Dumb Dumber Dumberer Dumbest

John O'Donnell	Kealan McCormack	Lukasz Filanowski	Gerard Colman
18368983	18312236	18414616	18327576

Synopsis

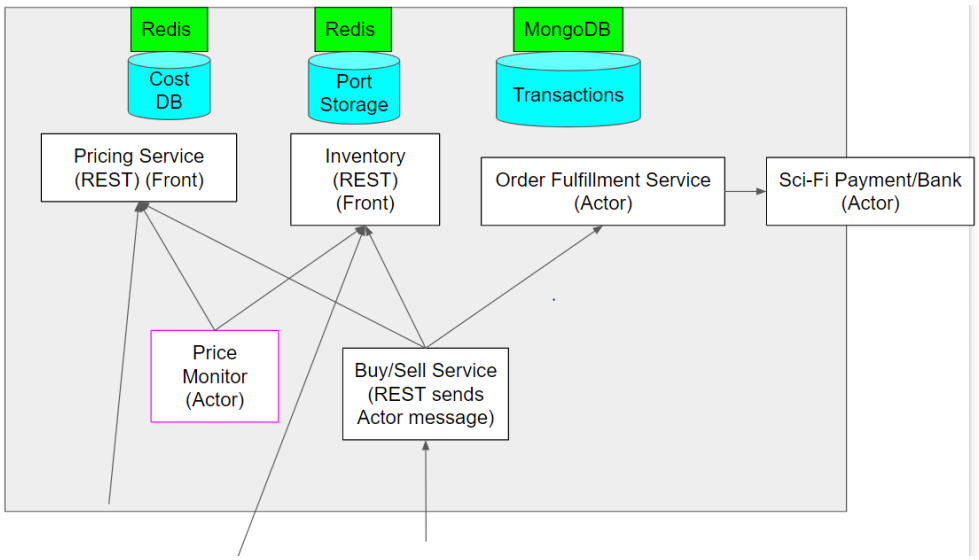
The application domain of this project is commerce and retail. The system is a simulation of a trading spaceport. The aim of this distributed system is to simulate the proceedings of an active spaceport where ships come in to buy or sell goods with the port. The goods are stored and catalogued in an internal inventory which the ships can access and buy/sell from. Once their trading is complete, a receipt will be sent to the ship, and it will depart from the port. The space port receives money from an external space bank to pay for the goods being sold.

Technology Stack

- **REST API's with Springboot**: Used to model InventoryAPI, Market, PriceMonitor and PriceAPI.
- **Actors**: Used to model SpaceBank and Orders.
- **Redis databases**: Key value stores for Inventory and Price.
- **MongoDB database**: Storage of documents of transactions and orders.
- **Open API/Swagger**: Documenting Inventory, Market and Price API's
- **Docker**: Used to convert each component into docker images to run independently.

System Overview

The system consists of 7 main parts that combine into a representation of a spaceport. Firstly, the **InventoryAPI** is used to keep track of the type and number of items present in the port's storage. This is done using a REST service on top of a Redis database. The cost of each item in the inventory is held in a separate REST based **PriceAPI**. The prices and names of each item are again held in a Redis database. The **client** can request to buy or sell an item or a series of items via a request sent to the Buy/Sell Service known as the **Market**. The Inventory is queried regarding the available amount of the item using a REST request. Similarly, the price is retrieved from the PriceAPI. Once an order is confirmed, a message is sent to the Order Fulfilment Service in module **Orders**. The Orders module is an Actor built on top of a MongoDB database used to store transactions that have taken place and the receipts. The Orders module also contacts the **SpaceBank** to ensure funds are available to make the transaction. Once the transaction is confirmed by the bank, the receipt for the transaction is sent to the client and the ship can depart from the port. Lastly the **PriceMonitor** acts as a sort of price adjustment service and rough economy simulator. If a large amount of a certain item is sold to the port, the price will be lowered to accommodate for the additional goods. If there is very little of a specific item, the price will be increased as it is rarer.



When a request is made for an item to be bought or sold through the Buy/Sell service, a REST request is sent to the Inventory and Price APIs to retrieve the pricing and amount of the item from the corresponding Redis databases. Checks are made to ensure the item is present in the inventory. The Buy/Sell service then communicates with the Order Fulfilment Service to finalize the transaction. The Order Service contacts the Bank to make sure the funds are available and saves the approved transaction into the MongoDB database. A receipt of the transaction is sent back to the client and the trade is complete. The Price Monitor then checks the amount and price of the items in the inventory and adjusts them accordingly.

Due to the use of NoSQL databases, the project is suited for scaling and both MongoDB and Redis are very scalable. Since each of the modules acts as a microservice, they can be easily deployed independently which provides fault tolerance.

Contributions

John O'Donnell 18368983

Work on Client and improvements to Market

Kealan McCormack 18312236

Work on PriceAPI, InventoryAPI and PriceMonitor

Lukasz Filanowski 18414616

Work on Market, Report writing and commenting

Gerard Colman 18327576

Work on Orders and SpaceBank

We all collectively worked on dockerising and deploying the project.

Reflections

The main challenges encountered in this project were teamwork and coordination issues. Due to having to work online and not being able to have in person team meetings due to Covid19, coordination was difficult. Time management was also an issue we encountered. This was particularly an issue when it came to the Orders service. Due to these constraints on time, the service couldn't be completed to a satisfactory degree.

If we could start the project again, we would focus on better prototyping of data types and APIs. We would also have clearer data types as this caused issues in the project. We would have also paid more attention to REST request standards and use standardised headers.

We have learned how to use and implement Redis databases. We have also expanded our knowledge on MongoDB databases and how they are implemented. This included learning about backend connections between REST and databases. The team gained insight into using Postman throughout the project's development and using Swagger for documenting. We have also refined our skills when it comes to implementing REST services and Actors.