Paper Trading Application

Date: 2023

Centre Number: [redacted]

Candidate Number: [redacted]
Centre Name: [redacted]
Subject Code: [redacted]

NON-EXAMINED ASSESSMENT DOCUMENTATION

ARAVIND KUMAR

Contents

Analysis

Project Hypothesis

In today's day and age, financial investment has become increasingly more common and more popular, as the world constantly shifts with new technological advancements. For many people, along with earning a given salary, they then use a portion of these earnings in investing in stock from different companies as a more long-term option in sustainably keeping themselves financially stable for the long term and especially when they retire, they will have successful funding due to these investments. Although investment can be exciting and very rewarding, there is an ever-growing chance that individuals' funds could dwindle if not conditioned to have that trading mind and use a correct strategy. Even today, prices of many stocks and funds are influenced by politics and the individual actions of many celebrities or entrepreneurs - a once thriving investment could result in a detrimental loss at any second.

What's important is for someone to be financially aware of how to handle investments and using a paper trader is the best way to do it. The paper trader emulates investment management of stocks with virtual money instead of live trading which has real money. In this case, since a user is practicing, there will be no financial loss if the user makes a wrong decision. It replicates what it would be like to have an investment account both making a user aware of the scenarios within investment and improving their experience in handling and spending their money.

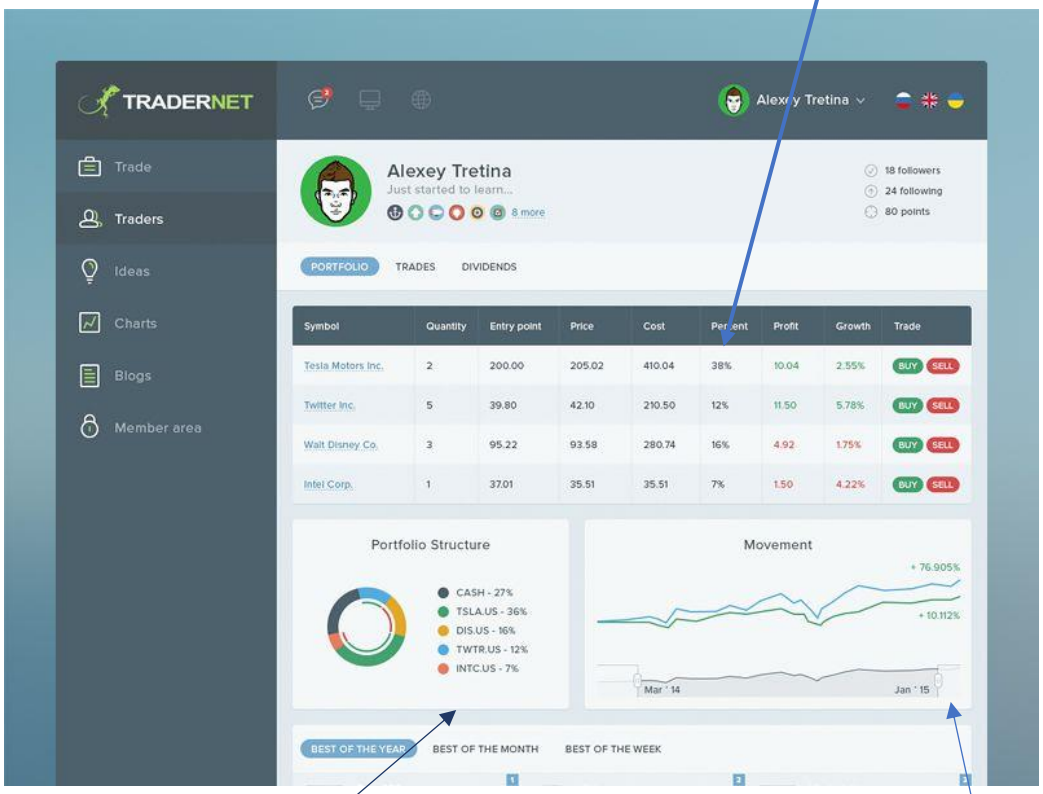
The role of this application is to provide a medium to generate confidence for the user to make their investments with building their emotional integrity in making decisions even if the decisions turn to a poor result, it will guide them in making sure they can make the correct changes to make a difference.

Additionally, with the current state of the UK economy and how much stock prices have been fluctuating, it is important for users to invest smartly given the current situation and especially if the state that we live in lasts for a longer time, it will be crucial to train users to act and work upon their financial strategies.

Even going beyond stocks to activities such as cryptocurrency, the combination has become far more popular over the past few years with individuals buying and selling every day which really emphasises the giant community of financial investment. I hope with this application that when people do eventually go into the real world in trading stocks that they will feel confident and at ease.

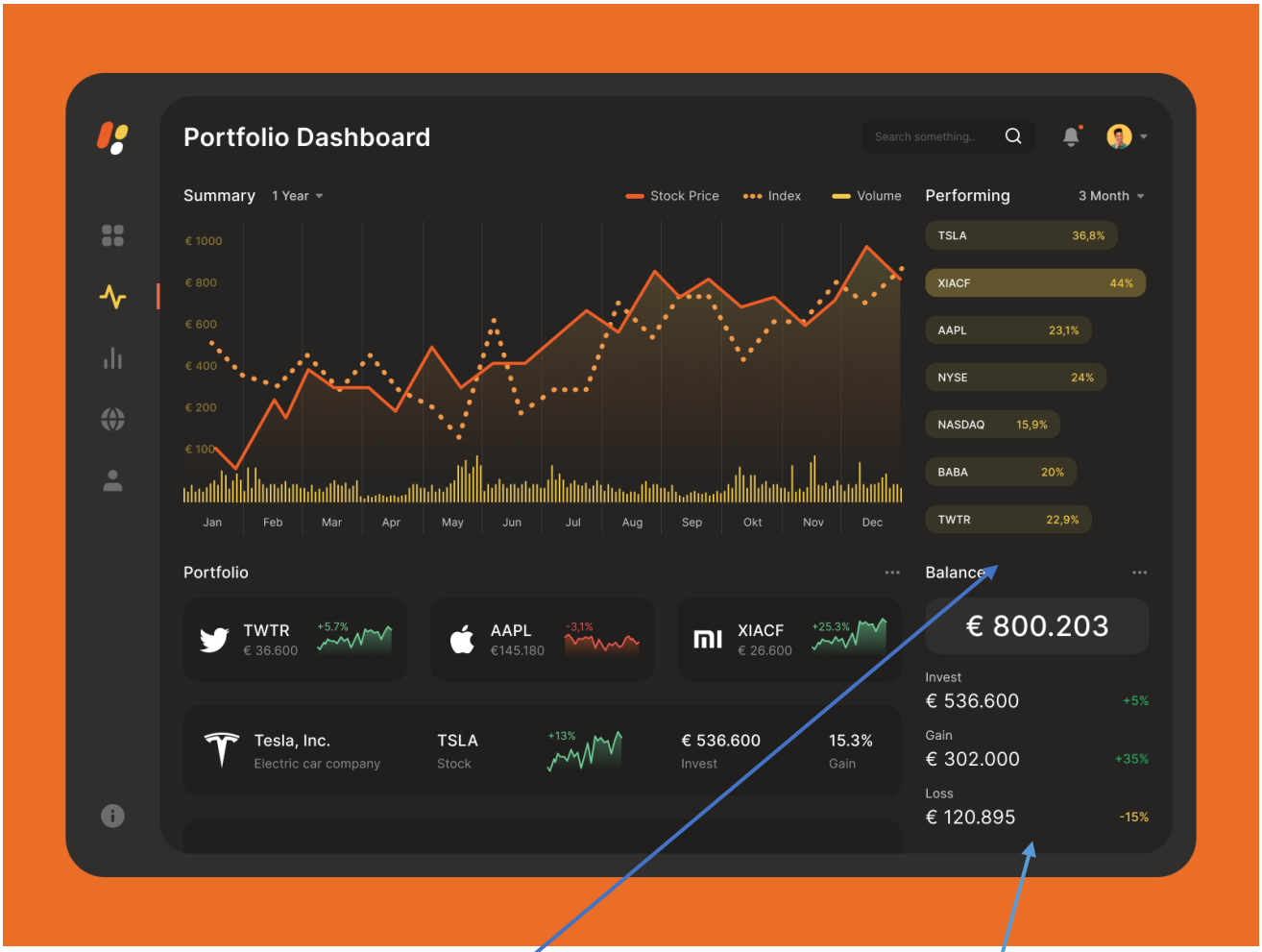
Overview of other solutions

Tables like these give the generic information on stock performance e.g., Current Price, growth, and percent change



A user's investments can be represented as a portfolio allowing the user to analyse their stock investments in a more visual representation. The use of the pie chart especially can be used to

For money that the user invests into stocks, we can review their overall movement of how their wallet is performing over a time duration.

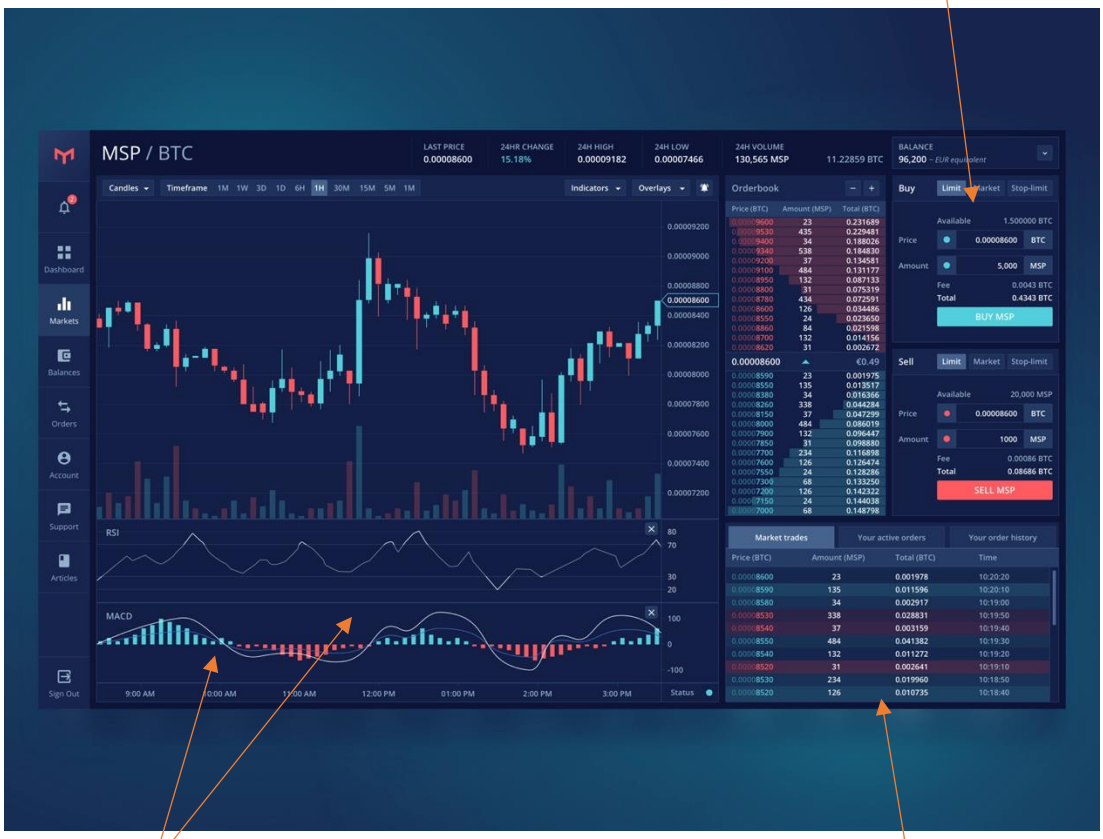


For companies that users have invested stock in, this feature will allow how their stock are performing and this performance can be dependent on multiple of different factors:

1) Supply and Demand
2) Company financial performance
3) Economic trends

For a user's portfolio page, it will provide an opportunity to analyse more closely of a user's spendings and in this case show the proportion of money invested and how much has turned into profit or loss. Having this sort of representation will provide a more distinct way of showing how their investments are performing and if immediate action to change investment strategies is needed.

Within the dashboard, users can make quick trades given how they view the behaviour of a given stock. The quantity of stock can be selected as a parameter



Using other statistical indicators, we can add momentum points to show the flow and movement of a stock's price over a longer period allowing traders to see both how a stock has behaved over time and he they can use it to predict future prices.

Users can easily view their previous orders on what stock they have bought and shows them the timeline on when they were bought. Other visual representations can be used to represent the performance of given stock on if it was a profit or loss.

Analysis of Other Solutions

For almost all solutions available, apps aim to simulate real-life conditions and behaviour of the current stock market. This involves current stock prices and statistics related to each company involving measures such as a change in stock price, volume, and high and low prices within the NYSE (New York Stock Exchange). As a result, I aim to implement an API that would provide this information to provide the most realistic environment for the user.
What my end-user commented on various other solutions is how some of them resorted to using very bare bones activities such as just buying or selling stocks and replied that there was a lack of substance in the process and almost felt unrewarding and monotonous. He suggested having some sort of statistical investigation of firstly the user's portfolio which would really provide better feedback towards the user as it provides a more personal quality to the system and really connects the app with the user.

Pros of other applications:

- Since the role of a paper trading application is to invest using virtual money, if an investment that a user made turns out to be a failure, many current applications provide feedback on the users' decisions when placing money on buying stock. This may be telling the user both what they did well in the situation and the decisions that made them lose money.
- Since paper trading is a very common tool for beginners who are looking to invest their time in trading, many applications are free of charge which provides makes the program more user-friendly and applicable to a wide range of people as there is no pre-payment. This is also a good addition as it can attract more people to use their application which provides a popularity benefit for the company using it.

Cons of other applications:

- Some applications work through only allowing trades to go through at and available. trading hours since stock markets hold a strict timing structure – once the time has passed, there is no chance to make further advancements and become more familiar.
- Since there are many paper trading applications around to use, many of them don't provide and allow advanced features that can be very crucial when using real trading apps. As a result, those traders that are very advanced and look to have a realistic environment one-to-one to their real trading applications may have a hard time truly replicating their work and improving on their strategies. (Farley, 2021)

Acceptable Limitations

Since my application utilises virtual money, I believe it would not be possible to incorporate actual financial transactions including real money as that would be first very difficult to maintain the security of the system in terms of encrypting transactions as bank details would be required for transactions therefore other systems would be needed to secure those.
Furthermore, my current API that I use supports 9 API calls per minute meaning that I won't be able to perform simultaneously many API calls. Additionally, due to my current

SQL infrastructure, the model's concurrency prevents me from performing multiple SQL queries at the same time limiting the rate of updates after an operation.

Questionnaire

End User: Ashwin Kumar

Occupation: Trader

1.

What do you look out for in Trading simulators?

The main thing that catches my eye when using paper trading applications is the user-friendly experience. Especially for the UI, I want something that I can understand in terms of the features and make sure I'm not left in the dark in terms of what to do. However, I would like to have an application that still contains all the features that are within the real trading apps as I don't want practice trading strategies on one system and then move onto another one which has no correspondence to the original one.

2.

What do you think are the drawbacks to the current systems available?

One of the most important factors involved with trading and the way stocks are traded are decisions such as risk tolerance and investment objectives. What I have noticed is that some trading simulators don't dwell on this and just use a buy and sell policy which seems very redundant in terms of learning the environment of trading stocks. Another problem which I believe affects most applications is the lack of historical data. Especially for systems that tend to be free, they don't have data that scales back to over 15 years and with data being such a crucial part in terms of making trading strategies and observing various trends, the less we have, the more sceptical we become when making decisions.

3.

Who do you recommend trading simulators to?

Anyone because there is a need for practice in any investment scenario where if you are new and want to get involved in investments where it is almost vital to have practice before going into the real world. Even for more experienced traders, they strive to keep up, improving and refining their decisions and choices.

4.

What existing features do you find useful when using trading simulations that you can apply to real-life trading?

What I find very useful is how connected you can be with the app and your investments. Especially for a traders' portfolio, I would like to have some analysis of my wallet understanding how my money has been spent and even better allowing the trader to

add financial objectives which both provides an organisation element and some motivation for the trader to achieve their objectives.

5.

Why do you think trading simulators play such a pivotal role in the finance industry?

I believe the most important aspect for a simulator is to perfect your trading strategies. To have the mindset of a trader plays such an important role in making sure that your success is maintained. Getting used to system makes a great difference as well as if you do go into the real-world trading with actual money, having a smooth transition from virtual investments to real investments is vital.

Modelling of the solution

What plays an essential role in my project is the use of statistical modelling and its use in looking at technical analysis. This analysis is formed from hundreds of different indicators to help forecast and trade depending on price movements. Within my project I aim to include different statistical analysis techniques that will identify trends in data using mathematical formulae which will present to the user the patterns in how data is flowing for a duration of time to which this can be used to for the user as an essential piece of information to make trading decisions.

Sentiment/Technical analysis

The use of technical analysis is the study of historical market data which involves processes such as market psychology and, behavioural economics. The main takeaways from this are:

- Technical analysis attempts to predict future price movements, providing traders with the information needed to make a profit.
- Traders apply technical analysis tools to charts to identify entry and exit points for potential trades. (Chen, 2021)

The main role for the technical analysis is to complement both the technical indicators to generate a model that will show the current state of a current stock showing the user if a stock is worth buying or not. The way this is interpreted is through if a stock is bullish or bearish.
The notion around bullish and bearish sentiments is the fact that if a stock is bullish then the model believes that prices will go up and if it is bearish, then prices may go down.
The model itself provides a response which can inform the user to decide to sell or buy a given stock.
I will aim to implement a model that will replicate this system in the form of a confidence meter.

Geometric Brownian Motion

Geometric Brownian Motion is both recognised as a Markov process and is also known as a very common statistical model used in Monte Carlo Simulations.

The formula states as shown:

$$\frac{\Delta S}{S} = \mu \Delta t + \sigma \epsilon \sqrt{\Delta t}$$

where:

$S =$ the stock price

$\Delta S =$ the change in stock price

$\mu =$ the expected return

$\sigma =$ the standard deviation of returns

$\epsilon =$ the random variable

$\Delta t =$ the elapsed time period

To rearrange to calculate the change in stock price over a given period, we find the equation manipulates to:

$$\Delta S = S \times (\mu \Delta t + \sigma \epsilon \sqrt{\Delta t})$$

Figure 1, GBM rearranged equation for price change.

Drift

Shock

With this model, we assume that after a distinct time, the price of the stock will drift by the expected return (**μ** = constant). However, this drift will undergo a collision through a random shock.

This "shock" constant is calculated by:

Standard deviation (s) x random normal number (∈)

Once the process is complete with the dependent parameters of being the number of simulations to number of trading days, a graph will be created for the user to analyse:

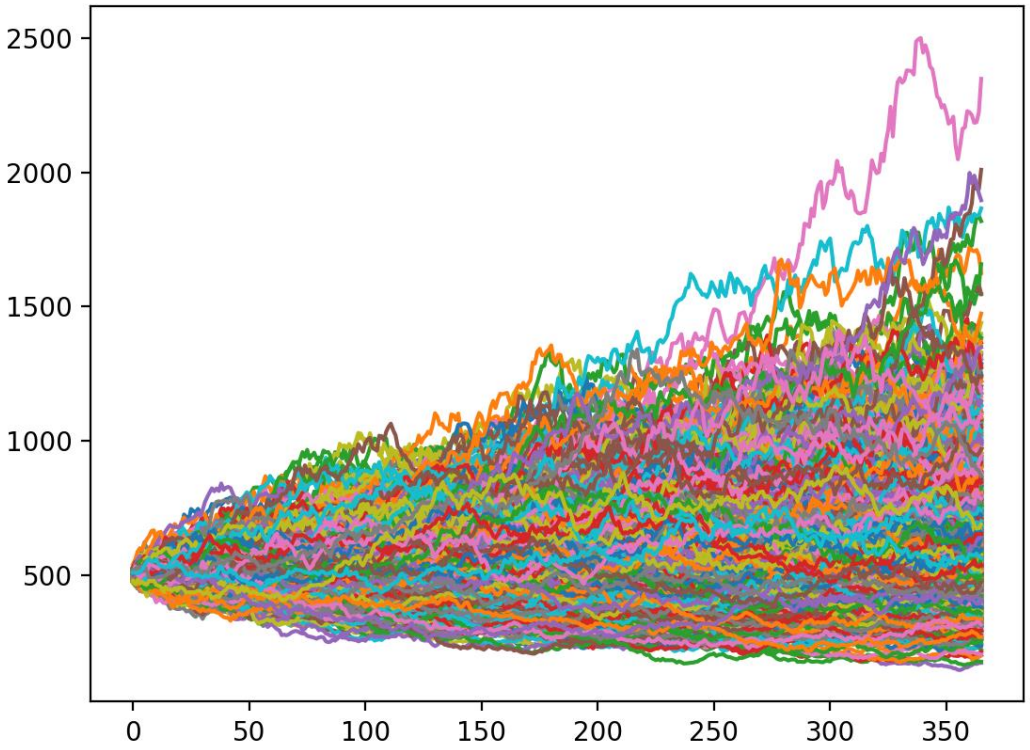


Figure 2, A graph representing a GBM simulation (x-axis – Simulation Day. Y-axis – Stock price)

Figure 3, Graph showing the relationship between bullish and bearish behaviour.

Prototype

I have chosen to utilise an API that would supply the program with the essential information of any stock within the NYSE. An API plays such an important role in both creating an efficient and fluid experience for users to easily find what they want to view and to also make it an easier process when applying the API data in later methods and algorithms.

My current API prototype works under a console application, but I aim to implement and link it to a GUI for a more professional approach for the data.
I will use asynchronous methods to retrieve data as it is a quicker in more parallel tasks as it will do multiple operations sequentially when using the app.

```
public async Task<Dictionary<string, double>> Dict_choice(string ticker, string startdate, string enddate, Dictionary<string, double> dictvals, string choice)
{
    HttpClient client = new HttpClient(); // Creates an new HttpClient
    var request = new HttpRequestMessage
    {

        Method = HttpMethod.Get,
        RequestUri = new Uri($"https://api.twelvedata.com/time_series?symbol={ticker}&start_date={startdate}&end_date={enddate}&interval=" +
        $"1month&apikey=0df69dce9d184e34a7b477c9e9a224d3&format=JSON"), // Web API string used to retrieve data depending on given parameters and factors
    };
    using (var response = await client.SendAsync(request))
    {
        response.EnsureSuccessStatusCode(); // used to ensure API response was successful
        var JSON_response = await response.Content.ReadAsStringAsync(); //Converts the JSON response to read it as a string.
        JObject jobject = JObject.Parse(JSON_response); // Parses JSON message so individual items can be processed
        // TEST IF ARRAY EMPTY
        foreach (var val in jobject["values"]) // loops through outer part of JSON string to view the inner parts.
        {
            dictvals.Add(Convert.ToString(val["datetime"]), Convert.ToDouble(val[choice])); // Adds items to the dictionary so they can be accessed in the main program.

        }
        return dictvals;
    }
}
```

Within my API, I will be utilising various important methods which will aid in the mathematical calculations and processes. A simple method above provides data of a given stock over a given time which will be called as shown:

```
List<string> times = new List<string>(); // Creates a new times list that will store the times the user enters
APICLIENT AC = new APICLIENT(); // Creates a new instance of the APICLIENT Class
times = AC.Initalise_timeseries(times); // Method used to initalise the dates that will be used
Console.WriteLine("Enter a ticker"); // Asks user to enter stock symbol
string ticker = Console.ReadLine();
string startDate = times[1];
string EndDate = times[0];
Dictionary<string, double> Prices = new Dictionary<string, double>(); // Creates new dictionary to store both dates and corresponding price

Prices = await AC.Dict_choice(ticker, startDate, EndDate, Prices, "close"); // Calls Web API to retrieve information and stores in 'Prices' Dictionary
foreach (var val in Prices)
{
    Console.WriteLine($"Date: {val.Key}, Price: {val.Value}"); // loops through dictionary to display data.
}
```

With the output being:

```
Amount of months of data from the past?:
5
Enter a ticker
AAPL
Date: 2022-10-01, Price: $145.66
Date: 2022-09-01, Price: $138.2
Date: 2022-08-01, Price: $157.22
Date: 2022-07-01, Price: $162.51
Date: 2022-06-01, Price: $136.72
```

Example of Apple's Historical stock data

Objectives

1. Login and Sign-up Menu

- 1.1. User can either make an account or login
- 1.2. If user doesn't have account, they can sign-up where data will be stored on an account database – both username and password will be stored.
  - 1.2.1. Passwords will be hashed and will be stored on the database.
  - 1.2.2. When user logs in, it will hash the input password and make a comparison to password stored in database table. If same, allow entry. If not, deny entry. Ask again.
    - 1.2.2.1. Once valid, open application.
  - 1.2.3. If user is signing up, the password has the match the confirm password field. If not ask again.
    - 1.2.3.1. once they generate the account, they will have a new wallet which will initially have a balance of $10,000 inserted.
- 1.3. There should also be an admin account
  - 1.3.1. There should be a separate password for accessing the admin account.
    - 1.3.1.1. The user should repeatedly be asked if incorrect.
  - 1.3.2. It should allow them to view all users' transactions and portfolios. There should be a tally showing the number of users.
  - 1.3.3. There should also be an option to delete an account from the application.

2. Analysis of Historical Stock Data

- 2.1. A search bar can be used to select a given stock to view their prices over a duration of time in a more visual representation – this is in the form of line graphs.
  - 2.1.1. If they do not enter a valid stock symbol or valid times, then they will be alerted to try again.
  - 2.1.2. Users will be able to select to view historical data over given time periods which if button is clicked for given period, the graph will be updated.
  - 2.1.3. Users will be able to view their previous searches of a stock when they search for one and this will be accomplished through a stack where each search will push itself to the top of the list.
- 2.2. Besides the information, there will be a technical analysis of that given stock which will provide strong feedback on if a stock should be bought or not. This will be in the form of a confidence percentage.
  - 2.2.1. This will be using the MACD algorithm in conjunction with Exponential Moving Averages to measure if a stock should be bought or sold depending on trends of price. It should also calculate percentage confidence of that result occurring.
- 2.3. Within an area of the screen will hold the user's wallet where they can view their remaining balance.

2.4. A user will also be able to buy a stock buy selecting both the name and the quantity. Again, checks should be made.
  2.4.1. Checks on if the stock exists on the NYSE
  2.4.2. Check the quantity is greater than 0.
  2.4.3. It also checks if there is the enough money for transaction.
2.5. The program should update the user's balance, update the portfolio, and add the transaction to the table transactions.

3. Portfolio Management

3.1. A main table will be used to display the shares that a user has in different companies. Once an order is complete, the order will be added to this list.
  3.1.1. The table will display their order history of their recent orders.
  3.1.2. This will contain information about the price at transaction, quantity, Transaction price, Time of Purchase, and transaction type (buy/sell)
3.2. A user's portfolio will be able to be viewed with a drop-down menu showing all their stocks and once clicked, it will provide data such as proportion and the amount of investment in their stock.
  3.2.1. This will use both cross-table parametrised SQL and aggregate functions to analyse and calculation stock performance.
3.3. A user will also be able to view their whole statistical performance of their portfolio through identifying most expensive investment and most invested company.
  3.3.1. Users will also be able to view their portfolio profit of their whole wallet which again will use cross table parameterised SQL and aggregate functions.
3.4. A user will also be able to sell a given stock of certain number of shares. This process is more intensive in terms of the validation.
  3.4.1. Check a stock symbol is valid and check the quantity field is greater than 0.
  3.4.2. If stock symbol is valid, the program should check if it is within the user's portfolio. This will involve retrieving the user's invested stocks from the database and performing a recursive search to identify if the stock they want to sell is within their portfolio.
  3.4.3. It should then check if they have enough stocks to sell.
  3.4.4. It should then retrieve the price of transaction using both the API and quantity field.
  3.4.5. It should then change the user's balance, add the transaction to the database and update the user's portfolio.

4. Financial Objectives Tracker

4.1. A user will be able to make financial objectives on what they would like to achieve soon. The table displayed will order the objectives utilising a priority queue for this process.

4.1.1. A menu will be presented containing a table showing the structure of their objectives with the most important (Critical) being at the top and least important (Not Important) being at the bottom. Earlier dates take priority.

4.1.2. A menu on the right-hand side will allow users to add objectives to their objective list.

4.1.2.1. It will perform checks to validate if the inputs are valid-All fields entered and valid date.

4.1.3. For a user wanting to edit a certain objective, there will be an edit button which will utilise both the SQL queries and priority queue implementation.

4.1.3.1. Needs to check if objective has been selected before editing.

4.1.3.2. The user will be able to change aspects such as their title and description as well as the priority and deadline which will change the objective's placement in the queue.

4.1.3.3. Once a user clicks Ok, the table will automatically update changing the placement of the queue objectives.

4.1.3.4. A user can also mark an objective complete which will remove it from the objective list.

5. Geometric Brownian Motion Stock Price Simulation

5.1. A user will be given boxes to fill depending on the given conditions they want to set for their stock.

5.1.1. For the volatility and drift constants, a set of checks will be in place to prevent users from entering erroneous data that would in turn crash the program. E.g., It shouldn't allow users to enter negative volatility and drift constants.

5.1.2. It should make sure that all fields are entered and filled in.

5.2. When a user is selecting the stock, they would like to simulate, there should be a check to see if the stock symbol is valid and part of the NYSE.

5.3. Once the user has clicked the button to simulate a screen will be presented showing all the data from the simulation.

5.3.1. This will be in a table showcasing the Sim Number, Date of time of that current simulation and the corresponding stock price simulated.

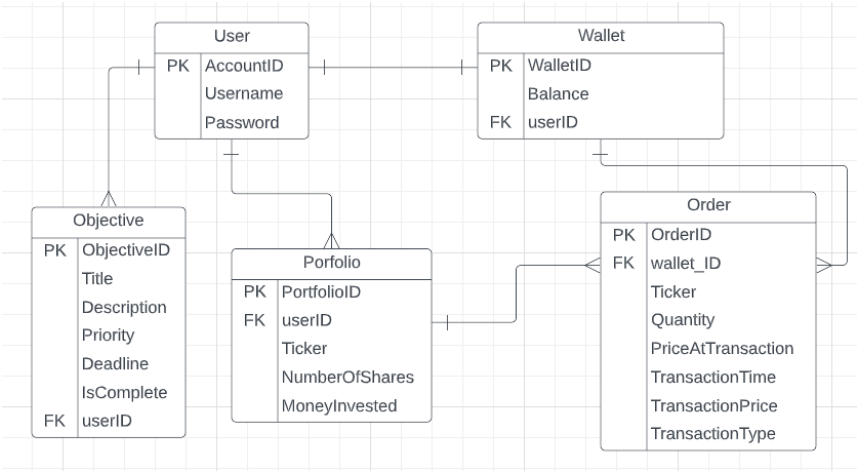
Documented Design

Below shows how my app aims to be structured:

I chose to separate each section and decompose the larger problem into small subproblems and task which negated a tunnel vision way of tackling the problem but instead allowed me to solve each problem with a greater understanding.

Linked to my objectives, I decided to create a structural diagram that would provide the visual capability to see how my app would both structure and function and how different parts would interlink with each other.

To begin with, I decided to generate an Entity-Relationship diagram to provide a greater picture on how I would structure my tables and understand the relationships between each one. Below presents my representation:



Since my program is dependent on user interactions and statistical analysis, it is important that databases are involved to store the information so it can be easily accessed and structured in the correct way.

Whenever a user makes a stock transaction, a database will store this purchase history mentioning given facts.

Below presents the **Orders** table:

Field	Key	Data Type	Validation
OrderID	Primary	auto	
Ticker		string	3
Quantity		integer	
PriceAtTransaction		float	
TransactionPrice		float	
TransactionTime		Date	
TransactionType		string	
Wallet_ID	Foreign	auto	

Whenever a user makes an order, the following details will be entered to the database so it can be viewed in the portfolio section. Within the program, this data along with the data in the account portfolio database will be used to analyse the portfolio performance.

Connected to the Orders database will be the Portfolio database which will be tracking the users' investments holding individual information on their accounts' current value, total profits, and percentage changes around their investments.

Below presents a design for the **Portfolio** table:

Field	Key	Data Type	Validation
PortfolioID	Primary Key	float	
Ticker		string	
NumberOfShares		int	
MoneyInvested		float	
userID	Foreign Key		

The **Orders** and **Portfolio** database will be closely linked in terms of sharing the data from transactions to influence the user's portfolio.

The portfolio will show statistics related to their investments and their whole wallet which will be directly sourced from the Orders database.

CASE generated DDL script was used to construct my tables and provide any suitable properties and constraints for different fields.

```
CREATE TABLE USERS(
AccountID INTEGER not null primary key AUTOINCREMENT,
Username TEXT not null,
Password TEXT not null
);

CREATE TABLE [Objectives](
  [ObjectiveID] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  [Title] TEXT NOT NULL,
  [Description] TEXT NOT NULL,
  [Priority] TEXT NOT NULL,
  [Deadline] TEXT NOT NULL,
  [IsComplete] BOOLEAN NOT NULL,
  [userID] INTEGER NOT NULL REFERENCES [USERS]([AccountID]) ON DELETE CASCADE);

CREATE TABLE WALLET(
WalletID INTEGER not null primary key AUTOINCREMENT,
Balance REAL not null CHECK (Balance >= 0),
userID INTEGER not null,
FOREIGN KEY (userID) REFERENCES USERS(AccountID) ON DELETE CASCADE);

CREATE TABLE [Orders](
  [OrderID] INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  [Ticker] TEXT NOT NULL,
  [Quantity] INTEGER NOT NULL,
  [PriceAtTransaction] REAL NOT NULL,
  [TransactionTime] TEXT NOT NULL,
  [TransactionPrice] REAL NOT NULL,
  [TransactionType] TEXT NOT NULL,
  [wallet_ID] INTEGER NOT NULL REFERENCES [WALLET]([WalletID]) ON DELETE CASCADE);

CREATE TABLE Portfolio(
PortfolioID INTEGER not null primary key AUTOINCREMENT,
Ticker TEXT not null,
NumberOfShares INTEGER not null,
MoneyInvested REAL not null,
userID INTEGER not null,
FOREIGN KEY (userID) REFERENCES USERS(AccountID) ON DELETE CASCADE
);
```

As we can see, I have specified in the primary keys that they are autoincrement and will not be required to have an input for generation. Furthermore, I have placed further checks especially for the balance to measure if a wallet is empty or not using **CHECK** functions. I've also placed statements such as **ON DELETE CASCADE** for wallet to if I delete a user in the admin section, it will erase all records in other tables – this will help reduce data redundancy and promoting normalisation.

Function of API and its role in Predicting Stock Movements

The role of the API plays at the heart of my program as almost all activities within the program both lie and depend on retrieving the stock data and statistics to both perform the mathematical analysis of trends and to also provide for when a user buys or sells stock.

As a result, within the API, I decided to implement a custom object which would hold this type of data, and this is known as the **Values** object.

This values object holds information such as:

1) High price
2) Low price
3) Open price
4) Close Price
5) Percent Change
6) Previous Close

Each part of this object plays an important role in both identifying trends within a certain stock when applying the technical analysis on the stock.

Especially for the API, its foundation is to provide data in a quick and efficient fashion which as a result pushed me to use the API methods being asynchronous allowing for a more fluid way of gaining the information both improving performance and user experience. Furthermore, I will also include task threading which will improve the responsiveness from the API when handling large amounts of stock data.

One of the most important methods is the **Prices** method which has its aim of retrieving financial data of a certain company over a time and over a particular interval.

When we get the financial data, I aimed to store it in a list and especially a list of the **Values** object.

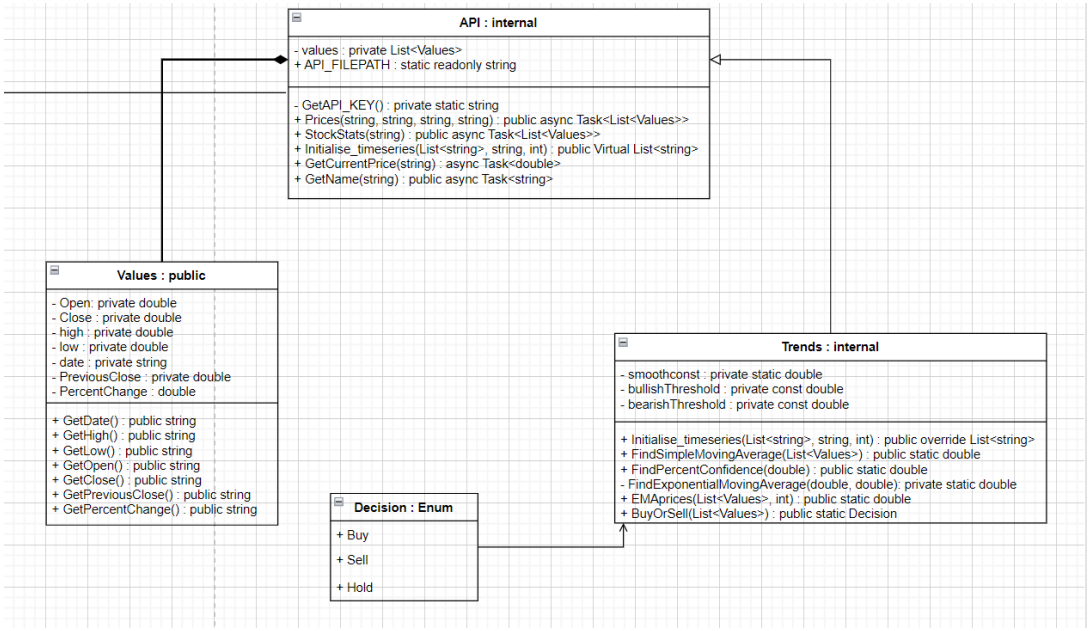
Below presents my algorithm to gain stock prices over an interval:

```
FUNCTION async Task<List<Values>> Prices (startDate, EndDate, ticker,
interval)
HTTPclient client = new HTTPclient()
var request = new HTTPrequest
{
Request=HTTPMethod.Get(https://api.twelvedata.com/time_series?symbol={ticker}&start_date={
startDate}&end_date={EndDate}&interval={interval}&apikey={API.GetAPI_KEY()}&format=JSON")

}
var response = await client.SendAsynchronousResponse(Request)
{
      response.CheckSuccessResponse()
      string JSON_Response = await response.ReadAsStringAsync()
      JObject object = JObject.Parse(JSON_Response)
      if(object[“code”] != 400) THEN
        foreach value in object[“values”]
           List.Add(stockPriceData)
      return List

}
else return null
```

We can see through this method that we are using the asynchronous procedure of retrieving the stock data via the API and we are also doing validations to check the response code is not equal to 400 which would signify the API retrieval process failed.
Below presents a class diagram highlighting how each method is interlinked.



Our Values object will store the data of each date and its corresponding stock data and since we have used the values properties as being private, I have used **encapsulation** to return the given property as a cohesive module. Through the diagram, we can see that I have used association with the API class.

Through this class diagram, we can see that I have used the values object as a collection in the API class and used **composition** to create an instance of a list of the object **values**. Since the **API** and **Trends** class are heavily linked on how the Trends class requires the stock data to make the stock predictions, I have inherited the methods from the API class into this class. Furthermore, since initialising dates is different for both displaying the user data and making the predictions, I have made the method 'Initialise_timeseries' virtual and overridden this method in the Trends class as it performs a different action and process to generate the dates for stock data collection.

Association has been used within my API class as I have instantiated a list of the class 'Values' as within my API retrieval of stocks, each date of stocks will have an open price, close price etc. This makes it far easier to retrieve this information when I output it onto the graph and do statistical analysis through the Trends class.

Application of my API in Stock prediction

One of my main uses of my API to supply daily data for my MACD algorithm which measures if a stock is bullish or bearish. These terms constitute to if a

stock behaves in a way that its price will increase or decrease respectively. The MACD (Moving Average Convergence Divergence) aims to calculate the rate of exponential decay and increase and is closely linked to the Exponential Moving Average (EMA) to solve this. The main equation that we use to provide our MACD algorithm is:

12 Period EMA – 26 Period EMA

As a result, as we calculate the MACD value, we use the formula stated to produce our EMA values using:

(Current Price * α) + (Current EMA value * (1 - α))

Where α is the **smooth constant** and is calculated by:

$$\frac{2}{1+n}$$

Where n is the number of days in the period.

Furthermore, to this, to calculate the percentage confidence of an outcome of buy or sell, I will use the bullish and bearish constants of bullish being 0.1 and bearish being -0.1.

In order to generate my EMA value, I will use a static function FindExpotentialMovingAverage() accompanied with the smooth constant which is calculated within the constructor of the class. We also calculate the Simple Moving Average which the average of the sum of stock prices over the interval.

```
STATIC FUNCTION FindExpotentialMovingAverage(todayPrice, currentEMA)
return (todayPrice * smoothConst) + (currentEMA * (1 - smoothConst))
```

This function takes in parameters and outputs the new EMA value after the previous period. As a result, we a use this now within the EMAPrices method to calculate the Final EMA value:

```
STATIC FUNCTION EMAPrices(List<Values> Prices, Interval)
double EMAval = FindSimpleMovingAverage(Prices)
FOR i → Interval TO Prices.Length
      EMAVal = FindExpotentialMovingAverage(Prices[i].GetClose(), EMAval)

ENDFOR
RETURN EMAval
```

Once we have calculated the EMA value for both the 12th period and 26th period, we can utilise the MACD algorithm to receive our result.

In terms of representing if a stock is a buy or a sell, I decided to use class enumeration in the form of a Decision which allowed me to create a new data type that would signify a buy, sell or a hold signal. This made it far easier to view and maintain the code than signifying a signal on numbers of Boolean expressions.

This technique has been used in my method to show if a stock should be bought or sold:

```
Enum Decision:
Buy,
Sell,
Hold
```

This is useful so when I signify a buy or sell, I can use it below:

```
FUNCTION Decision BuyOrSell(List<Values> Prices)
double MACD = GetMACDvalues(Prices) // Returns MACD value using equation

IF (MACD > 0)
      RETURN Decision.Buy
ELSE IF (MACD < 0)
      RETURN Decision.Sell
ELSE
      RETURN Decision.Hold
```

In accompany with this, to find our percentage confidence, I will utilise those constants:

```
FUNCTION FindPercentageConfidence(double MACD)

IF(MACD > 0)
      double distance = Math.Absolute(MACD - bullishthreshold)
      return distance
ELSE
      double distance = Math.Absolute(MACD - bearishthreshold)
      return distance
```

This will find the absolute distance from the MACD value to the bullish/bearish threshold which would constitute to the percentage of confidence.

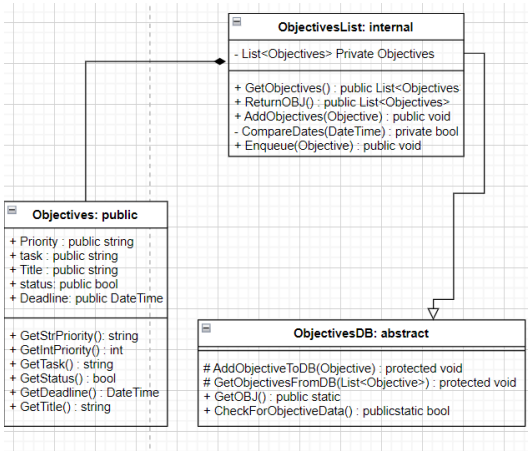
<u>Explanation of Objective Lists using Priority Queues (Scheduling method)</u>

Whilst the user goes along with the investment journey, they will be able to make simple financial objectives that they will set themselves and will aim to complete depending on both the priority of the objective and the deadline they have set themselves. As a result, when they do add a new objective, a use of a priority queue will be used in displaying the objectives but also acting as a way of displaying those objectives that are more important or have a closer deadline at the top.

As a result, I will implement two classes:

1) Objective
2) ObjectivesDB

The **Objective** class will be used for creating an instance of an objective to which it will be fed to the Objective List to be sorted and ordered to be in the correct position. It will act as a sub-class to the Objective List.



As we can see, I have separated both the database and queue class as a way of separating the front end and backend. To improve my design, I intend to make most of my database classes abstract allowing them to be used to provide a common set of functionality and behaviour for the **ObjectiveList** class to inherit the methods.

Each Objective will have a priority which will be the determining factor on where the objective will be placed in the list. The Task will hold the information the user has entered. The status will be used to as an indicator for the database to if it should be there. As a result, when the user removes one of the objectives, the status will be changed indicating that it should no longer there.

In terms of ordering the objectives, this will all be down the Objective List which will do checks to determine where an objective will be placed. This is specifically down to the Enqueue Method.

Within this class, a new List instance of type objective will be made:

```
List<Objective> objectivesList
```

Pseudocode for Enqueue Method:

```
FUNCTION Enqueue (Objective Objective)

Placement = 0
CorrectPosition = false

WHILE Placement < objectiveList.Length AND CorrectPosition == false


 If Objective.GetPriority() > ObjectiveList[placement].GetPriority() THEN
      CorrectPosition = true

 Else if Objective.GetPriority() == ObjectiveList[placement].GetPriority() THEN
     if CompareDates(Objective.GetDeadline(), ObjectiveList[placement].GetDeadline()) == true THEN
        CorrectPosition = true
     else
        CorrectPosition = false
        Placement = Placement + 1
     ENDIF

 Else
    CorrectPosition = false
    Placement = Placement + 1

 ENDIF

 ENDWHILE

 INSERT INTO OBJECTIVES, Objective AT POSITION CorrectPosition
```

The enqueue method works by retrieving the priority of the objective to be inserted and compare it with the priority of the items already in the priority queue. However, if the priority of the objective to be inserted is equal to the priority of an item in the objectives list, it will then compare the dates and check which one has an earlier deadline.

```
COMPARE DATES:

FUNCTION CompareDates (DateTime date1, DateTime date2)

{
       if date1 has a deadline earlier than date2 THEN
          return true
       else
          return false

}
```

Once a user makes an objective, for long-term storage, each objective will be stored onto a database so each time the application is ran, it will automatically be reloaded onto the application.

To accomplish the problem, I will implement a relational database which will connect each objective and a list of objectives.

Objective Database:

Field	Key	Data Type	Validation
ObjectiveID	Primary	integer	4
Title		Text	<50 characters
Description		Text	
Priority		String	
IsComplete		Boolean	True/False
Deadline		Date	
UserID	Foreign	integer	5

Within the **Objective** Database, the UserID field will act as the foreign key so when a user loads up their objectives, I will make a parametrised query holding the UserID which will be held in a static property so only their objectives are displayed.

The justification of utilising a priority queue for this process is its dynamic ability to shift around the objectives as soon as one objective has been either removed or its contents have been edited – a sorting algorithm won't be needed to reorder as it is having been ordered automatically. It also plays a great role as a scheduling algorithm in terms of structuring the user's objectives as since we have both the priority of each objective and deadline to compare as a compromise to if the priorities of two objectives are the same, this will allow for a more efficient method to order the objectives.

Financial Transactions

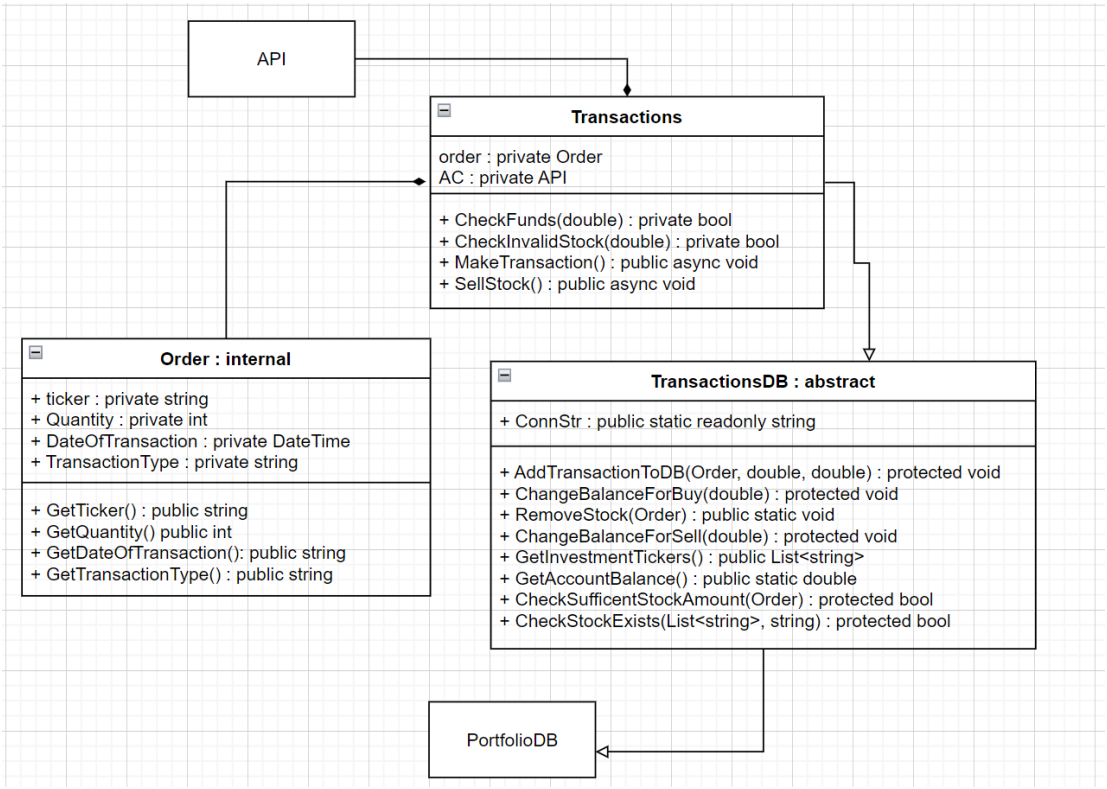
One of the core components of my trading application will be the financial purchases a user makes on given stocks that they believe will increase their profit.

For a transaction to take place, I aim to implement two classes:

1) Transactions
2) Order

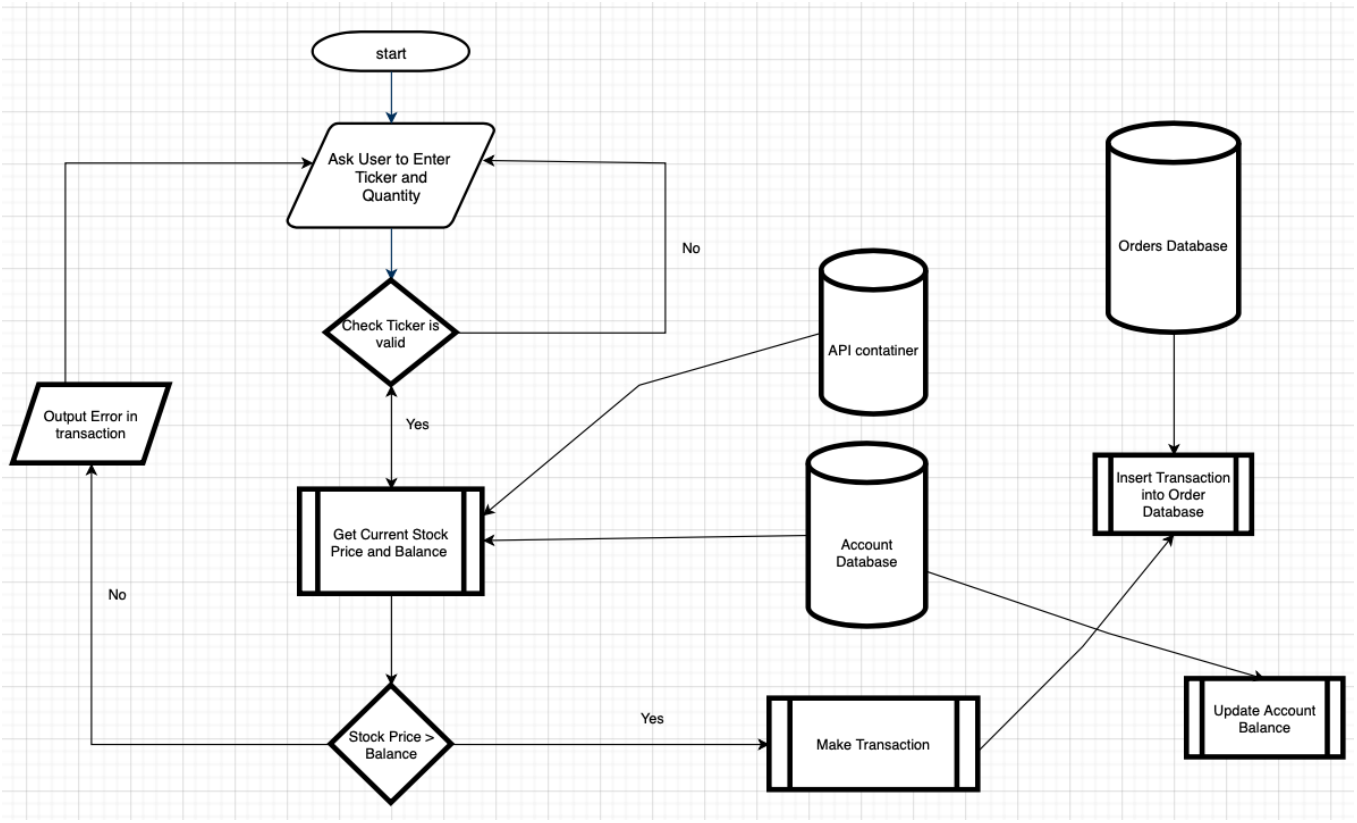
For each transaction, an instance of both Order and Account will be made to provide for the purchase.

Below shows the class diagram of the process:



As we can see through the class diagram, I have used **composition** to the fact that the Transactions class contains instances of both the API and Order as they are two of the core components in terms of making a financial transaction as I need the API to connect to the NYSE stock exchange prices and an Order instance to be there to retrieve the user's personalised order choice. Both these instances with undergo instantiation within the constructor of the Transactions class.

Flowchart of a Financial Buy Transaction



NOTE: Ticker is a stock symbol. E.g. AAPL (Apple), MSFT (Microsoft) etc.

When a user sells a given stock, there are more checks needed to make sure the transaction can be completed. As a result, a few more finalisations from the database are needed to ensure a smooth process.

This method takes in an instance of API (api) and Order (order) which is linked to the instance of the order made by the user. This utilises composition within the class

Pseudocode for selling stock:

```
FUNCTION ASYNC SellStock()
PriceOfStock = api.GetCurrentPrice(order.GetTicker())
if(CheckInvalidStock(PriceOfStock) == FALSE AND order.GetQuantity > 0)
       if(CheckStockExists(GetInvestmentTickers(), order.GetTicker()) == TRUE)
              if(CheckSufficientStockAmount(order) == TRUE)
                            PriceOfPurchase = PriceOfStock * order.GetQuantity()
                            AddTransactionToDB(order, PriceOfStock, PriceOfPurchase)
                            UpdatePortfolio(order, PriceOfPurchase, TRUE)
                     ENDIF
           ENDIF

ENDIF
```

Through this method, the program is first retrieving the Price of the stock given the ticker from the order object, it is first then checking if that stock is valid, and that the quantity entered in by the user is greater than 0. It then checks if the stock they have entered exists within the user's portfolio. This again links with the TransactionDB class to retrieve the user's stock which in turn plays a part in recursively going through each stock to check if a stock exists. Finally, it then checks that if the stock does exist in the user's portfolio that they have enough shares and this again links with the database.

The last three lines are all formulated through three other functions. These are:

1) ChangeBalanceForSell
2) AddTransactionToDB
3) UpdatePortfolio

ChangeBalanceForSell Function

For this function, we are simply changing the balance of the user account when we make a 'sell' transaction. We can use the following SQL query:

UPDATE Account SET Balance = GetAccountBalance() + transactionprice WHERE AccountID = {USERID}

Through this Query, I am using parametrised SQL to insert the data made within the front-end entered by the user and linking it to the backend for the database

NOTE. The GetAccountBalance() method will retrieve the account balance.

AddTransactionToDB Function

For this function, we are inputting the transaction within the orders database. We can do this using the following SQL query:

INSERT INTO Orders(OrderID, Ticker, Quantity, PriceAtTransaction, TransactionDate, TransactionPrice, TransactionType) VALUES({order.GetOrderID()}, {order.GetTicker()}, {order.GetQuantity()}, {PriceAtTransaction},{order.GetDateOfTransaction()} {TransactionPrice}, {order.GetTransactionType()})

For the Update Portfolio Function, there will have to be more checks undertaken by firstly checking the type of transaction to if it is a 'buy' or 'sell' meaning that the users' portfolios will therefore behave differently. Furthermore, depending on if the user has a certain stock in their portfolio, it will update the portfolio differently.

Method CheckStockExists ()

```
FUNCTION CheckStockExists (List<string> stocks, string TickerToSearch)
{
    If length(stocks) == 0
        return false
    else if stock [0] == TickerToSearch
        return true
    else
        stocks.Remove(0)
        return CheckStockExists(stocks, TickerToSearch)
}
```

This method utilises a **recursive** search which retrieves the user's invested stocks and the ticker to search. This will make it far more efficient as especially if the user chooses to invest in a lot of stocks in which using a recursive search will be suitable. The method is primarily used to check if a stock already exists in a user portfolio so if it does, then we can update the already existing record of that stock in the Portfolio database and if not, we should create a brand-new record existing of that stock. We can also see that I have utilised the list operation so that if the stock we are looking to search is not at the given index, we will remove it from the list and continue to the next one.

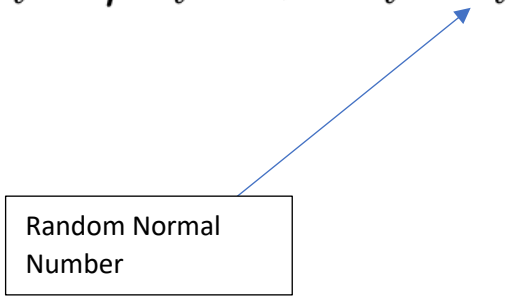
Geometric Brownian Motion Simulation

Brownian Motion is a stochastic process meaning that the process of the simulation is affected by a set of randomly generated numbers.

In this case, we are dealing with random standard normal numbers that will affect the way each simulation goes under a certain day.

Recalling back to the analysis stage where a display of the stochastic differential equation was in the form:

$$dS_t = \mu S_t\,dt + \sigma S_t\,dW_t$$



This number itself follows the concept of the Wiener Process which is heavily used when dealing with continuous time martingales (sequence of random variables).

One of the main reasons why this is so integral is the stock price movement is very much random and is affected by a plethora of varied factors whether it is financial shifts or political influences. As a result, the randomness of the process allows for a far more accurate result of the simulation.

The two classes which are integral to this process is:

1) Generating Random Normal Numbers
2) Simulation Process

The entire process is covered over a two-dimensional array which holds the Number of simulation days within the x-matrix (rows) and the number of simulations in the y-matrix (columns).

Method of generating random numbers

When generating the random number table, I will utilise a method called the Box-Muller Transform which utilises two uniform random numbers and transforms it into a standard normal number. This can be shown through:

```
STATIC FUNCTION GenerateRandomNum (double u1, double u2)
Return SQRT (-2.0 * naturalLog * (1 - u1)) * Sin (2.0 * PI * (1 - u2))
```

The return statement provides the Box-Muller transform which generates these random numbers.

The role of this method is to provide the standard normal number which can be later applied to the set of stock prices.

We can then utilise this static method for the generation of the normal numbers. This will use a function where we iterate through each index in the 2d array.

```
FUNCTION RandomNumbersGenerator()

Double [,] RandomArray = new Double[SimDays, NumOfSims]
FOR i → 0 to NumberOfSims
      FOR j → 0 to SimulationDays
          double RandVal1 = 1.0 - RandomVal.GenerateUniformNumber()
          double RandVal2 = 1.0 - RandomVal.GenerateUniformNumber()

          RandomArray[i-1 , j-1]  = GenerateRandomNum(RandVal1, RandVal2)
return RandomArray
```

Once we have generated the 2d array containing the standard normal random numbers, we can then apply it to the prices for simulation. The function below presents how we will use the differential equation to model the stock price:

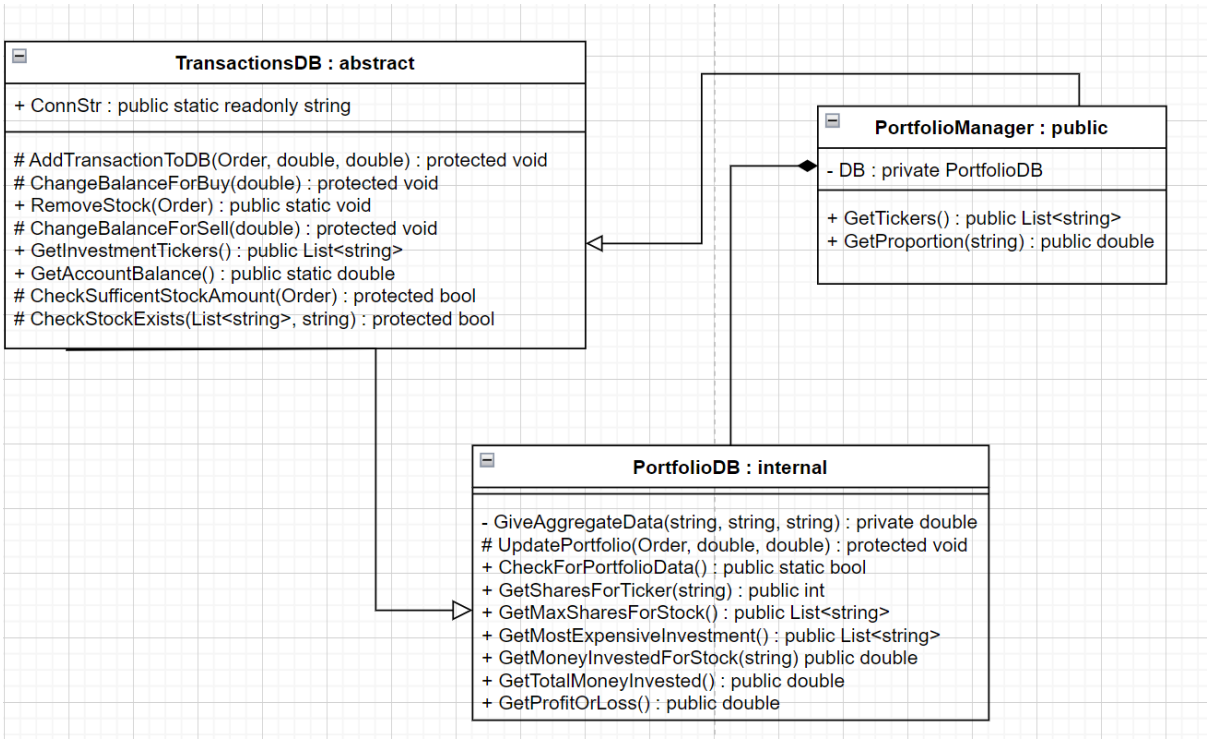
```
FUNCTION SimulateGBM()
FOR sim → 0 to SimulationNum
     FOR day → 0 to SimulationDays
          Simulated[day, sim] = SimulatedStockPrices[day-1,sim]
          + SimulatedStockPrices[day-1, sim] * Mu +
          SimulatedStockPrices[day-1, sim] * Sigma *
          RandomNumberArray[day-1, sim]
```

Where the Mu and Sigma are the annualised drift and annualised volatility constants respectively. We can see how we have applied the random process differing the behaviour on the stock price each time.

Role of user portfolio and its link to financial transactions

By far the most integral part of my program is to keep a track and record the user's investments as without these components, the user would not be able to see the performance of their trades. This part of my program heavily relies on the database link and gathering information from the Portfolio and Orders tables to calculate statistical analysis of a portfolio.

Below presents my class diagram for portfolio processes:



Through my class diagram, I proposed that both the classes of portfolio and transactions would be linked together. Within each transaction, there must be updates to the portfolio as well changing the number of shares or money invested.

Within my PortfolioDB class, I have implemented methods that provide a link between the front and back end with the calculations.

I have used composition as I create an instance of PortfolioDB in the PortfolioManager.

The GiveAggregateData() provides an easy cohesive way to return any aggregate value given parameters such as table, field and function:

```
PRIVATE FUNCTION GiveAggregateData (string Functiontype, string table, string column)
string Query = $"SELECT {FunctionType}({column}) FROM {table} NOLOCK WHERE userID = {UserCredsDB.UserID}";
double result = Query.ExecuteQuery()

Return result
```

The basis of this function is to provide aggregate data but as a single sealed function so that when I want to find the value using functions such as 'Max' and 'Min', It will be easier to retrieve it.

This can be showed within my GetMaxSharesForStock() which calculates the stock which a user has invested the most shares in.

The query that will complete this is:

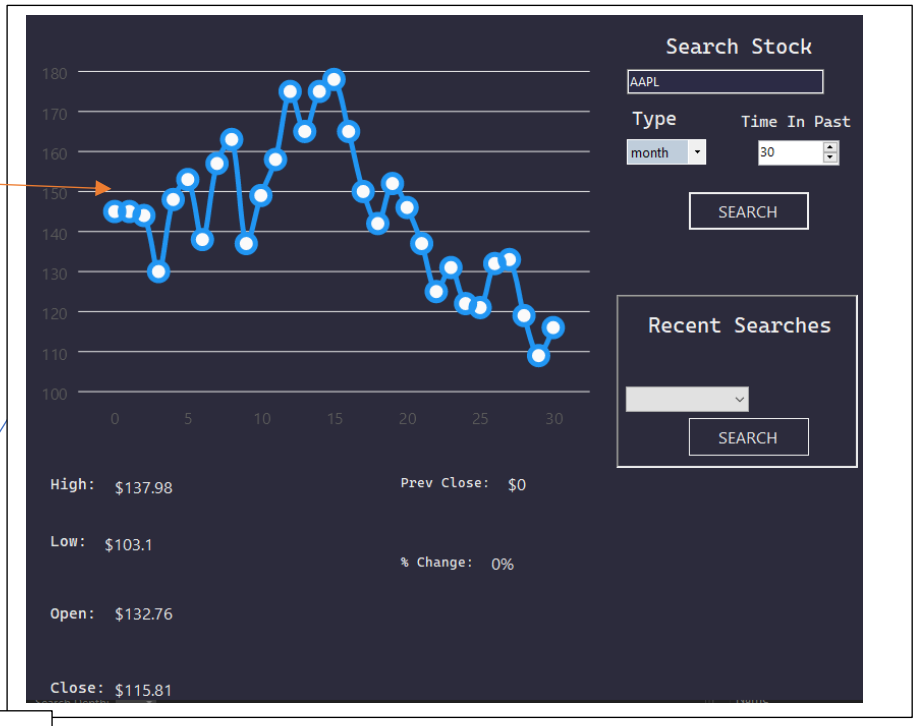
```
SELECT portfolio.Ticker, portfolio.NumberOfShares FROM Portfolio portfolio JOIN USERS user ON portfolio.userID
= user.AccountID WHERE portfolio.NumberOfShares = {GiveAggregateData("MAX", "Portfolio", "NumberOfShares")
```

I have used cross-table parametrised SQL to join both the portfolio and user table and used the GiveAggregateData () function to provide the constraint value to the maximum number of shares.

Another instance of using both cross-table parameterised SQL is to calculate the profit of a user's portfolio as it connects both the Order and Portfolio tables. A query below would be able to accomplish this:

```
SELECT SUM(TransactionPrice - MoneyInvested) FROM Portfolio portfolio JOIN Orders orders ON
portfolio.Ticker = orders.Ticker WHERE portfolio.userID = {UserCredsDB.UserID} AND
orders.TransactionType = 'BUY'";
```

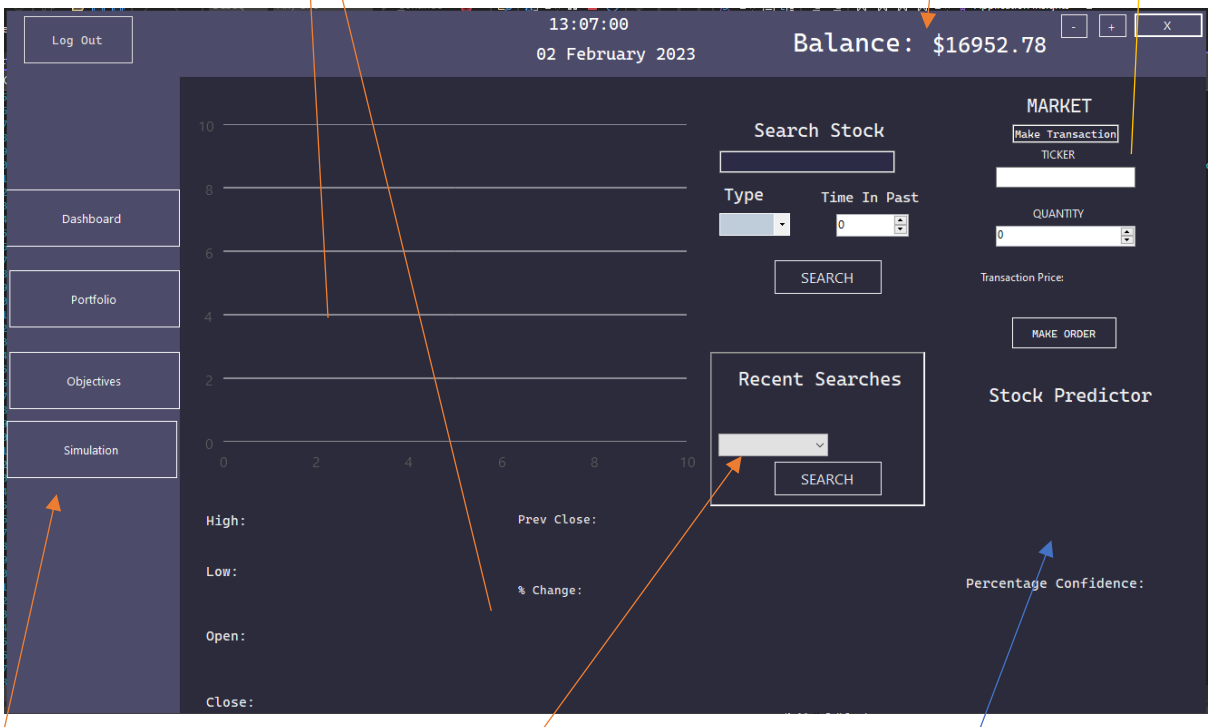
Through the query, I have utilised the SUM function to calculate the total between all records of all the logged in user investments' and adding a constraint in which the transaction type was a buy. This will provide the user with profit of their portfolio.



This graph utilises the LiveCharts 2 external library allowing an easy use of adding new data points and making it clear to see.

Whenever a stock is searched, data will be represented via graphical interpretation and via stating the overview of the stock.

Users can view their balance remaining making it easier to make decisions when buying stocks.
Users can also make financial transactions by selected any stock from the NYSE and select how much they want to buy



Side panel allows users to easily traverse between

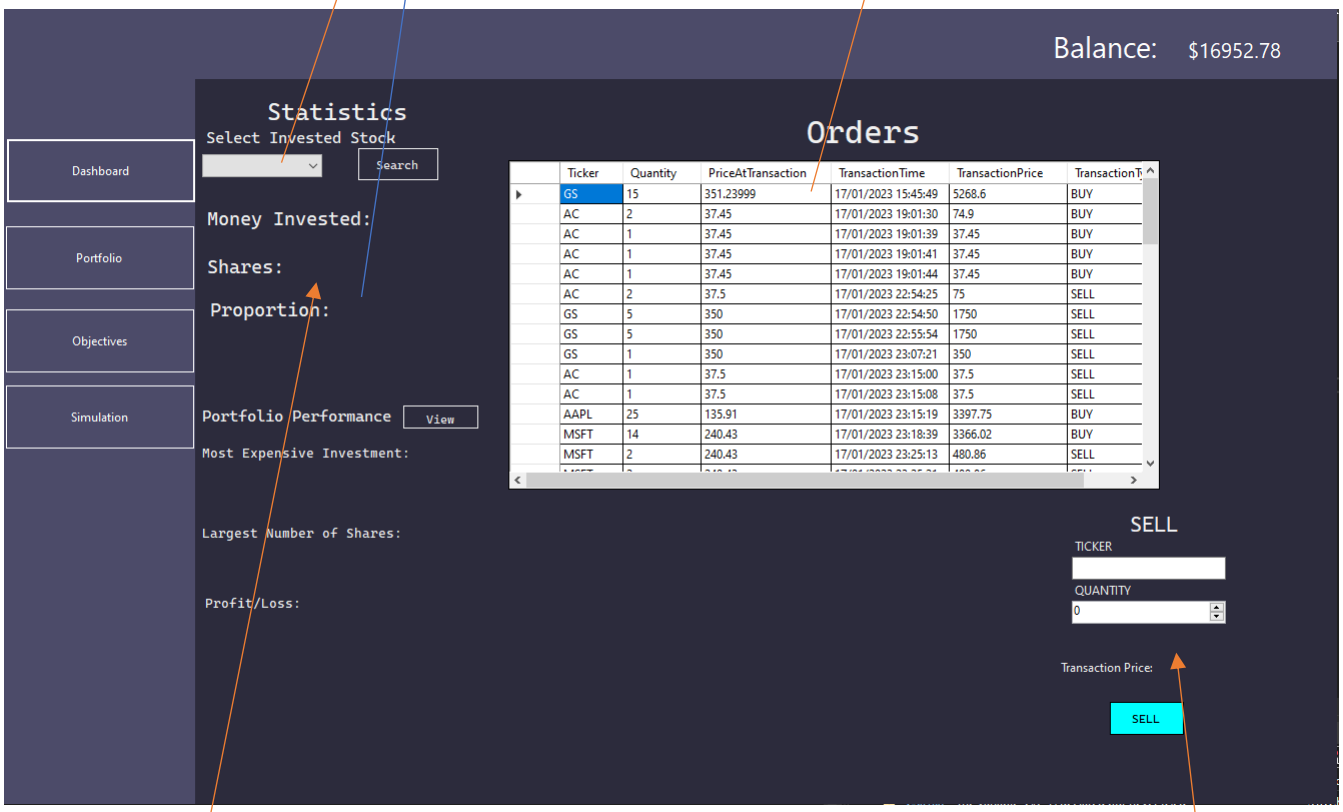
Drop down menu presents all the user's search history of previous stocks.

Once a user has selected a stock, the MACD algorithm will be called to measure if a stock should be bought or sold. Additionally, the percentage confidence of this will be

Portfolio Viewer

Drop down menu allows users to selected

User can view their own orders every time they make transaction

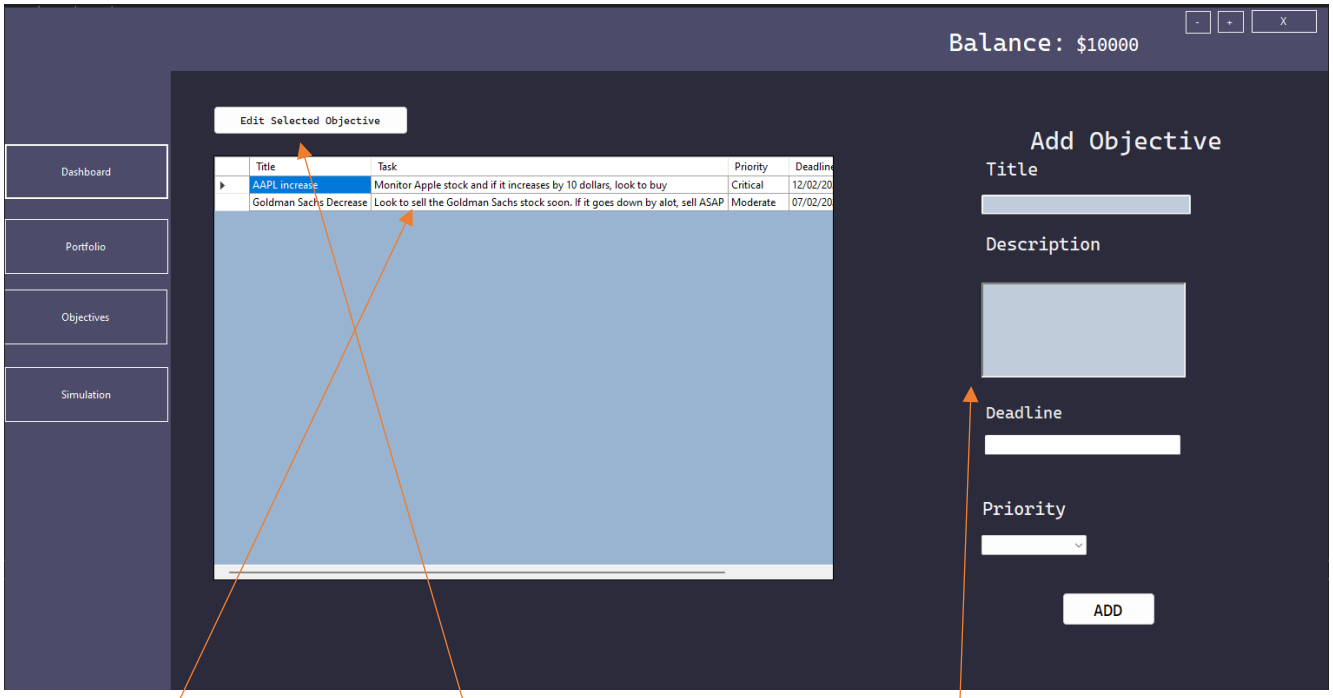


Once stock has been selected and searched, it will give statistics of that stock such as share numbers and proportion of stock within portfolio.

Users can sell the stock they have purchased again stating how much they want to sell.

Panel showing the portfolio statistics such as profit of user's wallet and their most expensive investment.

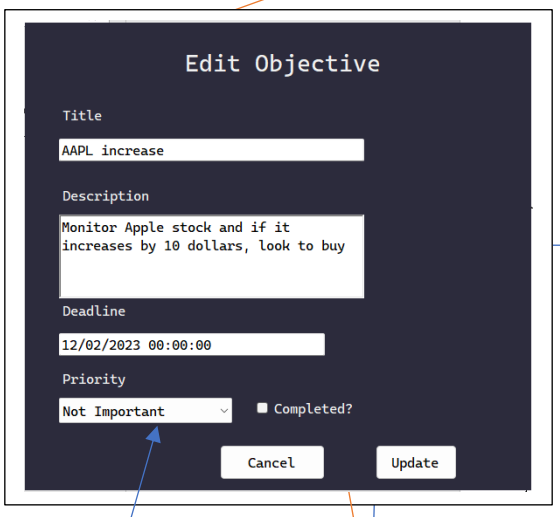
Objectives To Do List

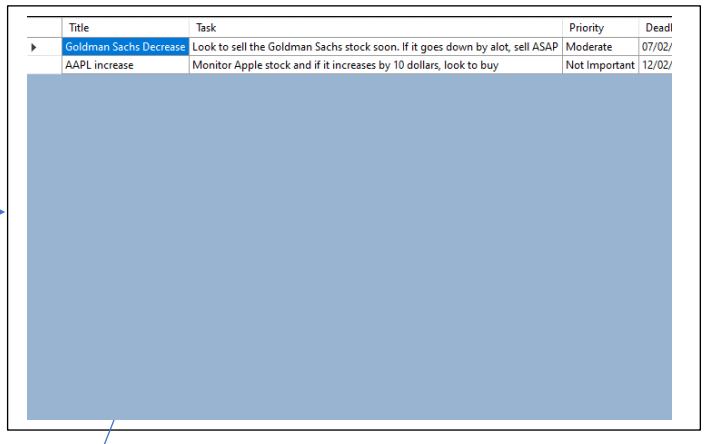


User can view any objectives they have made in the to-do list. We can see that the list has been ordered from most important to least important in priority.

User can add the objectives they want specifying the title, description, deadline and most importantly the priority. The priority selected will be a factor changing the order of the to do list

Users can select and click edit button to change the objective they want.





We can see that the Apple objective has been changed from critical to not important

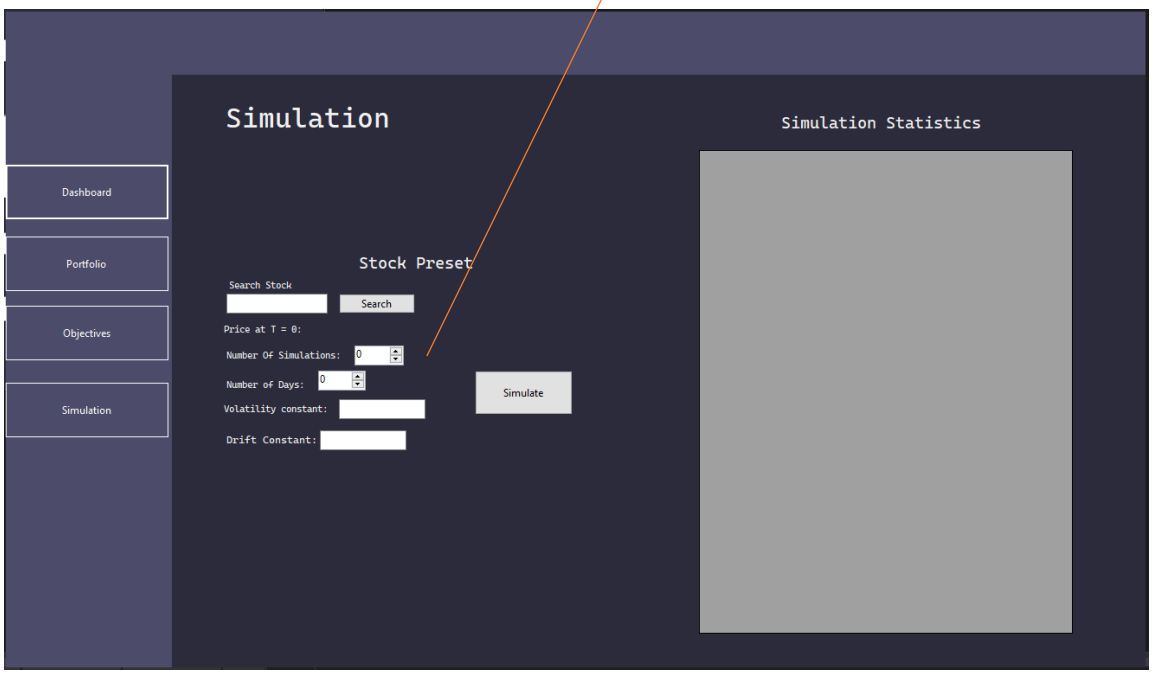
Users can change the contents of their objective or change the priority of the objective.

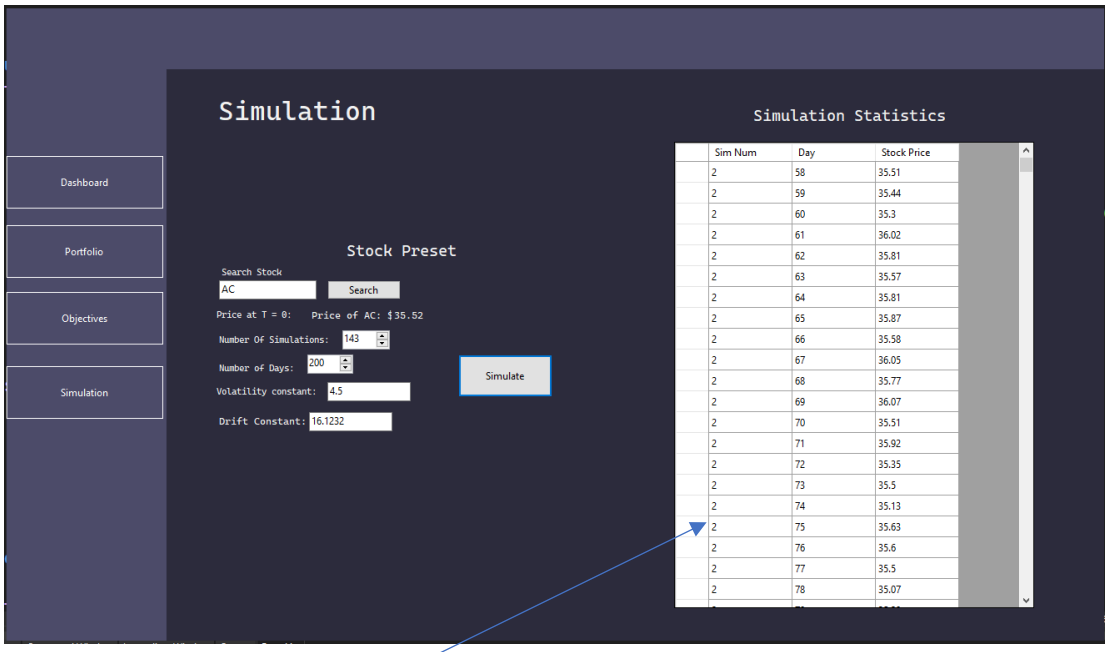
The Apple objective has now moved to the bottom since its priority is lower than the other objective

Geometric Brownian Motion Simulation

Fields user can enter to simulate and model stock prices. This includes volatility and drift which will affect how the stock prices differ.

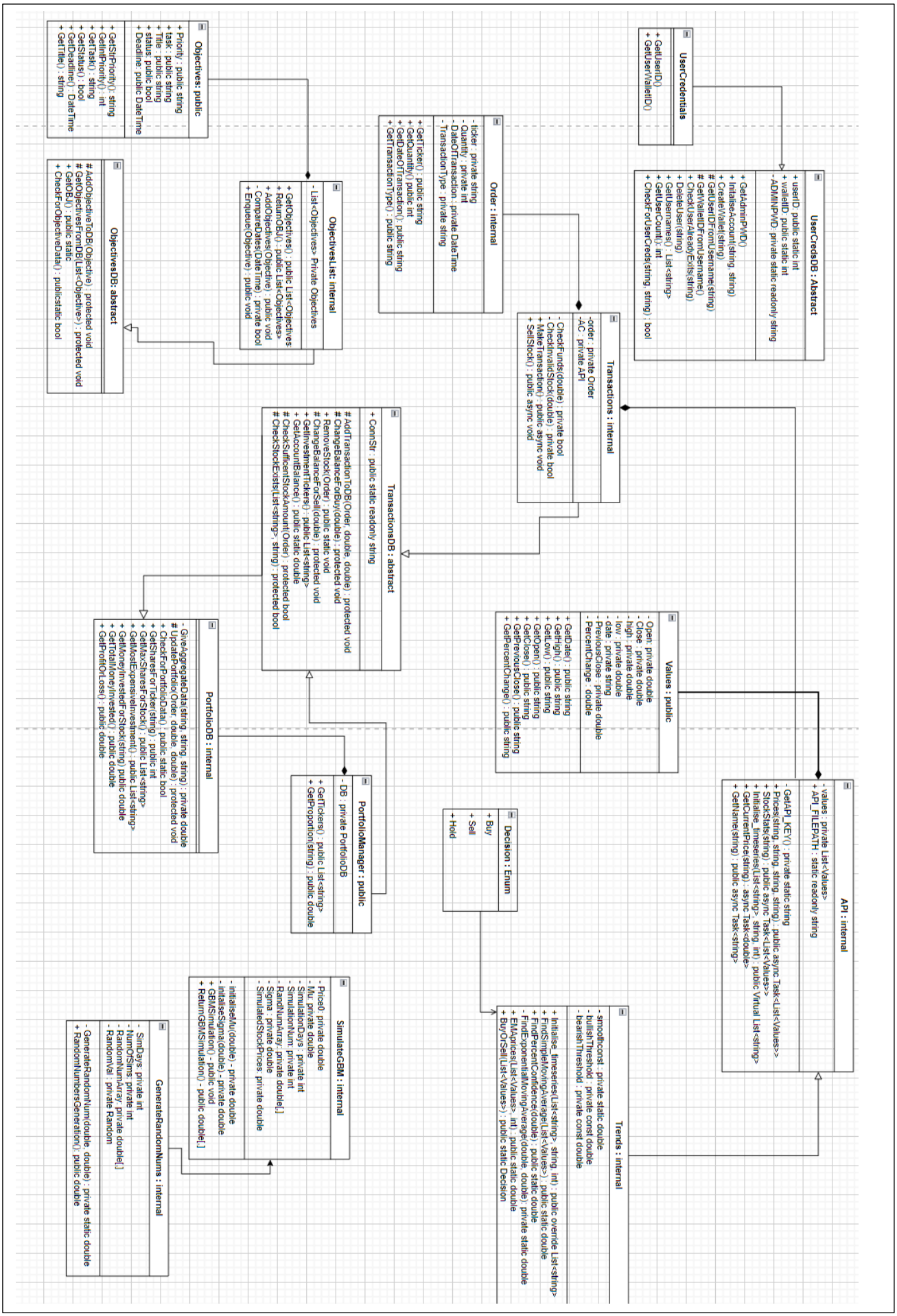
Users can enter a certain stock from the NYSE via the API.





From simulation results, we can see the results from the simulation stating the number simulation, day, and price at the time.

Program UML Diagram



My program involves both a login system including both a sign up and sign in process. Furthermore, it also includes an admin system that allows the admin the view other user's data and make actions such as viewing their orders, portfolio's and even deleting certain users.

One of the main aspects for maximising security was implementing a hashing process that would collect the input password and hash the string when storing it within the database.
Below presents the hashing process I used:

FUNCTION HashPassword(string Password)
Byte[] PasswordStringBytes = EncodePassowordIntoBytes(Password)
Byte[] HashedBytes = Hash.ComputeHash(PasswordStringBytes)

RETURN Convert.ToBaseString(HashedBytes)

Through this process, it allowed a level of anonymity for the user for their passwords and within the table of USERS, it will prevent passwords from being exposed providing a defensive aspect to the program.

To provide a completeness to my program, I also implemented an admin section which provides a user to observe other users' portfolio and order history. This provides a sense of separation of responsibility between the user's and provides the program with a more complete aspect in its development.

Within it, it will allow the admin to observe how many users are within the account through SQL queries such as:

SELECT COUNT(Username) FROM USERS

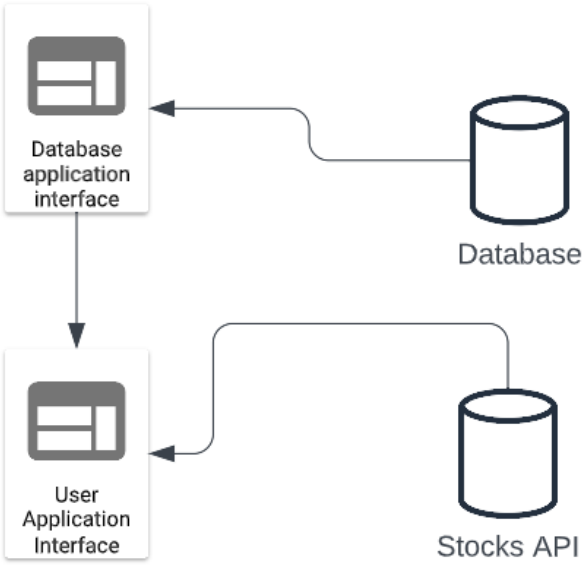
Which will give the admin a clear indication of the number of users operating in the application.

Technical Solution

Within my project, I have utilised C# as a language with that aid of Microsoft Visual Studio as an IDE to support my program. Related to my user interface, I have used WinForms to provide a good foundation to allow users to easily traverse around the system. With my program being heavily linked with database usage in terms of handling user portfolios and transactions, I have utilised SQLite for the querying and SQLite Expert Personal to provide a platform to observe and track how the storage of the data is organised.

With real stock data being a very core component of my program, having access the TwelveData API allowed me to gather the stock data aiding me in allowing financial transactions to go ahead and provide an accurate representation for providing analysis for various stocks and their trends. Linked with my API, I utilised the NewtonSoft external library in aid for the JSON parsing and allowing easy collection and allocation of the data.

Below presents a System Diagram displaying how my program will be structured and the various connections between each component.



NOTE. Classes highlighted in yellow are related to the Windows Form autogenerated classes for the design of the UI. However, code is still present that is used to provide the functionality for the program.

Contents:

API class

```
internal class API
    {

        private List<Values> values;
        private static readonly string API_FILEPATH = "C:\\Users\\kumar6972\\OneDrive - Bradford
Grammar School\\Home Drive\\computer science\\Year 13 - Year 1\\NEA 10JAN\\testtttt\\NEA -
19DEC\\NEA\\API_KEY.txt";

        public API()
        {
            values = new List<Values>();

        }
        #region APICODE

        private static string GetAPI_KEY()
        { // Retrieves API key from text file referenced in the filepath.
            try
            {
                using (StreamReader reader = new StreamReader(API_FILEPATH, true))
                {

                    string path = reader.ReadLine();
                    return path;

                }

            }
            catch (Exception exception)
            {
                throw exception;

            }

        }

        public  async Task<List<Values>> Prices(string ticker, string startdate, string enddate,
string interval)
            // Used async methods and a task to utltise a separate thread just for the API
response - provides quicker interface
        {
            HttpClient client = new HttpClient(); // Creates new client to link with web API
            var request = new HttpRequestMessage
            {
                Method = HttpMethod.Get,
                RequestUri = new
Uri($"https://api.twelvedata.com/time_series?symbol={ticker}&start_date={startdate}&end_date={en
ddate}&interval={interval}&apikey={API.GetAPI_KEY()}&format=JSON"),
                // sends API request to retrieve stock data from API
            };
            using (var response = await client.SendAsync(request))
            {
                try
                {
                    response.EnsureSuccessStatusCode(); // checks if API request has sent data
successfully
                    string JSON_response = await response.Content.ReadAsStringAsync(); // reads
data as an asynchronous string

                    JObject jobject = JObject.Parse(JSON_response); // Parses the string into
JSON format for easy access of data.
                    // TEST IF ARRAY EMPTY

                    foreach (var val in jobject["values"])
                    {
```

```
                        values.Add(new Values(Convert.ToString(val["datetime"]),
Convert.ToDouble(val["high"]), Convert.ToDouble(val["low"]), Convert.ToDouble(val["open"]),
Convert.ToDouble(val["close"]), Convert.ToDouble(val["previous_close"]),
Convert.ToDouble(val["percent_change"])));
                        // Adds each data point over a data into the values list
                    }
                    return values;
                }
                catch (Exception e)
                {
                    MessageBox.Show("Invalid Ticker", "Error", MessageBoxButtons.OKCancel);
                    return null;
                    // Exception handling to check if user has entered a valid Stock ticker.


                }

            }
        }


        public async Task<List<Values>> StockStats(string ticker)
        {
            // Retrieves stock data at current day and time to be displayed on dashboard to
user. Again using async methods and Task threading to improve user experience.
            try
            {
                HttpClient client = new HttpClient();
                var request = new HttpRequestMessage
                {
                    Method = HttpMethod.Get,
                    RequestUri = new
Uri($"https://api.twelvedata.com/quote?symbol={ticker}&apikey={API.GetAPI_KEY()}"),

                };
                using (var response = await client.SendAsync(request))
                {
                    response.EnsureSuccessStatusCode();
                    var JSON_response = await response.Content.ReadAsStringAsync();
                    JObject jobject = JObject.Parse(JSON_response);

values.Add(new Values(Convert.ToString(jobject["datetime"]), Convert.ToDouble(jobject["high"]),
Convert.ToDouble(jobject["low"]), Convert.ToDouble(jobject["open"]),
Convert.ToDouble(jobject["close"]), Convert.ToDouble(jobject["previous_close"]),
Convert.ToDouble(jobject["percent_change"])));


                    return values;
                }

            }
            catch (Exception exc)
            {
                MessageBox.Show("Unable to Retrieve Stock Stats", "error",
MessageBoxButtons.OKCancel);
                return values = null;
            }


        }




public virtual List<string> Initalise_timeseries(List<string> timeseries, string intervaltype,
int option)
            // Method Virtual to connect with Trends class to override method to create
different time range.
```

```
        {
            // Creates a set of two dates to create a start and end to retrieve stock data from
specified dates. Adds dates to
            string dateTime = DateTime.Now.ToString();
            string now = Convert.ToDateTime(dateTime).ToString("yyyy-MM-dd");
            if (intervaltype == "month")
            {
              var minustime = DateTime.Today.AddMonths(-option);
              string time = Convert.ToDateTime(minustime).ToString("yyyy-MM-dd");
              timeseries.Add(now);
              timeseries.Add(time);
              return timeseries;
            }

            else if (intervaltype == "day")
            {
              var minustime = DateTime.Today.AddDays(-option);
              string time = Convert.ToDateTime(minustime).ToString("yyyy-MM-dd");
              timeseries.Add(now);
              timeseries.Add(time);
              return timeseries;
            }
            else
            {
                return null;

            }


        }

        public async Task<double> GetCurrentPrice(string ticker)
        {
            // Gets current price of a ticker which is used in the buying and selling of stocks
and used in the Geometric Brownian Motion of stocks.
            HttpClient client = new HttpClient();
            var request = new HttpRequestMessage
            {
                Method = HttpMethod.Get,
                RequestUri = new
Uri($"https://api.twelvedata.com/price?symbol={ticker}&apikey={API.GetAPI_KEY()}&format=JSON"),

            };
            using (var response = await client.SendAsync(request))
            {
                response.EnsureSuccessStatusCode();
                var body = await response.Content.ReadAsStringAsync();
                JObject jobject = JObject.Parse(body);
                double Current_price = Convert.ToDouble(jobject["price"]);
                return Current_price;
            }

        }


public  async Task<string> Get_Name(string ticker)
        {
            // Retrieves Full name of company given the ticker

            HttpClient client = new HttpClient();
            var request = new HttpRequestMessage
            {
                Method = HttpMethod.Get,
                RequestUri = new
Uri($"https://api.twelvedata.com/quote?symbol={ticker}&apikey={API.GetAPI_KEY()}&format=JSON"),

            };
            using (var response = await client.SendAsync(request))
            {
                response.EnsureSuccessStatusCode();
```

```
                var body = await response.Content.ReadAsStringAsync();
                JObject jobject = JObject.Parse(body); // Parses JSON
                string FullName = Convert.ToString(jobject["name"]);
                return FullName;
            }

        }
        #endregion
    }
```

Values class

```
public class Values
    {
        private string date;
        private double high;
        private double low;
        private double open;
        private double close;
        private double PreviousClose;
        private double PercentChange;

        public Values(string date, double high, double low, double open, double close, double
PreviousClose, double PercentChange)
        {
            this.date = date;
            this.high = high;
            this.low = low;
            this.open = open;
            this.close = close;
            this.PercentChange = PercentChange;
            this.PreviousClose = PreviousClose;

        }
        public Values()
        {



        }
// Using encapsulation to keep the properties private but created public methods to retrieve
information
        public string GetDate()
        {
            return date;


        }
        public double GetHigh()
        {
            return high;

        }
        public double GetLow()
        {
            return low;

        }
        public double GetOpen()
        {
            return open;

        }
        public double GetClose()
        {
            return close;
```

```
        }

        public double GetPreviousClose()
        {
            return PreviousClose;

        }

        public double GetPercentChange()
        {
            return PercentChange;

        }

    }
```

SearchHistory Class

```
internal class SearchHistory
    {
        private List<string> History;

        public SearchHistory()
        {
            History = new List<string>();

        }

        public List<string> ShowHistory()
        {
            List<string> items = new List<string>();
            foreach (var item in History)
            {
                items.Add(item); // Adds a new stock into the search history

            }

            return items;

        }

        public void AddSearch(string ticker)
        { //Method used to add item to top of search history

            History.Insert(0,ticker); // Pushes item to the top of the stack. This is done once a search is added.

        }
}
```

Trends class

```
internal class Trends: API
    {


        private static double smoothconst;
        private const double bullishThreshold = 0.1; // Creates constants for both bullish and
bearish scenarios - used in comparing the confidence of buying or selling a stock.
        private const double bearishThreshold = -0.1;


        public Trends(List<Values> values)
        {
            smoothconst = 2 / (1 + values.Count);
        }
        #region TrendsCode
        public override List<string> Initalise_timeseries(List<string> timeseries, string
interval, int type)

            // Overriding method from API class to intitalise dates for API retrieval of data.
        {

            string dateTime = DateTime.Now.ToString();
            string now = Convert.ToDateTime(dateTime).ToString("yyyy-MM-dd");
            string minustime = DateTime.Today.AddDays(-1000).ToString();
            string time = Convert.ToDateTime(minustime).ToString("yyyy-MM-dd");
            timeseries.Add(now);
            timeseries.Add(time);
            return timeseries;

        }

        private static double FindSimpleMovingAverage(List<Values> prices)
        {
            // Method used to retrieve the first value for calculating the expotential moving
average of stock data
            // Used simple moving average algorithm
            double total = 0;
            for (int j = 0; j < prices.Count; j++)
            {
                total += prices[j].GetClose();

            }
            double avg = total / prices.Count;
            return avg;

        }
        public static double FindPercentConfidence(double MACD)
        {
            // Method of calculating the percent confidence of stock by calculating the absolute
distance the MACD value is from threshold value.
            // Threshold value differs depending on its trend status.
            if (MACD > 0)
            {
                double distance = Math.Abs(MACD - bullishThreshold);
                return distance;
            }
            else
            {
                double distance = Math.Abs(MACD - bearishThreshold);
                return distance;


            }

        }
```

```
        private static double FindExpotentialMovingAverage(double todayPrice, double currentEMA)
        {
            return (todayPrice * smoothconst) + (currentEMA * (1 - smoothconst));
            // Exponential Moving Average algorithm
        }

        public static double EMAPrices(List<Values> Prices, int interval)
        {
            // Method of looping through each price in the Prices list via API. Uses SMA value
as the beginning value.
            double EMAVal = FindSimpleMovingAverage(Prices);
            for (int i = interval; i < Prices.Count; i++)
            {
                // Changes the exponential moving average model depending on the previous value
of the EMA and current price at index i
                EMAVal = FindExpotentialMovingAverage(Prices[i].GetClose(), EMAVal);

            }
            return EMAVal;
        }


        public static Decision BuyOrSell(List<Values> Prices) // Create method of form
'Decision' linking with the enumeration
        {

            // MACD = 12 - 26
            double MACD = GetMACDval(Prices); // Retrieves MACD value through the equation of
calculating the EMA value of the 12 day period - 26 day period

            if (MACD > 0)
            {
                return Decision.Buy; // Uses MACD model to provide outcome of analysis

            }
            else if (MACD < 0)
            {
                return Decision.Sell;

            }
            else
            {
                return Decision.Hold;
            }

        }
        public static double GetMACDval(List<Values> prices)
        {
            return EMAPrices(prices, 12) - EMAPrices(prices, 26);

        }

        #endregion
    }


    public enum Decision // Use of class enumeration. Since there are three different outcomes
to a price prediction, an enum is suitable to provide a definive outcome of the analysis in a
readable way.
    {
        Buy,
        Sell,
        Hold
    }

    }
```

GenerateRandomNums class

```
internal class GenerateRandomNums
    {
        private int SimDays;
        private int NumOfSims;
        private double[,] RandomNumArray;
        private Random RandomVal;

        public GenerateRandomNums(int NumofSimulationDays, int NumofSimulations)
        {
            SimDays = NumofSimulationDays - 1;
            NumOfSims = NumofSimulations;
            RandomVal = new Random();

        }
        private static double GenerateRandomNum(double u1, double u2)
        {
            return Math.Sqrt(-2.0 * Math.Log(1 - u1)) * Math.Sin(2.0 * Math.PI * (1 - u2)); //
Box-Muller Transform
        }

        public double[,] RandomNumbersGeneration()
        {
            RandomNumArray = new double[SimDays, NumOfSims];
            for (int Column = 1; Column < NumOfSims; Column++)
            {
                for (int Row = 1; Row < SimDays; Row++)
                {
                    double randval1 = 1.0 - RandomVal.NextDouble(); // Generates two uniform
random numbers
                    double randval2 = 1.0 - RandomVal.NextDouble();

                    RandomNumArray[Row - 1, Column - 1] = GenerateRandomNum(randval1, randval2);
// Performs the Box-Muller transform on the two uniform random numbers and adds to array.
                }
            }

          return RandomNumArray;
        }

    }
```

SimulateGBM Class

```
internal class SimulateGBM
    {
        private double Price0;
        private double Mu;
        private int SimulationDays;
        private int SimulationNum;
        private double[,] RandNumArray;
        private double Sigma;
        private double[,] SimulatedStockPrices;

        private double initaliseMu(double mu)
        {
            return mu / 100 / 252; // annualises the drift

        }
        private double initialiseSigma(double voltality)
        {
            return voltality / 100 / Math.Sqrt(252); // annualises the volatility.

        }
        public SimulateGBM(double[,] RandArray, double PriceAt0, double drift, double volatlity)
        {
            SimulationDays = RandArray.GetLength(0); // Gets interval points for both the sim
days and number of sims
            SimulationNum = RandArray.GetLength(1);
            RandNumArray = RandArray; // Has new instance of the 2d array to be assigned to the
random array generated earlier.
            SimulatedStockPrices = new double[SimulationDays + 1, SimulationNum];
            this.Price0 = PriceAt0;
            Mu = initaliseMu(drift); // Initliased the drift and voltatlity constants entered by
users to be annualised for use
            Sigma = initialiseSigma(volatlity);

            for (int column = 1; column < SimulationNum+1; column++)
            {
                SimulatedStockPrices[0, column - 1] = Price0; // Sets the first simulation as
the initial price so each simulation starts on the same price specified by user.

            }

        }

          public double[,] returnGBMSimulation()
        {
            return SimulatedStockPrices;
        }

        public void GeometricBrownianMotion()
        {
            for (int sim = 0; sim < SimulationNum; sim++)
            {
                for (int day = 1; day <= SimulationDays; day++)
                {
                    SimulatedStockPrices[day, sim] = Math.Round(SimulatedStockPrices[day - 1,
sim] + SimulatedStockPrices[day - 1, sim] * Mu +SimulatedStockPrices[day - 1, sim] * Sigma
*RandNumArray[day - 1, sim], 2); // Stochastic diffrential process.
                }
            }

        }



    }
```

Hash class

```
static class Hash
    {
        private static byte[] hashstring; // creates property of hashstring of form bytes which
will store the hashed bytes.
        private static byte[] hashedbytes;

        public static string HashPassword(string password)
        {
            hashstring = Encoding.UTF8.GetBytes(password); // Gets the bytes via method
            using (SHA256 shahash = SHA256.Create())
            {
                hashedbytes = shahash.ComputeHash(hashstring); // Computes hash and stores bytes
hashedbytes object


            }
            return Convert.ToBase64String(hashedbytes); // Returns the hashed bytes property as
a base64 string

        }



     }
```

Objectives Class

```
public class Objective
    {
        public int ObjectiveID;
        public string Priority { get; set; } // 3 - Critical, 2 - Moderate, 1 - Not important
        public string task;
        public string Title;
        public bool Status;
        public DateTime Deadline;

        public Objective()
        {


        }

        public int GetObjID()
        {
            return ObjectiveID;
        }
        public string GetStrPriority()
        {
            return Priority;

        }
        public int GetIntPriority()
        {
            return GetImportance(Priority); // Retrieves the integer priority to be used in
priority queue method.

        }
        public string GetTask()
        {
            return task;

        }
        public bool GetStatus()
        {
            return Status;

        }

        public DateTime GetDeadline()
        {
            return Convert.ToDateTime(Deadline.ToString("dd/MM/yyyy"));
        }
        public string GetTitle()
        {
            return Title;

        }


        // Used to convert string priorities into values. A value of 3 has highest priority and
priority 1 has lowest.
        private int GetImportance(string Priority)
        {
            if (Priority == "Not Important")
            {
                return 1;


            }
            else if (Priority == "Moderate")
            {
                return 2;
```

```
            }
            else
            {
                return 3;

            }


        }

    }
```

ObjectivesList class

```
internal class ObjectiveList : ObjectiveDB // Again, Inheriting methods from the ObjectiveDB
class - separation of front-end and backend
    {
        private List<Objective> Objectives; // creates a list of objectives - class collection -
association


        public ObjectiveList()
        {
            Objectives = new List<Objective>();

        }

        public List<Objective> GetObjectives() // Retrieves Objectives from the Database.
        {
            return GetObjectivesFromDB();

        }
        public List<Objective> ReturnOBJ() // Returns the Objectives after being enqueued to the
priority queue and ordered.
        {
            return Objectives;
        }

        public void AddObjective(Objective obj)
        {
            AddObjectiveToDB(obj); // Adds new Objective to database to be ready for retrieval.

        }

        private bool CompareDates(DateTime t1, DateTime t2) // method used to compare if the
date of objective being added is earlier than others - used as a alternative check if priorities
of objectives are same.
        {
            if (DateTime.Compare(t1, t2) < 0)
            {
                return true;

            }
            else
            {
                return false;

            }
        }



 public void Enqueue(Objective objective) // priority scheduling method
        {
```

```
            int placement = 0; // initialises index of comparison to 0 - start of list
            bool OkPosition = false; // Sets

            while (placement < Objectives.Count && OkPosition == false) // Checks if we have
looped through all objectives and the correct position hasn't been reached.
            {
                if (objective.GetIntPriority() > Objectives[placement].GetIntPriority()) //
Checks if priority of objective being inserted has a greater priority that objective at index
placement - if so, placement has been reached.
                {
                    OkPosition = true;

                }
                else if (objective.GetIntPriority() == Objectives[placement].GetIntPriority())
// Checks if objective being inserted has the same priority of objective at index placement.
                {
                    if (CompareDates(objective.GetDeadline(),
Objectives[placement].GetDeadline())) // If the priority is the same, it will then compare the
deadlines. If the objective has earlier deadline, placement has reached.
                    {
                        OkPosition = true;

                    }
                    else
                    {
                        placement++; // Increment used to go through whole objective list.

                    }

                }
                else
                {
                    placement++;

                }

            }

            Objectives.Insert(placement, objective); // List operation - Once placement is
reached, it will be inserted into the priority queue at that given position.
        }



    }
```

Transactions Class

```
 1. internal class Transactions : TransactionDB
 2.     // Class inherits methods from TransactionDB - makes it easier to access methods, provides separation
between front-end and backend.
 3.
 4.     {
 5.         private Order order;
 6.         private API api;
 7.         // Use of composition and dynamic generation of objects through both the order and having an API
instance.
 8.
 9.         public Transactions(Order ORDER)
10.
11.         {
12.             this.order = new Order(ORDER.GetTicker(), ORDER.GetQuantity(),
Convert.ToDateTime(ORDER.GetDateOfTransaction()), ORDER.GetTransactionType());
13.             // Initialising both the order object and API instance for use.
                // Dynamic generation of Object Order.
14.             api = new API();
15.
16.
17.         }
18.         public Transactions()
19.         {
20.
21.
22.
23.         }
24.
25.         private bool CheckFunds(double TransactionPrice)
26.         {
27.             return GetAccountBalance() > TransactionPrice;
28.             // Retrieves account balance and measures if a user has sufficient funds for a transaction buy -
links to Wallet table via TransactionDB class.
29.
30.         }
31.         private bool CheckInvalidStock(double price)
32.         {
33.             return price == 0; // API check to find out if the stock they have searched is valid for selling
34.         }
35.
36.         public async void MakeTransaction()
37.         {
38.             try
39.             {
40.                 double PriceOfStock = await api.GetCurrentPrice(order.GetTicker()); // retrieves stock price via
API.
41.                 if (!CheckInvalidStock(PriceOfStock)) // Validation of checking for a correct stock symbol.
42.                 {
43.                     double PriceOfPurchase = Math.Round(PriceOfStock * order.GetQuantity(), 2);
44.                     if (CheckFunds(PriceOfPurchase)) // Compares user's balance and purchase price if it is
allowed.
45.                     {
46.                         if (CheckStockExists(GetInvestmentTickers(), order.GetTicker()))
47.                             // Collects the user's portfolio companies and the stock they have picked and
performs recursive search to check if stock exists within portfolio
48.                         {
49.
50.                             ChangeBalanceForBuy(PriceOfPurchase); // Changes user's balance
51.                             AddTransactionToDB(order, PriceOfStock, PriceOfPurchase); // Adds transaction to
table 'Orders'
52.                             UpdatePortfolio(order, PriceOfPurchase, true); // Updates user's portfolio statisics
(stock already exists) - changes the number of shares and amount of money invested.
53.                             MessageBox.Show($"Transaction Successful", "Confirmed", MessageBoxButtons.OKCancel);
// Suitable message responses
54.                         }
55.                         else // If User doesn't already have stock in portfolio, it will add stock to portfolio
table but as a fresh field.
56.                         {
57.
58.                             ChangeBalanceForBuy(PriceOfPurchase); // Changes user's balance
59.                             AddTransactionToDB(order, PriceOfStock, PriceOfPurchase); // Adds transaction to
table 'Orders'
60.                             UpdatePortfolio(order, PriceOfPurchase, false); // Updates user's portfolio
statistics (stock doesn't exist)
61.                             MessageBox.Show($"Transaction Successful", "Confirmed", MessageBoxButtons.OKCancel);
62.
63.
64.                         }
65.                     }
66.                     else
67.                     {
```

```
68.                         MessageBox.Show("Insufficient funds for transaction", "Error",
MessageBoxButtons.OKCancel); // Suitable error message for insufficientfunds
69.                     }
70.                 }
71.                 else
72.                 {
73.                     MessageBox.Show("Invalid Ticker Entered", "Error", MessageBoxButtons.OKCancel);
74.
75.                 }
76.             }
77.             catch (Exception e)
78.             {
79.                 MessageBox.Show("Transaction Failed. Technical Error", "Error", MessageBoxButtons.OKCancel);
80.
81.
82.
83.             }
84.
85.
86.         }
87.         public async void SellStock()
88.         {
89.            try
90.             {
91.
92.                 double PriceOfStock = await api.GetCurrentPrice(order.GetTicker()); // retrieves price of Stock
via API
93.                 if (!CheckInvalidStock(PriceOfStock) && order.GetQuantity() > 0) // Checks if  the stock is
valid from the NYSE and also checks if order field is greater than 0
94.                 {
95.
96.                     if (CheckStockExists(GetInvestmentTickers(), order.GetTicker())) // Recursive search which
collects both the user's invested companies and order stock symbol to check
97.                         // If the stock exists in the portfolio
98.                     {
99.
100.                        if (CheckSufficentStockAmount(order)) // Compares the number of stocks of that certain
stock already invested and checks if it greater than the order amount.
101.                        {
102.
103.                            double PriceOfPurchase = Math.Round(PriceOfStock * order.GetQuantity(), 2); //
Calculates Total Price.
104.                            ChangeBalanceForSell(PriceOfPurchase); // Changes the user's wallet balance
105.                            AddTransactionToDB(order, PriceOfStock, PriceOfPurchase); // Adds the transaction to
the Database table 'Orders'
106.                            UpdatePortfolio(order, PriceOfPurchase, true); // Updates user portfolio
107.                            MessageBox.Show($"Transaction Successful ", $"Confirmed",
MessageBoxButtons.OKCancel);
108.
109.                        }
110.                    else
111.                        {
112.                            MessageBox.Show($"Insufficient Number Of Stock for transaction ", $"Error",
MessageBoxButtons.OKCancel);
113.
114.                        }
115.                    }
116.                    else
117.                    {
118.                        //UpdatePortfolio(order, PriceOfPurchase, false);
119.                        MessageBox.Show($"Stock doesn't exist in portfolio", "Error",
MessageBoxButtons.OKCancel);
120.                    }
121.
122.                }
123.                else
124.                {
125.                    MessageBox.Show("Invalid Ticker or Invalid Stock Amount", "Error",
MessageBoxButtons.OKCancel);
126.                }
127.
128.
129.            }
130.            catch (Exception e)
131.            {
132.
133.                MessageBox.Show($"Transaction Failed: {e}", "Error", MessageBoxButtons.OKCancel);
134.            }
135.
136.        }
137.    }
138.
```

TransactionsDB class

```
1. abstract class TransactionDB : PortfolioDB // Again using abstract class to be a hub for inherting the methods for the Transactions class - Also inherits from PortfolioDB class as both classes are heavily interlinked.
  2.     {
  3.
  4.         public static readonly string ConnStr = $"Data Source=\"{Environment.GetFolderPath(Environment.SpecialFolder.UserProfile)}\\OneDrive - Bradford Grammar School\\Home Drive\\computer science\\Year 13 - Year 1\\NEA 10JAN\\testtttt\\NEA - 19DEC\\NEA\\STOCKUI2\\bin\\Debug\\net6.0-windows\\databaseFINAL.db\";";
  5.         // Connection string of database - used parmaterised file paths to make the code versatile and useable on all devices.
  6.
  7.
  8.         protected void AddTransactionToDB(Order order, double PriceAtTransaction, double TransactionPrice) // Protected identifier - Also used within transactions class.
  9.         {
 10.             // Method which adds new transaction to database. Takes the order made, Price at transaction for stock and total transaction price.
 11.             using (SQLiteConnection conn = new SQLiteConnection(ConnStr))
 12.             {
 13.                 try
 14.                 {
 15.                     conn.Open();
 16.                     string SQLQuery = $"INSERT INTO Orders(Ticker, Quantity, PriceAtTransaction, TransactionTime, TransactionPrice, TransactionType, wallet_ID) VALUES('{order.GetTicker()}', {order.GetQuantity()}, {PriceAtTransaction}, '{order.GetDateOfTransaction()}', {TransactionPrice}, '{order.GetTransactionType()}', {UserCredsDB.WalletID});";
 17.                     // Parameterised SQL - used in
 18.                     using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
 19.                     {
 20.                         comm.ExecuteNonQuery();
 21.
 22.                         //comm.Dispose();
 23.
 24.                     }
 25.                     conn.Close();
 26.                 }
 27.                 catch (Exception exc)
 28.                 {
 29.                     MessageBox.Show($"{exc}");
 30.                 }
 31.
 32.             }
 33.         }
 34.         protected void ChangeBalanceForBuy(double transactionPrice)
            // Made protected as it is used within the transactions class inheriting from this class.
 35.         {
 36.             // Changes wallet balance of user.
 37.             using (SQLiteConnection conn = new SQLiteConnection(ConnStr))
 38.             {
 39.                 try
 40.                 {
 41.                     conn.Open();
 42.                     double balance = Math.Round(GetAccountBalance() - transactionPrice, 2);
 43.                     string SQLQuery = $"UPDATE WALLET SET Balance = {balance} WHERE userID = {UserCredsDB.UserID};";
 44.                     using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
 45.                     {
 46.                         comm.ExecuteNonQuery();
 47.                         comm.Dispose();
 48.                     }
 49.                     conn.Close();
 50.
 51.                 }
 52.                 catch(Exception exc)
 53.                 {
 54.
 55.                 }
 56.             }
 57.
 58.         }
 59.         public static void RemoveStock(Order order)
 60.         {
 61.             // Method used to remove a stock from a portfolio if the number of shares is zero when the make a sell.
 62.             using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
 63.             {
 64.                 conn.Open();
```

```
65.                 string SQL = $"DELETE FROM Portfolio WHERE Ticker = '{order.GetTicker()}' AND userID =
{UserCredsDB.UserID}";
66.                 using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
67.                 {
68.                     comm.ExecuteNonQuery();
69.                     comm.Dispose();
70.                 }
71.                 conn.Close();
72.             }
73.         }
74.         protected void ChangeBalanceForSell(double transactionprice)
75.         {
76.             // Changes user balance for sell
77.             using (SQLiteConnection conn = new SQLiteConnection(ConnStr))
78.             {
79.                 try
80.                 {
81.                     conn.Open();
82.                     double balance = Math.Round(GetAccountBalance() + transactionprice, 2);
83.                     string SQLQuery = $"UPDATE WALLET SET Balance = {balance} WHERE userID =
{UserCredsDB.WalletID}";
84.                     using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
85.                     {
86.                         comm.ExecuteNonQuery();
87.                         //comm.Dispose();
88.                     }
89.
90.                     conn.Close();
91.                 }
92.                 catch
93.                 {
94.
95.                 }
96.             }
97.
98.
99.
100.         }
101.         public List<string> GetInvestmentTickers()
102.         {
103.             // Retrieves users' invested stocks - used in both showcasing them in the portfolio section for
analysis and sell method.
104.             try
105.             {
106.                 List<string> Companies = new List<string>();
107.                 using (SQLiteConnection conn = new SQLiteConnection(ConnStr))
108.                 {
109.                     conn.Open();
110.                     string SQLQuery = $"SELECT Ticker FROM Portfolio NOLOCK WHERE UserID =
{UserCredsDB.UserID}";
111.
112.                     using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
113.                     {
114.                         using (SQLiteDataReader reader = comm.ExecuteReader())
115.                         {
116.                             while (reader.Read())
117.                             {
118.                                 Companies.Add(reader.GetString(0));
119.                             }
120.                             reader.Close();
121.                         }
122.                     }
123.
124.                     conn.Close();
125.                     return Companies;
126.                 }
127.             }
128.             catch
129.             {
130.                 return null;
131.             }
132.         }
133.
134.         public static double GetAccountBalance()
135.         {
136.             // Get User account balance.
137.             using (SQLiteConnection conn = new SQLiteConnection(ConnStr))
138.             {
139.                 conn.Open();
140.                 string SQLQuery = $"SELECT Balance FROM WALLET NOLOCK WHERE userID = {UserCredsDB.UserID} ";
141.
142.
143.                 using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
144.                 {
145.                     using (SQLiteDataReader reader = comm.ExecuteReader())
```

```
146.                     {
147.                         while (reader.Read())
148.                         {
149.                             double balance = Convert.ToDouble(reader.GetDouble(0));
150.                             conn.Close();
151.                             return balance;
152.
153.                         }
154.                         reader.Close();
155.                         return -1;
156.                     }
157.                 }
158.             }
159.         }
160.
161.         protected bool CheckSufficentStockAmount(Order order)
162.         {
163.             // Checks if the number of shares
164.             return order.GetQuantity() <= GetSharesForTicker(order.GetTicker());
165.
166.         }
167.
168.         protected bool CheckStockExists(List<string> stocks, string TickerToSearch)
169.         // Recursive Search which takes in two parameters, a list of the user's invested stocks and the order
stock
170.         {
171.             if (stocks.Count == 0) // Checks if the list length is 0 - this means that the stock doesn't exists
172.             {
173.                 return false;
174.             }
175.             else if (stocks[0] == TickerToSearch) // If the stock at index 0 is equal to the order stock, it
exists
176.             {
177.                 return true;
178.             }
179.             else
180.             {
181.                 stocks.RemoveAt(0); // Remove the item at index 0
182.                 return CheckStockExists(stocks, TickerToSearch); // Recursive call to run function again.
183.             }
184.         }
185.
186.     }
187.
```

PortfolioDB class

```
internal class PortfolioDB
    {

        private double GiveAggregateData(string FunctionType, string table, string column)
        {
            // Create method to make it more accessible to find the aggregate data passing in a table, column and
type of SQL function
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
                string SQLQuery = $"SELECT {FunctionType}({column}) FROM {table} NOLOCK WHERE userID =
{UserCredsDB.UserID}";

                using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                {
                    SQLiteDataReader _reader = comm.ExecuteReader();

                    while (_reader.Read())
                    {

                        double result = _reader.GetDouble(0); // not empty
                        conn.Close();
                        return result;


                    }
                    return -1;

                }

            }

        }

        protected void UpdatePortfolio(Order order, double transactionPrice, bool StockExists)
        { // Method used to update portfolio depending on if they have bought or sold stock if they have or have not
previously owned the stock.
            try
            {
                if (order.GetTransactionType() == "BUY") // Checks if the transaction type is a buy
                {
                    if (StockExists) // Condition depending on if the stock exists in portfolio
                    {
                        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
                        {
                            conn.Open();
                            string SQLQuery = $"UPDATE Portfolio SET NumberOfShares =
'{GetSharesForTicker(order.GetTicker()) + order.GetQuantity()}' WHERE Ticker = '{order.GetTicker()}' AND userID =
{UserCredsDB.UserID}";
                            // Updates the number of shares to be the sum of the current number of shares in
portfolio and the order quantity.
                            using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                            {
                                comm.ExecuteNonQuery();
                            }
                            conn.Close();
                            conn.Open();
                            string Query = $"UPDATE Portfolio SET MoneyInvested =
'{GetMoneyInvestedForTicker(order.GetTicker()) + transactionPrice}' WHERE Ticker = '{order.GetTicker()}' AND userID =
{UserCredsDB.UserID}";
                            // Sets the money invested in the stock to the original amount of money invested and
transaction price.
                            using (SQLiteCommand comm = new SQLiteCommand(Query, conn))
                            {
                                comm.ExecuteNonQuery();
                            }
                            conn.Close();
                        }

                    }
                    else
                    { // If the stock doesn't exist in user portfolio, we should create a new field containing that.
                        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
                        {
                            conn.Open();
```

```
                            string SQLQuery = $"INSERT INTO Portfolio (Ticker, NumberOfShares, MoneyInvested, userID)
VALUES ('{order.GetTicker()}', {order.GetQuantity()}, {transactionPrice}, {UserCredsDB.UserID})";
                            using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                            {
                                comm.ExecuteNonQuery();
                            }
                            conn.Close();
                        }

                    }


                }
                else if (order.GetTransactionType() == "SELL")
                {
                    // Same principle as above but for selling.
                    if (StockExists == true)
                    {
                        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
                        {
                            conn.Open();
                            int ShareAmount = GetSharesForTicker(order.GetTicker()) - order.GetQuantity();
                            if (ShareAmount == 0)
                            {
                                TransactionDB.RemoveStock(order);
                            }
                            else
                            {
                                string SQLQuery = $"UPDATE Portfolio SET NumberOfShares = {ShareAmount} WHERE Ticker
= '{order.GetTicker()}' AND userID = {UserCredsDB.UserID}";
                                using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                                {
                                    comm.ExecuteNonQuery();
                                }
                                conn.Close();
                                conn.Open();

                                string Query = $"UPDATE Portfolio SET MoneyInvested =
{GetMoneyInvestedForTicker(order.GetTicker()) - transactionPrice} WHERE Ticker = '{order.GetTicker()}' AND userID =
{UserCredsDB.UserID}";
                                using (SQLiteCommand comm = new SQLiteCommand(Query, conn))
                                {
                                    comm.ExecuteNonQuery();
                                }
                                conn.Close();
                            }
                        }

                    }


                }



            }
            catch (Exception ex)
            {
                MessageBox.Show("Transaction Failed", "Error", MessageBoxButtons.OKCancel);


            }



        }


        public static bool CheckForPortfolioData()
        {
            //Check used so when user opens portfolio section, if they don't have data, they won't be able to search
and access information.,
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();

                string SQL = $"SELECT Ticker FROM Portfolio NOLOCK WHERE userID = {UserCredsDB.UserID}";

                using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
                {
                    using (SQLiteDataReader reader = comm.ExecuteReader())
                    {
                        if (reader.Read())
                        {
                            conn.Close();
```

```
                        return true;
                    }
                    else
                    {
                        conn.Close();
                        return false;

                    }

                }
            }



        }

    }


    public int GetSharesForTicker(string ticker)
    {

        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
        {
            conn.Open();
            string SQLQuery = $"SELECT NumberOfShares FROM Portfolio NOLOCK WHERE Ticker = '{ticker}' AND userID
= {UserCredsDB.UserID} ";
            // use of cross-table parameterised SQL queries
            int shares = 0;
            using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
            {
                using (SQLiteDataReader _reader = comm.ExecuteReader())
                {
                    while (_reader.Read())
                    {
                        shares = _reader.GetInt32(0);
                    }
                    _reader.Close();


                }
            }
            conn.Close();
            return shares;

        }
    }

    public List<string> GetMaxSharesForStock()
    {
        List<string> data = new List<string>();
        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
        {
            conn.Open();

            string SQL = $"SELECT portfolio.Ticker, portfolio.NumberOfShares FROM Portfolio portfolio JOIN USERS
user ON portfolio.userID = user.AccountID WHERE portfolio.NumberOfShares = {GiveAggregateData("MAX", "Portfolio",
"NumberOfShares")}";
            // Again a representation of parameterised JOIN SQL queries collecting the maximimum shares on a
user's portfolio
            using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
            {
                using (SQLiteDataReader _reader = comm.ExecuteReader())
                {
                    if (_reader.Read())
                    {
                        data.Add(_reader.GetString(0));
                        data.Add(Convert.ToString(_reader.GetInt32(1)));

                    }
                    _reader.Close();
                }

            }
            conn.Close();
            return data;

        }
    }
    public List<string> GetMostExpensiveInvestment()
    {
// Method utilises the Aggregate function MAX to return the most expensive investment in user's portfolio.
        List<string> data = new List<string>();
        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
        {
```

```
                conn.Open();

                string SQL = $"SELECT Ticker, MAX(TransactionPrice) FROM Orders NOLOCK WHERE wallet_ID =
{UserCredsDB.WalletID} AND TransactionType = 'BUY'";

                using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
                {
                    using (SQLiteDataReader _reader = comm.ExecuteReader())
                    {
                        if (_reader.Read())
                        {
                            data.Add(_reader.GetString(0));
                            data.Add(Convert.ToString(_reader.GetDouble(1)));

                        }
                        _reader.Close();
                    }


                }
                conn.Close();
                return data;


            }




        }
        public double GetMoneyInvestedForTicker(string ticker)
        {
// Returns the amount of money invested for a given stock symbol. Used when user searches for specific stock
statistics.
            try
            {
                using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
                {
                    conn.Open();
                    //"Data Source=<path to database file>;Version=3;New=True;Compress=True;"
                    string SQLQuery = $"SELECT MoneyInvested FROM Portfolio NOLOCK WHERE Ticker = '{ticker}' AND
userID = {UserCredsDB.UserID}";

                    using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                    {
                        SQLiteDataReader _reader = comm.ExecuteReader();

                        while (_reader.Read())
                        {

                            double MoneyInvested = _reader.GetDouble(0);
                            _reader.Close();
                            conn.Close();
                            return MoneyInvested;



                        }


                        return -1;

                    }


                }

            }
            catch
            {
                return -1;

            }



        }

        public double GetTotalMoneyInvested()
// Utilising the GiveAggregateData method to return the total amount of money spent in the portfolio.
        {
            return GiveAggregateData("SUM", "Portfolio", "MoneyInvested");

        }
```

```
        public double GetProfitOrLoss()
        {

            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
                string SQL = $"SELECT SUM(TransactionPrice - MoneyInvested) FROM Portfolio portfolio JOIN Orders
orders ON portfolio.Ticker = orders.Ticker WHERE portfolio.userID = {UserCredsDB.UserID} AND orders.TransactionType =
'BUY'";
                // Conjuction of both aggregate and join table parameterised SQL calculating the profit of a user
portfolio.

                using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
                {
                    using (SQLiteDataReader _reader = comm.ExecuteReader())
                    {

                        _reader.Read();
                        double Profit = _reader.GetDouble(0);
                        conn.Close();
                        return Profit;

                    }
                }
            }
        }
    }
```

UserCredsDB class

```
  abstract class UserCredsDB
    {
        public static int UserID;
        public static int WalletID;
        private static readonly string ADMINPWD =
$"{Environment.GetFolderPath(Environment.SpecialFolder.UserProfile)}\\OneDrive - Bradford Grammar School\\Home
Drive\\computer science\\Year 13 - Year 1\\NEA 10JAN\\testtttt\\NEA - 19DEC\\NEA\\API_KEY.txt";
        public static string GetAdmin_PWD()
        {
//Retrieves admin password from the text file. No exposure of passwords in the source code – defensive programming.
            try
            {
                using (StreamReader reader = new StreamReader(ADMINPWD, true))
                {

                    reader.ReadLine();
                    string Password = reader.ReadLine();
                    return Password;

                }

            }
            catch (Exception exception)
            {
                throw exception;

            }

        }

        public void InitialiseAccount(string Username, string Password)
        {
```

```
            Random rnd = new Random();
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
// Creates new user account and adds their username and the corresponding hashed password.

                string SQLQuery = $"INSERT INTO USERS (Username, Password) VALUES ('{Username}',
'{Hash.HashPassword(Password)}');";
                using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                {
                    comm.ExecuteNonQuery();

                }
                conn.Close();


            }
        }

        public void CreateWallet(string username)
        {
            // Creates user wallet inserting an initial balance of 10,000 dollars.
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
                string SQLQuery = $"INSERT INTO WALLET (Balance, userID) VALUES (10000,
'{GetUserIDFromUsername(username)}');";
                using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                {
                    comm.ExecuteNonQuery();

                }
                conn.Close();
            }


        }
        protected static int GetUserIDFromUsername(string username)
        {
            try
            {
                using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
                {// Selects the account ID from  the table where the username is equal to the one entered. This is
used when applying the accountID to the static integer userID - very important when making changes to accounts.
                    conn.Open();
                    string SQLQuery = $"SELECT AccountID FROM USERS NOLOCK WHERE Username = '{username}'";

                    using (SQLiteCommand cmd = new SQLiteCommand(SQLQuery, conn))
                    {
                        using (SQLiteDataReader reader = cmd.ExecuteReader())
                        {
                            while (reader.Read())
                            {
                                int id = Convert.ToInt32(reader.GetInt32(0));
                                reader.Close();
                                conn.Close();
                                return id;

                            }

                            return -1;

                        }

                    }
                }

            }
            finally
            {


            }

        }
        protected static int GetWalletIDFromUsername(string username)
        {
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
                        // Selects WalletID where the userID equals to the corresponding user signing in.
                string SQLQuery = $"SELECT WalletID FROM WALLET NOLOCK WHERE userID = '{UserCredsDB.UserID}'";

                using (SQLiteCommand cmd = new SQLiteCommand(SQLQuery, conn))
                {
```

```
                using (SQLiteDataReader reader = cmd.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        int id = Convert.ToInt32(reader.GetInt32(0));

                        conn.Close();

                        return id;

                    }
                    return -1;
                }


            }
        }
    }

    public bool CheckUserAlreadyExists(string username)
    {
        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
        {
            conn.Open();
            //Query used to select all users to which the username field equals the username entered by user
            string SQLQuery = $"SELECT * FROM USERS NOLOCK WHERE Username = '{username}'";

            using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
            {
                SQLiteDataReader _reader = comm.ExecuteReader();

                if (_reader.Read())
                {
                    conn.Close();
                    return true;


                }
                else
                {
                    conn.Close();
                    return false;
                }
            }
        }

    }
    public void DeleteUser(string username)
    {
        // Deletes user from database.
        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
        {
            conn.Open();
            string SQLQuery = $"DELETE FROM USERS WHERE AccountID = {GetUserIDFromUsername(username)}";
            using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
            {
                comm.ExecuteNonQuery();

            }
            conn.Close();
        }

    }
    public static List<string> GetUsernames()
    {
        using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
        {
            conn.Open();
            List<string> Usernames = new List<string>();
            string SQL = "SELECT Username FROM USERS";
            using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
            {
                using (SQLiteDataReader reader = comm.ExecuteReader())
                {
                    while (reader.Read())
                    {
                        Usernames.Add(reader.GetString(0));

                    }
                }
                conn.Close();


            }
```

```
                return Usernames;
            }


        }

        public static int GetUserCount() {

            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
                int count = 0;
                string SQL = "SELECT COUNT(Username) FROM USERS";
            // Aggregate functions used to display how many users exist in the app.
                using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
                {
                    using (SQLiteDataReader reader = comm.ExecuteReader())
                    {
                        while (reader.Read())
                        {
                            count = reader.GetInt32(0);
                        }
                    }
                    conn.Close();


                }

                return count;
            }



        }
        public bool CheckForUserCreds(string username, string password)
        {
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();

                string SQLQuery = $"SELECT * FROM USERS NOLOCK WHERE Username = '{username}' AND Password =
'{Hash.HashPassword(password)}'";
 // Query used to check if the username or password entered exists in the database.
                using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                {
                    SQLiteDataReader _reader = comm.ExecuteReader();

                    if (_reader.Read())
                    {
                        conn.Close();
                        return true;


                    }
                    else
                    {
                        conn.Close();
                        return false;
                    }
                }


            }

        }
    }
```

ObjectiveDB class

```
abstract class ObjectiveDB // Made class abstract to prevent multiple instances of the database class being created.
Used to provide methods of functionality for classes both inhertiting and in the UI.
    {

        protected void AddObjectiveToDB(Objective obj)
        {
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {

                conn.Open();
                string SQLquery = $"INSERT INTO Objectives (Title, Description, Priority, Deadline, IsComplete,
userID) VALUES ('{obj.GetTitle()}', '{obj.GetTask()}', '{obj.GetStrPriority()}',
'{Convert.ToString(obj.GetDeadline())}', {false}, '{UserCredsDB.UserID}');";
                // Query to add an objective to table

                using (SQLiteCommand comm = new SQLiteCommand(SQLquery, conn))
                {
                    comm.ExecuteNonQuery();
                    conn.Close();
                }
            }
        }

        protected List<Objective> GetObjectivesFromDB()
        {
            List<Objective> objectives = new List<Objective>();
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();

                string SQLquery = $"SELECT * FROM Objectives NOLOCK WHERE userID = {UserCredsDB.UserID}";
                // Returns all the objectives from the table as a List of 'Objective'. This is used to gather all
objectives to display on table

                using (SQLiteCommand comm = new SQLiteCommand(SQLquery, conn))
                {
                    using (SQLiteDataReader _reader = comm.ExecuteReader())
                    {
                        while (_reader.Read())
                        {
                            Objective obj = new Objective();
                            obj.ObjectiveID = Convert.ToInt32(_reader.GetInt32(0));
                            obj.Title = Convert.ToString(_reader.GetString(1));
                            obj.task = Convert.ToString(_reader.GetString(2));
                            obj.Priority = Convert.ToString(_reader.GetString(3));
                            obj.Status = Convert.ToBoolean(_reader.GetBoolean(5));
                            obj.Deadline = Convert.ToDateTime(_reader.GetString(4));
                            objectives.Add(obj);
                        }
                        conn.Close();
                        return objectives;
                    }

                }

            }

        }

        public static int GetOBJID()
        {
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
                string SQLQuery = $"SELECT ObjectiveID FROM Objectives NOLOCK WHERE Title = '{ObjectivesUI.title}'
AND Description = '{ObjectivesUI.Description}';";
                // Returns the objective ID of a selected objective so when a user clicks the edit button to another
form, it will return the ID to display the selected Objective

                using (SQLiteCommand cmd = new SQLiteCommand(SQLQuery, conn))
                {
                    using (SQLiteDataReader reader = cmd.ExecuteReader())
                    {
```

```
                        while (reader.Read())
                        {
                            int id = Convert.ToInt32(reader.GetInt32(0));

                            conn.Close();
                            return id;
                        }
                        return -1;
                    }
                }
            }
        }
        public static void EditObj(Objective obj)
        {
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {

                try
                {
                    conn.Open();
                    string status = Convert.ToString(obj.GetStatus()).ToLower();
                    string SQLQuery = $"UPDATE Objectives SET Title = '{obj.GetTitle()}', Description =
'{obj.GetTask()}', Priority = '{obj.GetStrPriority()}', Deadline = '{obj.GetDeadline()}', IsComplete =
{Convert.ToBoolean(status)} WHERE ObjectiveID = {ObjectiveDB.GetOBJID()}";
                    // Used to edit a given objective changing what has been typed into the edit columns
                    using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                    {
                        comm.ExecuteNonQuery();

                    }
                    conn.Close();
                }
                catch
                {

                }
            }



        }
        public static bool CheckForObjectiveData()
        {
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();

                string SQL = $"SELECT Title FROM Objectives NOLOCK WHERE userID = {UserCredsDB.UserID}";
                // Query used to check if a user has any objectives. Used to prevent exception of trying to gather
objectives that don't exist.

                try
                {
                    using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
                    {
                        using (SQLiteDataReader reader = comm.ExecuteReader())
                        {
                            if (reader.Read())
                            {
                                return true;
                            }
                            else
                            {
                                return false;
                            }
                        }
                    }


                }
                catch
                {
                    return false;

                }

            }
        }


    }
```

Order class

```
internal class Order
    {
        private string ticker; // privatising properties - Using encapsulation to get these properties.
        private int Quantity;
        private DateTime DateOfTransaction;
        private string TransactionType;

        public Order()
        {



        }

        public Order(string ticker, int Quantity, DateTime DateOfTransaction, string TransactionType)
        {
            this.ticker = ticker;
            this.Quantity = Quantity;
            this.DateOfTransaction = DateOfTransaction;
            this.TransactionType = TransactionType;

        }

        public string GetTicker()
        {
            return ticker.ToUpper();
        }

        public int GetQuantity()
        {
            return Quantity;
        }

        public string GetDateOfTransaction()
        {
            return string.Format("{0:G}", DateOfTransaction);  // formats date in the correct form for processing
within database.
        }

        public string GetTransactionType()
        {
            return TransactionType;
        }

    }
```

PortfolioManager Class

```
internal class PortfolioManager : TransactionDB
    {
        private PortfolioDB db;

        public PortfolioManager()
        {
            db = new PortfolioDB();
        }
        public List<string> GetTickers()
        {
            return GetInvestmentTickers();
        }

        public double GetProportion(string ticker)
        {
            return Math.Round((db.GetMoneyInvestedForTicker(ticker) / db.GetTotalMoneyInvested())*100,3);

        }
    }

```

Dashboard class

```
public partial class Dashboard : Form
    {

        List<string> LIST = new List<string>();
        private SearchHistory history;



        public Dashboard()
        {

            history = new SearchHistory();
            InitializeComponent();
        }




        private void label2_Click(object sender, EventArgs e)
        {

        }

        private void Form1_Load(object sender, EventArgs e)
        {
            this.SearchHistoryBox.DropDownStyle = ComboBoxStyle.DropDownList;
            this.TypeButton.DropDownStyle = ComboBoxStyle.DropDownList;
            TypeButton.Items.Add("day");
            TypeButton.Items.Add("month");
            label5.Text = Convert.ToString($"${Math.Round(TransactionDB.GetAccountBalance(), 2)}");
            Refresh();

            Refresh();
```

```
34.
35.
36.         }
37.
38.         private void timer1_Tick(object sender, EventArgs e)
39.         {
40.             label1.Text = DateTime.Now.ToLongTimeString();
41.             label2.Text = DateTime.Now.ToLongDateString();
42.         }
43.
44.         private void label3_Click(object sender, EventArgs e)
45.         {
46.
47.         }
48.
49.         private async void button1_Click(object sender, EventArgs e)
50.         {
51.
52.
53.
54.         }
55.
56.         private async void button5_Click(object sender, EventArgs e)
57.         {
58.
59.
60.         }
61.
62.         private async void label5_Click(object sender, EventArgs e)
63.         {
64.
65.         }
66.
67.         private void cartesianChart1_Load(object sender, EventArgs e)
68.         {
69.
70.         }
71.
72.
73.
74.         private void formsPlot2_Load(object sender, EventArgs e)
75.         {
76.
77.         }
78.
79.         private void formsPlot1_Load(object sender, EventArgs e)
80.         {
81.
82.
83.         }
84.
85.         private void panel4_Paint(object sender, PaintEventArgs e)
86.         {
87.
88.         }
89.
90.         private void Exit_Click(object sender, EventArgs e)
91.         {
92.             Application.Exit();
93.         }
94.
95.         private void TestObj_Click(object sender, EventArgs e)
96.         {
97.
98.         }
99.
100.        private void richTextBox1_TextChanged(object sender, EventArgs e)
101.        {
102.
103.
104.        }
105.
106.        private void button7_Click(object sender, EventArgs e)
107.        {
108.            PortfolioPage pg = new PortfolioPage();
109.            pg.Show();
110.
111.            this.Visible = false;
112.
113.        }
114.
115.        private void label3_Click_1(object sender, EventArgs e)
116.        {
117.
118.        }
```

```
119.
120.         private void MaxScreen_Click(object sender, EventArgs e)
121.         {
122.             this.WindowState = FormWindowState.Maximized;
123.             MaxScreen.Visible = false;
124.             MaxScreen.Location = MaxScreen.Location;
125.             MaxScreen.Visible = true;
126.         }
127.
128.         private void MinScreen_Click(object sender, EventArgs e)
129.         {
130.             this.WindowState = FormWindowState.Normal;
131.             MinScreen.Visible = false;
132.             MinScreen.Location = MinScreen.Location;
133.             MinScreen.Visible = true;
134.         }
135.
136.         private async void button10_Click(object sender, EventArgs e)
137.         {
138.
139.             StockLabel.ForeColor = Color.White;
140.
141.             try
142.             {
143.
144.                 cartesianChart1.TooltipPosition = LiveChartsCore.Measure.TooltipPosition.Bottom;
145.                 cartesianChart1.LegendTextSize = 1;
146.                 cartesianChart1.AnimationsSpeed = TimeSpan.FromMilliseconds(500);
147.
148.                 string ticker = StockLabel.Text;
149.                 try
150.                 {
151.
152.
153.                         API api = new API(); // Creates new instance of API
154.                         List<Values> values = await api.StockStats(ticker); // Retrieves the current statistics
for the stock entered
155.                         List<string> times = new List<string>();
156.
157.
158.
159.
160.                         if (values != null )
161.                             {
162.
163.                         times = api.Initalise_timeseries(times, TypeButton.Text,
Convert.ToInt32(DurationUpDown.Text)); // initalises the times
164.                         if (times != null)
165.                             {
166.                                 List<Values> PriceData = new List<Values>(); // Creates a list of values to
provide a container for the historical stock data.
167.                                 PriceData = await api.Prices(ticker, times[1], times[0], $"1{TypeButton.Text}");
// Makes API retrieval response
168.                                 cartesianChart1.Refresh();
169.
170.                                 double[] prices = new double[PriceData.Count]; // Creates an array to hold the
Stock Price data points
171.
172.                                 for (int i = 0; i < prices.Length; i++)
173.                                 {
174.                                     prices[i] = Math.Round(PriceData[i].GetClose()); // Adds each data point
from the API response into the array.
175.                                 }
176.                                 cartesianChart1.ZoomMode = LiveChartsCore.Measure.ZoomAndPanMode.X;
177.
178.                                 cartesianChart1.TooltipPosition = LiveChartsCore.Measure.TooltipPosition.Bottom;
179.
180.                                 cartesianChart1.Series = new ISeries[]
181.                                 {
182.                                 new LineSeries<double>
183.                                 {
184.                                     Name = "Price",
185.                                     LineSmoothness = 1,
186.                                     DataLabelsSize = 0.10,
187.                                     LegendShapeSize = 0,
188.                                     Values = prices,
189.                                     IsVisibleAtLegend = true,
190.                                     Fill = null,
191.                                 }
192.
193.                                 };
194.
195.
196.
197.
```

```
198.                                 foreach (Values vals in values)
199.                                 {
200.                                     DisplayHighPrice.Text = $"${Convert.ToString(Math.Round(vals.GetHigh(),
2))}"; // For each stock, it will display the current main data points, e.g. high
201.                                     DisplayClose.Text = $"${Convert.ToString(Math.Round(vals.GetClose(), 2))}";
202.                                     DisplayOpen.Text = $"${Convert.ToString(Math.Round(vals.GetOpen(), 2))}";
203.                                     DisplayLowPrice.Text = $"${Convert.ToString(Math.Round(vals.GetLow(), 2))}";
204.                                     DisplayPerChange.Text =
$"{Convert.ToString(Math.Round(vals.GetPercentChange(), 2))}%";
205.                                     DisplayPrevClose.Text =
$"${Convert.ToString(Math.Round(vals.GetPreviousClose(), 2))}";
206.
207.                                 }
208.
209.
210.
211.                                 history.AddSearch(ticker); // Adds each ticker search to the history stack
object
212.                                 LIST = history.ShowHistory(); // Adds each item from stack to List
213.                                 SearchHistoryBox.Items.Clear();
214.                                 foreach (var item in LIST)
215.                                 {
216.                                     SearchHistoryBox.Items.Add(item); // Adds each search to search box list.
217.
218.                                 }
219.
220.                                 List<Values> Prices = new List<Values>(); // Creates new list object holding
prices used for MACD algorithm
221.                                 List<string> TIMES = new List<string>();
222.                                 Trends trend = new Trends(Prices); // Creates new instance of Trends that inputs
price data.
223.                                 times = trend.Initalise_timeseries(TIMES, "day", 100); // Initialises the times.
224.                                 Prices = await api.Prices(StockLabel.Text, TIMES[1], TIMES[0], "1day");
225.                                 Decision result = Trends.BuyOrSell(Prices); // Creates Decision Object which
holds the outcome of the MACD algorithm
226.                                 double confidence = Trends.FindPercentConfidence(Trends.GetMACDval(Prices));
227.                                 if (result == Decision.Buy)
228.                                 {
229.                                     DecisionLabel.ForeColor = Color.IndianRed;
230.                                     DecisionLabel.Text = "BUY";
231.                                     PercentageConfidenceLBL.Text = $"{Math.Round(confidence, 2)}%";
232.
233.
234.
235.                                 }
236.                                 else if (result == Decision.Sell)
237.                                 {
238.                                     DecisionLabel.ForeColor = Color.LawnGreen;
239.                                     DecisionLabel.Text = "SELL";
240.                                     PercentageConfidenceLBL.Text = $"{Math.Round(confidence, 2)}%";
241.
242.                                 }
243.                                 else
244.                                 {
245.                                     DecisionLabel.Text = "HOLD";
246.                                     PercentageConfidenceLBL.Text = $"{Math.Round(confidence, 2)}%";
247.
248.                                 }
249.
250.
251.                             }
252.                             else
253.                             {
254.                                 MessageBox.Show("Times are not inputted correctly", "Error",
MessageBoxButtons.OKCancel);
255.
256.                             }
257.
258.                     }
259.                     else
260.                     {
261.                         MessageBox.Show("Ticker field is either empty or invalid", "Cannot produce stock
infomation", MessageBoxButtons.OK);
262.
263.                     }
264.                 }
265.                 catch (System.NullReferenceException ex)
266.                 {
267.                     MessageBox.Show("Unable to retrieve stock data for ticker. One or more fields may be empty",
"Error", MessageBoxButtons.OKCancel);
268.
269.                 }
270.
271.
272.
```

```
273.
274.             }
275.
276.             catch (System.NullReferenceException ex)
277.             {
278.                 MessageBox.Show("Invalid Ticker or Invalid field", "Error", MessageBoxButtons.OKCancel);
279.
280.
281.             }
282.
283.
284.
285.
286.
287.         }
288.         private void MakeChart(double[] prices)
289.         {
290.
291.
292.
293.         }
294.         private void HighLbl_Click(object sender, EventArgs e)
295.         {
296.
297.         }
298.
299.         private void button4_Click(object sender, EventArgs e)
300.         {
301.             ObjectivesUI objUI = new ObjectivesUI();
302.             objUI.Show();
303.             this.Visible = false;
304.
305.
306.         }
307.
308.         private void panel1_Paint(object sender, PaintEventArgs e)
309.         {
310.
311.         }
312.
313.         private void label12_Click(object sender, EventArgs e)
314.         {
315.
316.         }
317.
318.         private void label1_Click(object sender, EventArgs e)
319.         {
320.
321.         }
322.
323.         private void panel6_Paint(object sender, PaintEventArgs e)
324.         {
325.
326.         }
327.
328.         private void label8_Click(object sender, EventArgs e)
329.         {
330.
331.         }
332.
333.         private void textBox4_TextChanged(object sender, EventArgs e)
334.         {
335.
336.         }
337.
338.         private void textBox2_TextChanged(object sender, EventArgs e)
339.         {
340.
341.         }
342.
343.         private async void button1_Click_1(object sender, EventArgs e)
344.         {
345.             try
346.             {   int quantity = Convert.ToInt32(QuantityTEXTBOX.Text);
347.                 if (quantity != 0)
348.                 {
349.                     Order order = new Order(TickerBuyTextBox.Text, Convert.ToInt32(QuantityTEXTBOX.Text),
DateTime.Now, "BUY"); // Dynamic Generation of Object
350.                     DialogResult d = MessageBox.Show($"Would you like to make this payment?", "WARNING",
MessageBoxButtons.YesNo); // Creates Object to be the result of warning label
351.                     if (d == DialogResult.Yes)
352.                     {
353.
354.                         Transactions transaction = new Transactions(order); // Creates transaction object to
hold the order.
```

```
355.                         transaction.MakeTransaction();
356.                         Refresh();
357.                         Dashboard form = new Dashboard();
358.                         form.Show();
359.                         this.Visible = false;
360.                     }
361.
362.                 }
363.
364.                 else
365.                 {
366.                     MessageBox.Show("Transaction Failed. Quantity cannot be 0", "Error",
MessageBoxButtons.OKCancel);
367.                 }
368.
369.             }
370.             catch (Exception ex)
371.             {
372.                 MessageBox.Show("Transaction Failed. Invalid Ticker", "Error", MessageBoxButtons.OKCancel);
373.
374.             }
375.
376.
377.
378.         }
379.
380.         private async void button7_Click_1(object sender, EventArgs e)
381.         {
382.
383.         }
384.
385.         private void button3_Click(object sender, EventArgs e)
386.         {
387.             GeometricBrownianMotion gbm = new GeometricBrownianMotion();
388.             gbm.Show();
389.             this.Visible = false;
390.
391.         }
392.
393.         private void defaultLegend1_Load(object sender, EventArgs e)
394.         {
395.
396.         }
397.
398.         private void cartesianChart1_Load_1(object sender, EventArgs e)
399.         {
400.
401.         }
402.
403.         private void button5_Click_1(object sender, EventArgs e)
404.         {
405.             WelcomePage pg = new WelcomePage();
406.             pg.Show();
407.             this.Visible = false;
408.         }
409.
410.         private async void button8_Click(object sender, EventArgs e)
411.         {
412.             try
413.             {
414.
415.                 if (SearchHistoryBox.Text != null && TypeButton.Text != null &&
Convert.ToInt32(DurationUpDown.Text) != 0)
416.                 {
417.                     API api = new API();
418.                     string ticker = SearchHistoryBox.Text;
419.                     List<string> times = new List<string>();
420.                     times = api.Initalise_timeseries(times, TypeButton.Text,
Convert.ToInt32(QuantityTEXTBOX.Text));
421.                     List<Values> PriceData = await api.Prices(ticker, times[1], times[0],
$"1{TypeButton.Text}");
422.                     List<Values> values = await api.StockStats(ticker);
423.                     double[] prices = new double[PriceData.Count];
424.                     for (int i = 0; i < prices.Length; i++)
425.                     {
426.                         prices[i] = Math.Round(PriceData[i].GetClose());
427.                     }
428.
429.
430.                     cartesianChart1.ZoomMode = LiveChartsCore.Measure.ZoomAndPanMode.X;
431.                     cartesianChart1.BorderStyle = BorderStyle.Fixed3D;
432.
433.                     cartesianChart1.TooltipPosition = LiveChartsCore.Measure.TooltipPosition.Bottom;
434.
435.                     cartesianChart1.Series = new ISeries[]
```

```
436.                     {
437.                     new LineSeries<double>
438.                     {
439.                         Values = prices,
440.                         IsVisibleAtLegend = true,
441.                         Fill = null
442.
443.                     }
444.                     };
445.                     foreach (Values vals in values)
446.                     {
447.                         DisplayHighPrice.Text = $"${Convert.ToString(Math.Round(vals.GetHigh(), 2))}";
448.                         DisplayClose.Text = $"${Convert.ToString(Math.Round(vals.GetClose(), 2))}";
449.                         DisplayOpen.Text = $"${Convert.ToString(Math.Round(vals.GetOpen(), 2))}";
450.                         DisplayLowPrice.Text = $"${Convert.ToString(Math.Round(vals.GetLow(), 2))}";
451.                         DisplayPerChange.Text = $"{Convert.ToString(Math.Round(vals.GetPercentChange(), 2))}%";
452.                         DisplayPrevClose.Text = $"${Convert.ToString(Math.Round(vals.GetPreviousClose(), 2))}";
453.
454.                     }
455.                 }
456.                 else
457.                 {
458.
459.                     MessageBox.Show("Unable to retrieve stock data for ticker", "Error",
MessageBoxButtons.OKCancel);
460.
461.                 }
462.
463.
464.             }catch(NullReferenceException exc)
465.             {
466.                 MessageBox.Show("Search History field is missing");
467.
468.             }
469.         }
470.
471.         private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
472.         {
473.             StockLabel.Text = SearchHistoryBox.SelectedItem.ToString();
474.         }
475.
476.         private void richTextBox1_TextChanged_1(object sender, EventArgs e)
477.         {
478.
479.         }
480.
481.         private void panel5_Paint(object sender, PaintEventArgs e)
482.         {
483.
484.         }
485.     }
486.
```

PortfolioPage class

```
 1. public partial class PortfolioPage : Form
 2.     {
 3.         private PortfolioManager pm;
 4.         private Transactions transaction;
 5.
 6.
 7.          public PortfolioPage()
 8.          {
 9.              transaction = new Transactions();
10.              pm = new PortfolioManager();
11.              InitializeComponent();
12.          }
13.
14.          private void PortfolioPage_Load(object sender, EventArgs e)
15.          {
16.              this.InvestedStocksList.DropDownStyle = ComboBoxStyle.DropDownList;
17.              this.StartPosition = FormStartPosition.CenterScreen;
18.              BalanceLBL.Text = Convert.ToString($"${TransactionDB.GetAccountBalance()}");
19.              DisplayOrders();
20.              if (PortfolioDB.CheckForPortfolioData())
21.              {
22.
23.                  List<string> tickers = pm.GetInvestmentTickers();
24.
25.                  foreach (string ticker in tickers)
26.                  {
27.                      InvestedStocksList.Items.Add(ticker);
28.
29.                  }
30.
31.
32.              }
33.              else
34.              {
35.                  SearchButton.Enabled= false;
36.                  PortfolioViewButton.Enabled = false;
37.
38.              }
39.
40.          }
41.
42.          private void DisplayOrders()
43.          {
44.              using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
45.              {
46.                  conn.Open();
47.                     // used to retrieve the orders to be placed into the table.
48.                  SQLiteDataAdapter adp = new SQLiteDataAdapter($"SELECT Ticker, Quantity, PriceAtTransaction,
TransactionTime, TransactionPrice, TransactionType FROM Orders NOLOCK WHERE wallet_ID = {UserCredsDB.WalletID}",
conn);
49.                  DataSet ds = new System.Data.DataSet();
50.                  adp.Fill(ds);
51.                  dataGridView1.DataSource = ds.Tables[0];
52.                  conn.Close();
53.
54.
55.
56.              }
57.
58.
59.          }
60.          private void button2_Click(object sender, EventArgs e)
61.          {
62.              Dashboard form = new Dashboard();
63.              form.Show();
64.              this.Visible = false;
65.          }
66.
67.          private void label5_Click(object sender, EventArgs e)
68.          {
69.
70.          }
71.          private void timer1_Tick(object sender, EventArgs e)
72.          {
73.
74.
75.          }
76.          private void pieChart1_Load(object sender, EventArgs e)
77.          {
```

```
 78.
 79.         }
 80.
 81.         private void button5_Click(object sender, EventArgs e)
 82.         {
 83.
 84.             DialogResult d = MessageBox.Show("Would you like to make this payment?", "WARNING",
MessageBoxButtons.YesNo);
 85.             if (d == DialogResult.Yes)
 86.             {
 87.                 Order Order = new Order(TickerBox.Text, Convert.ToInt32(QuantityBox.Value), DateTime.Now,
"SELL");
 88.                 Transactions TRANSACTION = new Transactions(Order);
 89.                 TRANSACTION.SellStock();
 90.                 PortfolioPage pg = new PortfolioPage();
 91.
 92.                 pg.Show();
 93.                 this.Visible = false;
 94.             }
 95.             else
 96.             {
 97.
 98.
 99.             }
100.
101.
102.         }
103.
104.         private void label1_Click(object sender, EventArgs e)
105.         {
106.
107.         }
108.
109.         private void button7_Click(object sender, EventArgs e)
110.         {
111.             string ticker = InvestedStocksList.Text;
112.             if(InvestedStocksList.SelectedItem != null)
113.             {
114.                 ProportionLBL.Text = $"{Math.Round(pm.GetProportion(ticker), 2)}%";
115.                 MoneyInvestedLabel.Text = $"${Math.Round(pm.GetMoneyInvestedForTicker(ticker), 2)}";
116.                 ShareLabel.Text = Convert.ToString(pm.GetSharesForTicker(ticker));
117.
118.
119.             }
120.             else
121.             {
122.                 MessageBox.Show("No ticker selected", "Error", MessageBoxButtons.OKCancel);
123.
124.
125.             }
126.
127.
128.
129.
130.         }
131.
132.         private void panel4_Paint(object sender, PaintEventArgs e)
133.         {
134.
135.         }
136.
137.         private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
138.         {
139.
140.         }
141.
142.         private void panel1_Paint(object sender, PaintEventArgs e)
143.         {
144.
145.         }
146.
147.         private void button6_Click(object sender, EventArgs e)
148.         {
149.             if (PortfolioDB.CheckForPortfolioData())
150.             {
151.                 List<string> MaxShares = pm.GetMaxSharesForStock();
152.                 List<string> MaxInvestment = pm.GetMostExpensiveInvestment();
153.                 LargestShares.Text = $"Stock: {MaxShares[0]}, Number: {MaxShares[1]}";
154.                 MostExpensiveInvestment.Text = $"Stock: {MaxInvestment[0]}, Price: ${MaxInvestment[1]}";
155.                 // string ProfitText = Convert.ToString(Math.Round(TransactionDB.GetAccountBalance() -
pm.GetProfitOrLoss(), 2));
156.                 string ProfitText = Convert.ToString(Math.Round(pm.GetProfitOrLoss(), 2));
157.                 if (ProfitText[0] == '-')
158.                 {
159.                     ProfitTicker.ForeColor = Color.Red;
```

```
160.
161.                 }
162.                 else
163.                 {
164.                     ProfitTicker.ForeColor = Color.LawnGreen;
165.
166.                 }
167.                 ProfitTicker.Text = $"${ProfitText}";
168.
169.             }
170.             else
171.             {
172.                 List<string> MaxInvestment = pm.GetMostExpensiveInvestment();
173.                 MostExpensiveInvestment.Text = $"Stock: {MaxInvestment[0]}, Price: ${MaxInvestment[1]}";
174.                 LargestShares.Text = $"No Stocks Invested In";
175.                 ProfitTicker.Text = "No Stocks Currently Invested In. No Profit/Loss";
176.
177.
178.             }
179.
180.         }
181.
182.         private void button4_Click(object sender, EventArgs e)
183.         {
184.             ObjectivesUI ui = new ObjectivesUI();
185.             ui.Show();
186.             this.Visible = false;
187.         }
188.
189.         private void button3_Click(object sender, EventArgs e)
190.         {
191.             GeometricBrownianMotion gm = new GeometricBrownianMotion();
192.             gm.Show();
193.             this.Visible = false;
194.         }
195.
196.         private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
197.         {
198.
199.         }
200.
```

WelcomePage class

```
1. public partial class WelcomePage : Form
2.     {
3.         public WelcomePage()
4.         {
5.             InitializeComponent();
6.         }
7.
8.         private void WelcomePage_Load(object sender, EventArgs e)
9.         {
10.
11.         }
12.
13.         private void button2_Click(object sender, EventArgs e)
14.         {
15.             SignUp signUp = new SignUp();
16.             this.Visible = false;
17.
18.             signUp.Show();
19.             Refresh();
20.         }
21.
22.         private void button1_Click(object sender, EventArgs e)
23.         {
24.             this.Visible = false;
25.             SignIn form = new SignIn();
26.             form.Show();
27.         }
28.
29.         private void button3_Click(object sender, EventArgs e)
30.         {
31.             Application.Exit();
32.         }
33.
34.         private void button4_Click(object sender, EventArgs e)
35.         {
36.           SignInADMIN form = new SignInADMIN();
37.             form.Show();
```

```
38.             this.Visible= false;
39.         }
40.     }
41.
```

SignIn class

```
 1. public partial class SignIn : Form
 2.     {
 3.
 4.         private UserCredentials creds;
 5.         public SignIn()
 6.         {
 7.             creds = new UserCredentials();
 8.             InitializeComponent();
 9.         }
10.
11.         private void SignIn_Load(object sender, EventArgs e)
12.         {
13.
14.         }
15.
16.         private void button1_Click(object sender, EventArgs e)
17.         {
18.             if (Username.Text != string.Empty || Password.Text != string.Empty)
19.             {
20.                 if (creds.CheckForUserCreds(Username.Text, Password.Text))
21.                 {
// Checks that the fields are not empty and the username and password is valid.
22.                     UserCredsDB.UserID = UserCredentials.GetUserId(Username.Text);
23.                     UserCredsDB.WalletID = UserCredentials.GetUserWalletID();
24.                     MessageBox.Show($"Welcome {Username.Text}", "WELCOME", MessageBoxButtons.OK);
25.
26.                     Dashboard form = new Dashboard();
27.                     this.Visible = false;
28.                     form.Show();
29.
30.                 }
31.                 else
32.                 {
33.                     MessageBox.Show("No account available with these credientials");
34.
35.                 }
36.
37.
38.             }
39.
```

```
40.         }
41.
42.         private void button2_Click(object sender, EventArgs e)
43.         {
44.             WelcomePage pg = new WelcomePage();
45.             pg.Show();
46.             this.Visible = false;
47.         }
48.     }
49.
```

SignUp class

```
 1. public partial class SignUp : Form
 2.     {
 3.         private UserCredentials creds;
 4.         public SignUp()
 5.         {
 6.             creds = new UserCredentials();
 7.             InitializeComponent();
 8.         }
 9.
10.         private void Sin_Click(object sender, EventArgs e)
11.         {
12.
13.             if (CheckEmptyFields(UsernameField, PasswordField, ConfirmPassword) == false)
14.             {
15.                 if (!creds.CheckUserAlreadyExists(UsernameField.Text))
16.                 {
17.                     if (CheckSignCorrect(PasswordField.Text, ConfirmPassword.Text) == true)
18.                     {
19.                         creds.InitialiseAccount(UsernameField.Text, PasswordField.Text);
20.                         creds.CreateWallet(UsernameField.Text);
21.                         MessageBox.Show("Account Created!");
22.                         this.Visible = false;
23.
24.                         WelcomePage form = new WelcomePage();
25.                         form.Show();
26.
27.
28.                     }
29.                     else
30.                     {
31.                         MessageBox.Show("Passwords don't match each other. Please re-enter password", "Fault",
MessageBoxButtons.OK);
32.
33.                     }
34.
35.
36.                 }
37.                 else
38.                 {
39.                     MessageBox.Show("Username already exists. Please try again", "Fault", MessageBoxButtons.OK);
```

```
40.
41.                 }
42.
43.
44.             }
45.             else
46.             {
47.
48.                 MessageBox.Show("One or more fields are empty", "Error", MessageBoxButtons.OK);
49.
50.             }
51.
52.
53.
54.
55.
56.
57.
58.         }
59.
60.         private bool CheckSignCorrect(string Password, string CorrectPassword)
61.         {
62.             return Password == CorrectPassword;
63.
64.
65.         }
66.         private bool CheckEmptyFields(TextBox b1, TextBox b2, TextBox b3)
67.         {
68.             if (b1.Text == string.Empty || b2.Text == string.Empty || b3.Text == string.Empty)
69.             {
70.                 return true;
71.
72.             }
73.             else
74.             {
75.                 return false;
76.             }
77.
78.
79.
80.         }
81.
82.         private void button1_Click(object sender, EventArgs e)
83.         {
84.             WelcomePage pg = new WelcomePage();
85.             pg.Show();
86.             this.Visible = false;
87.         }
88.     }
89.
```

SignInADMIN class

```
1. public partial class SignInADMIN : Form
2.     {
3.         private static int tries = 3;
4.         public SignInADMIN()
5.         {
6.             InitializeComponent();
7.
8.         }
9.
10.        private void button1_Click(object sender, EventArgs e)
11.        {
12.            if (SignInADMIN.tries == 0)
13.            {
14.                AdminLoginButton.Enabled= false;
15.                label2.Text = "Locked out of admin use.";
16.                Thread.Sleep(5000);
17.
18.
19.            }
20.            if (AdminPasswordField.Text == UserCredsDB.GetAdmin_PWD())
21.            {
22.                Reports reports = new Reports();
23.                reports.Show();
24.                this.Visible = false;
25.            }
26.            else
27.            {
28.                MessageBox.Show($"Incorrect password", "Error", MessageBoxButtons.OKCancel);
29.                SignInADMIN.tries --;
30.
31.
32.            }
33.        }
34.
35.        private void button2_Click(object sender, EventArgs e)
36.        {
37.            WelcomePage pg = new WelcomePage();
38.            pg.Show();
39.            this.Visible=false;
40.        }
41.
42.        private void SignInADMIN_Load(object sender, EventArgs e)
43.        {
44.
45.        }
46.    }
47.
```

Report class

```
 1. public partial class Reports : Form
 2.     {
 3.         private UserCredentials creds;
 4.         public Reports()
 5.         {
 6.             InitializeComponent();
 7.            creds= new UserCredentials();
 8.         }
 9.
10.         private void Reports_Load(object sender, EventArgs e)
11.         {
12.             this.UsernameLBL.DropDownStyle = ComboBoxStyle.DropDownList;
13.             this.ChoiceLBL.DropDownStyle = ComboBoxStyle.DropDownList;
14.             ChoiceLBL.Items.Add("Orders");
15.             ChoiceLBL.Items.Add("Portfolio");
16.             ChoiceLBL.Items.Add("Delete User");
17.             NumberUserLBL.Text = Convert.ToString(UserCredsDB.GetUserCount());
18.             List<string> Usernames = UserCredsDB.GetUsernames();
19.
20.             foreach (string username in Usernames)
21.             {
22.                 UsernameLBL.Items.Add(username);
23.
24.
25.             }
26.
27.         }
28.
29.         private void checkedListBox1_SelectedIndexChanged(object sender, EventArgs e)
30.         {
31.
32.         }
33.
34.         private void button1_Click(object sender, EventArgs e)
35.         {
36.             if (ChoiceLBL.Text == "Orders")
37.             {
38.                 using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
39.                 {
40.                     conn.Open();
41.
42.                     SQLiteDataAdapter adp = new SQLiteDataAdapter($"SELECT Ticker, Quantity, PriceAtTransaction,
TransactionTime, TransactionPrice, TransactionType FROM Orders WHERE wallet_ID =
{UserCredentials.GetUserId(UsernameLBL.Text)}", conn);
43.
44.                     DataSet ds = new System.Data.DataSet();
45.                     adp.Fill(ds);
46.                     ReportGrid.DataSource = ds.Tables[0];
47.                     conn.Close();
48.
49.                 }
50.
51.
52.             }
53.             else if (ChoiceLBL.Text == "Portfolio")
54.             {
55.                 using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
56.                 {
57.                     conn.Open();
58.
59.                     SQLiteDataAdapter adp = new SQLiteDataAdapter($"SELECT Ticker, NumberOfShares, MoneyInvested
FROM Portfolio WHERE userID = {UserCredentials.GetUserId(UsernameLBL.Text)}", conn);
60.
61.                     DataSet ds = new System.Data.DataSet();
62.                     adp.Fill(ds);
63.                     ReportGrid.DataSource = ds.Tables[0];
64.                     conn.Close();
65.
66.                 }
67.
68.
69.             }
70.             else
71.             {
72.                 DialogResult d = MessageBox.Show($"Are you sure you would like to Delete the account
{UsernameLBL.Text}?", "WARNING", MessageBoxButtons.YesNo);
73.                 if (d == DialogResult.Yes)
74.                 {
75.
76.                     creds.DeleteUser(UsernameLBL.Text);
77.                     MessageBox.Show("Account deleted successfully", "Complete", MessageBoxButtons.OK);
78.                     Reports Reportform = new Reports();
79.                     this.Close();
80.                     Reportform.Refresh();
```

```
                    Reportform.Show();
                }


            }


        }

        private void button2_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }

        private void button3_Click(object sender, EventArgs e)
        {
            WelcomePage pg = new WelcomePage();
            pg.Show();
            this.Visible = false;
        }

        private void UsernameLBL_SelectedIndexChanged(object sender, EventArgs e)
        {

        }
    }

```

PortfolioPage class

```
public partial class PortfolioPage : Form
    {
       private PortfolioManager pm;
       private Transactions transaction;


        public PortfolioPage()
        {
            transaction = new Transactions();
            pm = new PortfolioManager();
            InitializeComponent();
        }

        private void PortfolioPage_Load(object sender, EventArgs e)
        {
            this.InvestedStocksList.DropDownStyle = ComboBoxStyle.DropDownList;
            this.StartPosition = FormStartPosition.CenterScreen;
            BalanceLBL.Text = Convert.ToString($"${TransactionDB.GetAccountBalance()}");
            DisplayOrders();
            if (PortfolioDB.CheckForPortfolioData())
            {

                List<string> tickers = pm.GetInvestmentTickers();

                foreach (string ticker in tickers)
                {
                    InvestedStocksList.Items.Add(ticker);

                }


            }
            else
            {
                SearchButton.Enabled= false;
                PortfolioViewButton.Enabled = false;

            }

        }

        private void DisplayOrders()
        {
            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
```

```
47.
48.                 SQLiteDataAdapter adp = new SQLiteDataAdapter($"SELECT Ticker, Quantity, PriceAtTransaction,
TransactionTime, TransactionPrice, TransactionType FROM Orders NOLOCK WHERE wallet_ID = {UserCredsDB.WalletID}",
conn);
49.                 DataSet ds = new System.Data.DataSet();
50.                 adp.Fill(ds);
51.                 dataGridView1.DataSource = ds.Tables[0];
52.                 conn.Close();
53.
54.
55.
56.             }
57.
58.
59.         }
60.         private void button2_Click(object sender, EventArgs e)
61.         {
62.             Dashboard form = new Dashboard();
63.             form.Show();
64.             this.Visible = false;
65.         }
66.
67.         private void label5_Click(object sender, EventArgs e)
68.         {
69.
70.         }
71.         private void timer1_Tick(object sender, EventArgs e)
72.         {
73.
74.         }
75.         private void pieChart1_Load(object sender, EventArgs e)
76.         {
77.
78.         }
79.
80.         private void button5_Click(object sender, EventArgs e)
81.         {
82.
83.             DialogResult d = MessageBox.Show("Would you like to make this payment?", "WARNING",
MessageBoxButtons.YesNo);
84.             if (d == DialogResult.Yes)
85.             {
86.                 Order Order = new Order(TickerBox.Text, Convert.ToInt32(QuantityBox.Value), DateTime.Now,
"SELL");
87.                 Transactions TRANSACTION = new Transactions(Order);
88.                 TRANSACTION.SellStock();
89.                 PortfolioPage pg = new PortfolioPage();
90.                 //dataGridView1.Update();
91.                 pg.Show();
92.                 this.Visible = false;
93.             }
94.             else
95.             {
96.
97.
98.             }
99.
100.
101.         }
102.
103.         private void label1_Click(object sender, EventArgs e)
104.         {
105.
106.         }
107.
108.         private void button7_Click(object sender, EventArgs e)
109.         {
110.             string ticker = InvestedStocksList.Text;
111.             if(InvestedStocksList.SelectedItem != null)
112.             {
113.                 ProportionLBL.Text = $"{Math.Round(pm.GetProportion(ticker), 2)}%";
114.                 MoneyInvestedLabel.Text = $"${Math.Round(pm.GetMoneyInvestedForTicker(ticker), 2)}";
115.                 ShareLabel.Text = Convert.ToString(pm.GetSharesForTicker(ticker));
116.
117.
118.             }
119.             else
120.             {
121.                 MessageBox.Show("No ticker selected", "Error", MessageBoxButtons.OKCancel);
122.
123.
124.             }
125.
126.
127.
```

```
128.
129.         }
130.
131.         private void panel4_Paint(object sender, PaintEventArgs e)
132.         {
133.
134.         }
135.
136.         private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
137.         {
138.
139.         }
140.
141.         private void panel1_Paint(object sender, PaintEventArgs e)
142.         {
143.
144.         }
145.
146.         private void button6_Click(object sender, EventArgs e)
147.         {
148.             if (PortfolioDB.CheckForPortfolioData())
149.             {
150.                 List<string> MaxShares = pm.GetMaxSharesForStock();
151.                 List<string> MaxInvestment = pm.GetMostExpensiveInvestment();
152.                 LargestShares.Text = $"Stock: {MaxShares[0]}, Number: {MaxShares[1]}";
153.                 MostExpensiveInvestment.Text = $"Stock: {MaxInvestment[0]}, Price: ${MaxInvestment[1]}";
154.                 string ProfitText = Convert.ToString(Math.Round(pm.GetProfitOrLoss(), 2));
155.                 if (ProfitText[0] == '-')
156.                 {
157.                     ProfitTicker.ForeColor = Color.Red;
158.
159.                 }
160.                 else
161.                 {
162.                     ProfitTicker.ForeColor = Color.LawnGreen;
163.
164.                 }
165.                 ProfitTicker.Text = $"${ProfitText}";
166.
167.             }
168.             else
169.             {
170.                 List<string> MaxInvestment = pm.GetMostExpensiveInvestment();
171.                 MostExpensiveInvestment.Text = $"Stock: {MaxInvestment[0]}, Price: ${MaxInvestment[1]}";
172.                 LargestShares.Text = $"No Stocks Invested In";
173.                 ProfitTicker.Text = "No Stocks Currently Invested In. No Profit/Loss";
174.
175.
176.             }
177.
178.         }
179.
180.         private void button4_Click(object sender, EventArgs e)
181.         {
182.             ObjectivesUI ui = new ObjectivesUI();
183.             ui.Show();
184.             this.Visible = false;
185.         }
186.
187.         private void button3_Click(object sender, EventArgs e)
188.         {
189.             GeometricBrownianMotion gm = new GeometricBrownianMotion();
190.             gm.Show();
191.             this.Visible = false;
192.         }
193.
194.         private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
195.         {
196.
197.         }
198.     }
199.
```

ObjectivesUI class

```
 1. public partial class ObjectivesUI : Form
 2.     {
 3.         private ObjectiveList obj;
 4.         public static string title; // Use of public static variables to hold the data of the selected objective
to transfer to the next form in EditObjectives.
 5.         public static string Description;
```

```
 6.          public static string priority;
 7.          public static DateTime deadline;
 8.          public static bool status;
 9.          private DateTime DATECHECKER;
10.
11.
12.          public ObjectivesUI()
13.          {
14.              obj = new ObjectiveList();
15.              InitializeComponent();
16.          }
17.
18.          private void ObjectivesUI_Load(object sender, EventArgs e)
19.          {
20.              this.PriorityLBL.DropDownStyle = ComboBoxStyle.DropDownList;
21.              this.StartPosition = FormStartPosition.CenterScreen;
22.              if (ObjectiveDB.CheckForObjectiveData()) // Only if a user has data present will it present any
objectives - prevents exception
23.               {
24.                  DisplayObjectives();
25.
26.              }
27.
28.              PriorityLBL.Items.Add("Critical");
29.              PriorityLBL.Items.Add("Moderate");
30.              PriorityLBL.Items.Add("Not Important");
31.              label5.Text = $"${TransactionDB.GetAccountBalance()}";
32.
33.          }
34.          private void DisplayObjectives()
35.          {
36.              List<Objective> Objectives = obj.GetObjectives(); // Gets the list of objectives from the database
37.              foreach (Objective OBJECTIVE in Objectives)
38.              {
39.                  obj.Enqueue(OBJECTIVE); // Enqueues the objectives into the priority queue.
40.              }
41.
42.              List<Objective> menuObj = obj.ReturnOBJ(); // returns the objectives in the queue into another
objective list
43.              int i = 0;
44.              foreach (Objective OBJECTIVE in menuObj)
45.              {
46.                  if (!OBJECTIVE.GetStatus())
47.                  {
48.                      string[] row = new string[] { OBJECTIVE.GetTitle(), OBJECTIVE.GetTask(),
OBJECTIVE.GetStrPriority(), Convert.ToString(OBJECTIVE.GetDeadline()), Convert.ToString(OBJECTIVE.GetStatus()) };
49.                      ObjectiveViewer.Rows.Insert(i, row);
50.                      i++;
51.
52.                  }
53.
54.
55.              }
56.              ObjectiveViewer.Show();
57.              ObjectiveViewer.Visible = true;
58.          }
59.          private void toolStripMenuItem1_Click(object sender, EventArgs e)
60.          {
61.
62.          }
63.
64.          private void addToolStripMenuItem_Click(object sender, EventArgs e)
65.          {
66.
67.          }
68.
69.          private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
70.          {
71.
72.          }
73.          private void button7_Click(object sender, EventArgs e)
74.          {
75.              if (string.IsNullOrEmpty(TitleLBL.Text) || string.IsNullOrEmpty(DescriptionLBL.Text) ||
string.IsNullOrEmpty(DeadlineLBL.Text) || string.IsNullOrEmpty(PriorityLBL.Text))
76.              {
77.                  MessageBox.Show("One or more fields are empty! ", "Error", MessageBoxButtons.OKCancel);
78.
79.
80.              }
81.              if (!DateTime.TryParse(DeadlineLBL.Text, out DATECHECKER)) // Tries parsing the input date to check
if it is a valid date.
82.              {
83.                  MessageBox.Show("Not a valid deadline, Please try Again");
84.              }
85.              else
```

```
            {
                try
                {

                    Objective OBJ = new Objective()
                    {
                        Title = TitleLBL.Text,
                        task = DescriptionLBL.Text,
                        Deadline = Convert.ToDateTime(DeadlineLBL.Text),
                        Status = false,
                        Priority = PriorityLBL.Text,

                    };
                    obj.AddObjective(OBJ);
                    MessageBox.Show("Objective Successfully Added", "Confirmed", MessageBoxButtons.OKCancel);
                    ObjectivesUI ui = new ObjectivesUI();
                    ui.Show();
                    this.Visible = false;
                }
                catch (Exception ex)
                {

                    MessageBox.Show($"Objective Unsuccessfully Added", "Error", MessageBoxButtons.OKCancel);

                }

            }
        }

        private void button1_Click(object sender, EventArgs e)
        {
            if (ObjectiveViewer.SelectedRows.Count == 0)
            {
                MessageBox.Show("Row hasn't been selected", "Error", MessageBoxButtons.OK);


            }
            else
            {
                if (ObjectiveViewer.Rows.Count > 0)
                {

                    DataGridViewRow selectedrow = ObjectiveViewer.SelectedRows[0]; // The selected row is
collected to gather all info through an object
                    ObjectivesUI.title = selectedrow.Cells[0].Value.ToString(); //
                    ObjectivesUI.Description = selectedrow.Cells[1].Value.ToString();
                    ObjectivesUI.priority = selectedrow.Cells[2].Value.ToString();
                    ObjectivesUI.deadline = Convert.ToDateTime(selectedrow.Cells[3].Value);
                    ObjectivesUI.status = Convert.ToBoolean(selectedrow.Cells[4].Value);
                    EditObjectives editObj = new EditObjectives();
                    editObj.Show();
                    this.Visible = false;
                }
                else
                {
                    MessageBox.Show("No Objective has been selected to edit. Please try again", "Error",
MessageBoxButtons.OK);

                }

            }



        }

        private void button5_Click(object sender, EventArgs e)
        {

        }

        private void PortfolioClick_Click(object sender, EventArgs e)
        {

        }

        private void button9_Click(object sender, EventArgs e)
        {

        }

        private void button2_Click(object sender, EventArgs e)
        {
            Dashboard form = new Dashboard();
            this.Visible = false;
```

```
169.             form.Show();
170.         }
171.
172.         private void PortfolioClick_Click_1(object sender, EventArgs e)
173.         {
174.             PortfolioPage form = new PortfolioPage();
175.             this.Visible = false;
176.             form.Show();
177.         }
178.
179.         private void button3_Click(object sender, EventArgs e)
180.         {
181.             GeometricBrownianMotion form = new GeometricBrownianMotion();
182.             this.Visible = false;
183.             form.Show();
184.         }
185.
186.         private void panel1_Paint(object sender, PaintEventArgs e)
187.         {
188.
189.         }
190.
191.         private void textBox2_TextChanged(object sender, EventArgs e)
192.         {
193.
194.         }
195.
196.         private void Exit_Click(object sender, EventArgs e)
197.         {
198.             Application.Exit();
199.         }
200.
201.         private void dataGridView1_CellContentClick_1(object sender, DataGridViewCellEventArgs e)
202.         {
203.
204.         }
205.
206.         private void MinScreen_Click(object sender, EventArgs e)
207.         {
208.
209.         }
210.
211.         private void label1_Click(object sender, EventArgs e)
212.         {
213.
214.         }
215.
216.         private void button4_Click(object sender, EventArgs e)
217.         {
218.
219.         }
220.     }
221.
```

EditObjectives class

```
1. public partial class EditObjectives : Form
2.     {
3.         private DateTime DATECHECKER;
4.         public EditObjectives()
5.         {
```

```
 6.
 7.              InitializeComponent();
 8.
 9.
10.          }
11.
12.          private void label4_Click(object sender, EventArgs e)
13.          {
14.
15.          }
16.
17.          private void EditObjectives_Load(object sender, EventArgs e)
18.          {
19.              this.PriorityLBL.DropDownStyle = ComboBoxStyle.DropDownList;
20.              PriorityLBL.Items.Add("Critical");
21.              PriorityLBL.Items.Add("Moderate");
22.              PriorityLBL.Items.Add("Not Important");
23.              TitleLBL.Text = ObjectivesUI.title;
24.              DescriptionLBL.Text = ObjectivesUI.Description; // Gets the information contained within the static
properties that contain the objective to edit.
25.              PriorityLBL.Text = ObjectivesUI.priority;
26.              DeadlineLBL.Text = Convert.ToString(ObjectivesUI.deadline);
27.              CompletedLBL.Checked = Convert.ToBoolean(ObjectivesUI.status);
28.          }
29.
30.          private void button1_Click(object sender, EventArgs e)
31.          {
32.              ObjectivesUI ui = new ObjectivesUI();
33.              ui.Show();
34.              this.Visible = false;
35.          }
36.
37.          private void button2_Click(object sender, EventArgs e)
38.          {
39.              if (!DateTime.TryParse(DeadlineLBL.Text, out DATECHECKER))
40.              {
41.                  MessageBox.Show("Not a valid Deadline", "Try Again", MessageBoxButtons.OK);
42.
43.
44.              }
45.              else
46.              {
47.                  try
48.                  {
49.                      string status = Convert.ToString(CompletedLBL.Checked.ToString().ToLower());
50.                      Objective obj = new Objective()
51.                      {
52.                          Title = TitleLBL.Text,
53.                          task = DescriptionLBL.Text,
54.                          Priority = PriorityLBL.Text,
55.                          Deadline = Convert.ToDateTime(DeadlineLBL.Text),
56.                          ObjectiveID = ObjectiveDB.GetOBJID(),
57.                          Status = Convert.ToBoolean(status)
58.
59.
60.                      };
61.                      ObjectiveDB.EditObj(obj);
62.                      ObjectivesUI form = new ObjectivesUI();
63.                      form.Show();
64.                      this.Visible = false;
65.
66.                  }
67.                  catch (Exception exc)
68.                  {
69.                      MessageBox.Show("Edit Unsuccessful", "Error", MessageBoxButtons.OK, MessageBoxIcon.Error);
70.
71.                  }
72.
73.              }
74.
75.
76.
77.
78.
79.
80.
81.          }
82.      }
83.
```

GeometricBrownianMotion Class

```
1. public partial class GeometricBrownianMotion : Form
2.     {
3.         private API AC;
4.         public GeometricBrownianMotion()
5.         {
6.             InitializeComponent();
7.             AC = new API();
8.         }
9.
10.        private void button2_Click(object sender, EventArgs e)
11.        {
12.            Dashboard form = new Dashboard();
13.            form.Show();
14.            this.Visible = false;
15.
16.        }
17.        private void GeometricBrownianMotion_Load(object sender, EventArgs e)
18.        {
19.
20.
21.        }
22.        private async void button7_Click(object sender, EventArgs e)
23.        {
24.            try
25.            {
26.                double Price = await AC.GetCurrentPrice(StockLBL.Text);
27.                if (Price == 0) // Checks if the ticker is valid from the API resposne.
28.                {
29.                    MessageBox.Show("Invalid Ticker", "Error", MessageBoxButtons.OKCancel);
30.
31.                }
32.                else
33.                {
34.                    ReferencePrice.Text = Convert.ToString(Price);
35.                    PriceAtStart.Text = $"Price of {StockLBL.Text}: $";
36.
37.
38.                }
39.
40.            }
41.            catch (Exception exception)
42.            {
43.                MessageBox.Show("Invalid Ticker or Invalid field", "Error", MessageBoxButtons.OKCancel);
44.
45.            }
46.
47.        }
48.
49.        private void numericUpDown8_ValueChanged(object sender, EventArgs e)
50.        {
```

```
51.
52.         }
53.
54.         private void label1_Click(object sender, EventArgs e)
55.         {
56.
57.         }
58.         private bool CheckValidFields()
59.         {
60.             if (SimNum.Value != 0 && DayNumLBL.Value != 0 && ReferencePrice.Text != String.Empty &&
Convert.ToDouble(VolatilityLBL.Text) > 0 && Convert.ToDouble(DriftLBL.Text) > 0 &&
Convert.ToDouble(VolatilityLBL.Text) <= 100 && Convert.ToDouble(DriftLBL.Text) <= 100 && VolatilityLBL.Text !=
String.Empty && DriftLBL.Text != String.Empty)
61.             {
62.                 return true;
63.             }
64.             else
65.             {
66.
67.                 return false;
68.             }
69.
70.
71.         }
72.         private void button5_Click(object sender, EventArgs e)
73.         {
74.             SimulationTable.DataSource = null;
75.             SimulationTable.Rows.Clear();
76.             SimulationTable.Columns.Clear();
77.             if (CheckValidFields()) // Checks if fields entered are valid.
78.             {
79.                 GenerateRandomNums randnums = new GenerateRandomNums(Convert.ToInt32(SimNum.Value),
Convert.ToInt32(DayNumLBL.Value)); // Creates instance for generating the 2D random number array.
80.                 double[,] RandomArray = randnums.GenerateRandomNumbers(); // Generates random numbers and
assigns it to RandomArray
81.                 SimulateGBM sim = new SimulateGBM(RandomArray, Convert.ToDouble(ReferencePrice.Text),
Convert.ToDouble(VolatilityLBL.Text), Convert.ToDouble(DriftLBL.Text)); // Creates instance of simulation holding the
user entered constants.
82.                 sim.Simulate(); // Simulates the brownian motion
83.                 double[,] PRICES = sim.returnSimulation(); // Returns Simulation results.
84.                 DataTable table = new DataTable();
85.
86.                 table.Columns.Add("Sim Num", typeof(int));
87.                 table.Columns.Add("Day", typeof(int));
88.                 table.Columns.Add("Stock Price", typeof(double));
89.                 double[] vals = new double[PRICES.Length];
90.                for (int i = 1; i < PRICES.GetLength(0); i++)
91.                 {
92.                     for (int j = 0; j < PRICES.GetLength(1); j++)
93.                     {
94.                         table.Rows.Add(i + 1, j + 1, PRICES[i, j]);
95.                         vals[i] = PRICES[i, j];
96.                     }
97.                 }
98.
99.                StatsLabel.Text = $"Simulation Statistics for {StockLBL.Text}";
100.               SimulationTable.DataSource = table;
101.
102.
103.
104.            }
105.            else
106.            {
107.                MessageBox.Show("Invalid Fields entered, Please try again", "Error",
MessageBoxButtons.OKCancel);
108.
109.
110.            }
111.
112.
113.
114.
115.        }
116.
117.        private void panel4_Paint(object sender, PaintEventArgs e)
118.        {
119.
120.        }
121.
122.        private void button1_Click(object sender, EventArgs e)
123.        {
124.
125.
126.
127.        }
```

```
128.
129.         private void GeometricBrownianMotion_Load_1(object sender, EventArgs e)
130.         {
131.
132.             this.StartPosition = FormStartPosition.CenterScreen;
133.         }
134.
135.         private void panel6_Paint(object sender, PaintEventArgs e)
136.         {
137.
138.         }
139.
140.         private void dataGridView1_CellContentClick(object sender, DataGridViewCellEventArgs e)
141.         {
142.
143.         }
144.
145.         private void button4_Click(object sender, EventArgs e)
146.         {
147.
148.         }
149.     }
150.
```

Design Implementations and justifications

Within this section, I will explain about various complex parts of my section related to my classes and the techniques and data structures implemented within them.

The sections which I aim to cover through this section are:

1) Priority Queue Implementation
2) API Class Structure
3) Geometric Brownian Motion Simulation
4) Database Classes and Queries

Priority Queue Implementation

Within my priority queue implementation, I implemented both a class called *Objective* which would contain the to do objective data and a class called *ObjectiveList* which would aim to order and sort each item within the priority queue.

Within the start of the ObjectiveList class:

```
internal class ObjectiveList : ObjectiveDB
    {
        private List<Objective> Objectives;

        public ObjectiveList()
        {
            Objectives = new List<Objective>();

        }
    }

```

We can clearly see that the class itself inherits from the objectiveDB class. This is a clear theme within other various classes which have its own database counterpart as since the database classes are structured as abstract, the main role of the class is to provide the methods and functions for the child class. The main role of this inheritance is to provide easy access to the previously made objectives from the users so they can be easily accessed and processed better when applying them into the priority queue.

For the property *Objectives*, we have utilised composition to act as a container to hold each item within the objective.

Later within the class, we are also introduced to other methods contained below:

```
public List<Objective> GetObjectives()
        {
            return GetObjectivesFromDB();

        }
        public List<Objective> ReturnOBJ()
        {
            return Objectives;
        }

        public void AddObjective(Objective obj)
        {
            AddObjectiveToDB(obj);

        }

```

My main aim with not just this class but other classes was to provide a separation between the front-end of the user interface and the backend of the database. As a result, creating these intermediate classes enabled me to provide easy access and manipulation of user interactions.

Within the *GetObjectives*() method, we are retrieving the previously contained objectives from the database which is both used when commencing the priority queue algorithm and also adding the objectives into the data table for the user to see.

For the ReturnOBJ() method, it will be used to return the ordered priority queue after the algorithm has been completed.

For the AddObjective() Method, it will be used to add an objective to the database when a user makes one within the add objective part of the menu.

Since I have another constraint to determine an objectives position within the priority queue, I used the method:

```
private bool CompareDates(DateTime t1, DateTime t2)
        {
            if (DateTime.Compare(t1, t2) < 0)
            {
                return true;

            }
            else
            {
                return false;

            }
        }

```

I have made this method private to encapsulate the method within the class to compare the dates of two objectives to measure if the one inserted was higher or lower within the queue.

<u>Enqueue Method</u>

```
1. public void Enqueue(Objective objective)
2.         {
3.             int placement = 0;
4.             bool OkPosition = false;
5.
6.             while (placement < Objectives.Count && OkPosition == false)
7.             {
8.                 if (objective.GetIntPriority() > Objectives[placement].GetIntPriority())
9.                 {
10.                    OkPosition = true;
11.
12.                }
13.                else if (objective.GetIntPriority() == Objectives[placement].GetIntPriority())
14.                {
15.                    if (CompareDates(objective.GetDeadline(), Objectives[placement].GetDeadline()))
16.                    {
17.                        OkPosition = true;
18.
19.                    }
20.                    else
21.                    {
22.                        placement++;
23.
24.                    }
25.
26.                }
27.                else
28.                {
29.                    placement++;
30.
31.                }
32.
33.            }
34.
35.            Objectives.Insert(placement, objective);
36.        }
37.
```

Algorithm Walkthrough

1) Initialise a **placement** variable to 0 and place **OkPosition** to false – this Boolean variable determines if the correct position has been met.
2) A while loop commences and keeps running until the end of the queue has been met or the position has not yet been determined.
3) In each iteration of the while loop, it compares the priority of the soon to be inserted objective with the **Objective** object at the current **placement** index.
4) If the priority of the new objective to be inserted is higher than the objective at index, **placement**, then the new **Objective** object should be inserted at this position.
5) If the priorities of the two **Objective** objects are equal, then it compares their respective deadlines. If the deadline of the objective to be inserted is earlier than the deadline of the **Objective** object at the current **placement** index, it takes higher priority and is placed in that position in front of the compared objective.
6) The variable placement is then incremented if neither of the conditions are met. This signifies to keep on moving.
7) The **OkPosition** variable is then set to the respective true/false if the placement is/is not in the right position.
8) When the appropriate placement index is found, the new **Objective** object is inserted into the priority queue using the Insert() method.

Use of Client-Server interface and Web API implementation

Within this region of my program, I aimed to implement both a container to hold each data point of each stock in each time interval and the API class itself.

Within the **Values** class:

```
 1. public class Values
 2.     {
 3.         private string date;
 4.         private double high;
 5.         private double low;
 6.         private double open;
 7.         private double close;
 8.         private double PreviousClose;
 9.         private double PercentChange;
10.
11.         public Values(string date, double high, double low, double open, double close, double PreviousClose,
double PercentChange)
12.         {
13.             this.date = date;
14.             this.high = high;
15.             this.low = low;
16.             this.open = open;
17.             this.close = close;
18.             this.PercentChange = PercentChange;
19.             this.PreviousClose = PreviousClose;
20.
21.         }
22.         public Values()
23.         {
24.
25.
26.
27.         }
28.
29.         public string GetDate()
30.         {
31.             return date;
32.
33.
34.         }
35.         public double GetHigh()
36.         {
37.             return high;
38.
39.         }
40.         public double GetLow()
41.         {
42.             return low;
43.
44.         }
45.         public double GetOpen()
46.         {
47.             return open;
48.
49.         }
50.         public double GetClose()
51.         {
52.             return close;
53.
54.         }
55.
56.         public double GetPreviousClose()
57.         {
58.             return PreviousClose;
59.
60.         }
61.
62.         public double GetPercentChange()
63.         {
64.             return PercentChange;
65.
66.         }
67.
68.     }
69.
```

Again, with this class, I have utilised encapsulation within privatising the properties providing that layer of abstraction to my program. As a result, to

complement, I implemented the getter methods which provided a way to gather the information of the objects state in a controlled manner.

Within the beginning of the API class, we initialise the class and two variables:

```
internal class API
    {

        private List<Values> values;
        private static readonly string API_FILEPATH =
$"{Environment.GetFolderPath(Environment.SpecialFolder.UserProfile)}\\OneDrive - Bradford Grammar School\\Home
Drive\\computer science\\Year 13 - Year 1\\NEA 10JAN\\testtttt\\NEA - 19DEC\\NEA\\API_KEY.txt";

        public API()
        {
            values = new List<Values>();

        }

```

Through the List Values, I have utilised composition as for the list itself, it will hold each data point for a certain date over a time. We can see that I have initialised the list of the object Values.

The API_FILEPATH also plays a crucial role as a way of accessing the API key from the provider but instead placing it within the source code which would place some integrity problems, I have included defensive programming to access this key via text file within the user's directory. I have also made this parameterised making it universal for all users. I have also made this *readonly* so it can remain constant and prevent change during runtime.

Within the API class, one of the main methods is the *Prices* method:

```
public async Task<List<Values>> Prices(string ticker, string startdate, string enddate, string interval)
            // Used async methods and a task to utltise a separate thread just for the API response - provides
quicker interface
        {
            HttpClient client = new HttpClient(); // Creates new client to link with web API
            var request = new HttpRequestMessage
            {
                Method = HttpMethod.Get,
                RequestUri = new
Uri($"https://api.twelvedata.com/time_series?symbol={ticker}&start_date={startdate}&end_date={enddate}&interval={inte
rval}&apikey={API.GetAPI_KEY()}&format=JSON"),
                // sends API request to retrieve stock data from API
            };
            using (var response = await client.SendAsync(request))
            {
                try
                {
                    response.EnsureSuccessStatusCode(); // checks if API request has sent data successfully
                    string JSON_response = await response.Content.ReadAsStringAsync(); // reads data as an
asynchronous string

                    JObject jobject = JObject.Parse(JSON_response); // Parses the string into JSON format for
easy access of data.
                    // TEST IF ARRAY EMPTY
                    if (Convert.ToInt32(jobject["code"])  != 400)
                    {
                        foreach (var val in jobject["values"])
                        {

                            values.Add(new Values(Convert.ToString(val["datetime"]),
Convert.ToDouble(val["high"]), Convert.ToDouble(val["low"]), Convert.ToDouble(val["open"]),
Convert.ToDouble(val["close"]), Convert.ToDouble(val["previous_close"]), Convert.ToDouble(val["percent_change"])));
                            // Adds each data point over a data into the values list
                        }
                        return values;
```

```
29.
30.                     }
31.                     else
32.                     {
33.                         return null;
34.
35.                     }
36.
37.
38.                 }
39.                 catch (NullReferenceException e)
40.                 {
41.
42.                     return null;
43.                     // Exception handling to check if user has entered a valid Stock ticker.
44.                 }
45.
46.             }
47.         }
48.
```

To which I have initialised the method name and its parameters, I have classified the method as being **async** and a **Task** of the object *Values.* I made this decision with the most integral part of the API being fluid and high performance and preventing or inhibiting the UI thread. This provided are far more responsive approach to the program and user experience.

The decision to use a task has well played very well as it allowed me to apply a separate thread which prevented long delays when there was any UI interaction. This allowed multiple concurrent operations ranging from the front-end of the user interactions to the backend operations within the API.

The program then uses an instance of a HTTP client to send a parameterised GET request to the API to retrieve the information required depending on the user requirements. The client then sends an async request to which from the response checks if the response was received successfully. It then parses the API response as a JSON formatted string to which we can easily retrieve the data via accessing the various indexes which hold the information. We then use the dynamic Values List and add each stock data point to it. By formatting the API request into JSON format allowed me to easily access each part of the response when storing it into the list.

With both the Trends and API class being quite linked together in terms of calculating the patterns in stock movements, I decided to make the method Initialise_timeseries virtual within the class:

```
 1. public virtual List<string> Initalise_timeseries(List<string> timeseries, string intervaltype, int option)
 2.             // Method Virtual to connect with Trends class to override method to create different time range.
 3.         {
 4.             // Creates a set of two dates to create a start and end to retrieve stock data from specified dates.
Adds dates to
 5.             try
 6.             {
 7.                 string dateTime = DateTime.Now.ToString();
 8.                 string now = Convert.ToDateTime(dateTime).ToString("yyyy-MM-dd");
 9.                 if (intervaltype == "month" && option > 0)
10.                 {
11.                     var minustime = DateTime.Today.AddMonths(-option);
12.                     string time = Convert.ToDateTime(minustime).ToString("yyyy-MM-dd");
13.                     timeseries.Add(now);
14.                     timeseries.Add(time);
15.                     return timeseries;
16.                 }
17.
```

```
18.                 else if (intervaltype == "day" && option > 0)
19.                 {
20.                     var minustime = DateTime.Today.AddDays(-option);
21.                     string time = Convert.ToDateTime(minustime).ToString("yyyy-MM-dd");
22.                     timeseries.Add(now);
23.                     timeseries.Add(time);
24.                     return timeseries;
25.                 }
26.                 else
27.                 {
28.                     return null;
29.
30.                 }
31.
32.
33.
34.             }
35.             catch (NullReferenceException ex)
36.             {
37.                 return null;
38.             }
39.
40.
41.
42.         }
43.
```

The reason for this is that when we calculate the MACD algorithm, there should be a fixed time interval and not one that can be altered. As a result, within the Trends class, it has become overridden to accommodate the change in time interval for receiving the stock data when calculating the MACD value when predicting stock prices.

Database Classes and Queries

One of the main behaviours with my database classes is the fact that some of the classes are abstract. The thinking behind is since it allows a layer of abstraction as it acts as a separation between the front-end and the back end. The front end can perform all the database operations e.g., update balance, updating number of shares without any need of knowing the underlying database structure.

Whereas with the back end, it can implement these methods for each specific class and becomes a parent for the child classes it inherits to.

With a core component of my project relating to the portfolio analysis of user's accounts, the use of parameterised SQL and aggregate functions was evident.

Within the method:

```
1. public double GetProfitOrLoss()
2.         {
3.             using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
4.             {
5.                 conn.Open();
```

```
                string SQL = $"SELECT SUM(TransactionPrice - MoneyInvested) FROM Portfolio portfolio JOIN Orders orders ON portfolio.Ticker = orders.Ticker WHERE portfolio.userID = {UserCredsDB.UserID} AND orders.TransactionType = 'BUY'";


                using (SQLiteCommand comm = new SQLiteCommand(SQL, conn))
                {
                    using (SQLiteDataReader _reader = comm.ExecuteReader())
                    {

                        _reader.Read();
                        double Profit = _reader.GetDouble(0);
                        conn.Close();
                        return Profit;

                    }
                }
            }
        }

```

We can see through the SQL query, we are joining the portfolio and order tables and calculating the sum of the transaction prices and the corresponding money spent for each stock. This allows us to find the difference which in turn gives us out profit. This was also very useful involvement of joining the tables orders and portfolio increasing query responsiveness.

With aggregate mathematical functions being very essential when calculating portfolio analysis, including a sole module for handling every possible combination seemed useful

Through the method:

```
private double GiveAggregateData(string FunctionType, string table, string column)
        {

            using (SQLiteConnection conn = new SQLiteConnection(TransactionDB.ConnStr))
            {
                conn.Open();
                string SQLQuery = $"SELECT {FunctionType}({column}) FROM {table} NOLOCK WHERE userID = {UserCredsDB.UserID}";

                using (SQLiteCommand comm = new SQLiteCommand(SQLQuery, conn))
                {
                    SQLiteDataReader _reader = comm.ExecuteReader();

                    while (_reader.Read())
                    {

                        double result = _reader.GetDouble(0); // not empty
                        conn.Close();
                        return result;


                    }
                    return -1;

                }

            }

        }

```

We can see through the method that we are performing:

```
$"SELECT {FunctionType}({column}) FROM {table} NOLOCK WHERE userID = {UserCredsDB.UserID}";
```

Through the query, we pass the parameters FunctionType, column and table as parameters for the method. The FunctionType constitutes to an aggregate function e.g., SUM, MAX, MIN, the column regards to field of interest and the table signifies which table they would like to choose. This played very efficiently within my code as I didn't need to keep writing functions to get me what specific aggregate data but instead, I could call the function with the respective information I wanted.

```
1. GiveAggregateData("MAX", "Portfolio", "NumberOfShares")
```

Through this function, we are getting the stock that we have invested in most.

Geometric Brownian Motion Simulation

The inner workings of this algorithm stems from generating the random numbers in a 2d array to then utilise this for simulating the stock prices.

Within the class GenerateRandomNums, we initialise 4 properties:

```
1. private int SimDays;
2. private int NumOfSims;
3. private double[,] RandomNumArray;
4. private Random RandomVal;
5.
```

In this case, the RandomNumArray will store the random numbers we generate within the method.

I then initialised a function to generate a random number taking in two uniform random numbers generated by the RandomVal instance. This performs the Box-Muller Transform.

```
1. private static double GenerateRandomNum(double u1, double u2)
2.         {
3.             return Math.Sqrt(-2.0 * Math.Log(1 - u1)) * Math.Sin(2.0 * Math.PI * (1 - u2)); // Box-Muller
Transform
4.         }
5.
```

This return module performs the formula onto the two random uniform numbers and returns the standardised version.

The main function which holds the loop is within the RandomNumbersGeneration() method:

```
public double[,] RandomNumbersGeneration()
        {
            RandomNumArray = new double[SimDays, NumOfSims];
            for (int Column = 1; Column < NumOfSims; Column++)
            {
                for (int Row = 1; Row < SimDays; Row++)
                {
                    double randval1 = 1.0 - RandomVal.NextDouble(); // Generates two uniform random numbers
                    double randval2 = 1.0 - RandomVal.NextDouble();

                    RandomNumArray[Row - 1, Column - 1] = GenerateRandomNum(randval1, randval2); // Asigns the standard normal random number to every index in the 2d array.
                }
            }

          return RandomNumArray;
 }
```

We can see that it performs a nested iteration with the x-matrix ranging from 1 to the number of simulations and the y-matrix ranging from 1 to the number of simulation days. For each nested iteration, it generates two random normal numbers and makes it uniform by doing 1 minus the value.

It then performs the Box-Muller transform onto the two numbers and stores the standard normal random number into the respective index references by the position in the index. It then returns the 2d array filled with the random values.

Within the simulation class a set of properties are again initialised with two corresponding methods:

```
private double Price0;
private double Mu;
private int SimulationDays;
private int SimulationNum;
private double[,] RandNumArray;
private double Sigma;
private double[,] SimulatedStockPrices;

private double initaliseMu(double mu)
{
     return mu / 100 / 252; // anualises the drift constant

}
private double initialiseSigma(double voltality)
 {
      return voltality / 100 / Math.Sqrt(252); // annualises the volatility constant.

 }

```

The two methods above correspond to the drift and volatility constants derived by the user input. The reason for the calculation is to annualise the values to be in a yearly format where dividing by 100 converts it to a percentage and dividing by 252 places it into a trading day value (252 trading days in a year).

```
 1. public SimulateGBM(double[,] RandArray, double PriceAt0, double drift, double volatility)
 2.         {
 3.             SimulationDays = RandArray.GetLength(0); // Gets interval points for both the sim days and number of
sims
 4.             SimulationNum = RandArray.GetLength(1);
 5.             RandNumArray = RandArray; // Has new instance of the 2d array to be assigned to the random array
generated earlier.
 6.             SimulatedStockPrices = new double[SimulationDays + 1, SimulationNum];
 7.             this.Price0 = PriceAt0;
 8.             Mu = initaliseMu(drift); // Initliased the drift and voltatlity constants entered by users to be
annualised for use
 9.             Sigma = initialiseSigma(volatlity);
10.
11.             for (int column = 1; column < SimulationNum+1; column++)
12.             {
13.                 SimulatedStockPrices[0, column - 1] = Price0; // Sets the first simulation as the initial price
so each simulation starts on the same data specified by user.
14.
15.             }
16.
17.         }
18.
```

The constructor contains arguments from the user input such as random array generated from the previous class, drift values and volatility values.

1) Sets the number of days as the x-matrix of the random array and the number of simulations as the y-matrix of the random array.
2) Initialises a second 2d array which will hold the simulated stock prices.
3) Annualises the drift and volatility constants using functions stated above.
4) Loops through the first column placing the initial price as the first index for each day in a simulation – this means each simulation starts on the same price.

The program then performs the Brownian Motion equation using the function GBMSimulation():

```
 1. public void GBMSimulation()
 2.         {
 3.             for (int sim = 0; sim < SimulationNum; sim++)
 4.             {
 5.                 for (int day = 1; day <= SimulationDays; day++)
 6.                 {
 7.                     SimulatedStockPrices[day, sim] = Math.Round( SimulatedStockPrices[day - 1, sim] +
SimulatedStockPrices[day - 1, sim] * Mu + SimulatedStockPrices[day - 1, sim] * Sigma * RandNumArray[day - 1, sim],
2); // Stochastic diffrential process.
 8.                     // follows the Geometric Brownian Motion equation.
 9.                 }
10.             }
11.
12.         }
13.
```

This performs a similar nested iteration to the generate random numbers function to which it assigns the index of the 2d array at position (day, sim) to the value calculated using the Brownian motion equation.

The function then returns return the simulation statistics through the function:

```
1. public double[,] returnGBMSimulation()
2.         {
3.             return SimulatedStockPrices;
4.         }
5.
```

Testing

The role of my Testing is to test both the integrity and functionality of my program to make sure the program is clean of errors and logic problems.

Below presents a URL which will go through a live walkthrough my program explaining the various tests and functionality with both the program and database incorporation.

https://youtu.be/IYRjPYVRNJw

Test Case	Objective	Test Data	Expected output	Actual Output	Reference	Pass /Fail
1	1.1	User can see and click on various options when logging in. **Normal**	Shows the options of login and sign up	Screen shows the options clearly	1	Pass
	1.2	User Creating account **Normal**.	Included in test case 2,3,4	Included in test case 2,3,4	Included in reference 2,3,4	Pass
2	1.2.1	When user creates account. **Normal**	User passwords should be hashed and stored in table.	Correct use of hashing and store within the table USERS.	2	Pass
3	1.2.3	User enters two different passwords when making and password and then confirming it. **Erroneous**	User should be prompted to re-enter password until both fields are equal when signing up.	Alerts are clearly present to if the user's password and confirm password field are different.	3	Pass
4	1.2.2	When user enters a set of a username and a password, and they are either valid or invalid according to the database. **Erroneous**	User's input should be hashed and compared with what is in the database. It should or should not provide access if this comparison is incorrect.	The code does perform this comparison and presents the user with a message saying if the password does/does not match within the database.	4	Pass
4.5	1.2.2.1	User clicks login **Normal**	User should be granted access to app.	User is allowed access with suitable welcome message.	4.5(a)	Pass
5	1.3.3.1	Admin user enters a password to access admin reports. **Normal/Erroneous**	User should ask user to enter admin password until correct.	Program keeps asking user to enter admin password for logging in.	5	Pass
6	1.3.1 and 1.3.2	User accessing the options to view	User should be able select an option to select Order history	Table clearly shows the user	6	Pass

		other user's portfolio Data. **Normal Data**	and portfolio. A table should show this data.	orders and portfolio, and users can switch between the users for their data.		
7	1.3.3	When an admin wants to delete a user account from the application. **Normal**	User should be able to delete a user from the application	User is successfully removed from the database and all records related in other tables	7	Pass
8	2.1.1	When a user tries and searches for a ticker with the stock name and times invalid. **Erroneous**	If user input isn't part of the NYSE or the time inputs are not correct, a suitable message should be presented.	Message is presented to user if the ticker inputted is not valid or the times are not valid.	8	Pass
9	2.1.3	When a user searches for a stock to look at its previous data for analysis. **Normal**	Every time a user makes a search, the ticker should be added to the search history where users can select the stock again.	Once a search has been done, the drop-down menu will contain the respective stock ticker which the user can select and make the search again	9	Pass
10	2.2 and 2.2.1	Whenever a user searches for a stock, the analysis should be presented. **Normal**	Program should display analysis to either buy, sell, or hold and should display the percentage confidence.	Program does calculate the percent confidence and give definitive guidance on if a stock should be bought or sold.	10	Pass
11	2.3 and 2.5	When a user makes a financial transaction which changes their wallet amount. **Normal**	There should be section of the program that displays the user's balance which automatically updates when user's make transactions	Balance updates if a transaction is made. However, the update only occurs if you traverse from that form to another and back. Does not do automatic update.	11	Pass
12	2.41 ,2.42, 2.43	When a user makes a buy transaction for a stock. **Normal/Erroneous**	A set of checks should be in place. Suitable alerts should be placed if the transaction stock is invalid, there is an incorrect quantity or insufficient funds for transaction	Correct responses if there is a problem with stock name, quantity, or balance problems	12	Pass
13	3.1, 3.1.1, 3.1.2	When a user traverses to the portfolio section	A table should be presented showing the user their latest transactions showing data	The table clearly shows the user's transactions	13	Pass

		and the orders section shows up onto the screen. **Normal**	like transaction price, time, and type.	ordered in the transactions latest. It follows the objective set of displaying all the data.		
14	3.2	Whenever a user selects the drop-down menu for looking at individual stocks and their statistics. **Normal**	A user should be able to see the stock companies they have invested in through a drop-down list. It will have to show information such as number of shares, money invested etc.	A user clearly shows the given stocks a user has invested in and information about their portfolio's	14	Pass
15	3.3, 3.3.1	A user clicks the view button to look at their portfolio statistics. **Normal**	The program should display the user's main information of their portfolio such as most expensive investment, largest shares in a company and the profit/loss of their wallet.	The program clearly shows the user this data displaying the company and its respective data. It also colour codes the profit/loss.	15	Pass
16	3.4.1, 3.4.2, 3.4.3	Whenever a user makes a sell transaction. **Normal/erroneous/ Boundary.**	The user should receive suitable error messages to tell them if the stock they have entered is invalid, not in their portfolio or don't have enough of that share.	User clearly gets a distinct error message on the individual violation that is made.	16	Pass
17	3.4.4, 3.4.5	User clicks sell to which the sell process starts. **Normal**	The program should retrieve the transaction price via the API and quantity fields. It then should update the user portfolio.	Program correctly calculates transaction price and alters the user portfolio in the suitable positions of change.	17	Pass
18	4.1, 4.1.1	User clicks the objectives menu to which they can view their recent objectives and their progress **Normal**	User should be able to see a table signifying all the objectives they have to complete to which it is ordered in priority. The objectives should be ordered from highest priority first. If priority is same, look at the deadline and go for the earlier one.	User can clearly see the to do list table showing the title, description, and the priority of the objectives. The program performs the correct comparison to if the priorities are different or the deadlines are different.	18	Pass
19	4.1, 4.1.2	User adds a new objective on the add objective menu. **Normal/Erroneous/ Boundary**	The user should be able to add a new objective to their account inputting the details of the objective and setting dates and priorities. Checks are in place that should	Program clearly adds the objective to account and the checks perform correctly observing if any	19	Pass

			prevent users from making an invalid objective.	fields are empty or the date field is inputted incorrectly.		
20	4.1.3.1	User selects the edit objective button. **Normal**	To edit an objective, the user should select the individual objective they would like to alter to which if no objective is selected, it should display a suitable message.	Program correctly displays the message that an objective hasn't been selected when a user is trying to edit the objective.	20	Pass
21	4.1.3.2, 4.1.3.3	User edits the objective changing information such as the priority or deadline. **Normal**	The program should allow the user to alter the objective to which when they click update, the order of the objective table should be automatically updated depending on the new priorities and deadlines.	Within the edit objectives menu, the user has clear access to changing any part of their objective and once they click update, the table is updated with any new changes made.	21	Pass
22	4.1.3.4	User clicks complete for the objective to signify it is done. **Normal**	The program should remove the objective from the table as its status is now complete. It should be erased from the user's account.	Objective is removed from the user account and no longer appears within the table.	22	Pass
23	5.1.1, 5.1.2	User enters the simulation conditions. **Erroneous**	The user should be prompted to make sure that all fields are entered in for the simulation and the volatility and drift constants are valid and are greater than 0 but less than 100.	User clearly gets error message signifying that the values entered are fully filled in and they are valid.	23	Pass
24	5.2	When a user selects stock by typing it into the search field and clicking search. **Erroneous**	If a user searches for a ticker that is invalid and not part of the NYSE, the user should be exposed to an error message to signify an invalid stock symbol	Error is clearly shown when user clicks the search button.	24	Pass
25	5.3, 5.3.1	When a user clicks the simulate button after entering in all the details for the simulation to commence. **Normal**	User should be able to see a table showing the simulation data with a column showing the simulation number, Date of simulation and stock price at that given time.	User clearly can see the simulation statistics respective of the simulation conditions they chose to which they can see the price fluctuations and affects as the simulations pass by.	25	Pass

Below are the objectives repeated making it easier to refer to for the testing.

Objectives

1. **Login and Sign-up Menu**
    - 1.1. User can either make an account or login
    - 1.2. If user doesn't have account, they can sign-up where data will be stored on an account database – both username and password will be stored.
        - 1.2.1. Passwords will be hashed and will be stored on the database.
        - 1.2.2. When user logs in, it will hash the input password and make a comparison to password stored in database table. If same, allow entry. If not, deny entry. Ask again.
            - 1.2.2.1. Once valid, open application.
        - 1.2.3. If user is signing up, the password has the match the confirm password field. If not ask again,
            - 1.2.3.1. once they generate the account, they will have a new wallet which will initially have a balance of $10,000 inserted.
    - 1.3. There should also be an admin account
        - 1.3.1. There should be a separate password for accessing the admin account.
            - 1.3.1.1. The user should repeatedly be asked if incorrect.
        - 1.3.2. It should allow them to view all users' transactions and portfolios. There should be a tally showing the number of users.
        - 1.3.3. There should also be an option to delete an account from the application.

2. **Analysis of Historical Stock Data**
    - 2.1. A search bar can be used to select a given stock to view their prices over a duration of time in a more visual representation – this is in the form of line graphs.
        - 2.1.1. If they do not enter a valid stock symbol or valid times, then they will be alerted to try again.
        - 2.1.2. Users will be able to select to view historical data over given time periods which if button is clicked for given period, the graph will be updated.
        - 2.1.3. Users will be able to view their previous searches of a stock when they search for one and this will be accomplished through a stack where each search will push itself to the top of the list.
    - 2.2. Besides the information, there will be a technical analysis of that given stock which will provide strong feedback on if a stock should be bought or not. This will be in the form of a confidence percentage.
        - 2.2.1. This will be using the MACD algorithm in conjunction with Exponential Moving Averages to measure if a stock should be bought or sold depending on trends of price. It should also calculate percentage confidence of that result occurring.
    - 2.3. Within an area of the screen will hold the user's wallet where they can view their remaining balance.

2.4. A user will also be able to buy a stock buy selecting both the name and the quantity. Again, checks should be made.
  2.4.1. Checks on if the stock exists on the NYSE
  2.4.2. Check the quantity is greater than 0.
  2.4.3. It also checks if there is the enough money for transaction.
2.5. The program should update the user's balance, update the portfolio, and add the transaction to the table transactions.

3. Portfolio Management

3.1. A main table will be used to display the shares that a user has in different companies. Once an order is complete, the data will be added to this list.
  3.1.1. The table will display their order history of their recent orders.
  3.1.2. This will contain information about the price at transaction, quantity, Transaction price, Time of Purchase, and transaction type (buy/sell)
3.2. A user's portfolio will be able to be viewed with a drop-down menu showing all their stocks and once clicked, it will provide data such as proportion and the amount of investment in their stock.
  3.2.1. This will use both cross-table parametrised SQL and aggregate functions to analyse and calculation stock performance.
3.3. A user will also be able to view their whole statistical performance of their portfolio through identifying most expensive investment and most invested company.
  3.3.1. Users will also be able to view their portfolio profit of their whole wallet which again will use cross table parameterised SQL and aggregate functions.
3.4. A user will also be able to sell a given stock of certain number of shares. This process is more intensive in terms of the validation.
  3.4.1. Check a stock symbol is valid and check the quantity field is greater than 0.
  3.4.2. If stock symbol is valid, the program should check if it is within the user's portfolio. This will involve retrieving the user's invested stocks from the database and performing a recursive search to identify if the stock they want to sell is within their portfolio.
  3.4.3. It should then check if they have enough stocks to sell.
  3.4.4. It should then retrieve the price of transaction using both the API and quantity field.
  3.4.5. It should then change the user's balance, add the transaction to the database and update the user's portfolio.

4. Financial Objectives Tracker

4.1. A user will be able to make financial objectives on what they would like to achieve soon. The table displayed will order the objectives utilising a priority queue for this process.
  4.1.1. A menu will be presented containing a table showing the structure of their objectives with the most important (Critical)

being at the top and least important (Not Important) being at the bottom. Earlier dates take priority.

4.1.2. A menu on the right-hand side will allow users to add objectives to their objective list.

4.1.2.1. It will perform checks to validate if the inputs are valid-All fields entered and valid date.

4.1.3. For a user wanting to edit a certain objective, there will be an edit button which will utilise both the SQL queries and priority queue implementation.

4.1.3.1. Needs to check if objective has been selected before editing.

4.1.3.2. The user will be able to change aspects such as their title and description as well as the priority and deadline which will be both change the objective's placement in the queue.

4.1.3.3. Once a user clicks Ok, the table will automatically update changing the placement of the queue objectives.

4.1.3.4. A user can also mark an objective complete which will remove it from the objective list.

5. Geometric Brownian Motion Stock Price Simulation

5.1. A user will be given boxes to fill depending on the given conditions they want to set for their stock.

5.1.1. For the volatility and drift constants, a set of checks will be in place to prevent users from entering erroneous data that would in turn crash the program. E.g., It shouldn't allow users to enter negative volatility and drift constants.

5.1.2. It should make sure that all fields are entered and filled in.

5.2. When a user is selecting the stock, they would like to simulate, there should be a check to see if the stock symbol is valid and part of the NYSE.

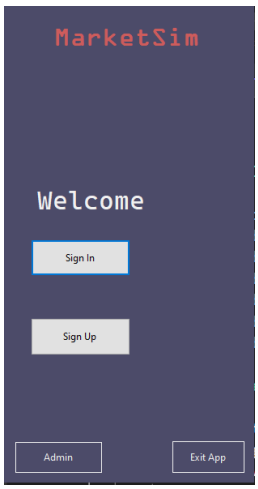
5.3. Once the user has clicked the button to simulate a screen will be presented showing all the data from the simulation.

5.3.1. This will be in a table showcasing the Sim Number, Date of time of that current simulation and the corresponding stock price simulated.

References are contained within the next page.

References

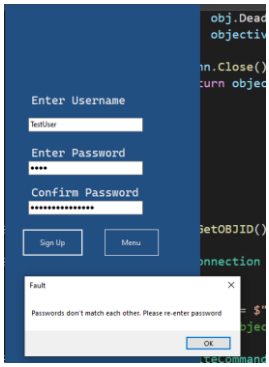
1)



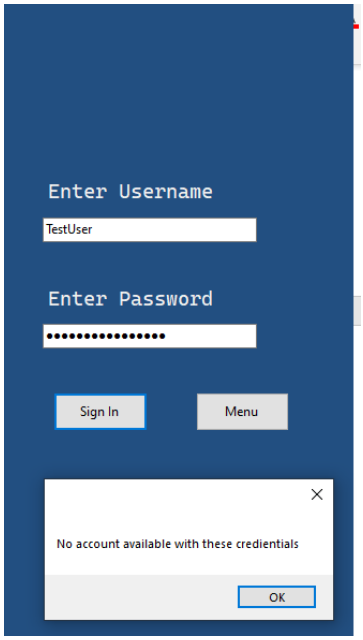
2)

rowid	AccountID	Username	Password
		Click here to define a filter	
4	4	123	pmWkWSBCL51Bfkhn79xPuKBKHz//H6B+mY6G9/eieuM=
5	5	admin1	n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4ls0V1sFbDwCgg=
13	13	1234	A6xnQhbz4Vx2HuGl4lXwZ5U2l8iziLRFnhP5eNfIRvQ=

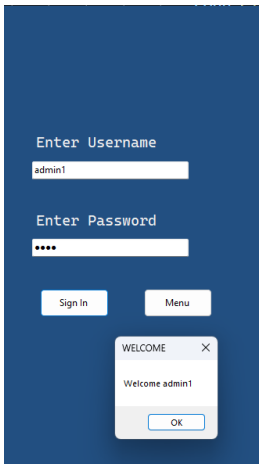
3)



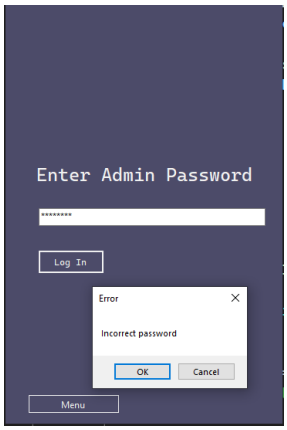
4)


Enter Username
TestUser
Enter Password
Sign In
Menu
No account available with these credientials
OK

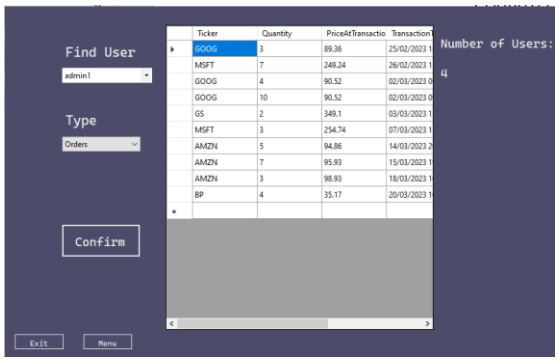
4.5(a)

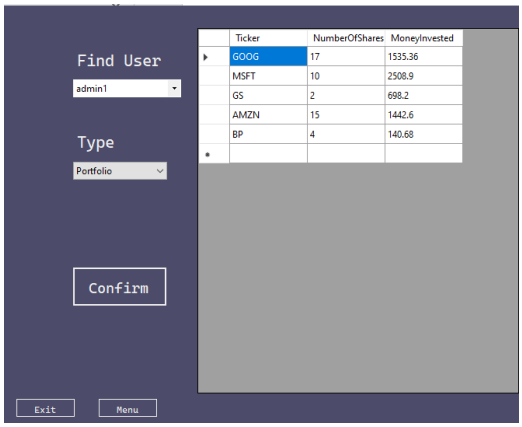

Enter Username
admin1
Enter Password
Sign In
Menu
WELCOME
Welcome admin1
OK

5)


Enter Admin Password
Log In
Error
Incorrect password
OK
Cancel
Menu

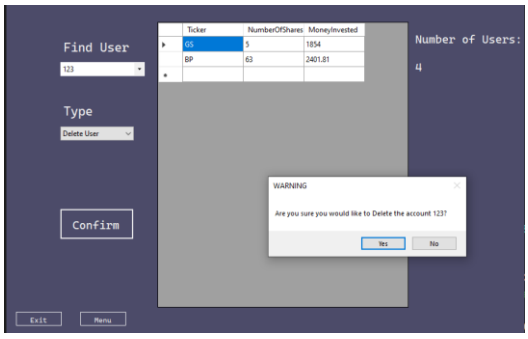
6)

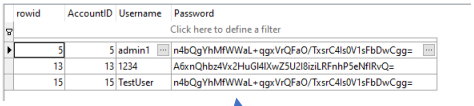




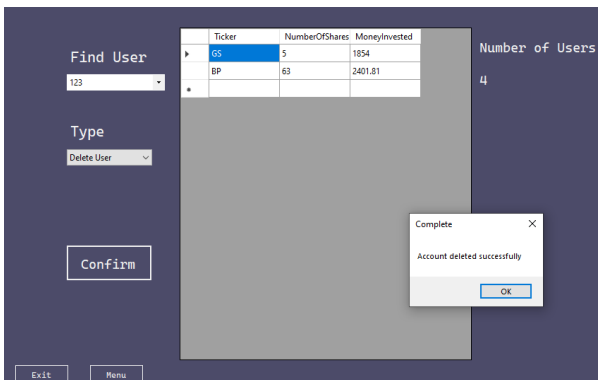
▶	4	4	123	pmWkWSBCL51Bfkhn79xPuKBKHz//H6B+mY6G9/eieuM=
	5	5	admin1	n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg=
	13	13	1234	A6xnQhbz4Vx2HuGl4lXwZ5U2l8iziLRFnhP5eNfIRvQ=
	15	15	TestUser	n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg=

7)



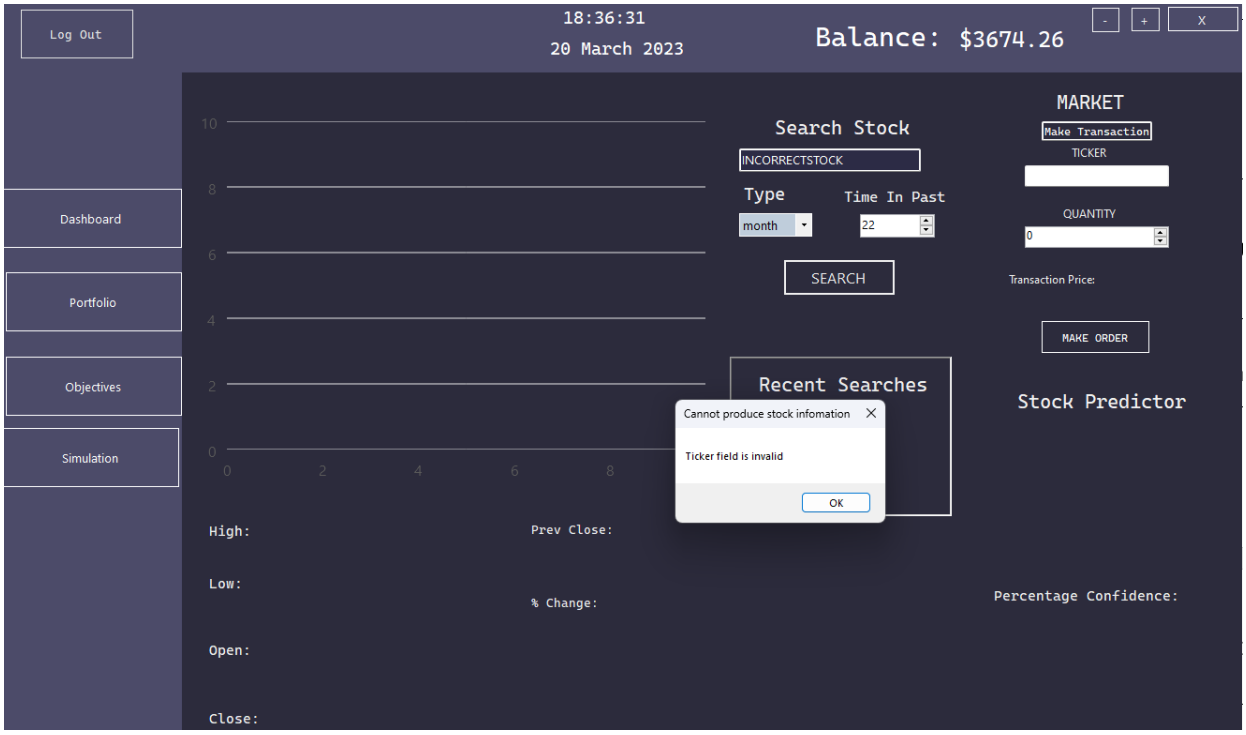


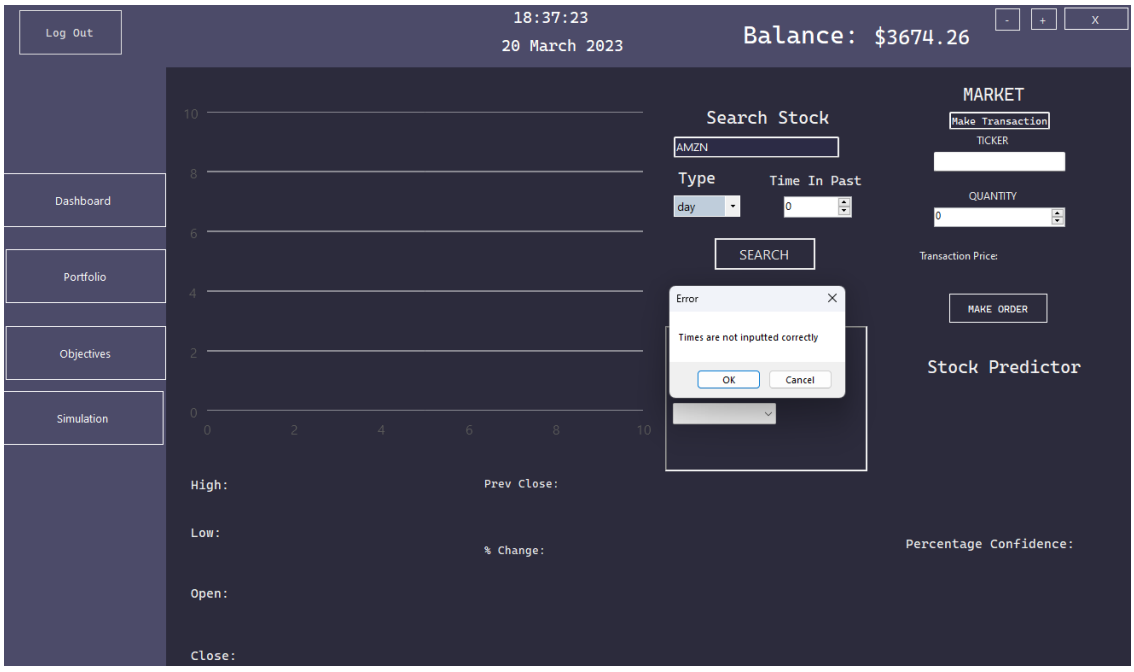
rowid	AccountID	Username	Password
			Click here to define a filter
5	5	admin1	n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg=
13	13	1234	A6xnQhbz4Vx2HuGl4lXwZ5U2l8iziLRFnhP5eNfIRvQ=
15	15	TestUser	n4bQgYhMfWWaL+qgxVrQFaO/TxsrC4Is0V1sFbDwCgg=



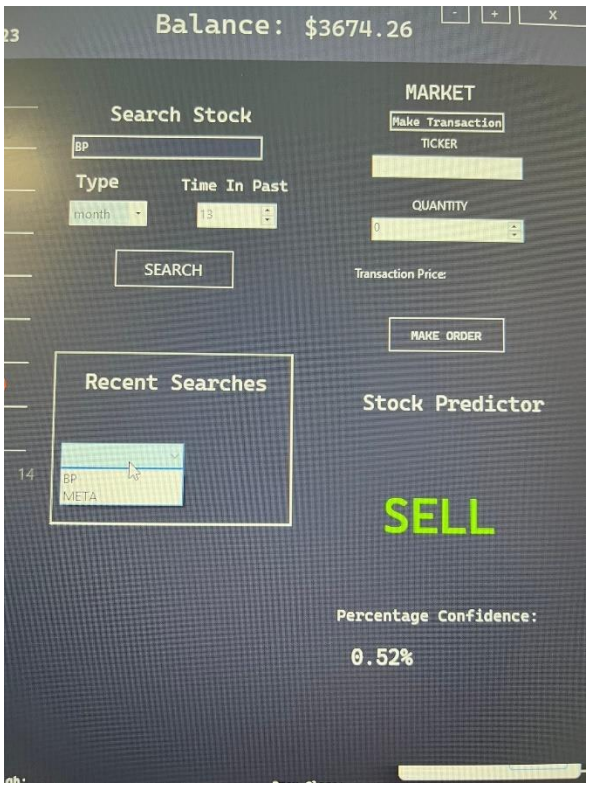
User removed from table

8)

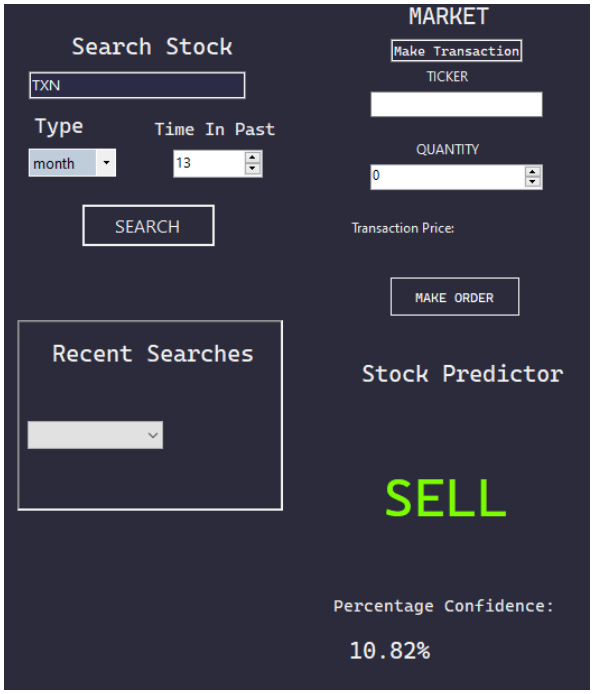

Log Out
18:36:31
20 March 2023
Balance: $3674.26
Dashboard
Portfolio
Objectives
Simulation
Search Stock
INCORRECTSTOCK
Type
month
Time In Past
22
SEARCH
Recent Searches
MARKET
Make Transaction
TICKER
QUANTITY
0
Transaction Price:
MAKE ORDER
Stock Predictor
Cannot produce stock infomation
Ticker field is invalid
OK
High:
Low:
Open:
Close:
Prev Close:
% Change:
Percentage Confidence:


Log Out
18:37:23
20 March 2023
Balance: $3674.26
Dashboard
Portfolio
Objectives
Simulation
Search Stock
AMZN
Type
day
Time In Past
0
SEARCH
MARKET
Make Transaction
TICKER
QUANTITY
0
Transaction Price:
MAKE ORDER
Stock Predictor
Error
Times are not inputted correctly
OK
Cancel
High:
Low:
Open:
Close:
Prev Close:
% Change:
Percentage Confidence:

9)



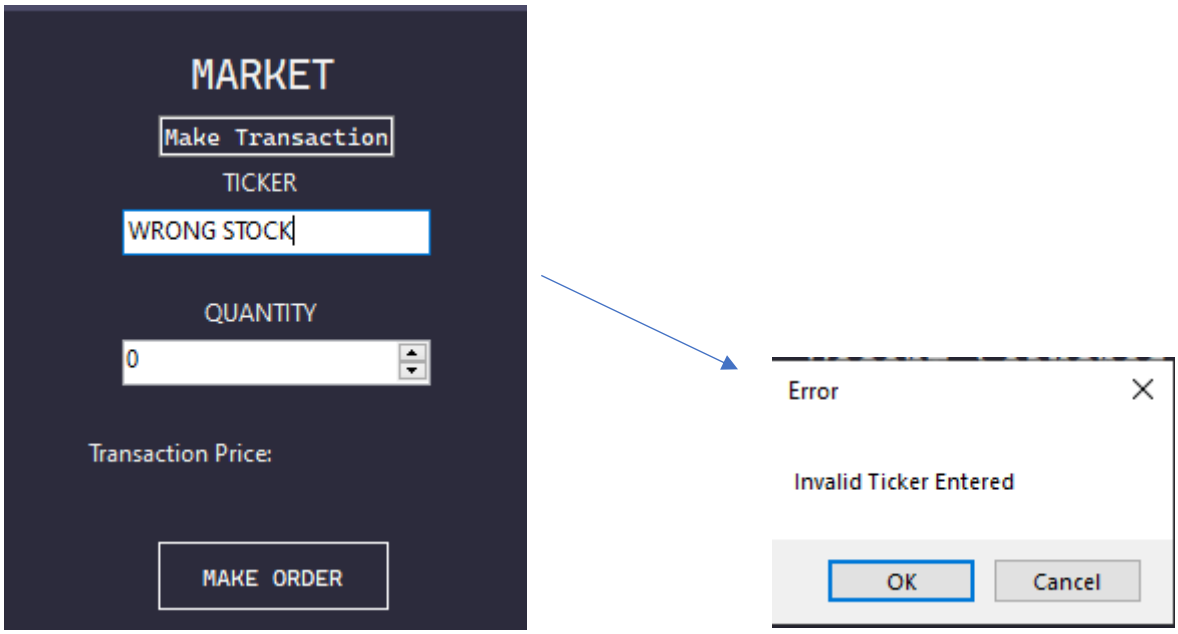
10)

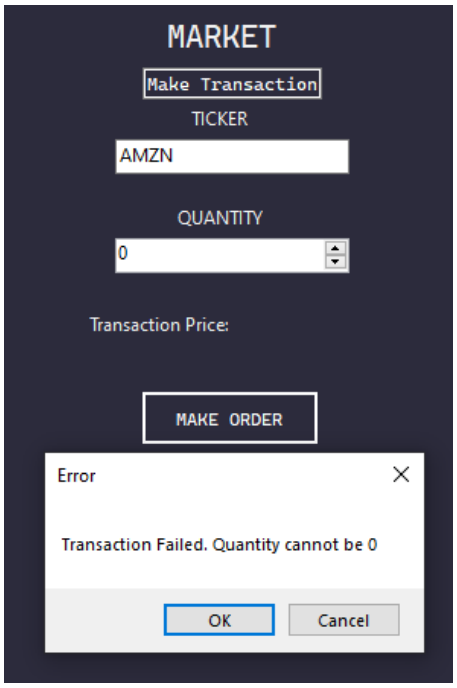


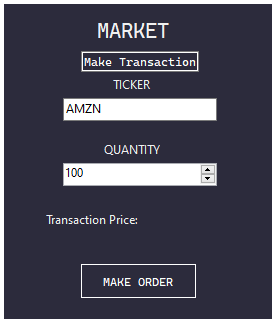
Stock prediction for Texas Instruments Ltd.

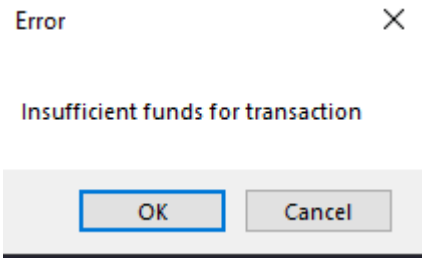
11.

12.

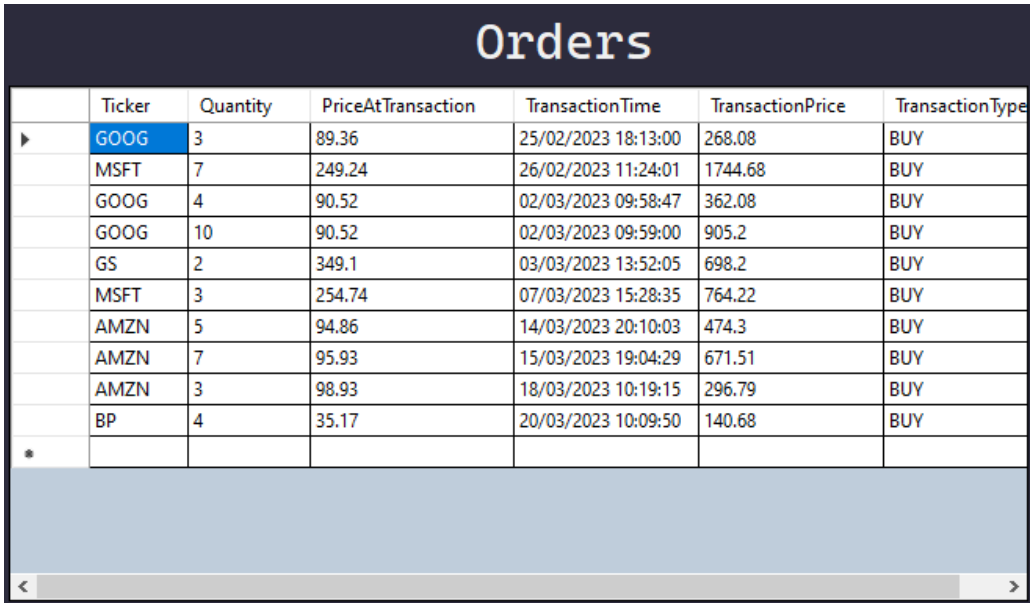

MARKET
Make Transaction
TICKER
WRONG STOCK
QUANTITY
0
Transaction Price:
MAKE ORDER
Error
Invalid Ticker Entered
OK
Cancel


MARKET
Make Transaction
TICKER
AMZN
QUANTITY
0
Transaction Price:
MAKE ORDER
Error
Transaction Failed. Quantity cannot be 0
OK
Cancel



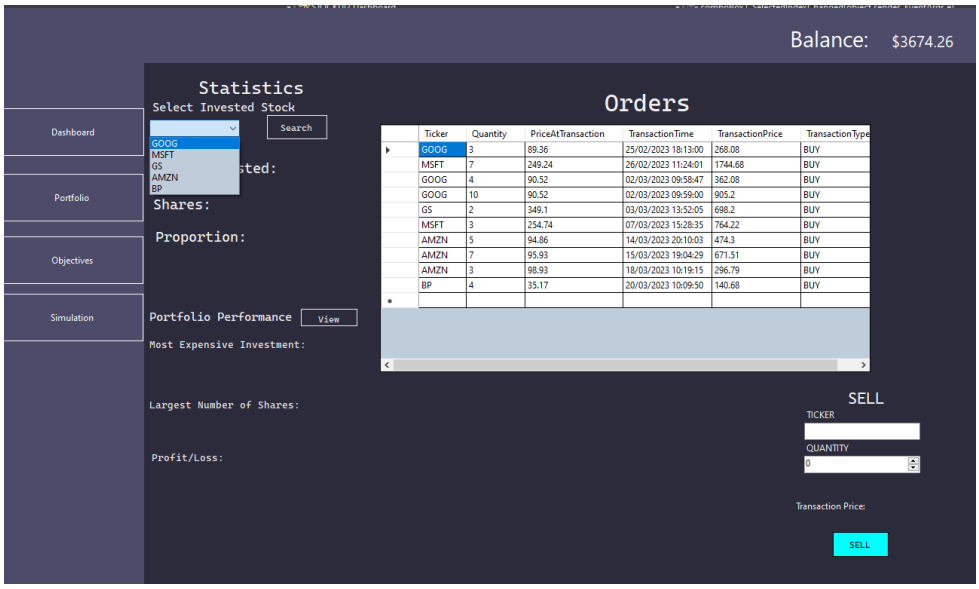


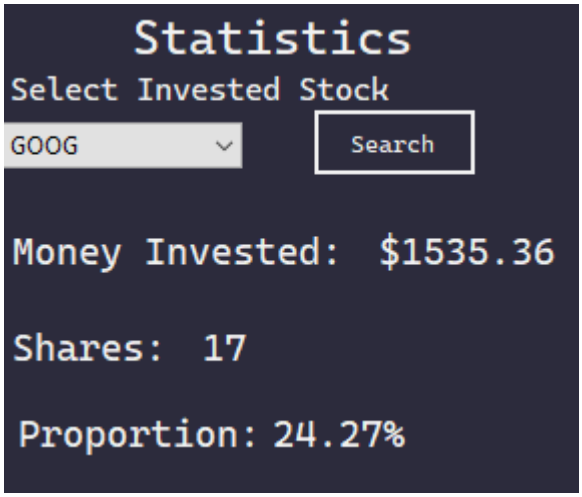
13)



	Ticker	Quantity	PriceAtTransaction	TransactionTime	TransactionPrice	TransactionType
▶	GOOG	3	89.36	25/02/2023 18:13:00	268.08	BUY
	MSFT	7	249.24	26/02/2023 11:24:01	1744.68	BUY
	GOOG	4	90.52	02/03/2023 09:58:47	362.08	BUY
	GOOG	10	90.52	02/03/2023 09:59:00	905.2	BUY
	GS	2	349.1	03/03/2023 13:52:05	698.2	BUY
	MSFT	3	254.74	07/03/2023 15:28:35	764.22	BUY
	AMZN	5	94.86	14/03/2023 20:10:03	474.3	BUY
	AMZN	7	95.93	15/03/2023 19:04:29	671.51	BUY
	AMZN	3	98.93	18/03/2023 10:19:15	296.79	BUY
	BP	4	35.17	20/03/2023 10:09:50	140.68	BUY
*						

14.

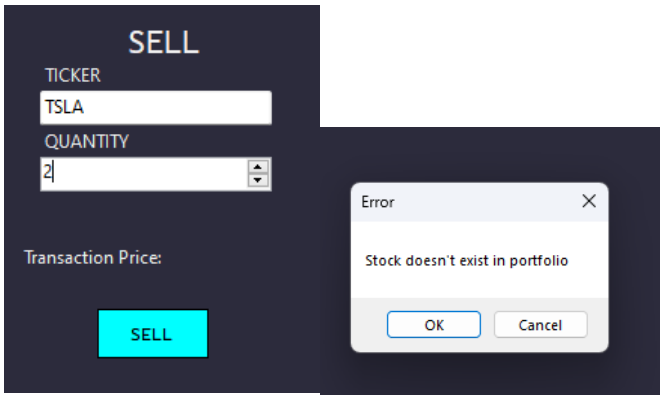


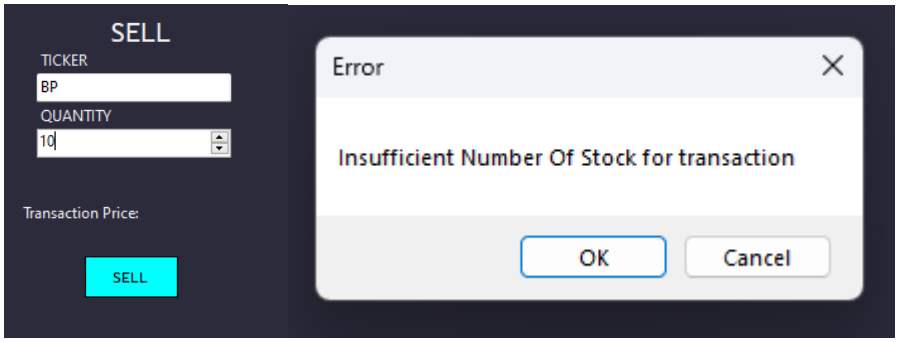

Statistics
Select Invested Stock
GOOG
Search
Money Invested: $1535.36
Shares: 17
Proportion: 24.27%

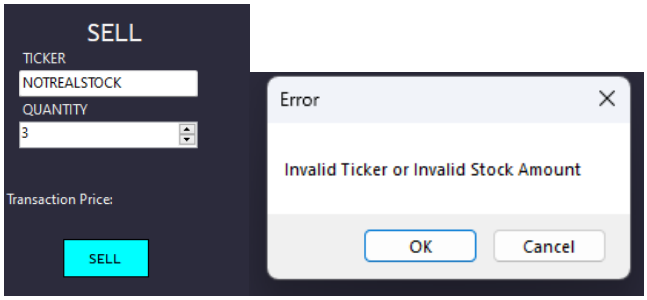
15.


Portfolio Performance
View
Most Expensive Investment:
Stock: MSFT, Price: $1744.68
Largest Number of Shares:
Stock: GOOG, Number: 17
Profit/Loss:
$-2961.57

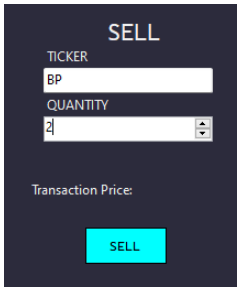
16.


SELL
TICKER
TSLA
QUANTITY
2
Transaction Price:
SELL
Error
Stock doesn't exist in portfolio
OK
Cancel

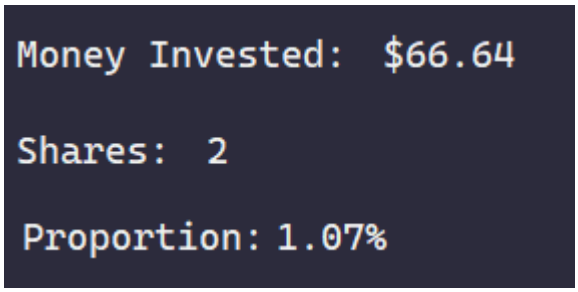




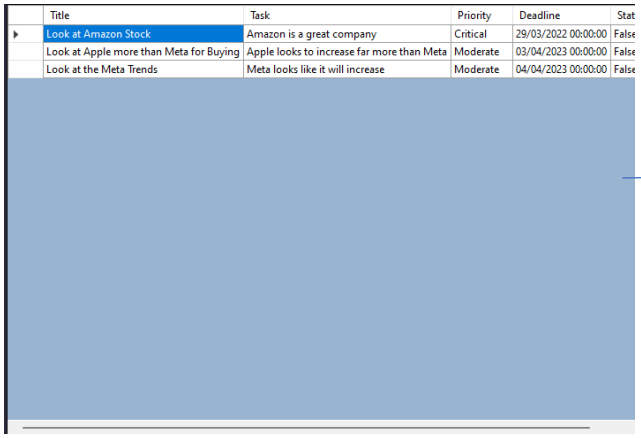
17.



BP	2	37.02	22/03/2023 12:13:52	74.04	SELL



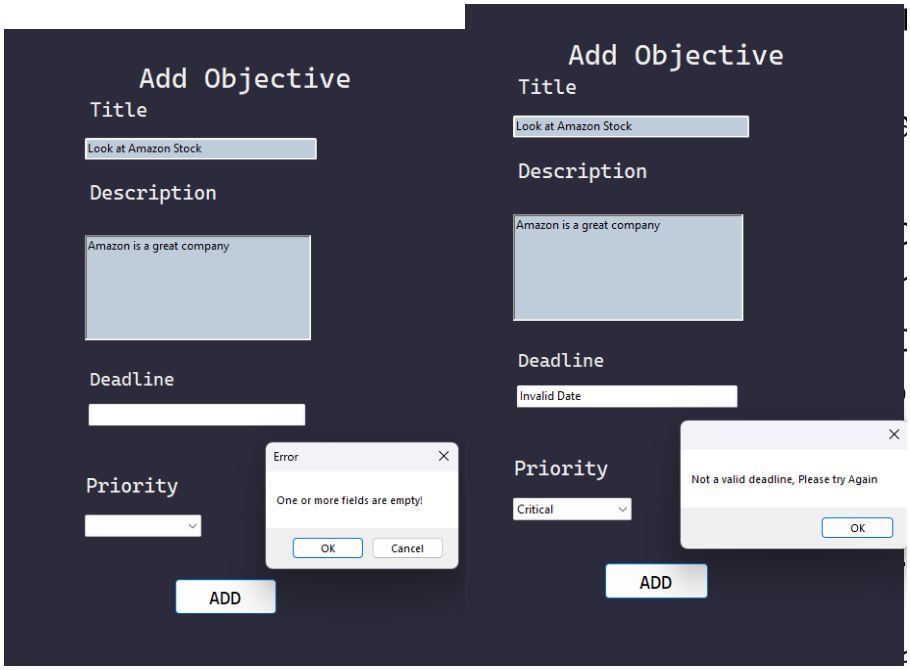
18.



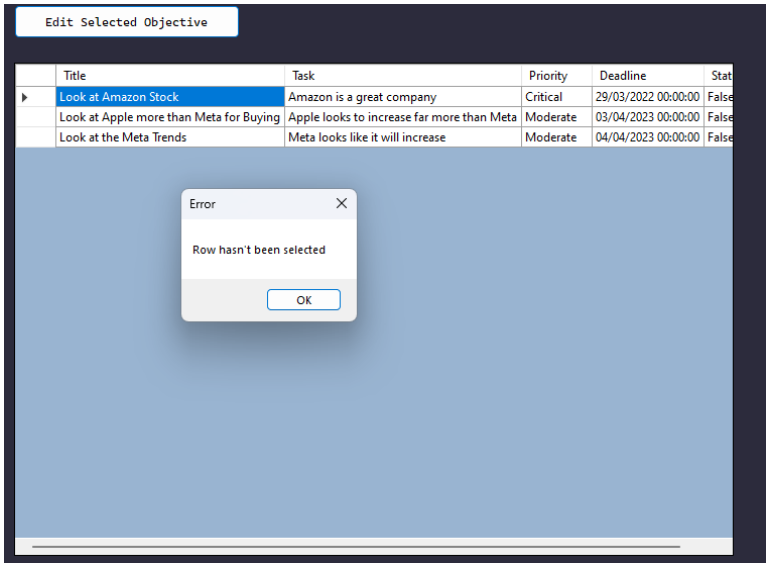
Title	Task	Priority	Deadline	Stat
Look at Amazon Stock	Amazon is a great company	Critical	29/03/2022 00:00:00	False
Look at Apple more than Meta for Buying	Apple looks to increase far more than Meta	Moderate	03/04/2023 00:00:00	False
Look at the Meta Trends	Meta looks like it will increase	Moderate	04/04/2023 00:00:00	False

We can see that since the Amazon Objective has a higher priority it is at the top. We can also see the date comparison seeing that even though the Apple objective has the same priority than the Meta Objective, it has an earlier deadline, so it is further up.

19.

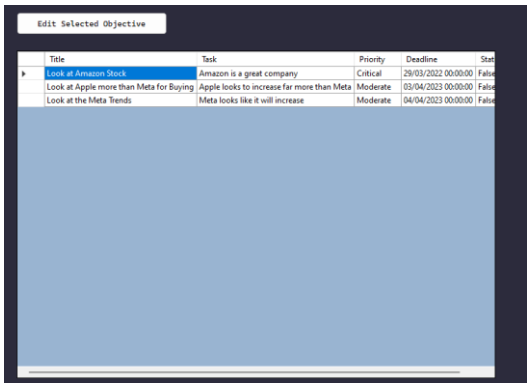


20.

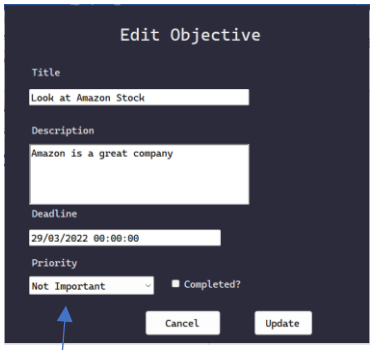


21.

Before:

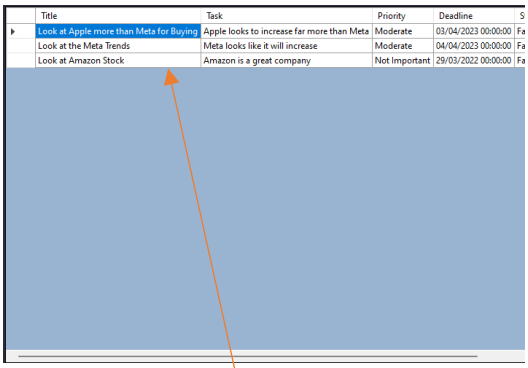


Change:



Objective on Amazon has now changed its priority to not important.

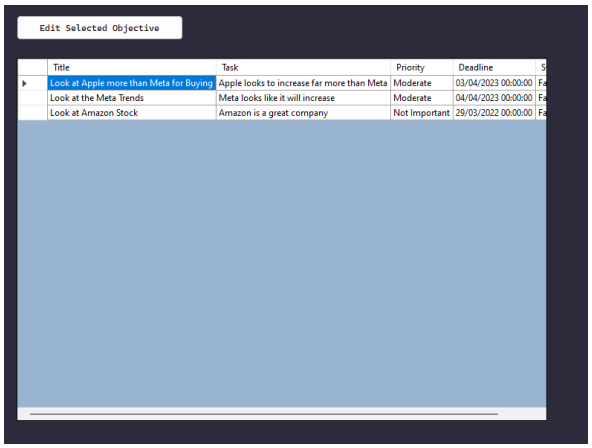
Result:



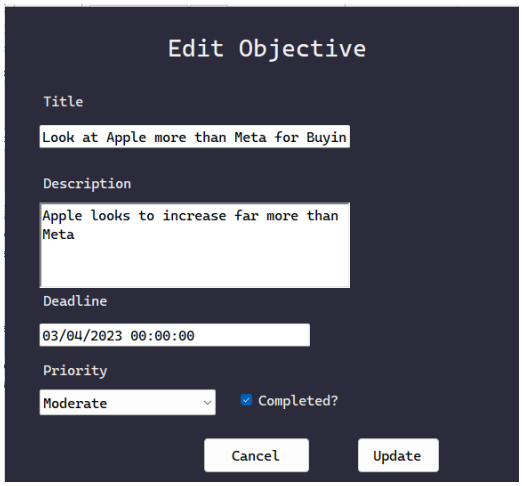
Since the Amazon Objective has now become not important, it is now the lowest priority and once updated will appear at the bottom of the table.

22.

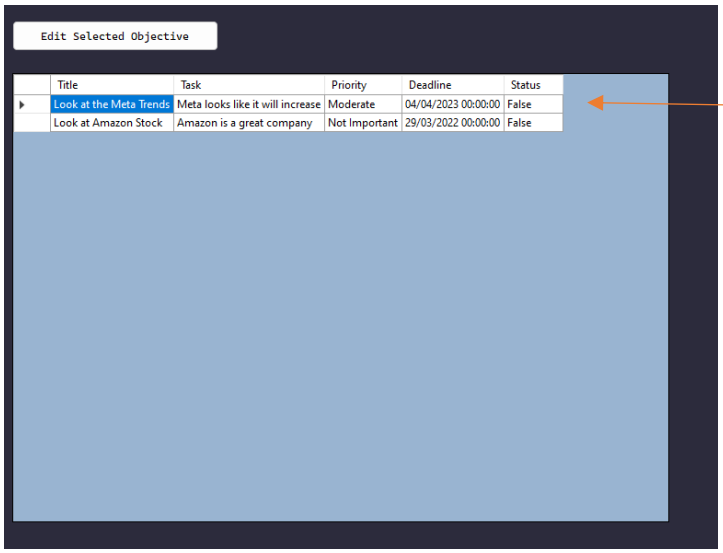
Before:



Complete option ticked:

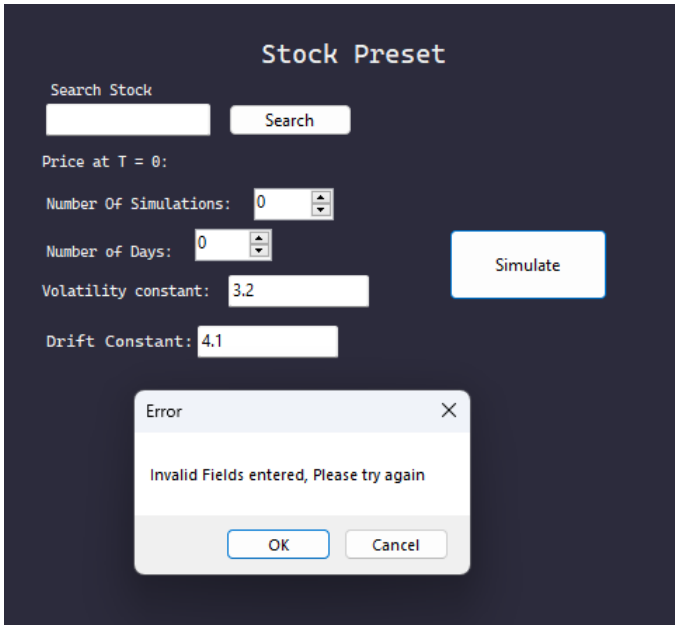


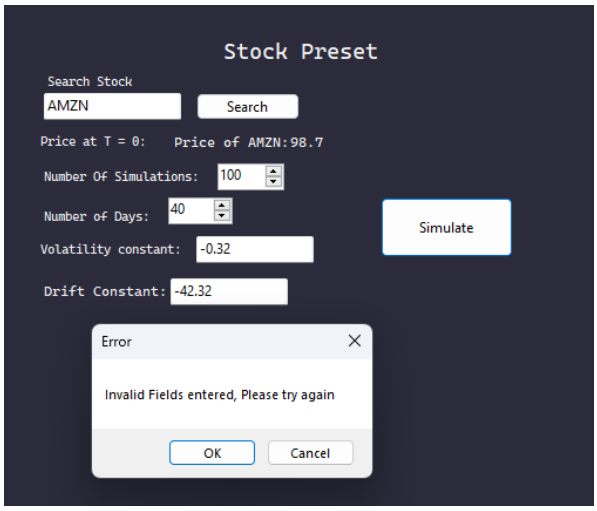
After completion of Objective:



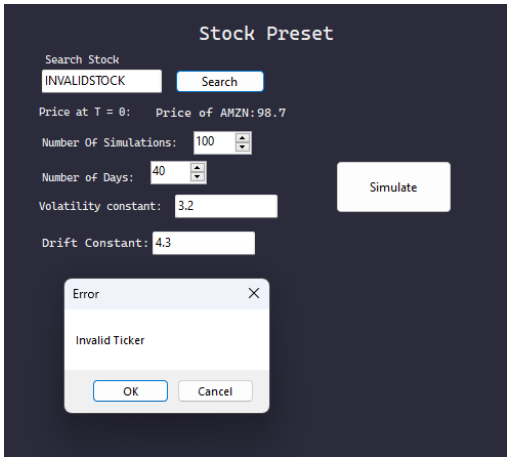
Objective about Apple is now removed from the user account and table.

23.

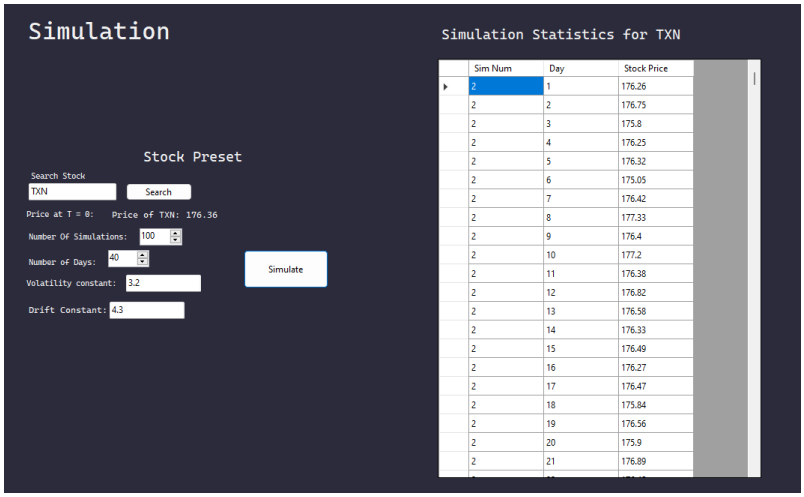

Stock Preset
Search Stock
Search
Price at T = 0:
Number Of Simulations: 0
Number of Days: 0
Volatility constant: 3.2
Drift Constant: 4.1
Simulate
Error
Invalid Fields entered, Please try again
OK
Cancel

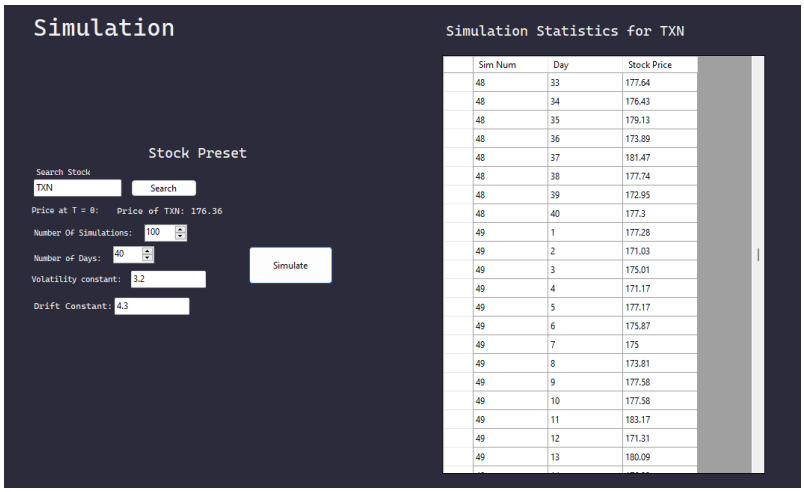

Stock Preset
Search Stock
AMZN
Search
Price at T = 0:
Price of AMZN:98.7
Number Of Simulations: 100
Number of Days: 40
Volatility constant: -0.32
Drift Constant: -42.32
Simulate
Error
Invalid Fields entered, Please try again
OK
Cancel

24.


Stock Preset
Search Stock
INVALIDSTOCK
Search
Price at T = 0:
Price of AMZN:98.7
Number Of Simulations: 100
Number of Days: 40
Volatility constant: 3.2
Drift Constant: 4.3
Simulate
Error
Invalid Ticker
OK
Cancel

25.


Simulation
Stock Preset
Search Stock
TXN
Search
Price at T = 0:
Price of TXN: 176.36
Number Of Simulations:
100
Number of Days:
40
Volatility constant:
3.2
Drift Constant:
4.3
Simulate
Simulation Statistics for TXN
Sim Num
Day
Stock Price
2 1 176.26
2 2 176.75
2 3 175.8
2 4 176.25
2 5 176.32
2 6 175.05
2 7 176.42
2 8 177.33
2 9 176.4
2 10 177.2
2 11 176.38
2 12 176.82
2 13 176.58
2 14 176.33
2 15 176.49
2 16 176.27
2 17 176.47
2 18 175.84
2 19 176.56
2 20 175.9
2 21 176.89


Simulation
Stock Preset
Search Stock
TXN
Search
Price at T = 0:
Price of TXN: 176.36
Number Of Simulations:
100
Number of Days:
40
Volatility constant:
3.2
Drift Constant:
4.3
Simulate
Simulation Statistics for TXN
Sim Num
Day
Stock Price
48 33 177.64
48 34 176.43
48 35 179.13
48 36 173.89
48 37 181.47
48 38 177.74
48 39 172.95
48 40 177.3
49 1 177.28
49 2 171.03
49 3 175.01
49 4 171.17
49 5 177.17
49 6 175.87
49 7 175
49 8 173.81
49 9 177.58
49 10 177.58
49 11 183.17
49 12 171.31
49 13 180.09

Evaluation

From my experience, I have considered this project to be a success with the program going further than what I expected and what I wanted to do. The project has given me the opportunity to further my understanding of the world of stock trading by both developing models for financial strategies and understanding how we can store and calculate trends in user portfolios to understand performance.

From the documentation standpoint, it has allowed me to gain experience in software development through scoping out my problem and gaining initial thoughts within the analysis, constructing data models and class diagrams within my documented design, showcasing the various methods and skills within the technical solution, and finally trialling the robustness and integrity of my program within the testing portion. With my program being both heavily invested on the application side as well as the database side, I was able to strengthen my skills and understanding within object-oriented programming utilising techniques such as abstract classes, enumeration, and encapsulation. Furthermore, it allowed me to see the immense impact of SQL by using measures such as constraints to maximise the integrity of the table designs. It also allowed me to look at how we can join tables to gather interlinking information and use aggregate functions to provide easier ways of calculating user performance and statistical analysis. From my experience, I found the implementation of API to be a challenging aspect of my program as it both tested my server-side skills in terms of validating responses and parsing web API responses into a correct format for easy access to other areas of the code. As a whole, I believe that my program satisfies the selected objectives and the end-user requirements that were set.

Some regions that I would have wanted to improve on were within my Stock Simulation section, including more conditions would have made the simulation far more complex and as well as more realistic to current conditions such as incorporating earnings growth or dividends. Furthermore, adding some analysis for the simulation would have proved to be very useful and provided the user with a far greater outlook on how the simulation has performed rather than just showing the stock data. Additionally, I would argue that the fluidity and performance of my program were inhibited regarding various concurrency issues within SQLite meaning that I wasn't able to perform multiple SQL transactions which prevented fast-paced SQL operations to be performed.

Objective	How was it met?	How well was it met?	Improvements?
User can either make an account or login.	Created button that allows user to sign in or create new account	Perfectly	No Improvements
If user doesn't have account, they can sign-up where data will be stored on an account database – both username and password will be stored.	Within the database table USERS, I have created fields that will store the username and password	Perfectly	No Improvements
Passwords will be hashed and will be stored on the database.	Once a signs up, the password string they enter will be hashed and stored on database	Perfectly	Maybe think about using different salts. This means that if a user enters a password like another person's password, then it will produce a completely different hash.
When user logs in, it will hash the input password and make a comparison to password stored in database table. If same, allow entry. If not, deny entry. Ask again.	When user enters username and password string, it will try and compare with all data available in database and if they match then allow entry.	Perfectly	Maybe add Forgot your password.
Once valid, open application.	Program opens when the username and password are valid.	Perfectly	No Improvements
If user is signing up, the password has the match the confirm password field. If not ask again.	Program keeps asking user to keep entering password until both fields are equal.	Perfectly	No Improvements
Once they generate the account, they will have a new wallet which will initially have a balance of $10,000 inserted.	When user signs up, an initial balance of 10,000 dollars will be inserted into the wallet as a starter fund package. This will show up when logging in.	Perfectly	Maybe have some options of allowing users to choose how much they would like to start with.

There should also be an admin account	There is a section for reports where the admin and manage and view other user's activities.	Perfectly	No Improvements
There should be a separate password for accessing the admin account.	Password is stored within a text file and not within the database table. This password cannot be changed.	Perfectly but maybe have the opportunity for the password to be changed.	Maybe allow admin to change the password.
The user should repeatedly be asked if incorrect.	Error message is shown repeatedly if incorrect	Perfectly	No Improvements
It should allow them to view all users' transactions and portfolios. There should be a tally showing the number of users	Drop down menus to traverse from each option to which when clicking search will show the respective data in a table	Perfectly	No Improvements
There should also be an option to delete an account from the application.	Once deleted, the user is removed from database and account is erased	Perfectly	Maybe have an option to recover an account.
A search bar can be used to select a given stock to view their prices over a duration of time in a more visual representation – this is in the form of line graphs.	Search engine allows users to enter a stock name along with parameters for time intervals to which when they search, the stock prices will be displayed onto the graph.	Perfectly	No Improvements
If they do not enter a valid stock symbol or valid times, then they will be alerted to try again.	User is displayed with suitable message saying that the inputs are not correct.	Perfectly	No Improvements
Users will be able to select to view historical data over given time periods which if button is clicked for given period, the graph will be updated.	User are allowed to change both the stock and time intervals to which if the search button is selected, the graph will automatically	Perfectly	No Improvements

	update for the user.		
Users will be able to view their previous searches of a stock when they search for one and this will be accomplished through a stack where each search will push itself to the top of the list.	Once a user searches for a stock, it is added to the top of the search list for a user to select and search again.	Perfectly	Maybe store other information such as times of the search to give the user more information.
Besides the information, there will be a technical analysis of that given stock which will provide strong feedback on if a stock should be bought or not. This will be in the form of a confidence percentage.	The section of the program displays the analysis of the stock that was just searched showing both an overall view on if a stock should be bought or sold and a corresponding percentage confidence for that belief	Perfectly	No Improvements
This will be using the MACD algorithm in conjunction with Exponential Moving Averages to measure if a stock should be bought or sold depending on trends of price. It should also calculate percentage confidence of that result occurring.	The MACD algorithm was built well and has behaved accurately compared to the trends of stock prices.	Good however I would argue that I haven't considered any other factors such as dividends or stock share ratios which may affect how prices fluctuate	No Improvements
Within an area of the screen will hold the user's wallet where they can view their remaining balance.	Within the top right of the screen displays the current balance of a user's wallet and automatically changes after transactions	Perfectly	Maybe include how the wallet behaves and its changes over time in terms of how much money is in there.
A user will also be able to buy a stock buy selecting both the name and the quantity. Again, checks should be made.	User can buy a stock by entering the stock name and quantity fields. The checks are then performed to validate the transaction.	Perfectly	Have a pop up that shows the user the transaction before they buy.
Checks on if the stock exists on the NYSE	Pop shows up to the user if any invalidated transactions are	Perfectly	No Improvements

Check the quantity is greater than 0. It also checks if there is the enough money for transaction.	made to the user. It will cause the transaction to fail and end and ask the user to enter again. A suitable error message is shown for any of these violations.		
The program should update the user's balance, update the portfolio, and add the transaction to the table transactions.	Whenever a user makes a transaction, the user portfolio's is changed making the correct updates and updating the user's investment portfolio.	Good however, once a transaction is completed and the balance is updated from the backend, it sometimes doesn't update the application in real time.	Maybe do separate threading for updating the user's balance and displaying this balance to the user.
A main table will be used to display the shares that a user has in different companies. Once an order is complete, the order will be added to this list.	Whenever a user clicks on the portfolio section, a table will show the orders to the user displaying their recent activities in transactions.	Perfectly	Allow user's to maybe filter their orders over a shorter or long-time interval.
The table will display their order history of their recent orders. This will contain information about the price at transaction, quantity, Transaction price, Time of Purchase, and transaction type (buy/sell)	Table has columns which shows the respective data and information of each order and its key information.	Perfectly	No Improvements.
A user's portfolio will be able to be viewed with a drop-down menu showing all their stocks and once clicked, it will provide data such as proportion and the amount of investment in their stock.	User can use a drop-down menu to select each stock to which its respective information about the number of shares and money invested etc. is shared.	Perfectly	Maybe add more information about the stock.
This will use both cross-table parametrised SQL and aggregate functions to analyse and calculation stock performance.	Within the code, I have utilised aggregate functions such as SUM to find total money	Perfectly	No Improvements

	invested and COUNT to identify the number of users in the application.		
A user will also be able to view their whole statistical performance of their portfolio through identifying most expensive investment and most invested company	Allow user to see a holistic view of their portfolio performance. Statistics such as most invested stock, most expensive investment, and profit. Again, using aggregate functions	Perfectly	Same with the drop-down menu for invested stock, there should be more information to display towards the user. E.g., dividends, stock per share ratios.
A user will also be able to sell a given stock of certain number of shares. This process is more intensive in terms of the validation.	User is allowed to enter the stock they would like to sell and enter the quantity that they would like to sell.	Perfectly	Rather than entering the stock name, maybe have a drop-down menu to display the stocks they have invested in. Also once perform transaction, show to user the amount they will gain.
Check a stock symbol is valid and check the quantity field is greater than 0. If stock symbol is valid, the program should check if it is within the user's portfolio. This will involve retrieving the user's invested stocks from the database and performing a recursive search to identify if the stock they want to sell is within their portfolio. It should then check if they have enough stocks to sell. It should then retrieve the price of transaction using both the API and quantity field. It should then change the user's balance, add the transaction to the database and update the user's portfolio.	These validations if compromised will display a suitable error message conveying the specific error they have made when making the transaction. Especially for the stock symbol in portfolio validation, the recursive algorithm is designed well for its purpose	Perfectly	No Improvements

A user will be able to make financial objectives on what they would like to achieve soon. The table displayed will order the objectives utilising a priority queue for this process.	SHOWN IN ROW BELOW	SHOWN IN ROW BELOW	SHOWN IN ROW BELOW
A menu will be presented containing a table showing the structure of their objectives with the most important (Critical) being at the top and least important (Not Important) being at the bottom. Earlier dates take priority.	User can see a table showing them the objectives they have made to which they are ordered in terms of priority.	Perfectly	Try and change table so it shows the full table – this removes the need of scrolling to see the end of the objectives as they are cut off.
A menu on the right-hand side will allow users to add objectives to their objective list.	Section of portfolio menu that allows user to add an objective and customising the various properties of that objective	Perfectly	No Improvements.
For a user wanting to edit a certain objective, there will be an edit button which will utilise both the SQL queries and priority queue implementation.	By selecting the objective and clicking the edit button, a new window will pop up to which the user can alter the objective's information	Perfectly	No Improvements
Needs to check if objective has been selected before editing.	If the objective in the table isn't selected, then a suitable error message is shown that the row hasn't been selected	Perfectly	No Improvements
The user will be able to change aspects such as their title and description as well as the priority and deadline which will change the objective's placement in the queue.	Within the edit objectives menu, users can alter the title, description and alter the priority or deadline which will place an effect on the objective's position in the priority queue	Perfectly	No Improvements
Once a user clicks Ok, the table will automatically	When we user clicks update,	Perfectly	Maybe having some indication showing

update changing the placement of the queue objectives.	the table updates showing the new arrangement of the priority queue.		what has been changed to allow the user to see what objectives have been edited.
A user can also mark an objective complete which will remove it from the objective list.	Once it is completed and updated. It is removed from the Objectives table and is no longer available to be seen.	Perfectly	Maybe have a section showing the objectives that have been completed to which can view and put back into the table.
A user will be given boxes to fill depending on the given conditions they want to set for their stock.	There are input fields allowing users to select what stock they would like to select and the various conditions	Perfectly	No Improvements
For the volatility and drift constants, a set of checks will be in place to prevent users from entering erroneous data that would in turn crash the program. E.g., It shouldn't allow users to enter negative volatility and drift constants.	If the user enters invalid values, an error message will be shown telling the user to re-enter the volatility or drift constant values.	Perfectly	Try and be more specific with the error message. Maybe highlight the part which is wrong.
It should make sure that all fields are entered and filled in.	An error message is displayed telling the user not all fields have been filled in.	Perfectly	No Improvements
When a user is selecting the stock, they would like to simulate, there should be a check to see if the stock symbol is valid and part of the NYSE.	An error message is displayed when the user searches a stock that the ticker is invalid and should be re-entered.	Perfectly	Include more stock exchanges such as LSE to allow more stocks to be simulated for their stock prices.
Once the user has clicked the button to simulate a screen will be presented showing all the data from the simulation.	Once user clicks simulate after entering valid inputs, the simulation will commence, and the data of the simulation is stored and displayed onto the data table.	Perfectly	Provide more analysis of the data – average stock price over each simulation, highest, lowest etc. Have selectable scenarios which automatically place the condition values depending on the values at the certain time e.g., recessions.

This will be in a table showcasing the Sim Number, Date of time of that current simulation and the corresponding stock price simulated.	Table contains columns listed in objective and for each simulation and day, each point is simulated successfully.	Good however slight problem with the simulation number starting from 2 rather than one.	Have another column showing the absolute difference from the previous day,

End-User Feedback

After conducting a review test of my project, the end-user supplied feedback on my project which gave me greater insight onto the workings and performance of each part of my program.

The references in brackets constitute to the corresponding objective criticised.

Below are the comments from the end-user:

- The menu UI is clean and very easy to navigate around.
- You have met the objectives I've set you in terms of providing some portfolio analysis and objective to-do lists. (2 and 3)
- I can see that your stock prediction works very well, but I would like to have some sort of graphical representation of how the trend behaves. (2.2.1)
- I would like to see some indication of the transaction price of when I am buying or selling stock. This would make it far clearer of how much money will come out or into my account. (2.4)
- The addition of using graphical representation of stock data makes the program very realistic to the real stock trading applications. Having that feature made the program more intuitive and made my feeling of making transactions more personal. (2.1)
- I really liked having the data such as the high, low, open, and close prices of the stock but what would extend the project is to include other information such as stock volume and rolling period averages. This would make it far better when analysing a stock trend. (2.2)
- Within the portfolio section, having a drop-down menu when selecting a stock for selling would be far more useful then manually inputting the stock symbols. (3.2)
- When completing an objective, it would be nice to have an option to see those objectives again and maybe place them back into the table. (4.1.3.4)
- For the simulation, I would think that having a graphical representation in addition to the table would make it far easier to analyse the trend of stock prices. Even better, having some analysis relating to the simulation itself. Maybe talking about the average price over the simulation or the highest/lowest would also be a bonus as it allows the user to gain some sort of overview of how the stock performed. (5.3.1)

Analysis of end-user feedback

I found the end-user to have really enjoyed using my system with the core compliments stemming from the realness of the system compared to others that use real money and have other complex features. Regarding the feedback, the comment related to the graphical representation of the stock prediction I feel would be easy to implement. If I were to shift the graph showing the stock prices and add another one, it would show how the trend was changing. After thought, I believe this would play a greater role for users especially when they make their financial decisions for buying or selling as graphical diagrams provide a bigger effect than just text.

Furthermore, with concern about showing the price of the stock before performing a transaction buy or sell, I personally believe that this issue should have been addressed as it plays an important role in the user experience. By Showing users the price of their transactions, it will allow them to make more informed decisions about their investments and will help them decide if a stock is worth buying or selling. This may be solved by making an API call before they are prompted with buying/selling stock to then calculate the price of the transaction and display it to the user. However, one of the limitations of this implementation arises from the fact that the API supplier I use limits 9 API calls per minute and calculating and showing the price of the transaction before the transaction process takes place means that it reduces the number of calls it makes. This may then inhibit a good user experience for the program if further exceptions regarding API limit emerge. Adding this feature would benefit this system.

Additionally, in relation to the comment about having the option to reinstate objectives after they have been completed, I personally believe that it is a logical and trivial improvement to my program as for users, having this feature will provide a safety net in case they accidentally delete an important objective from the list. This will prevent frustration for the user and will make the user feel more controlled with their data. With reference to my program, to combat this, I would simply need to change the status of the objective wanted to be rectified back to false so that when they are displayed within the table, they will show up. However, the rectification of these objectives may lead to data redundancy within the database as we must store objectives after they have been deleted – these objectives may not be touched again. As a result, this will make it difficult to maintain the data integrity diminishing the program's performance.

Finally, touching upon the comment made about the graphical analysis of the stock simulation, I would most likely combat this by including a graph which would plot these data points to which the user can view. I felt this would be much better than having the current system of a table as it provides a visual aid for the user to understand the trends and patterns far better. This would make it far easier to make more correctly informed decisions on the simulation of stock. Regarding the addition of simulation analysis, I would most likely add some of the analysis techniques mentioned using simple functions or maybe store the

data within the database which I can call aggregate functions to make it far easier. This would be far more impactful to the user as it would give the user a comprehensive overview of how the stock performed and behaved over time. However, one problem that poses to me is the performance of the solution. Since the simulation can involve many simulation days and simulation numbers, having graphs to showcase this data may cause delays in loading data graphically and may cause more time lost when calculating all the statistics required.

Final Remarks

From the feedback collated, I believe that overall, my end users and testers have found my project to be a success and from my view, it has allowed me to develop skills in both developing a project linked to my interests and has given me a new opportunity to see the construction of programs within software development. In my opinion, some areas such as my database classes could have been constructed far better as more of an interface rather than just classes and methods and that would have played well in terms of the interlink of the classes themselves. Furthermore, within the API classes, it would have played well if I introduced processes such as caching data which would remove the dependency of always going over the API limit mentioned within the provider's terms. This caching would've have played far more efficiently if a user was to search for a certain stock more frequently to which the API call wouldn't be needed.

To implement these changes would've provided a different structure to my program but, I would argue that my program completes the objectives that I set and the requirements of the end user to which I confirm this project as a success.

Bibliography

Chen, J. (2021, December 06). *Guide to Technical Analysis*. Retrieved from Investopedia: https://www.investopedia.com/terms/t/technical-analysis-of-stocks-and-trends.asp

Farley, A. (2021, September 29). *Pros and Cons of Paper Trading*. Retrieved from Investopedia: https://www.investopedia.com/articles/active-trading/072915/pros-and-cons-paper-trading.asp