**PleaseGiveUsAFirst**

Federico Carrocera

20434992

Kyle Draper

20500399

Nathan Betts

20358446

**Synopsis:**

*Describe the system you intend to create:*

*What is the application domain?*

The domain of our application is a stock trading system. It involves concepts like user accounts, stock trading, portfolio management, and financial transactions.

*What will the application do?*

This system aims to provide a Stock Market service to multiple clients. Key features include Signup, Login, News Reports, Stock Information Requests, Add Money, Withdraw Money, Buy Stock, Sell Stock, Get Portfolio Report, View Transaction History, and Advertisement Services.

***Services:***

***Advertisement Service****: Provides advertisements to be displayed on clients.*

***Data Service:*** *Exposes CRUD Database operations to other services.*

***Eureka Service:*** *Host Eureka Service for Service Registration and Discovery.*

***Management Service:*** *Performs Business logic such as generating portfolio report, buy stock, sell stock, etc.*

***StockAndNewsService:*** *Provides Stock and News information (Emulating Finance and News API’s)*

***User Service****: Manages User Logic such as login and create user, all client requests go here.*

The Data Service uses a H2 database to hold data, when the docker containers are composed for the first time the database will be empty, however, after the container has been composed, any data added to the database will be retained when the container is stopped/restarted. The H2 database itself is not distributed but this would be a good future step to take.

**Technology Stack**

*List of the main distribution technologies you will use*

* ***Eureka:*** *User for Service Registration and Discovery, Fault Tolerance.*
* ***Docker:*** *Used for Containerisation, Scalability, Fault Tolerance.*
* ***Swagger****: Used to keep a record of exposed APIs, Documentation.*
* ***SpringBoot:*** *Backend used to control and expose APIs.*
* ***OkHTTP:*** *Used to send HTTP requests.*

***Highlight why you used the chosen set of technologies and what was it about each technology that made you want to use it.***

**Eureka** was chosen for Service Registration and Discovery (which are essential components of microservices architectures) as it offers several advantages for microservices development, such as simplicity, resilience, and flexibility. It was easy to set up and use, with minimal configuration and dependencies.

**Docker** was chosen for the containerisation of each of the services due to its inherent ability to scale selective services. It also allows for the isolation of services from the User, allowing for important endpoints to be unreachable from the client application interface.

**Swagger** was chosen as it allows for an interpretable and easy-to-read interface for each of the service endpoints, as the services were split among the team members, it was invaluable for checking what endpoints were available from other members' services. It also allows for the ability to test these endpoints from the Swagger-UI interface.

**SpringBoot** was naturally chosen as the Framework for developing our application, given our experience in developing Java-based applications and the emphasis the module placed on Java-based implementations. Coupled with Maven, it allowed for easy configuration of necessary dependencies and handled a lot of configurations automatically. SpringBoot is also a production-ready Framework used in many Enterprise applications for its stability and flexibility which it offers.

**OkHTTP** was chosen as our HTTP client due to its popularity among Java-based applications. Its benefits which include its efficient handling of HTTP requests, automatic recovery in case of Network issues and easy-to-interpret and use API made it the natural choice for our application.

Leveraging the advantages which these components offer has enabled us to develop a fault-tolerant, scalable, distributed microservice architecture application.

**System Overview**

*Describe the main components of your system.*

The Main components of our application include

**Management Service**:

This service handles all business logic needs for User Portfolio Creation, Report Creation, Buying and Selling of Stocks. This Service manages User Transactions History, and relevant updating of User Balance and Transactions in case of Buy or Sell Option.

**DataService**:

This service holds the database and exposes CRUD database operation for the other services to perform. In addition to this, it exposes access to other services including the Advertisement service and StockAndNewsService.

**Advertisement Service**:

Basic service which serves advertisements to clients not logged in.

**StockAndNewsService**:

Provides Stock and News information to be served to the client. (Roughly emulates the responses from real finance and news APIs).

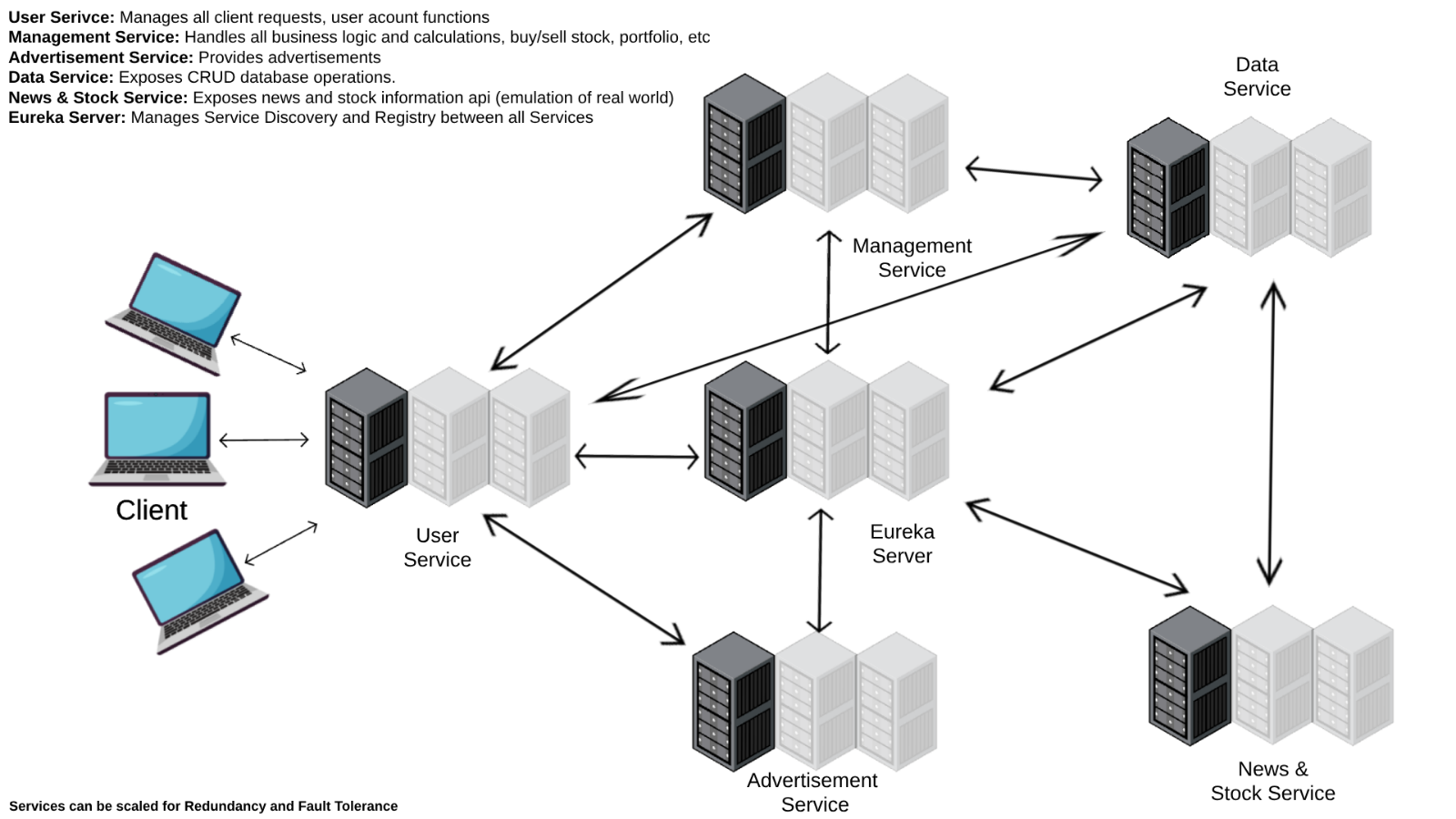
**Eureka Service:**

Manages Service Discovery and Service Registration to allow services to be added and removed as needed, allowing for a looser coupling between services.

**User Service:**

Acts as a gateway for all client requests and performs User related tasks, login/signup, etc.

*Include your system architecture diagram in this section.*

**

**Sequence Diagrams:**

**When a User is Logged In :**

1. Buy Stock

2. Sell Stock

3. Get Stock Information

4. Get Trending Stocks

5. Get Portfolio Report

6. Add Funds

7. Withdraw Funds

8. Logout

**A diagram of a company

Description automatically generated1 - Buy Stock**

**2 – Sell Stock**

**A diagram of a process

Description automatically generated**

**A diagram of a diagram

Description automatically generated3 – Get Stock Information**

**A diagram of a company

Description automatically generated4 – Get Trending Stocks**

**5 – Get Report**

**A diagram of a company

Description automatically generated**

**A diagram of a diagram

Description automatically generated5 - Get Portfolio**

**6 - Add Funds**

*Get Balance*

*Balance = Balance + amount*

**A diagram of a diagram

Description automatically generatedUpdate Balance :**

**7 – Withdraw Funds**

*Get Balance*

*Balance = Balance – amount*

**A diagram of a diagram

Description automatically generatedUpdate Balance:**

*\*Note, for the sake of brevity, communication with the Eureka server is only shown after the User Service in the above diagrams once, in reality, every time any of the services communicate with each other, the Eureka server is contacted to get the Hostname of the service, with that hostname being used to contact the service.*

Eureka Server: <http://localhost:8761/>

User Service Swagger: <http://localhost:8080/swagger-ui/#/user-controller/>

***Explain how your system works based on the diagram.***

The User is presented with multiple options of functions available to them before and after login.

The in-depth Sequence Diagrams above outline the individual interactions at a high level between the services to complete each of the functions. All requests from the Client go through the user service which performs the logic on user-related requests such as login or signup and forwards all other requests to the appropriate services. At each step of interaction between the services, the Eureka Server is used to ensure that a valid hostname is being used to contact the required service. This means if a container goes down and comes back up on a new hostname or a new container is launched in between requests, the required service can still be contacted.

Some functions and features such as the advertisement display feature, when a client isn’t logged in, just require one request from the user service to the service before returning the response to the user, however, most other requests require significant communication between the services such as the get report request which involves requests and communication from nearly all of the services.

***Explain how your system is designed to support scalability and fault tolerance.***

The service containers are equipped with an AUTO\_RESTART policy, designed to kick in and restart services should they encounter any failures. This coupled with Eureka Service Discovery, allows the seamless reconnection of services within the Application framework, ensuring a smooth continuation of request processing even after an interruption.

In terms of scalability, the Dockerisation of the application allows the scaling of each service selectively, tailoring the allocation of resources to match their individual usage and respond dynamically to changing demands.

**Contributions**

*Fede: UserService, Client Application Interface, Swagger, Dockerisation*

*Nathan: Management Service, Client Application Interface, Swagger, Dockerisation*

*Kyle: StockAndNewsService, DataService, Eureka Service, Advertisement Service, Client Application Interface, Swagger, Dockerisation*

**Reflections**

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

When Dockerising the application, the containers wouldn’t connect to each other or the Eureka Server as most didn’t have their ports mapped (purposely). To solve this problem, a virtual network was used leaving only the user service exposed to the client (And the Eureka Server but that is for demonstration purposes only).

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

Distributed Database – If we had additional time, we could have used a distributed database as another additional technology, this would have provided more fault tolerance, flexibility, scalability, and reliability.

Build a front end – A proper front end to interact with would be a nice touch to add for the user, the fact that our application is designed for total functionality via API is a positive, allowing for many platforms and versions of supporting frontend to be developed as long as they support REST API communication.

Additional Functionality – Adding additional functionality through a different language such as Python would’ve been an interesting, maybe a machine learning model in some sort of Federated Learning approach, showing the extreme benefits of distribution.

Tight Coupling – The User Service acts as a gateway for the Client to communicate with the other services while also performing user operations such as signup and login, this is good from a Security Perspective as the client does not know the inner workings of the systems, the problem with this is, for all functionality with the client, the user service must be running. Some possible solutions to this include making the client call all of the services directly (not ideal), adding an additional service which performs no logic but simply forwards on the requests (a gateway, this is similar to the user service but without the user logic) and replicate this service many times for fault tolerance and redundancy, could also act as a load balancer in a similar fashion.

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

The built-in SpringBoot HTTP client is very useful when compared to the standard Maven OkHTTP client. The SpringBoot client is easier to work with.

SpringBoot also makes exposing endpoints quite a simple task while allowing complex Java and OOP programs to be used as the backend for these APIs. Linking these APIs up to functions databases and other services is a satisfying experience and really shows the power of these distributed systems with regards to their ability to be language agnostic, we could have coded any of our services in Python, C# or a multitude of other languages and the client would be none the wiser.

For Service Discovery and Registration Eureka was invaluable to the project, allowing for the compliance of the application to industry standards. Real-world services make use of Eureka and including it in the application albeit tricky, was an incredible asset to the application’s capabilities as well as our own experience.

Swagger is great for understanding, referencing, and describing exposed APIs to allow for the development of software and applications which communicate with these APIs.

Docker, while sometimes tricky to use and setup is a very powerful tool, the fact that it allows for the orchestration of multiple containers at once through the docker-compose file and its ability to spin up / shut down containers as needed in response to faults or scalability needs is quite significant. Docker makes managing the entire distributed system a simpler task, rather than having to *‘mvn install’* and *‘mvn exec java:springboot -pl …’* multiple times, a simple *‘docker compose up’* can be run to achieve the same result and more. It also allows for extension of the system as needs change and when coupled with Eureka, allows for releases to be seamless and unnoticed for the end user by simply spinning up a new container and then shutting down the old one.

If this system was really going into large-scale production, a container orchestration tool such as Kubernetes would be good to consider allowing for advanced scalability with many containers across multiple zones to allow for a global distribution of the system.