GDP Final Report
Simulated Stock Exchange

Capula project

Project Aims

Capula

Capula (investment fund, 30Bn AUM) asked us to create a simulated stock exchange, the core of which would be an order book. It should not be a stock exchange that tracks stocks' values from other exchanges- it should determine the value based on the order book's behaviour. They gave us the idea of building it in Python, and then adding a database and a UI to make the order book actually feasible. They also suggested that down the line we could have market-maker bots to keep the market liquid. They gave us broadly free range on what kind of users we wanted this, and the specifics of the project. We decided to make this project with a focus on the database and order book functionality, as well as UI and usability. We also had plans to make this into a competition and to be able to give prizes to people with the highest returns. Overall, our goal was to create a simulated stock exchange that would help people gain a better understanding of the market.

Personas

Myles: Beginner, History first year
Myles is a first-year History student at Oxford with no prior knowledge of investing. He downloaded the app to have fun simulating trading and maybe win the prize. He doesn't know what volume, EBITDA, or order books are. He logs in just to check how his investments are doing and occasionally decides to buy or sell. He wants the app to be simple and intuitive.

Sofia: student interested in Quant, CS student 3rd year
Sofia is a third-year CS student with strong quant knowledge. She uses the app to test her own strategies and relies on the API to automate trades. She's serious about it but doesn't really talk about it with others. She sees the app as a space to experiment without risking real money.

Travis: student interested in discretionary investing, E&M second year
Travis is a second-year E&M student who knows a lot about companies and discretionary investing, but less about the mechanics of markets. He understands

volume and order books at a basic level. He uses the app to see if he could be good at quant trading. He enjoys betting on outcomes and seeing how sentiment affects prices.

Requirements

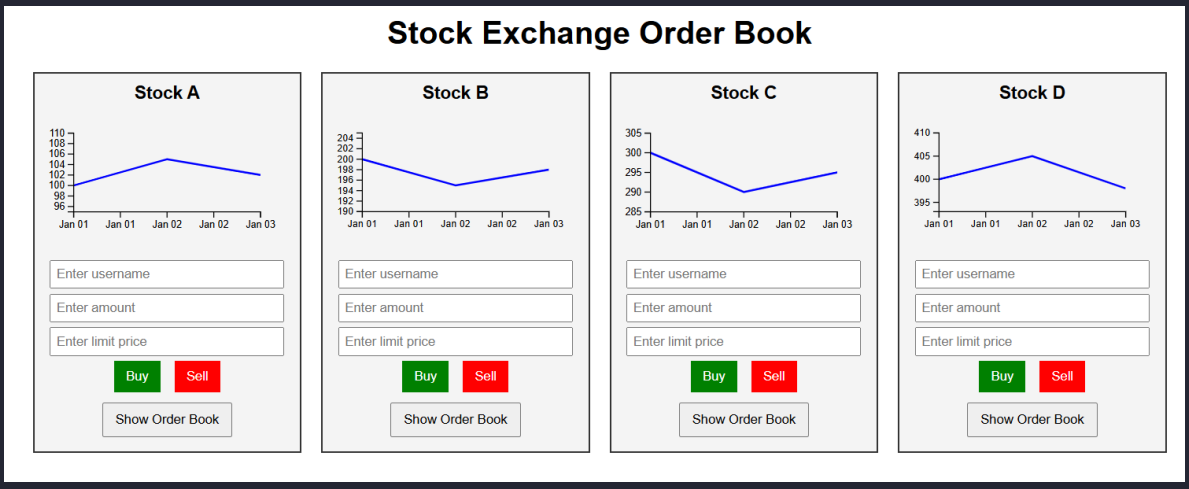
The system allows the user to bid for stock
The system allows the user to sell stock
The system allows the user to view their current inventory
The system allows the user to view up-to-date information of the most recent transactions
The order book matches a bid and a sell request such that the bid is higher than the sell request
The order book records buy and sell orders

The order book records the transaction history for all stocks and users
For a user the system should store their username and email
For a stock the system should store the owner and the price paid for it
The order book allows a trading bot to connect to it
The system authenticates users

Outcomes- current deliverable

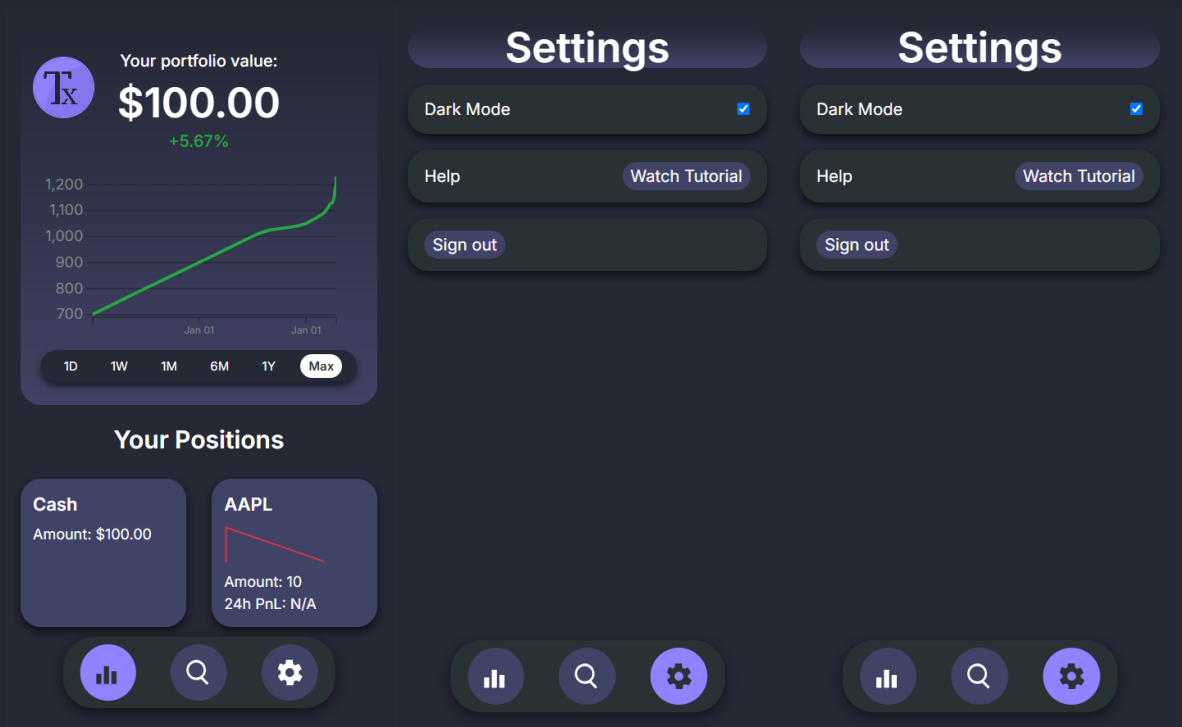
Front-end & Design:

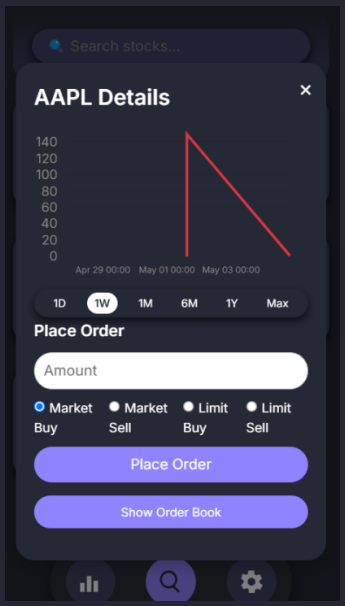
First version of the Front-end showed a very bare picture of everything that was necessary before any log-in or personalisation:



after this, it became crucial to improve the UI/UX after the core technical specifications were done. The UI is built with vanilla JS, HTML, and CSS. The graphs, for the stocks and the portfolio, is build with d3.js. Most of the inspiration came from Pinterest and Dribbble, and previous experince with Trading 212, which provided a good benchmark

for what a UI should include, and what could be improved. The UI should be both intuitive to users with little experience, whilst catering to the needs of those with a deeper knowledge of the markets. We decided to incorporate a three-page system with a settings view, a search view, and a portfolio view. (below).

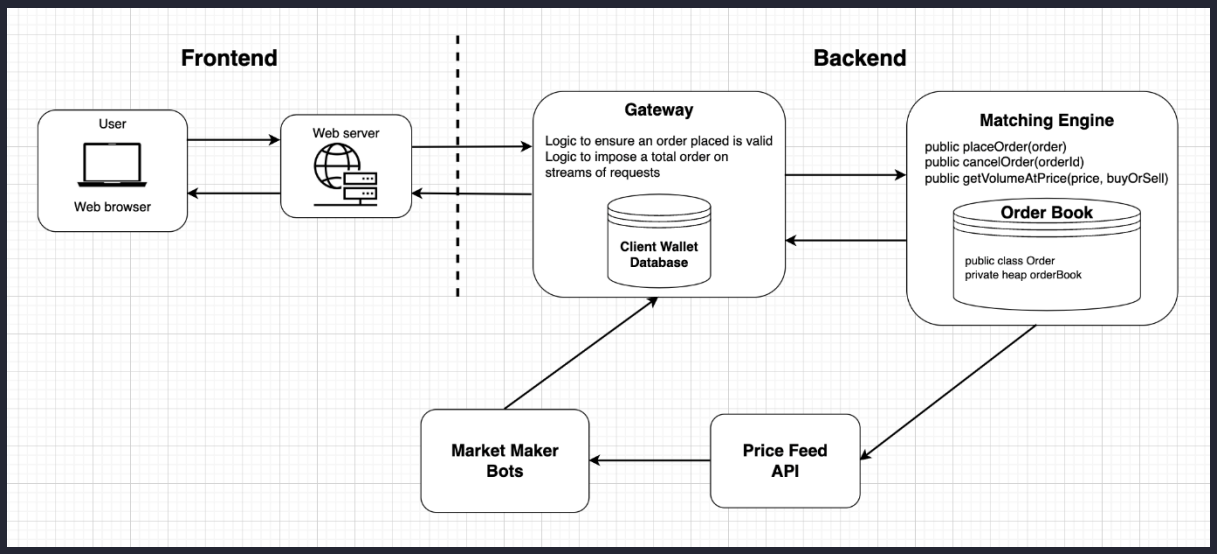




The graphs in each enable users to quickly understand the behaviour of their portoflio or of each stock, and were incroporate with d3.js, with data joins, multiple views, color automated changes on PnL, and ability to move the graph (updating in real time to axes). Each stock can be pressed to reveal a detailed page (left) which enables users to pace orders, look at the order book, and a more detailed graph of the stock's price.

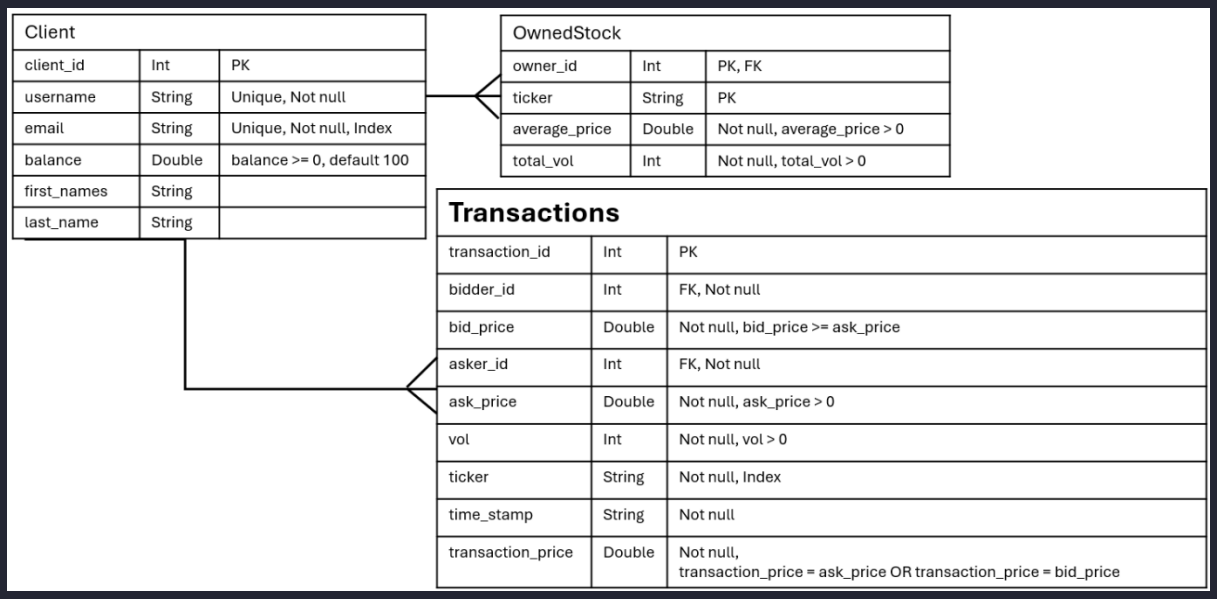
Back-end

System Design



The above system diagram shows the key components of the stock exchange that were considered early into the project. End-consumers interact with the app via the web/mobile UI, and we expose an API for developers to send automated trades.

Database



Client

client_id	Int	PK
username	String	Unique, Not null
email	String	Unique, Not null, Index
balance	Double	balance >= 0, default 100
first_names	String	
last_name	String	

OwnedStock

owner_id	Int	PK, FK
ticker	String	PK
average_price	Double	Not null, average_price > 0
total_vol	Int	Not null, total_vol > 0

Transactions

transaction_id	Int	PK
bidder_id	Int	FK, Not null
bid_price	Double	Not null, bid_price >= ask_price
asker_id	Int	FK, Not null
ask_price	Double	Not null, ask_price > 0
vol	Int	Not null, vol > 0
ticker	String	Not null, Index
time_stamp	String	Not null
transaction_price	Double	Not null, transaction_price = ask_price OR transaction_price = bid_price

To allow for faster processing in the backend, the database was designed to store the current state of the system so it could be restored in the case that the server went down. The data stored in the table is as followed:

- **Client:** For each user who logs onto the server, it stores their contact details to identify users, and their balance to record progress

- **OwnedStock:** Stores an accurate record of what stocks are owned by who, as well as the average price paid for them
- **Transactions:** Stores a complete history of all transactions made on the server, including bid and ask details

The tables have requirements to ensure they match the conditions placed by the order books, such as ensuring that transaction_price can only take ask_price or bid_price.

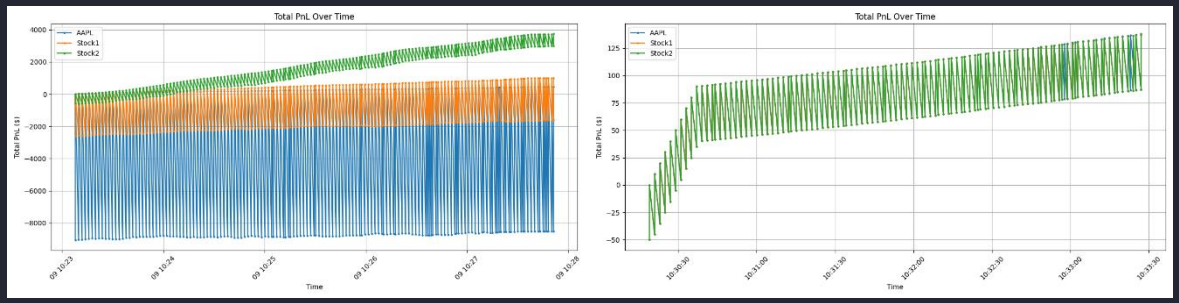
For the database implementation, sqlite3 was used due to its simplicity and ease of maintainability. It has limited typing, which is why time_stamp has been stored as a String, but due to no queries looking at the time_stamp, this is a limitation that has been avoided. Two indexes have been created, Client(email) and Transactions(ticker) to speed up searches used by the backend, and all database access has been wrapped in a class to allow it to be used as a library by other members of the team. It also has a test-suite, which passed all tests when test data was placed into the database, but this test data has since been removed to reduce the confusion with real data.

Order Book

The order book was designed to handle limit and market orders. We used python due to its ease of use. At the scale we are using the order book, we will not be held back by its efficiency, and due to the complexity of the order book logic the development would be fastest if using an easy-to-read language rather than a more efficient language that team members would have to learn.

Trading Bot

The app exposes a websocket API to stream live price feed data, as well as endpoints to place limit and market orders. A simple market maker bot was tested on the platform to place limit orders above and below the mid price for each ticker, in the absence of other market participants (below right) and again with a simple time series that models fluctuating price (below left). A fixed order size leads to heavy fluctuations after placing orders with high risk, but the bot manages to capture spread in both markets over time.



Successes & failures

As our project evolved, playtesting the UI/UX revealed to us some successes and failures of the order book. For example, we realized that stock values were only being updated periodically with the data. We hence implemented webSocket auto-updates for the data, as well as data joins with d3.js to update automatically when the data updates.

One failure of the database occurred during testing, where it revealed that the current functions could not catch errors originating from the database itself, such as when the Unique constraint was violated. This has resulted in pre-conditions for any functions involving an INSERT statement that any unique constraints are already satisfied.

Another failure of our design process was that our core order book logic was written in one file. While this doesn't have any impact on functionality, it is not conventional to have this many Python classes written in one file. Hence, if we had identified this potential problem earlier during the design process, we could have built more modular code with less co-dependencies between classes, hence allowing each class to be written on its own file.

Lessons learnt

Project management: assigning tasks to members of the team based on a combination of how much one wishes to learn that task and skill in that task is a good heuristic
Collaboration: communication is key, and having well-documented code even more so.
Order Books: efficiency is not always the end goal, reliability matters a lot more before considering speed.
UI/UX : prettier isn't always better - sometimes one must sacrifice a cool design for usability and cramming all the features that we want and making them accessible (e.g. buttons for order book and buying/selling)

Responsible Innovation

The system we have created is a platform for users to learn more about the stock market, and to build users' confidence in their trading strategies. This can get more people involved in the stock market and teach users to better navigate risk. It can also help investment companies gain experienced traders and provide safer training. However, as this is a simulation, it could lead to the users gaining false confidence, resulting in risky choices on the actual stock market and potentially high losses.

The system could also be used in competitions, to help companies identify budding traders, as well as a tool for campus recruitment. However, this will draw malicious

users to the platform, who will either try to sabotage the competition or find loopholes to ensure they win. However, we've put in protections to try to reduce this.

Potential fraud was a major ethical issue, so many protections were put in place regarding this. Firstly, it is impossible for the same account to be the bidder and asker of a transaction, blocking the case where the user artificially drives the value of the stock up or down. Secondly, we've used google authentication to the system, to prevent multiple accounts being made with false or random emails. Whilst this still allows users to have multiple accounts, this condition makes it harder to do on a large scale. As well, the database records allow accusations of fraud to be investigated by identifying the owners of accounts and analysing if accounts have made repeated trades with the same people.

Using google authentication also allows us to ensure that the user's accounts are secure, as the protections google has for their passwords are more secure that our server's protections could be with the resources we have available.