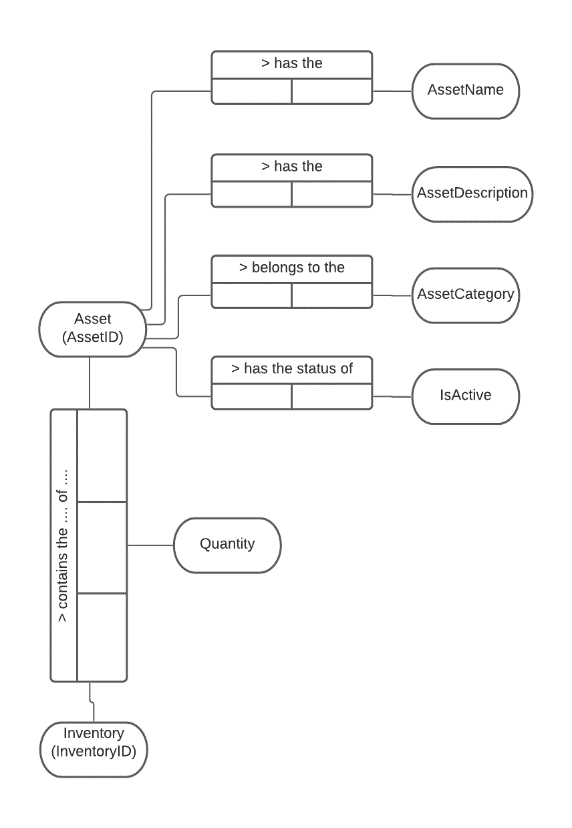
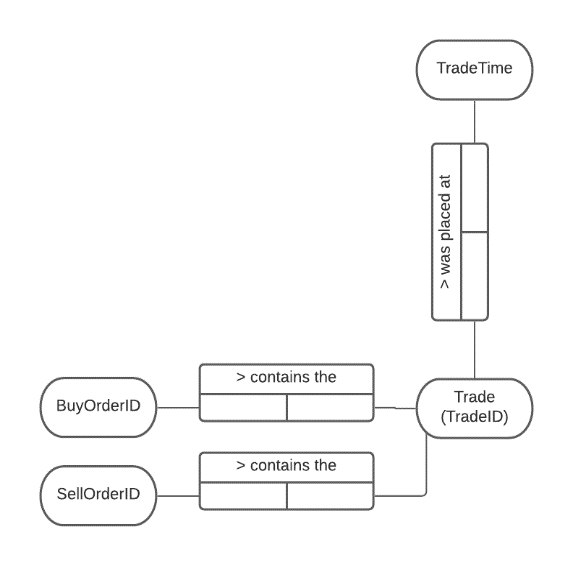
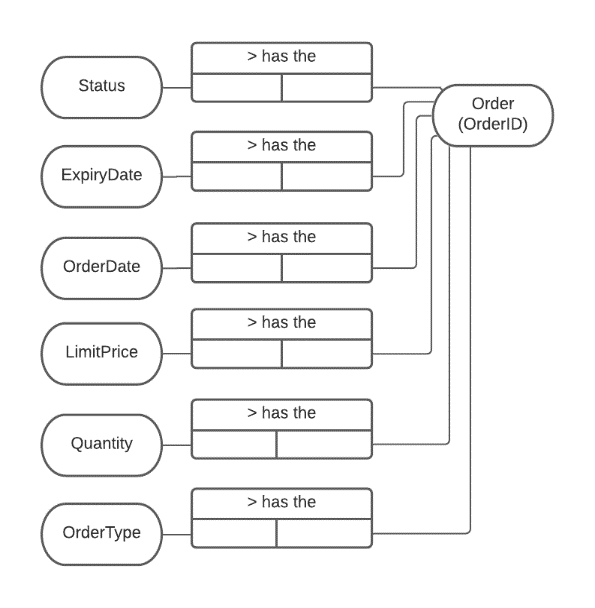


Figure : Database Object Role Model

## Appendix 2: Database Script

-- ----------------------------------------------------------------

-- DATABASE cab302db

-- ----------------------------------------------------------------

CREATE DATABASE cab302db

CHARACTER SET `latin1`

COLLATE `latin1\_swedish\_ci`;

-- ----------------------------------------------------------------

-- TABLE assetcategories

-- ----------------------------------------------------------------

CREATE TABLE cab302db.assetcategories

(

`AssetCatID` INT(20) NOT NULL COMMENT 'Unique Asset Category ID',

`AssetCatName` VARCHAR(30)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Asset Category name',

`AssetCatActive` TINYINT(1)

NOT NULL

DEFAULT 1

COMMENT 'Is the Asset Category currently active',

PRIMARY KEY(`AssetCatID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE assets

-- ----------------------------------------------------------------

CREATE TABLE cab302db.assets

(

`AssetID` INT(20) NOT NULL COMMENT 'Unique Asset ID',

`AssetName` VARCHAR(30)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Asset name',

`AssetDescription` VARCHAR(100)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Further information about the asset',

`AssetCategory` INT(20) NOT NULL COMMENT 'Asset category',

`AssetActive` TINYINT(1)

NOT NULL

DEFAULT 1

COMMENT 'Is the Asset currently active',

PRIMARY KEY(`AssetID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE inventories

-- ----------------------------------------------------------------

CREATE TABLE cab302db.inventories

(

`InventoryID` INT(20) NOT NULL COMMENT 'Unique Inventory ID',

`AssetID` INT(20) NOT NULL COMMENT 'Asset ID',

`OUID` INT(20) NOT NULL COMMENT 'OU ID',

`Quantity` INT(20)

NOT NULL

COMMENT 'Quantity of particular asset held by OU',

`InventoryActive` TINYINT(1)

NOT NULL

DEFAULT 1

COMMENT 'Is the Inventory currently active',

PRIMARY KEY(`InventoryID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE orders

-- ----------------------------------------------------------------

CREATE TABLE cab302db.orders

(

`OrderID` INT(20) NOT NULL COMMENT 'Unique Order ID',

`OrderType` TINYINT(1)

NOT NULL

COMMENT 'Buy(true) or Sell(false) order',

`UserID` INT(20) NOT NULL COMMENT 'User ID',

`AssetID` INT(20) NOT NULL COMMENT 'Asset ID',

`OUID` INT(20) NOT NULL COMMENT 'OU ID',

`Quantity` INT(20)

NOT NULL

COMMENT 'Quantity of particular asset to be bought/sold',

`LimitPrice` INT(20)

NOT NULL

COMMENT 'Limit price for an individual asset',

`OrderDate` DATETIME(0)

NOT NULL

DEFAULT CURRENT\_TIMESTAMP()

COMMENT 'The Timestamp for when the order was placed',

`OrderStatus` INT(1) NOT NULL COMMENT 'The current status of the order',

`ExpiryDate` DATETIME(0)

NOT NULL

COMMENT 'Timestamp for when the order will expire if not filled.',

PRIMARY KEY(`OrderID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE orderstatus

-- ----------------------------------------------------------------

CREATE TABLE cab302db.orderstatus

(

`OrderStatusID` INT(20) NOT NULL COMMENT 'Unique Order Status ID',

`OrderStatusDesc` VARCHAR(20)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Description for the order status',

PRIMARY KEY(`OrderStatusID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE ous

-- ----------------------------------------------------------------

CREATE TABLE cab302db.ous

(

`OUID` INT(20) NOT NULL COMMENT 'Unique Organisational Unit ID',

`OUName` VARCHAR(30)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Organisational Unit name',

`OUManager` INT(30)

NOT NULL

COMMENT 'UserID for Organisational Unit Manager',

`Credits` INT(30)

NOT NULL

COMMENT 'Number of credits held by the Organisational Unit',

`OUActive` TINYINT(1)

NOT NULL

DEFAULT 1

COMMENT 'Is the Organisational Unit currently active',

PRIMARY KEY(`OUID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE permissions

-- ----------------------------------------------------------------

CREATE TABLE cab302db.permissions

(

`PermissionsID` INT(20) NOT NULL COMMENT 'Unique Permissions Level ID',

`BuyPermissions` TINYINT(1)

NOT NULL

DEFAULT 0

COMMENT 'Does user have permissions to buy',

`SellPermissions` TINYINT(1)

NOT NULL

DEFAULT 0

COMMENT 'Does user have permissions to sell',

`ApprovePermissions` TINYINT(1)

NOT NULL

DEFAULT 0

COMMENT 'Does user have permissions to approve orders',

`EditUserPermissions` TINYINT(1)

NOT NULL

DEFAULT 0

COMMENT 'Does user have permissions to edit users',

`CreateUserPermissions` TINYINT(1)

NOT NULL

DEFAULT 0

COMMENT 'Does user have permissions to create users',

PRIMARY KEY(`PermissionsID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE trades

-- ----------------------------------------------------------------

CREATE TABLE cab302db.trades

(

`TradeID` INT(20) NOT NULL COMMENT 'Unique Trade ID',

`BuyOrderID` INT(20) NOT NULL COMMENT 'Buy Order ID',

`SellOrderID` INT(20) NOT NULL COMMENT 'Sell Order ID',

`AssetID` INT(20) NOT NULL COMMENT 'Asset ID',

`TradeTime` DATETIME(0)

NULL

DEFAULT NULL

COMMENT 'Timestamp for when trade was completed',

PRIMARY KEY(`TradeID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

-- ----------------------------------------------------------------

-- TABLE users

-- ----------------------------------------------------------------

CREATE TABLE cab302db.users

(

`UserID` INT(20) NOT NULL COMMENT 'Unique user ID Employee ID',

`FirstName` VARCHAR(30)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Users first name',

`Surname` VARCHAR(30)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Users surname',

`Email` VARCHAR(50)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Users email address',

`UsersOU` INT(20) NOT NULL COMMENT 'Users Organisational Unit',

`PermissionLevel` INT(20) NOT NULL DEFAULT 0 COMMENT 'Permission Level',

`Password` VARCHAR(20)

CHARACTER SET latin1

COLLATE latin1\_swedish\_ci

NOT NULL

COMMENT 'Users Password',

`UserIsActive` TINYINT(1)

NOT NULL

DEFAULT 1

COMMENT 'Is the user active',

PRIMARY KEY(`UserID`)

)

ENGINE MYISAM

COLLATE 'latin1\_swedish\_ci'

ROW\_FORMAT DEFAULT;

# 7 - Bibliography

1. MariaDB. 2014. "Why MariaDB? Advantages over MySQL | MariaDB." Last modified October 29, 2014. <https://mariadb.com/resources/blog/why-should-you-migrate-from-mysql-to-mariadb/#:~:text=First%20and%20foremost%2C%20MariaDB%20offers,large%20organizations%20and%20corporate%20users>..
2. Steam. “Community Market”. <https://steamcommunity.com/market/>
3. Wikipedia. “Technical Debt”. <https://en.wikipedia.org/wiki/Technical_debt>