COMP30220 Final Project

Ethan Epperson

Id: 25201495

Alex Waldman

Id: 25204001

David Martinez

Id: 25202624

Synopsis:

Describe the system you intend to create:

- We intend to create a system that simulates the stock market on a very small scale, allowing a player to use funds in a bank account to buy stocks from a couple companies whose stock prices fluctuate. The player will be able to build their portfolio, buying and selling shares.

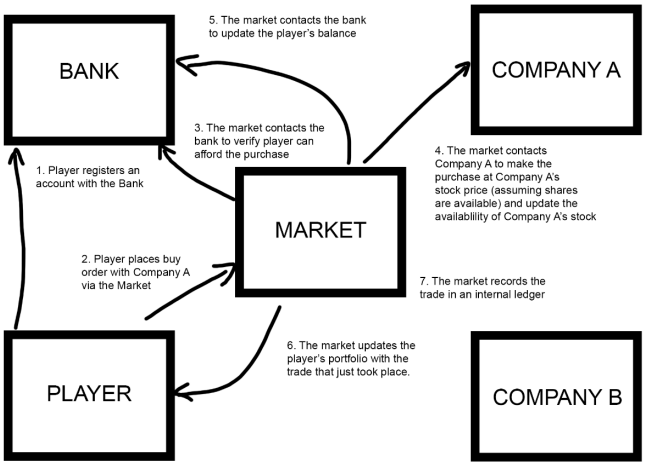
What is the application domain?

- Our application is split into several different services. There will be a service for the player, from which the player can manage their portfolio and check on their bank account. There will be a service representing the market that acts as a broker between the player and the companies on the stock exchange, facilitating buy/sell orders and updating the player's portfolio. There will also be a service for the bank, which the player will register with to make an account. The bank service will communicate with the market service during transactions to make sure the player's balance is accurate. Additionally, the bank will update the player's balance according to its interest rate. Lastly, there will be services for each company that will communicate with the market service during transactions.

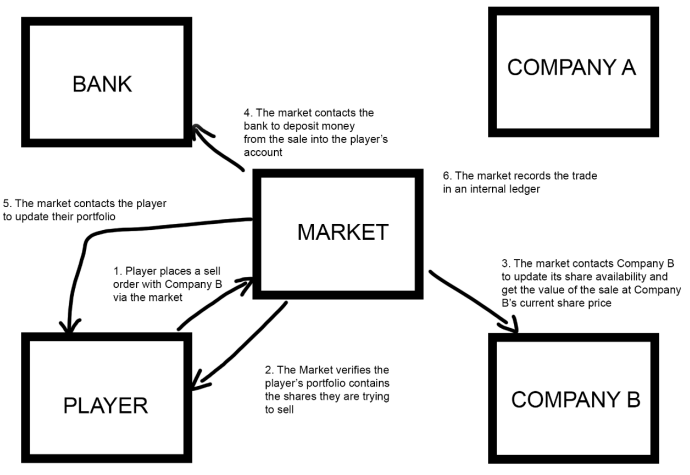
What will the application do?

- Upon startup, the player service will automatically register an account with the bank. From then on, the player will be able to buy and sell stock at their leisure by accessing the endpoints defined in our code, assuming they have the funds available and/or the necessary shares in their portfolio to complete the transaction. In addition to simply buying and selling, the player will be able to monitor their portfolio and bank account as they continue to make transactions in the market.

BUYING SHARES



SELLING SHARES



System Architecture:

Provide a system architecture diagram that highlights the main components of your system and flow of interactions. Explain how your system works based on the diagram.

To the right, we have a system architecture diagram that displays the flow of interaction between the various services in our system when the player is buying or selling stock. Aside from creating a bank account, which the

player does by contacting the bank service directly, all actions taken by the player in our system go through the market service, which acts as a broker between the player and the rest of the services in addition to having its own functionality (like keeping track of all the trades that have been carried out by the player). When initiating any buy/sell order, the player first contacts the market service to request the transaction take place. From there, the market service must verify the player has enough money (in the case of a buy) or that the player's portfolio contains the necessary holdings (in the case of a sell) to complete the transaction. From there, the market service contacts the company involved in the trade to verify/update relevant information before contacting the bank again to update the player's account balance. Lastly, upon updating the player's bank account balance, the market always records the details of the trade in an internal ledger, which allows the system to maintain accurate records and potentially roll back transactions if necessary.

Design Considerations:

Explain how your system is designed to support scalability and fault tolerance.
Discuss any Microservices Design Patterns you have used (and why)

- Due to the stateless nature of REST systems, our system is capable of handling a larger load of requests from the player (or multiple players) in the future because none of our services need to store data related to the client in between requests, meaning a higher volume of requests will not cause memory issues. Additionally, splitting the player from all of the other services allows each aspect of the system to be developed independently of one another, making the scaling process easier. Further, the use of client classes (e.g. BankClient, CompanyClient, etc.) within each service allows for abstracted inter-service communication. When a service is dependent on another service for a particular request, the client classes become the mechanism for adding this functionality. As such, services again can develop and scale independently.
- Our system contains lots of error handling code with meaningful error messages, meaning if certain services go down, our system can handle it without crashing and inform the user about what exactly is going wrong at the moment.

Contributions:

Ethan Contributions:

1. Core Module
    - Features the following classes:
        - BankAccount
        - Client (Player)
        - Company
        - ShareHolding
        - Trade
2. Market Module
    - Clients Package: BankClient, CompanyClient, PlayerClient
        - Classes in this package are intended to abstract the communication between the MarketController and other services. For instance, the MarketController delegates the withdrawal of a player's funds to the BankClient
    - MarketController
        - Endpoints:
            - GET /companies - get list of companies in the market
            - POST /companies - add a company to the market
            - GET /companies/{companyId} - get a specific company's info

        - DELETE /companies/{companyId} - remove a company from the market
        - POST /companies/{companyId}/buy - buy company shares
        - POST /companies{companyId}/sell - sell company shares
        - GET /trades?companyId=... - get all trades / trades related to specific company
- Market Package
    - Exception
        - Contains user defined exceptions
    - Handler
        - Defines a global exception handler. Maps exception types to handler methods. When exception are thrown, Spring searches for the matching exception handler.
    - MarketService
        - Handles the business logic of the market (e.g adding companies, storing trades, etc.)
- Utils Package:
    - TradeDTO
        - Data Transfer Object used as a return type in MarketController Endpoints
    - TradeMapper
        - Maps Trade Objects to TradeDTOs
    - TradeRequest
        - Record class used as a parameter to trade related market endpoints
    - TradeResponse
        - Record class used as a return type from CompanyClient
- Resources
    - Developed open api documentation to model the MarketController

3. Company Module
- CompanyController
    - Endpoints
        - POST /company/sell?quantity=... - sell company shares
        - POST /company/buy?quantity=... - buy company shares
- Resources
    - Developed open api documentation to model the CompanyController

Alex Contributions:

1. Bank Module
- BankService
    - Manages the players account by facilitating account creation/deletion, depositing/withdrawing funds, applying interest rates, and retrieving an account's balance
- BankController
    - Endpoints:
        - POST /accounts- create a new account at the bank
        - GET /accounts/{accountId}- retrieve an account by its unique ID
        - DELETE /accounts/{accountId}- delete an account by its unique ID
        - GET /accounts/{accountId}/balance- get the balance of an account
        - POST /accounts/{accountId}/deposit- deposit money into an account
        - POST /accounts/{accountId}/withdraw- withdraw money from an account
- Resources
    - Developed open api documentation to model the BankController

2. Wrote the Synopsis, System Architecture, Design Considerations, and Reflections section of the report

David Contributions:

1. Dockerizing our services

2. Player Module
    - Clients Package: BankServiceClient, MarketServiceClient
    - PlayerController
        - Endpoints:
            - GET /players- get all players
            - POST /players- create new player
            - GET /players/{playerId}- get specific player by their ID
            - PUT /players/{playerId}- update player information
            - DELETE /players/{playerId}- remove a player
            - GET /players/{playerId}/portfolio- get a player's portfolio
            - GET /players/{playerId}/portfolio/value- get the value of a player's portfolio
            - GET /players/{playerId}/assets- Get a player's total assets (bank balance + portfolio value)
            - POST /players/{playerId}/buy- buy shares for a player
            - POST /players/{playerId}/sell- sell shares for a player
            - GET /market/companies- get market information about all companies
            - GET /market/companies/{companyId}- get information about a specific company
    - PlayerService

Reflections:

What were the key challenges you have faced in completing the project? How did you overcome them?

- The main challenges we faced had to do with the overall scope of the project. Initially, we wanted the user to be able to input their own trades using the command line interface. We also had the idea of there being multiple other players that the user would compete against in an attempt to outperform them in the market. Ultimately, the implementation of these features proved to be very complex and time-consuming, so we chose to shrink the scope of our application in order to focus our efforts on core features like buying and selling stocks.

What would you have done differently if you could start again?

- If we could start again, we would likely begin designing our system and solidifying the features that would end up in our program a little earlier on, as that would give us more time to build out the program's functionality and potentially expand upon our original vision. It also probably would have been beneficial if we merged separate branches of our code more often, ensuring that our code stayed updated even as services were developed individually.

What have you learnt about the technologies you have used? Limitations? Benefits?

- In working on this project, we learned a lot about resource-oriented design in relation to REST and found it to be really intuitive. We also gained more experience using tools like OpenAPI and Postman in an effort to document and debug our API. The main benefit of working with REST is ease of use. While there is a small amount of boilerplate required when it comes to configuring the server and setting up Springboot for Maven, it is fairly easy to develop APIs once you understand the general design philosophy. The use of HTTP methods like GET, POST, DELETE, etc. also made it easier to design our API because they helped us concretely define the functionality of specific API calls.

Video:

https://drive.google.com/file/d/1YkOfp8yFHiEteRJBBlEDbhch-Va5S2s7/view?usp=drive_link