# Assignment 2
# Implementing a Globally Accessible Distributed Service

Distributed Systems - CS7NS6
Group 9

# Cafe Loyalty Program

| | | | |
|---|---|---|---|
| Anastasiia Khomenko | Charbel Nebo | Karan Dua | Shauna Gurhy |
| 22304231 | 18324228 | 21331391 | 18324848 |
| khomenka@tcd.ie | neboc@tcd.ie | duak@tcd.ie | gurhys@tcd.ie |

## *Introduction*

The main task of this project was to design a Distributed Cafe Loyalty Program application with high-level architecture of a globally accessible system. Therefore, a loyalty program was developed and created aiming to be potentially scalable and used by a large number of users who could perform the following actions: registration and creating new accounts, generating, getting and redeeming vouchers, viewing transaction history and updating transactions.

In order to achieve the set goals we decided to use microservice architecture, as it provides a lot of benefits in scalability (ability to increase and decrease the scale of certain services without impacting the system), flexibility (ability to change services independently without impacting the system), resilience (the ability of the system to continue working if one of the services is failed) and simplicity (convenience and clarity in working with microservices).

Spring Boot framework was used to develop all needed services, as we found it quite clear to use and helpful in performing the necessary tasks. A detailed description of using Spring Boot framework and the creation of microservices can be found later in the report.

Besides that, solving the problem of storing data related to customers and their transactions we decided to use MongoDB, as it is horizontally scalable which makes it the perfect choice for our application as it requires high availability and scalability and allow to handle a large number of high-speed, low-latency read and write operations required for application.

Also, in this project were used Docker and Kubernetes to containerise the Spring Boot microservices and to manage the containerised microservice what makes deployment, management of the app and managing multiple instances of the application across multiple servers easier. In addition, we tested all the requirements both functional and non-functional, using Postman as a tool for testing the REST API endpoints, and included the results in the report.

## *Requirements*

A Distributed Cafe Loyalty Program application was developed to reward customers of a global café chain for their transaction or purchase history with discounts on their in-store purchases. While it has been developed and all functional requirements have been met on a small scale to show design and functionality, considerations were also made for non-functional requirements such as performance, scalability, reliability and availability so that it could have the potential to be scaled up into a globally distributed system.

## *Functional Requirements*

The functional requirements of the Cafe Loyalty Program application are:

1. **API Gateway -** Required to provide a single entry point for clients to access multiple microservices.
2. **Service Registry -** Required for communication between various microservices..
3. **Create Customer -** Required to create a new customer entity based on the information provided.
4. **Fetch Details -** Required to retrieve customer details based on the information provided and to update the cache with updated customer details.
5. **Login -** Required to validate customer login credentials (username and password).
6. **Update Transaction -** Required to update transactions of customers.

7. **Get Transaction History -** The transaction history should accurately reflect the previous transactions.
8. **Generate Voucher -** Required to generate vouchers when enough points have been accrued by the customer.
9. **Get Voucher -** Required to output a  list of vouchers reflective of those generated by the user.
10. **Redeem Voucher -** Required to allow customers to redeem vouchers against their purchases.

## *Non-Functional Requirements*

**Performance**

To achieve high performance, Spring cache was implemented for fetch customer service. This process was implemented in order to reduce the consumption of resources and time in processing data of existing customers, since the system must check the history of purchases and transactions and determine whether the customer has the opportunity to receive a discount. We have added a mechanism to add customer details to cache when a customer has more than 10 transactions in a month. This is done in order to store details of frequently visiting customers in the cache for faster retrieval of customer details and overall performance improvement.

**Reliability**

To improve the reliability of our application, we have implemented Master-Slave replication strategy for our database. This will provide better reliability for our system as in the event of a database failure we can involve promoting a slave database instance to become the new master, or restoring from a backup.

**Scalability**

To achieve high scalability, we used microservice-based architecture, as it allows us to scale up and scale down certain services without impacting the system.

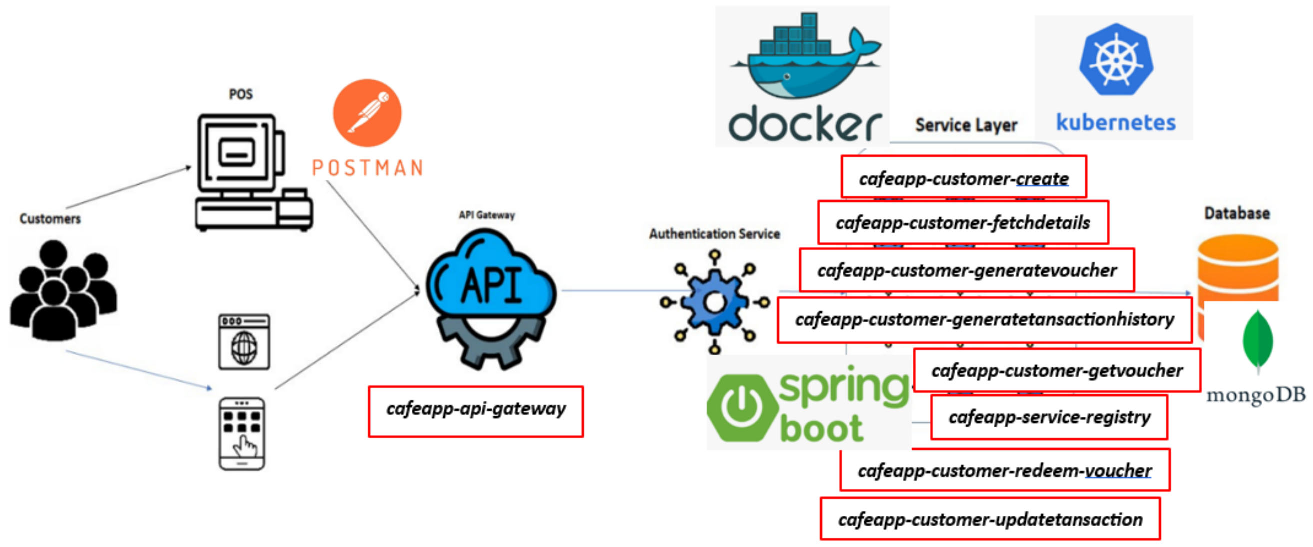
**Availability**

We have deployed our application on Kubernetes to ensure high availability of all microservices. Also, Kubernetes ensures equal distribution of load on all pods for a microservice, thereby ensuring that the workload is evenly distributed and the application remains responsive.

## *Architecture & Specification*

The Cafe Loyalty Program application has a spring boot microservices architecture, with each service running in a separate Docker container. The system is made up of several services. Each service is responsible for a specific function and communicates with other services through REST API's.

The figure below illustrates the architecture diagram for our application.


POS
POSTMAN
Customers
API Gateway
API
cafeapp-api-gateway
Authentication Service
docker
Service Layer
kubernetes
spring boot
cafeapp-customer-create
cafeapp-customer-fetchdetails
cafeapp-customer-generatevoucher
cafeapp-customer-generatetansactionhistory
cafeapp-customer-getvoucher
cafeapp-service-registry
cafeapp-customer-redeem-voucher
cafeapp-customer-updatetansaction
Database
mongoDB

## Spring Boot Microservices

Spring Boot is the framework that was used to build the microservices in our globally distributed system. It was chosen as it provides a simple, easy-to-use framework for building microservices and has a wide range of pre-built libraries and modules that can be used. Microservices built using Spring Boot can be easily scaled up or down based on demand, making it easier to handle large volumes of traffic and ensuring the system remains responsive even during peak periods. The microservices can be configured to automatically recover from failures and handle errors.

### Microservices Implemented in our Globally Distributed System

#### *cafeapp-api-gateway*

This microservice serves as an API gateway for the Café Application. This provides a single-entry point for clients to access multiple microservices. The CafeAppApiGateway routes incoming requests to our other microservices responsible for different functionalities of the application.

The application uses the Spring Cloud Gateway library to handle incoming requests and route them to the appropriate microservices. It includes several routes that specify the URI paths and filters to apply to incoming requests. Each route includes a fallback URL that specifies the endpoint to call if the microservice it routes to is unavailable. To include failure resistance, the Netflix Hystrix library which provides circuit breaker functionality was added to handle failures and prevent cascading failures across microservice.

#### *cafeapp-customer-create*

This microservice is responsible for creating new customers for the Café app. It contains a *collection* of all the data fields needed, a Spring REST *controller* that handles HTTP requests to create a new customer, a single endpoint ***/create*** that accepts a CreateCustomerPojo object as a request body and returns a ResponseEntity. It contains a MongoDB *repository* used to interact with the database and a *Service* for creating a new customer entity based on the information provided.If any errors occur during the creation of a customer, then an appropriate response message is returned.

### *cafeapp-customer-fetchdetails*

This microservice has two REST endpoints responsible for retrieving customer details based on the session information provided and for updating the cache with updated customer details, based on the number of transactions of the customer. It contains a *collection* of all the data fields needed with classes Customer, LoginDetails, ResponsePojo, Session, TransactionHistory and VoucherDetails whose contents are outlined in the previous section, a Spring REST *controller* that handles HTTP requests for the /fetchdetails and /updateCache endpoints, a MongoDB *repository* that is used to interact with the database and a *Service* for retrieving customer details based on the information provided and for updating the cache with updated customer details.

### *cafeapp-customer-login*

This microservice is responsible for validating customer login credentials (username and password) and interacting with the database and query for customer records. This service takes the credentials and checks the database for a corresponding customer record, if a matching record is found and the password matches, then a session is created. If customer details are not found or entered credentials are invalid, then it returns an error.

### *cafeapp-customer-generatevoucher*

This microservice is responsible for generating vouchers for the customers. We have implemented a logic that generates a €5 voucher for the customer for every 500 loyalty points that the customer has in his rewards balance. The customer can generate multiple vouchers at a time. Furthermore, if customer details are present in the cache, then cache is updated with new voucher details using the update cache endpoint in fetch customer microservice.

All vouchers have a validity period of one month from the date of generation.

The generateVoucher() method takes a map of headers as an input, which includes the customer's session ID. If the session ID is valid it retrieves the customer's data from the database and checks if the session ID matches the one provided in the headers, if it matches the method then checks if the customer has enough reward points to generate a voucher. The customer needs at least 100 reward points to generate a voucher, and the voucher amount depends on their reward points. The voucher is valid for 1 month from the date it is generated, and a unique voucher code created.

The customers reward points and the details of the generated voucher are returned as a response.

### *cafeapp-customer-generatetransactionhistory*

This microservice is responsible for retrieving the transaction history for a customer. It first checks if the session-id is present in the headers. If the session-id is valid, it returns the transaction history for the customer along with their membership ID and username. Otherwise it returns an error message saying no session was found for the user and they should login to the app to obtain a valid session-id.

### *cafeapp-customer-getvoucher*

This microservice is responsible for retrieving the voucher details for a given user session. It takes in a map of headers that contains the session ID of the user, then it retrieves the customer details from the database based on the ID. If the Customer is found it retrieves the voucher details, checking whether the vouchers are valid by comparing their expiry date with the current date and time. Any expired vouchers are removed from the list and the updated voucher list is saved back to the repository. This information is all sent as a response to the customer.

### *cafeapp-customer-redeem-voucher*

This microservice allows customers to redeem vouchers against their purchases for a given user session. The microservice first checks the validity of the voucher that customer has provided. If the voucher is valid, it checks if the billing amount is adequate to apply the voucher. Once all the things are verified, the voucher is applied and then it updates the transaction using update transaction microservice. The resultant balance of the customer is returned as a response.

This method takes in an object RedeemVoucherPojo and a map of HTTP headers as input parameters. The object contains the transaction details like transaction amount and voucher code, and the HTTP headers contain the session ID of the logged-in user.

### *cafeapp-customer-updatetransaction*

This microservice is responsible for updating transactions of customers.If the session is valid the method retrieves the customer details from the repository, if a customer is found, their reward points and transaction history are updated based on the new transaction details. This service also updates customer reward points. Customer gets 1 reward point for every euro spent. Furthermore, this service also checks if the customer has made more than 10 transactions in the last month including the current transaction. If the number of transactions are more than 10, then it adds the customer details in the cache using update cache endpoint expose in fetch customer microservice.

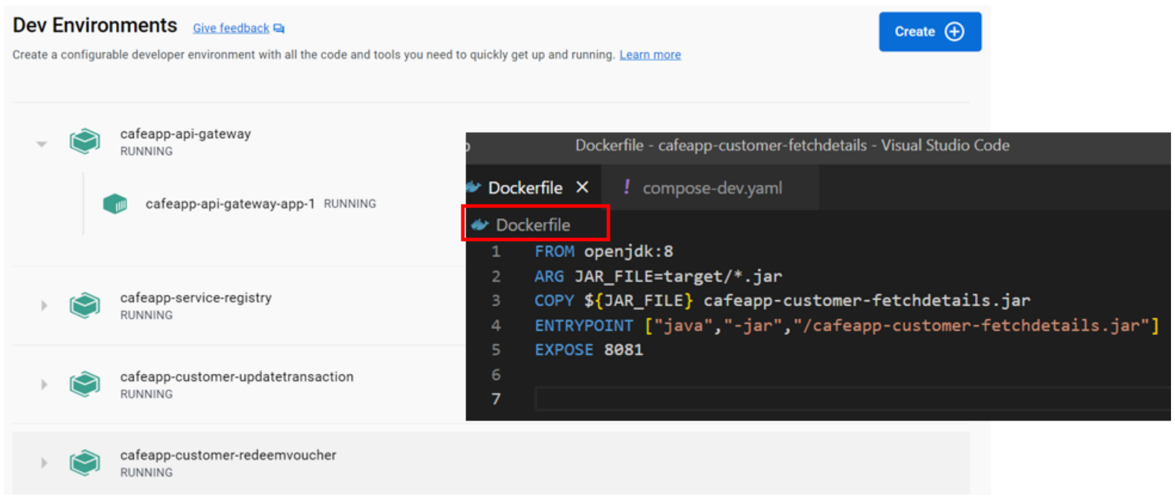
### *cafeapp-service-registry*

This microservice allows other microservices in the cafe app to register themselves and discover other services registered with the Eureka server. All the microservices self-register themselves to service registry during boot up.

## Docker and Kubernetes(K8) Containerisation

In the Cafe Loyalty Program application, Docker is used to containerise the Spring Boot Microservices. This allows for easier deployment and management of the app. Each microservice has its own Dockerfile, which specifies the dependencies and configurations required for that particular microservice. Using docker compose, the microservices can be started and stopped together as a single application, making it easier to test and deploy changes. Docker also provides a portable and consistent runtime environment, allowing the app to run reliably across different machines and platforms.

The Dockerfile defines a series of instructions to create a Docker image, these include commands to copy the application code into the image, install dependencies, configure the environment and run the application.


Dev Environments Give feedback

Create a configurable developer environment with all the code and tools you need to quickly get up and running. Learn more

Create

| | |
|---|---|
| cafeapp-api-gateway | RUNNING |
| cafeapp-api-gateway-app-1 | RUNNING |
| cafeapp-service-registry | RUNNING |
| cafeapp-customer-updatetransaction | RUNNING |
| cafeapp-customer-redeemvoucher | RUNNING |

Dockerfile - cafeapp-customer-fetchdetails - Visual Studio Code

Dockerfile | compose-dev.yaml

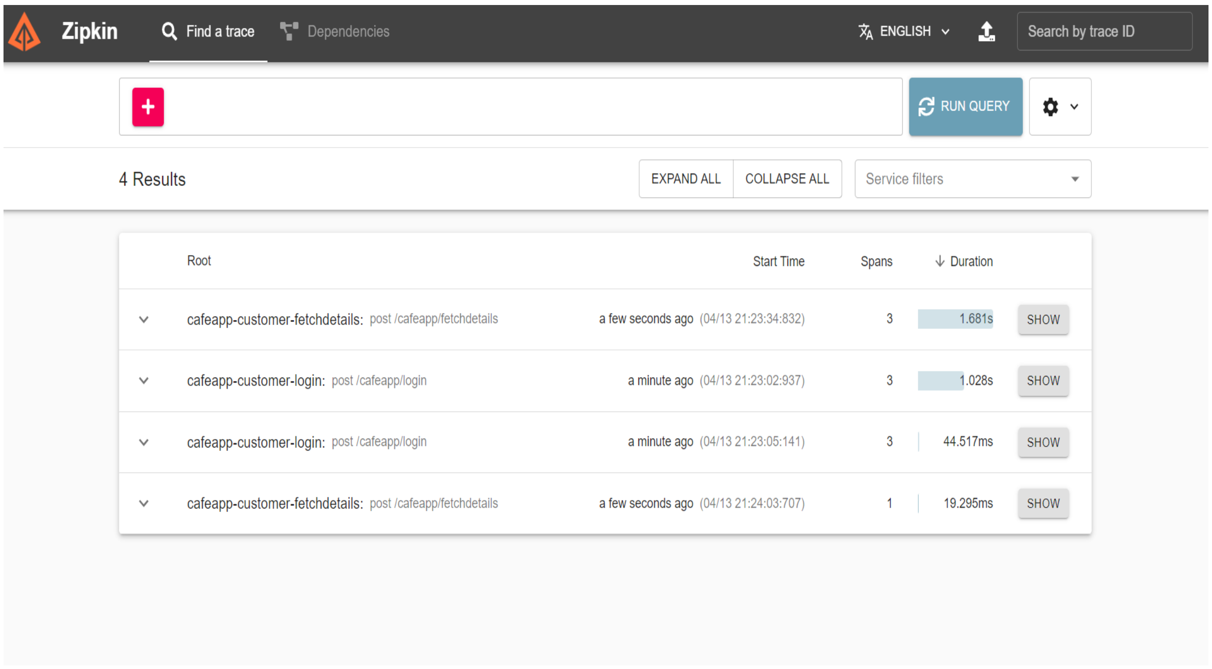
```
FROM openjdk:8
ARG JAR_FILE=target/*.jar
COPY ${JAR_FILE} cafeapp-customer-fetchdetails.jar
ENTRYPOINT ["java","-jar","/cafeapp-customer-fetchdetails.jar"]
EXPOSE 8081

```

Kubernetes is used to manage the containerised microservices that make up the application. It provides a platform for deploying, scaling and managing containerised applications across a cluster of machines making it easier to manage multiple instances of the application across multiple servers.

The aim of using Kubernetes is to allow the Cafe Loyalty Program application to easily scale up or down based on demand, ensure high availability, and roll out new features with ease, making the application more reliable.

## Zipkin and Sleuth

We have used Zipkin and Sleuth to enable distributed logging for all microservices. This can enable us to implement checkpoints in the logs by providing unique span-ids and trace-ids in logs for all the transactions. These can be utilised for implementing checkpoint mechanisms in case of failures.

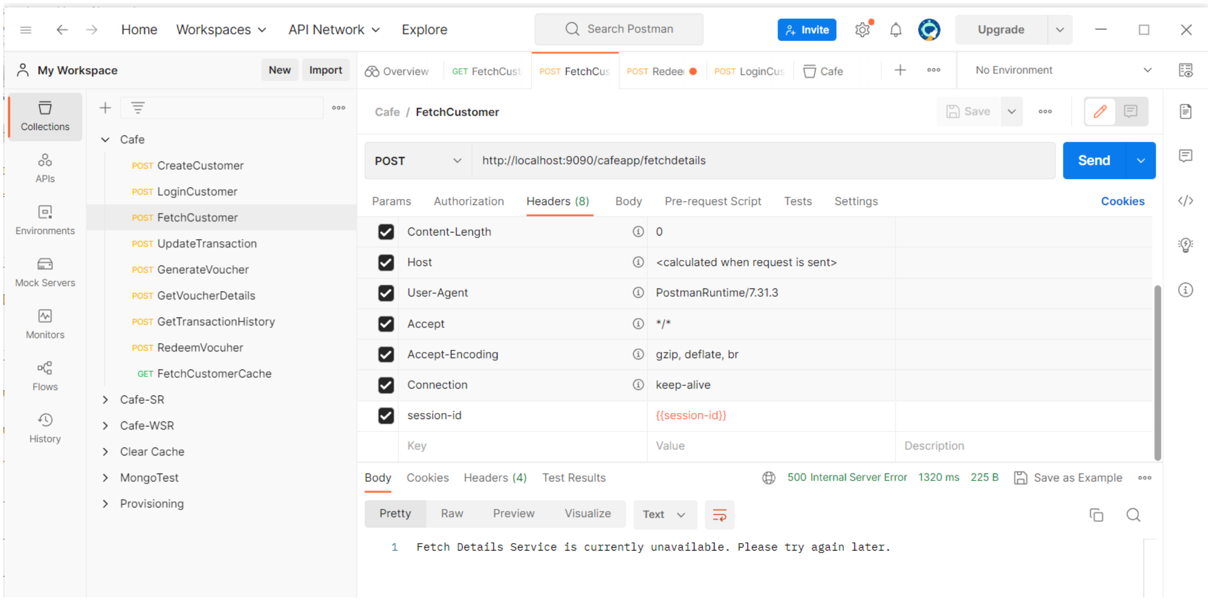


4 Results

| Root | Start Time | Spans | Duration | |
|---|---|---|---|---|
| cafeapp-customer-fetchdetails: post /cafeapp/fetchdetails | a few seconds ago (04/13 21:23:34:832) | 3 | 1.681s | SHOW |
| cafeapp-customer-login: post /cafeapp/login | a minute ago (04/13 21:23:02:937) | 3 | 1.028s | SHOW |
| cafeapp-customer-login: post /cafeapp/login | a minute ago (04/13 21:23:05:141) | 3 | 44.517ms | SHOW |
| cafeapp-customer-fetchdetails: post /cafeapp/fetchdetails | a few seconds ago (04/13 21:24:03:707) | 1 | 19.295ms | SHOW |

## Netfilx Hystrix

We have used Netfilx hystrix to implement failover mechanisms for all the microservices. This will enable us to ensure better reliability for our application. Currently we have given an error message but it can be modified to implement routing of traffic to a secondary server in case the master server fails.



POST http://localhost:9090/cafeapp/fetchdetails

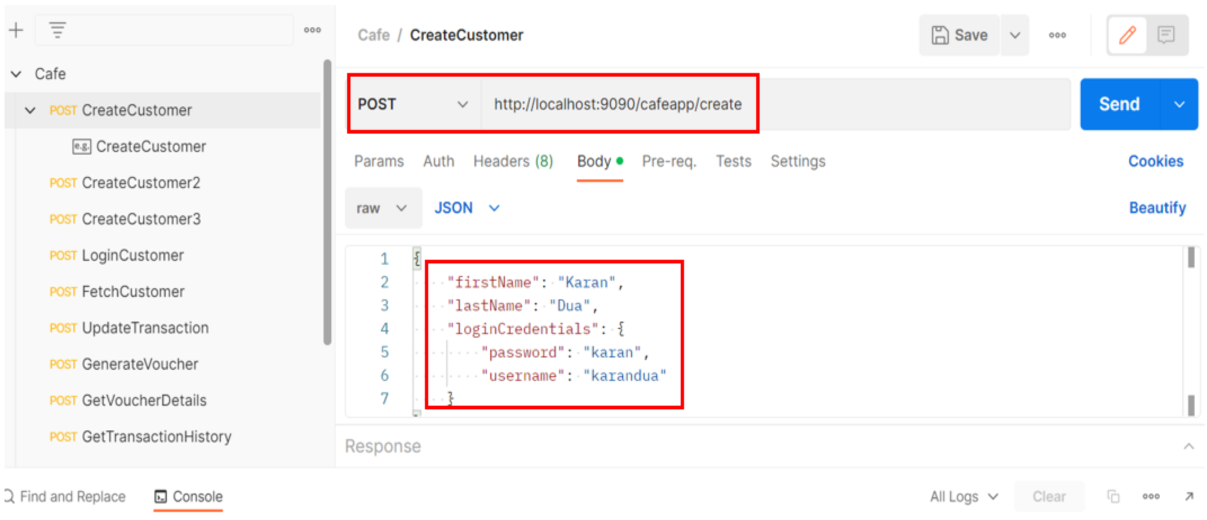
| Key | Value | Description |
|---|---|---|
| ✓ Content-Length | 0 | |
| ✓ Host | <calculated when request is sent> | |
| ✓ User-Agent | PostmanRuntime/7.31.3 | |
| ✓ Accept | */* | |
| ✓ Accept-Encoding | gzip, deflate, br | |
| ✓ Connection | keep-alive | |
| ✓ session-id | {{session-id}} | |

500 Internal Server Error 1320 ms 225 B

```
Fetch Details Service is currently unavailable. Please try again later.
```

## Postman

In the Cafe Loyalty Program application, Postman is a tool used for testing the REST API endpoints. It allows HTTP requests to be sent to the server and receives responses, making it an essential tool for testing the API's functionality and verifying it meets the requirements. Postman also provides features such as authentication, testing environments, and detailed request logs that can help in debugging and troubleshooting.



POST http://localhost:9090/cafeapp/create

```
{
    "firstName": "Karan",
    "lastName": "Dua",
    "loginCredentials": {
        "password": "karan",
        "username": "karandua"
    }
```

## *Implementation*

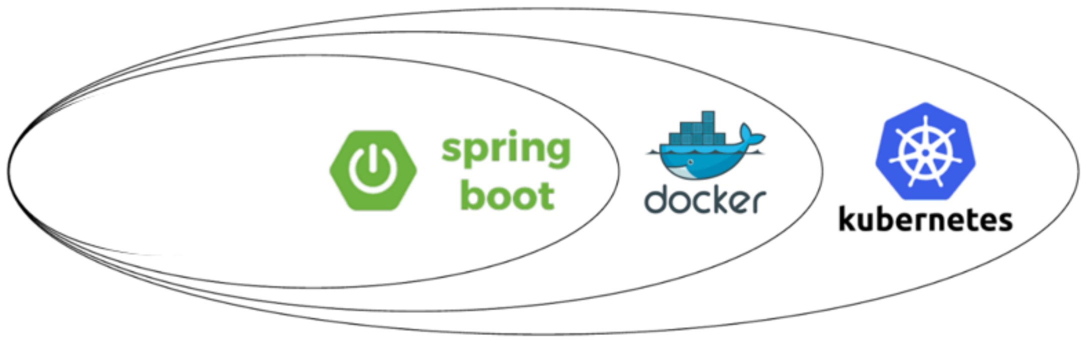
The Cafe Loyalty Program application was implemented in several key phases. Firstly all Spring Boot Microservices were created, with each microservice being responsible for a specific function within the application. A service registry was then created using Eureka, which enabled all microservices to register themselves, making it easier for them to communicate with each other.

To provide uniform access and fallback mechanisms for all microservices, an API gateway was implemented using Netflix Hystrix. This allowed incoming requests to be routed through a single entry point, with Hystrix providing a fallback mechanism in case any of the microservices became unavailable. To improve performance, Spring cache was implemented for the cafeapp-customer-fetchdetails service, which can help reduce the number of calls made to the database.

Docker support was then enabled for easier deployment and management of the application in a containerised environment with all images being pushed to Docker. Deployment files were then created for kubernetes which specified how the various microservices should be deployed and connected to each other within the kubernetes cluster.

To facilitate distributed logging, Zipkin and Sleuth were implemented, which allows logs to be collected from multiple microservices and brought into a single view.

The Spring Boot Microservices interact with the MongoDB database to retrieve and store data and Postman is used to interact with the individual microservices to test them.



**Kubernetes: Memory Issues on Startup**
Deploying Kubernetes on docker causes memory issues and the freezing of the Docker. Kubernetes can be resource-intensive, and if your machine doesn't have enough memory, CPU, or disk space it may cause issues.

**MongoDB: Failure Tolerance & Recovery**
Failures are tolerated through the use of replication of the database in what is known as a 'replica set'. Each replica set contains one primary node and multiple secondary nodes. The primary node is responsible for handling all of the write requests that update the database and asynchronously relaying this information to the remaining nodes. It stores all updates in its 'oplog' (operation log). The secondary nodes then replicate that oplog and apply the operations listed to their own data.

This way, when a node fails, the redundancy of data keeps the service available. A recovering node can rejoin the cluster seamlessly as a secondary node.

## *Testing*

This section outlines a test plan which contains the comprehensive testing strategy for the Cafe Loyalty Program System. Both functional and non-functional requirements of the system are tested.

### Functional Testing

The functionality of all implemented microservices are tested

1. **API Gateway -** Send requests to this gateway and validate the correct routing of the request to the appropriate microservice.
2. **Create Customer -** Test the creation of a customer's account. It should not be possible to create duplicate accounts with the same username. It should not be possible to create an account without providing all details correctly.
3. **Fetch Details -** Test the viewing of customer details. The details should be accurate and consistent independent of the location from which it is accessed.
4. **Login -** Test that customer gets a valid session id on successful login.
5. **Update Transaction -** The loyalty points of a customer should be correctly updated according to the scheme provided and added to their transaction history.
6. **Get Transaction History -** The transaction history should accurately reflect the previous transactions.
7. **Generate Voucher -** Vouchers should be generated only when enough points have been accrued by the customer.
8. **Get Voucher -** The output list of vouchers should be reflective of those generated by the user.
9. **Redeem Voucher -** Vouchers should be regarded as stale and unusable if the expiry date has passed. If the voucher is valid, the balance should be deducted and transaction history should be updated.
10. **Session-id**: Test that session id is validated for all microservices except login and create. A validar request must have a valid session-id, else the request should not execute

### Non-Functional Testing

**Reliability**

1. Test that when any node fails (primary or secondary), the system continues to run smoothly from the users perspective. This is done by forcefully stopping a node while the service is running.
2. Test of failover mechanism implemented using Netfilx Hystrix.
3. Test that a node can recover after failure and seamlessly catch up with the system.

**Availability**

1. Test that any updates are indeed being replicated across redundant databases.

**Scalability**

1. Test that particular microservice pod scale-up and scale-down by executing commands on Kubernetes.

**Performance**

1. Test that customer details are stored and fetched from cache.

## *Allocation of work*

We met with our group a few times, but mostly our communication and work on a project was online, using different platforms like WhatsApp or Microsoft Teams, so if questions arose during the work, everyone could ask for help in a group chat. We found that the distribution of responsibilities among team members has a positive effect on overall work productivity.

## *Summary*

The project involved developing a Distributed Cafe Loyalty Program application. The aim was to create a loyalty program application that could potentially be scaled up to accommodate a large number of users. The app allows customers to register and create new accounts, generate, redeem and receive vouchers, view transaction history and update transactions.

The application was built using Spring boot Microservice architecture, which can offer scalability, resilience and simplicity. The microservices were containerised using Docker and Postman was used to test all the functional and non-functional requirements. MongoDB was chosen to store data related to customers and their transactions. The application is highly scalable and distributed to handle high volumes of traffic from the transactions.