Bank Application using Microservices

Sulav Dhakal

18th Aug, 2023

Table of Contents

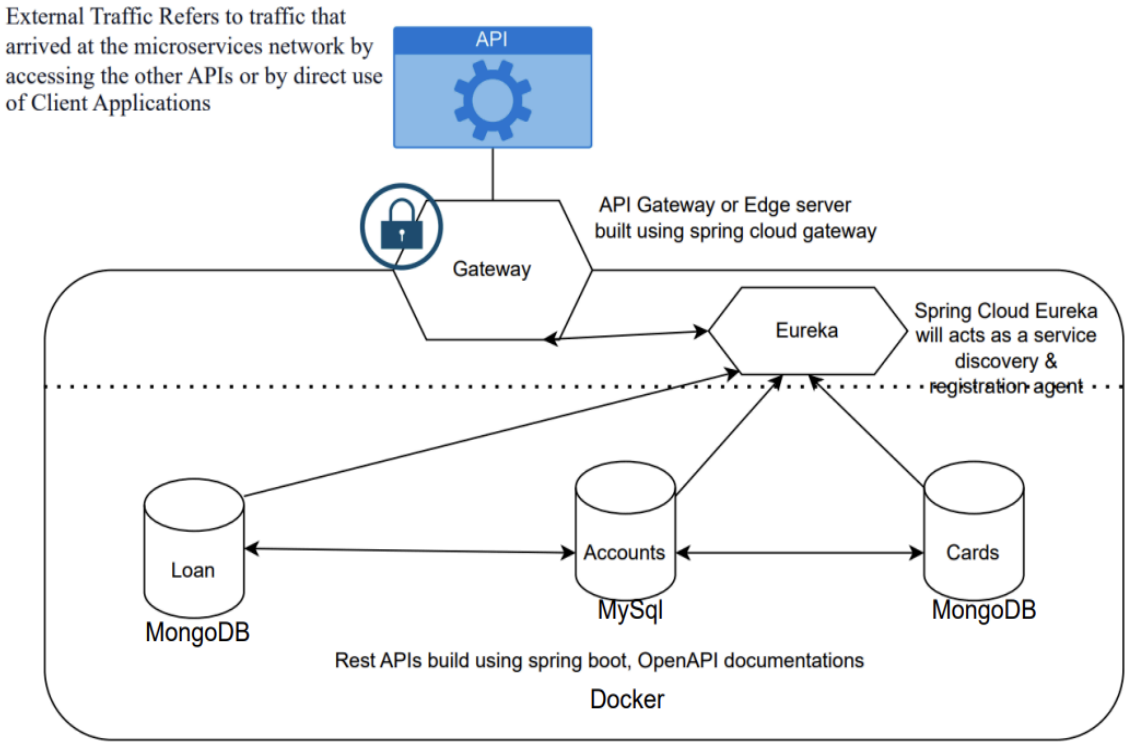
1. Introduction....4
2. Project Architecture....4
    2.1 Architecture of Microservices Bank Applications....4
    2.2 Services and Components:....5
3. Project Description....6
4. Implementation of services....7
    4.1 Interaction between the services....7
    4.2 Spring Cloud Framework....8
5. Database and its implementation....10
6. Deployment....13
    6.1 Docker....14
        6.1.1 Docker Compose....15
        6.1.2 Deploying applications in Docker using docker-compose....17
    6.2 Kubernetes setup:....18
        6.2.1 Minikube....18
7. Testing....22
    7.1 Current Approach:....22
    7.2 Changed Approach:....25
8. Resource Utilization....26
    8.2 Grafana....28
    8.3 Microservice Scaling and Performance Monitoring....29
Securing Enterprise Architecture for the Banking Application....30
    Architecture Design with security implementation....31
        Overview of Components....31
    Architecture Flow Outline:....33
Output....38
ScreenShot....48
Conclusion....56

1. Introduction

This project is based on microservices, a banking application that implements the concept of microservices. It includes three primary services: Basis: Accounting, Banking & Credit services. As stated earlier, every service involves managing the business logic of its respective domain; data persistence is accomplished by MySQL and MongoDB. They are developed using Spring Boot microservices and each service is registered with Eureka Service Registry. The Docker and Docker Compose are involved in processes of containerization and container orchestration.

2. Project Architecture

2.1 Architecture of Microservices Bank Applications



2.2 Services and Components:

The project architecture includes several key components:

- Service Discovery: Service discovery is a eureka - server, hosted in port 8761. It acts as a registry for the microservices.to register himself at runtime within the different ports.
- API Gateway: It receives all the client requests and should forward them to the correct microservice according to the request, Also, Security is implemented within the API

gateway Thus, the individual doesn't know that this is an internal function. It helps to rout the request to a related service. Here the API gateway is running under port 8080

- Microservices

1. Account Service: Currently responsible for simple account routines in the context of the utilization of the core banking application. Account service handles the client account data like account number, id, amounts. It is running under the ports 8089.
2. Card Service: Within the agency, they oversee affairs to do with credit cards. Cars service handles the cards related query it stores the data like card number, account number, type of card. It run under the port 8083.
3. Loan Service: Although its role mainly deals with loans, it supervises the operation of the organization in the same aspect. Loan service runs under port 8082 and handles the query like loan amount , loan id , account number.

- Database: Both MySQL and MongoDB databases are used for the different services. Here MySQL is used for account service, whereas MongoDB is used for loan and card services.
- Docker: The project is deployed using containerization. For easy deployment and making the services Docker containers, use within this project.

3. Project Description

In my project, the banking application, I have provided the micro service architecture to transform data. The application is designed to handle various banking operations through three distinct services:Three base operation services which the application offers are meant for executing various banking activities.

- Account Service: This service oversees all the account operations of the firm, detail for which is described in this report. While implementing the functionalities as an

independent service, the system should therefor handle an effectively large number of requests that relate to accounts and this should not have any impact on other services.

- Card Service: This service is for processing the operations for credit card. For instance, banking industry involves itself in the offering of credit cards, processing of credit card transactions as well as the processing of payments. Thus, although these processes are most interconnected, such a division of basic tasks will help the application to offer significantly more particular solutions and improvements regarding the credit card usage.
- Loan Service: From this service it is assumed that they are in overall charge of all the processes regarding the loan in question. Such as loan processing, a loan offered and loan repayments. In this way, loans and their management do not become integrated into other functions of the given application, which allows the module to carry out other more complex operations and calculations related to loans.

While using these services, the coordinative interaction of these services support the user in their banking problems, such as accounts service can call loan service facilities among others. The Microservices architecture ensures enough modularity, scalability, maintainability and positive alteration of the application which would not affect the previous microservices heavily.

4. Implementation of services

4.1 Interaction between the services

Specifically, regarding the microservice architecture in this project, Account, Loan, and Card services are designed to interact with other services through REST APIs. An overcoming moment occurs with the service discovery mechanism and Spring Cloud for the

convenient interaction of many services and the test is made fp. This paper will also highlight how all those services are connected.

```java
public class LoanServiceImpl implements LoanService {
    public LoanDTO getLoanById(String id) {  // Changed from Long to String
        return appendAccountDetail(loan);
    }

    @Override  1 usage  new *
    public List<Loan> getLoansByAccountId(String accountId) {
        return loanRepository.findByAccountId(accountId);
    }

    private LoanDTO appendAccountDetail(Loan loan) {  4 usages  Sulav Dhakal *
        String accountUrl = "http://account-service/account/" + loan.getAccountId();
        Account account = restTemplate.getForObject(accountUrl, Account.class);

        LoanDTO loanDTO = new LoanDTO();
        loanDTO.setId(loan.getId());
        loanDTO.setAmount(loan.getAmount());
        loanDTO.setInterestRate(loan.getInterestRate());
        loanDTO.setAccount(account);

        return loanDTO;
    }
}
```

Params Authorization Headers (9) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{
    "id": 5,
    "accountHolderName": "dhakal sulav ",
    "accountNumber": "12345887737",
    "balance": 100000.00
}
```

Body Cookies Headers (5) Test Results Status: 200 OK Time: 17 ms Size: 438 B Save as examp

Pretty Raw Preview Visualize JSON

```
[
    {
        "id": 1,
        "accountHolderName": "John Doe",
        "accountNumber": "123456789",
        "balance": 1000.00
    },
    {
        "id": 3,
        "accountHolderName": "sulav dhakal",
        "accountNumber": "22222222222",
        "balance": 1000.00
    },
    {
```

Loan service Calling Account Service

4.2 Spring Cloud Framework

- **Eureka: Service Discovery**

All the microservices require to have the Eureka server to which each service has to register when it starts. This being the case, the Eureka environment has a registry where there is a list of all the services, and also individual instances of the same. In cases where a microservice needs to interact with another service, the server address of the intended service is then gotten from Eureka. Service dependency for the Netflix-eureka-server has been incorporated in the Eureka in the resource server to cater to the requirements of the server.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Similarly, the configuration for the service discovery is done in 'application properties' and made it as registry.

```
spring.application.name=service-discovery
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false
eureka.server.enable-self-preservation=false
```

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.18.13:account-service:8081
BANK-API-GATEWAY	n/a (1)	(1)	UP (1) - 192.168.18.13:bank-api-gateway:8080
CARD	n/a (1)	(1)	UP (1) - 192.168.18.13:card:8083
LOAN	n/a (1)	(1)	UP (1) - 192.168.18.13:loan:8082

General Info

Name	Value
total-avail-memory	100mb
num-of-cpus	10
current-memory-usage	53mb (52%)

Services Registered in Eureka Server

- **API Gateway Integration with the services**

The API Gateway acts as an entry point of the application, and it forwards the incoming request to the right microservice depending on the URL path. It plays the role of an inverse proxy, which means that it relays clients' requests to the suitable backend server.

```
spring.application.name=bank-api-gateway

server.port=8080

eureka.client.service-url.default-zone=http://localhost:8761/eureka/

spring.cloud.gateway.routes[0].id=account-service
spring.cloud.gateway.routes[0].uri=lb://account-service
spring.cloud.gateway.routes[0].predicates[0]=Path=/account/**

spring.cloud.gateway.routes[1].id=card-service
spring.cloud.gateway.routes[1].uri=lb://card-service
spring.cloud.gateway.routes[1].predicates[0]=Path=/card/**

spring.cloud.gateway.routes[2].id=loan-service
spring.cloud.gateway.routes[2].uri=lb://loan-service
spring.cloud.gateway.routes[2].predicates[0]=Path=/loan/**
```

Integration with the services

5. Database and its implementation

For this project, there are two types of databases for three different services used here. Concerning the account services, MySQL is utilized, while MongoDB supports loan and card services. The databases for the services are configured in the 'application.properties' file.

- **Account Service:**

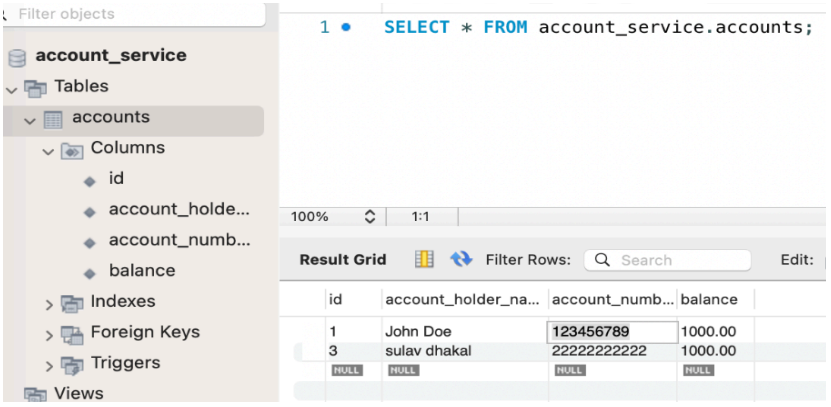
  MySQL is used to store information about the accounts such as the account number, name of the account holder, and balance for Account Service. For database operations, Spring Data JPA is used which is a framework that provides a generic repository for handling relational data in Java. Hence, the Account Service calls the MySQL database instance using the JDBC URL.

  application.property

```
spring.datasource.url=jdbc:mysql://localhost:3306/account_service
spring.datasource.username=root
#spring.datasource.password=${MY_SQL_DB_PASSWORD:}
spring.datasource.password=example
#spring.datasource.password=
```

Dependencies:

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```



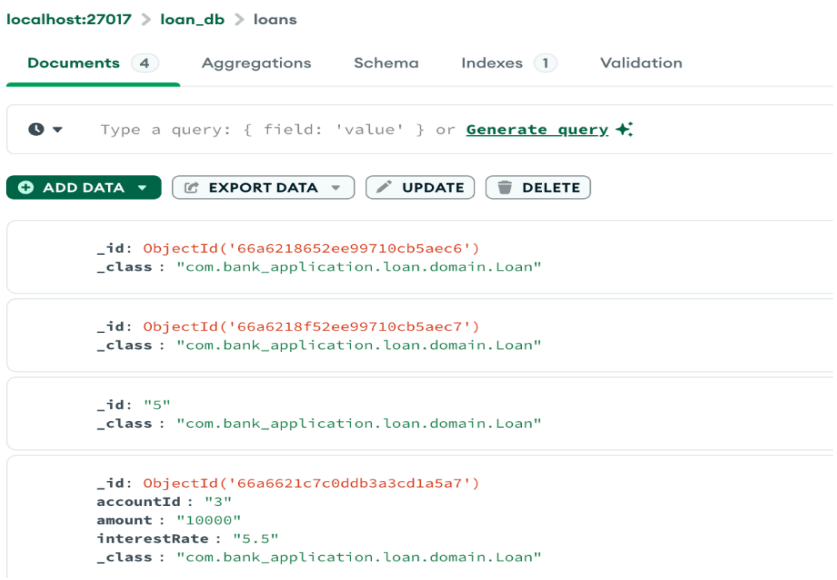
Datastore in MySQL DB

- **Loan & Card Services:**

  The Loan and Card Services both are configured in MongoDB to store information about loans and cards such as loan information which includes loan ID, account ID, amount, and interest rate, and for card information which includes card ID, account ID, and card type. For MongoDB, I have used Spring Data MongoDB for accessing the MongoDB database as it is an extension of Spring and provides easy data access layer coupled with Spring Boot for maintaining CRUD operations or querying. The services connect to MongoDB instances using MongoDB URI; they incorporate options like database name and connection string URI.

```
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
```

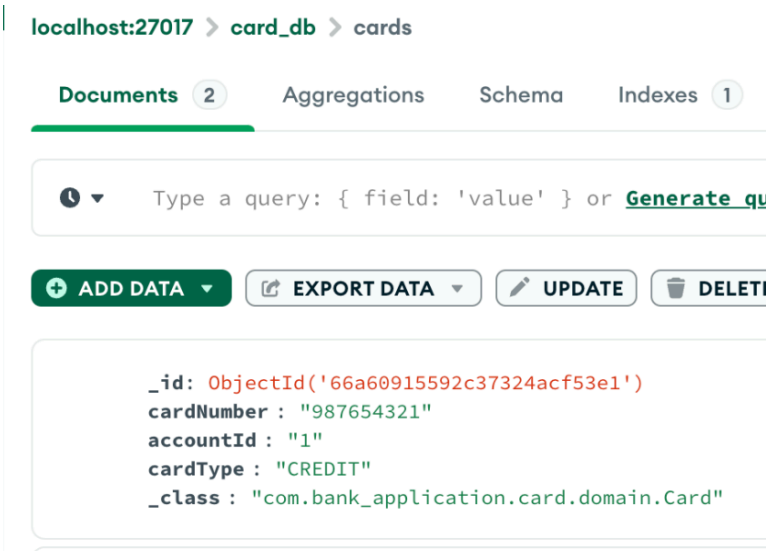
```
spring.application.name=loan
server.port=8082
eureka.client.service-url.defaultZone=http://${EUREKA_SERVER:localhost}:${EUREKA_SERVER_PORT:8761
spring.data.mongodb.uri=mongodb://localhost:27017/loan_db
#spring.data.mongodb.uri=mongodb://admin:admin@mongo:27017/loan_db
spring.data.mongodb.database=loan_db


management.endpoints.web.exposure.include=prometheus
```



Datastore in MongoDB for Loan Services

```
spring.application.name=card
server.port=8083

eureka.client.service-url.defaultZone=http://${EUREKA_SERVER:localhost}:${EUREKA_SERVER_P

spring.data.mongodb.uri=mongodb://localhost:27017/card_db

#spring.data.mongodb.uri=mongodb://admin:admin@mongo:27017/card_db
spring.data.mongodb.database=card_db


management.endpoints.web.exposure.include=prometheus
```

```java
package com.bank_application.card.domain;

import ...

@Data  18 usages  Sulav Dhakal *
@NoArgsConstructor
@AllArgsConstructor
public class CardDTO {
    private String id;
    private String cardNumber;
    private Account account;
    private CardType cardType;
    private Long accountId;
}
```



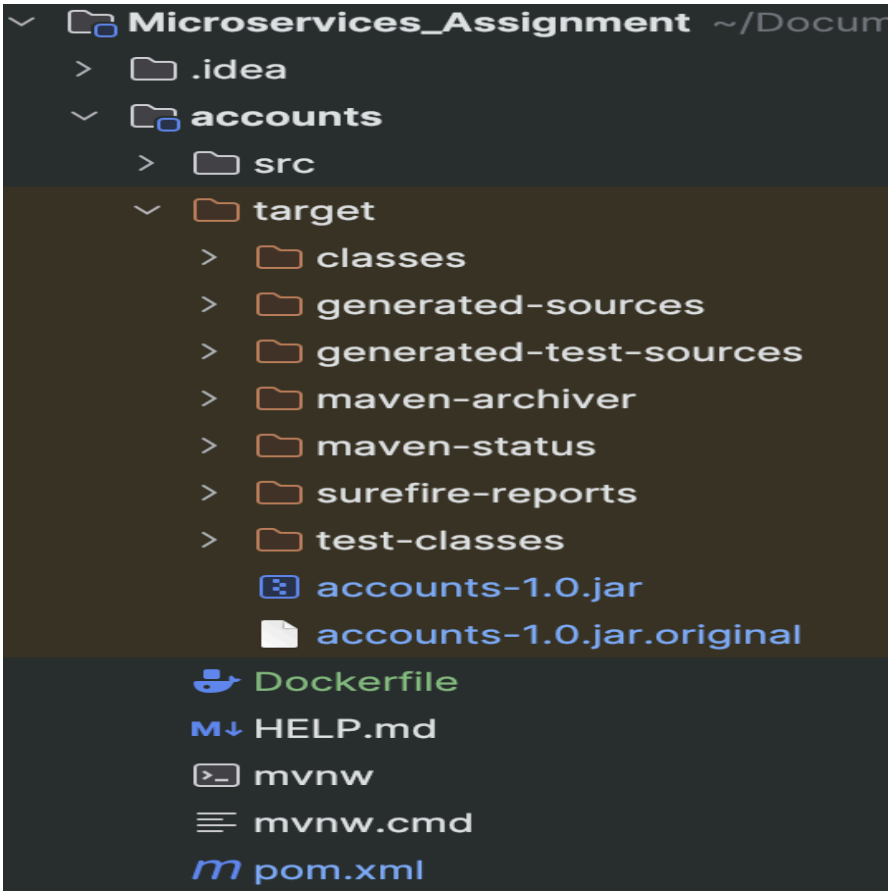
Datastore in MongoDB for Loan Services

6. Deployment

For the deployment, I have used a Docker container. Docker container provides consistency and makes easy deployment by allowing to management of multiple services in a multiple container, it isolates the dependencies and prevents conflicts.

6.1 Docker

Each microservice (Account Service, Loan Service, Card Service, Service Discovery, APIGateway) has its own Docker file. Docker files define how the Docker images are built. For instance, the Dockerfile, as shown in the below image for the Account Service, shows the configuration of the Docker image, a separate jar file is created for each service, which will include all the configuration and the service code to build an image in the Docker container. Jar files allow Java applications and their dependencies in one single executable file.



Jar File & DockerFIle for Docker

```
LoanServiceImpl.java        accounts/Dockerfile ×        loan
FROM openjdk:17-jdk-slim

MAINTAINER SULAV.DHAKAL

COPY target/accounts-1.0.jar accounts-1.0.jar

ENTRYPOINT ["java","-jar","accounts-1.0.jar"]
```

DockerFile

6.1.1 Docker Compose

Docker Compose is an official tool for writing and running Docker step by step applications. Using docker-compose. In the terminal with starting from docker-compose.yml file, we can describe what services, networks, and volumes of application needs and control them using just commands.

The docker-compose. yml file created in my project is used here. The yml file enables to set a configuration for the different services comprising account, loan, card microservices, API gateway, service discovery server besides the database.

Components in docker-compose.yml file:

1. Services in Microservices

- Account Service: Runs on 8081 which is connected with MySQL DB.
- Loan Service: Runs on 8082 which is connected with MongoDB.
- Card Services: Runs on 8083 which is connected with MongoDB.
- Bank-API-Gateway: Runs on 8080, which exposes routes for other services.
- Service Discovery: It's a eureka server, that acts as a registry and runs on default port 8761.

2. Database:

MySQL Database: It access 3306 port for account service.

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>8.0.33</version>
</dependency>
```

MongoDb Database: it access 27017 & 27018 respectively for loan and card services.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
```

In Docker Compose, the port mapping is defined as 8080:8081 to provide the identified port mapping of the Docker container with the host machine.

pplication.java service-discovery/.../application.properties service-discovery/Dock

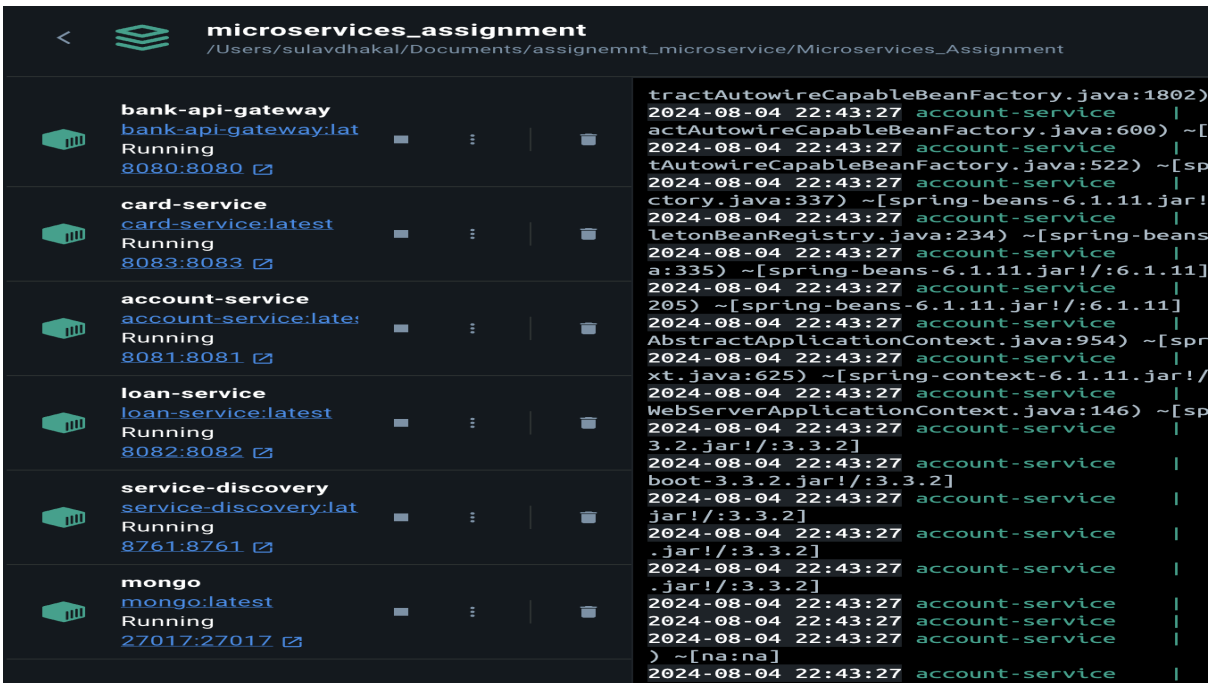
```
services:
  account-service:
    build:
      context: ./accounts
      dockerfile: Dockerfile
    ports:
      - "8081:8081"
    environment:
      SPRING_DATASOURCE_URL: jdbc:mysql://account-db:3306/accounts
      SPRING_DATASOURCE_USERNAME: root
      SPRING_DATASOURCE_PASSWORD: root
    depends_on:
      - account-db
    networks:
      - microservices-network

  loan-service:
    build:
      context: ./loan
      dockerfile: Dockerfile
    ports:
      - "8082:8082"
    environment:
      SPRING_DATA_MONGODB_URI: mongodb://loan-db:27017/loan
    depends_on:
      - loan-db
```

Docker-Compose.yml file

6.1.2 Deploying applications in Docker using docker-compose

Docker Compose is used to define and run multi-container Docker applications where I am able to specify and coordinate a number of Docker containers using a ‘docker-compose. This file coordinates the creation of multiple services such as `account-service`, `loan-service`, `card-service`, `bank-api-gateway`, and the service-discovery, which are services from different Dockerfiles. It also modifies MySQL for the account service and MongoDB for the loan and card services. The respective port mappings, dependencies on other services for inter-service communication in the `yml` file. When the command `docker-compose up` is run, Docker Compose handles the process of building new images to run alongside the containers and configuring ports of the application on the host and the containers being fired up simultaneously in the correct order.

```
(base) sulavdhakal@sulavs-MacBook-Pro accounts % docker images
REPOSITORY                TAG                    IMAGE ID       CREATED              SIZE
sulav999/accounts         microservice_account   d2da5d7990ce   About a minute ago   481MB
docker/welcome-to-docker  latest                 648f93a1ba7d   8 months ago         19MB
(base) sulavdhakal@sulavs-MacBook-Pro accounts %
```



Docker image for account service

6.2 Kubernetes setup:

Kubernetes: Kubernetes is an open-source container orchestration system in that is used for managing application containers with the use of generalized abstractions of Portable and Extensible Declarative approach and integration of Automations' as a basic 'Platform'. It has it has a large and growing network which is an indication that many people use it. To extend this list, we can enumerate quite many available Kubernetes services, supports and tools.

6.2.1 Minikube

Minikube is a development environment development tool that enables the creation of Kubernetes. If the footprints are small, the files can be clustered on a local PC or laptop. In principle, it can be regarded as a Kubernetes distribution, but since was designed to perform a task that few other distributions (like Rancher, OpenShift, EKS) do. It is for this reason that

people frequently refer to it as a tool. As it has been explained earlier, Minikube supports all of the major dints originally specified by Kalyptus and platforms such as Windows, Linux, and MacOS. But alas, it doesn't run on mobile devices.

Setting up Minikube:

1. Download minikube from the relative site https://minikube.sigs
2. Use docker and a container or a virtual machine as a manager
3. Start Kubernete, docker must be running in local use command 'minikube start' on a terminal

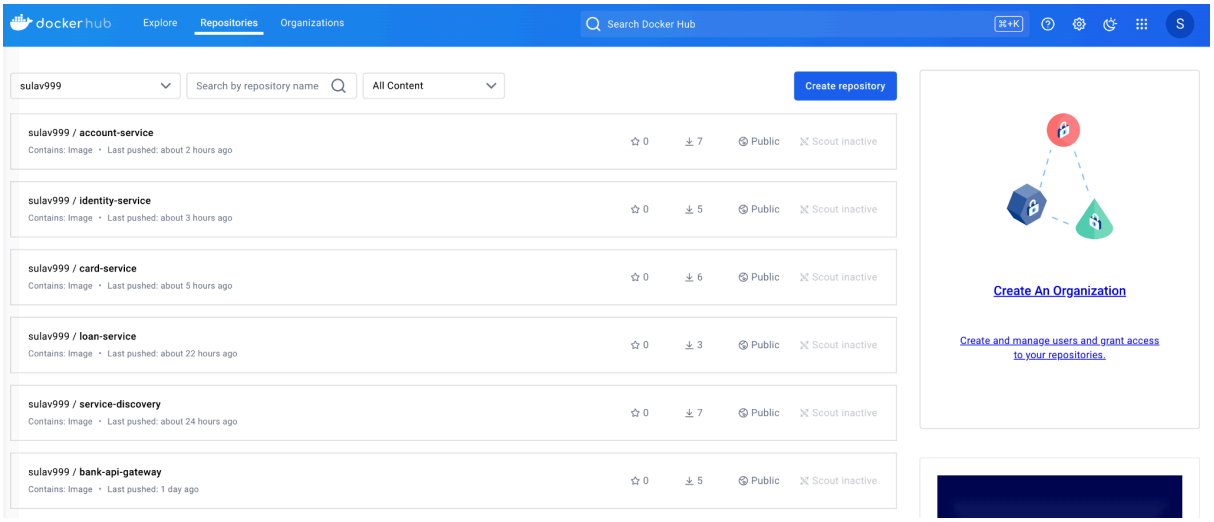
```
C:\Users\gaman>minikube start
* minikube v1.33.1 on Microsoft Windows 11 Home 10.0.22631.3880 Build 22631.3880
* Automatically selected the docker driver
* Using Docker Desktop driver with root privileges
* Starting "minikube" primary control-plane node in "minikube" cluster
* Pulling base image v0.0.44 ...
* Downloading Kubernetes v1.30.0 preload ...
    > preloaded-images-k8s-v18-v1...:  342.90 MiB / 342.90 MiB  100.00% 2.01 Mi
    > gcr.io/k8s-minikube/kicbase...:  481.58 MiB / 481.58 MiB  100.00% 2.42 Mi
* Creating docker container (CPUs=2, Memory=4000MB) ...
* Preparing Kubernetes v1.30.0 on Docker 26.1.1 ...
  - Generating certificates and keys ...
  - Booting up control plane ...
  - Configuring RBAC rules ...
* Configuring bridge CNI (Container Networking Interface) ...
* Verifying Kubernetes components...
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5
* Enabled addons: storage-provisioner, default-storageclass
* Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default

C:\Users\gaman>minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

The images of each microservices have been created using the Docker file (for example: The Docker specifications and build command refers to the Docker file which contains the specifications of the Docker image that is to be created and it is run by the command 'docker build -t docker image name -f docker file name . ' for example 'docker build -t eureka-server -f Dockerfile-eureka-server. '. After this, to give it a meaningful name and version identifier the created image has been tagged using the command 'docker tag <name>:'<name>' as '<docker username>/<name>:<version>'

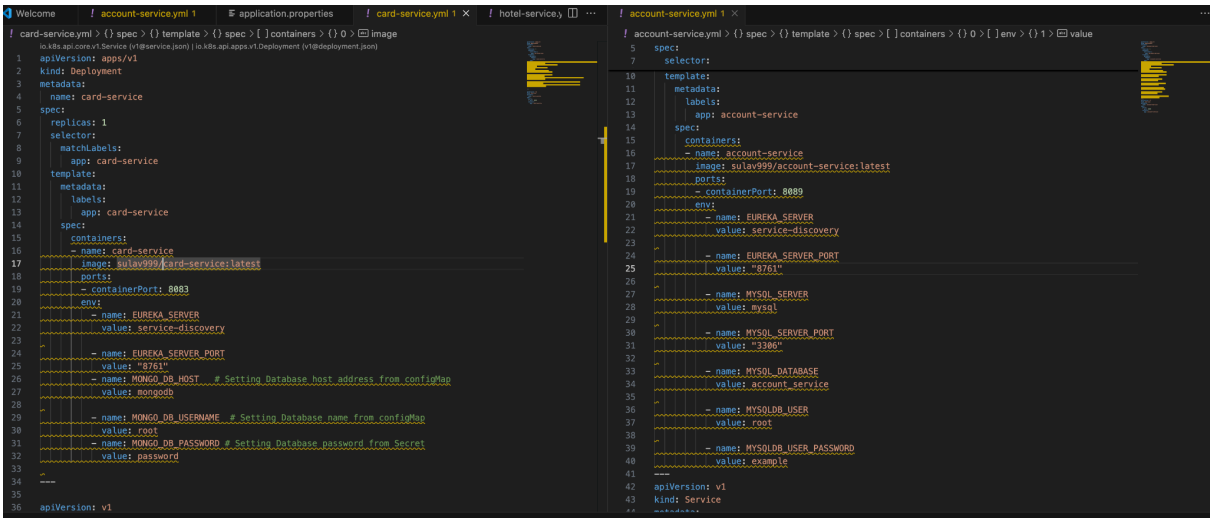
Push the image in ducker hub, for usinging need to provide unique identifier for here sulav999 is the unique identifier tagged to each service.

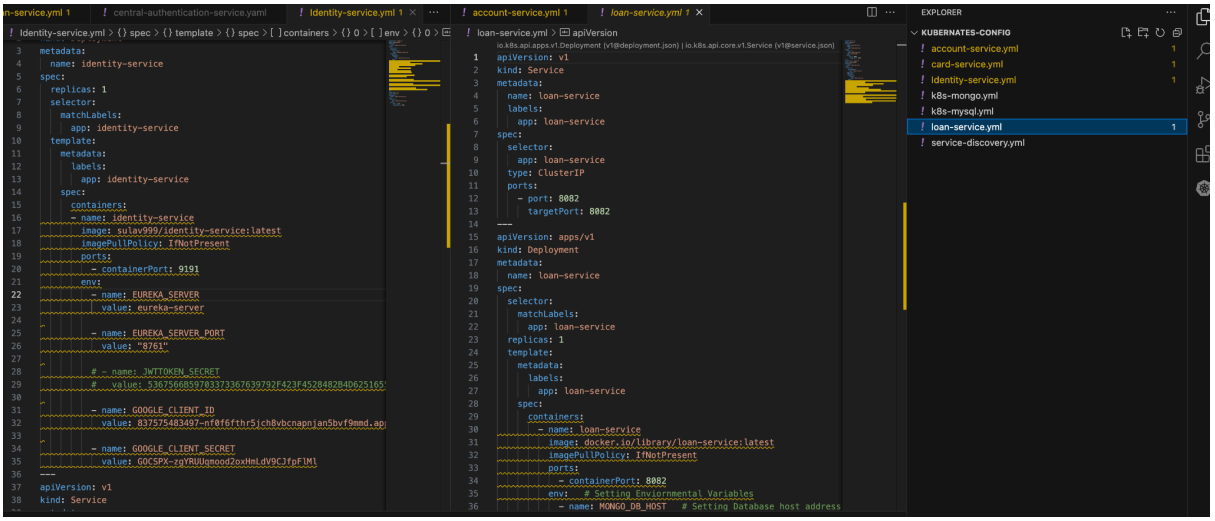
```
sulav999/account-service     latest    17ca788d7bbf   12 minutes ago   486MB
(base) sulavdhakal@sulavs-MacBook-Pro accounts % docker build -t sulav999/account-service:lates

[+] Building 1.0s (7/7) FINISHED                          docker:desktop-linux
 => [internal] load build definition from Dockerfile                        0.0s
 => => transferring dockerfile: 230B                                        0.0s
 => [internal] load metadata for docker.io/library/openjdk:17-jdk-slim      0.9s
 => [internal] load .dockerignore                                           0.0s
 => => transferring context: 2B                                             0.0s
 => [internal] load build context                                           0.0s
 => => transferring context: 73B                                            0.0s
 => [1/2] FROM docker.io/library/openjdk:17-jdk-slim@sha256:aaa3b3cb27e3e   0.0s
 => CACHED [2/2] COPY target/accounts-1.0.jar accounts-1.0.jar              0.0s
 => exporting to image                                                      0.0s
 => => exporting layers                                                     0.0s
 => => writing image sha256:17ca788d7bbf586425d202d3898727928f03c0650784f   0.0s
 => => naming to docker.io/sulav999/account-service:latest                  0.0s

What's next:
    View a summary of image vulnerabilities and recommendations → docker scout quickview
(base) sulavdhakal@sulavs-MacBook-Pro accounts % docker push sulav999/account-service:latest

The push refers to repository [docker.io/sulav999/account-service]
8709231bf99b: Layer already exists
c82e5bf37b8a: Pushed
2f263e87cb11: Layer already exists
f941f90e71a8: Pushed
```

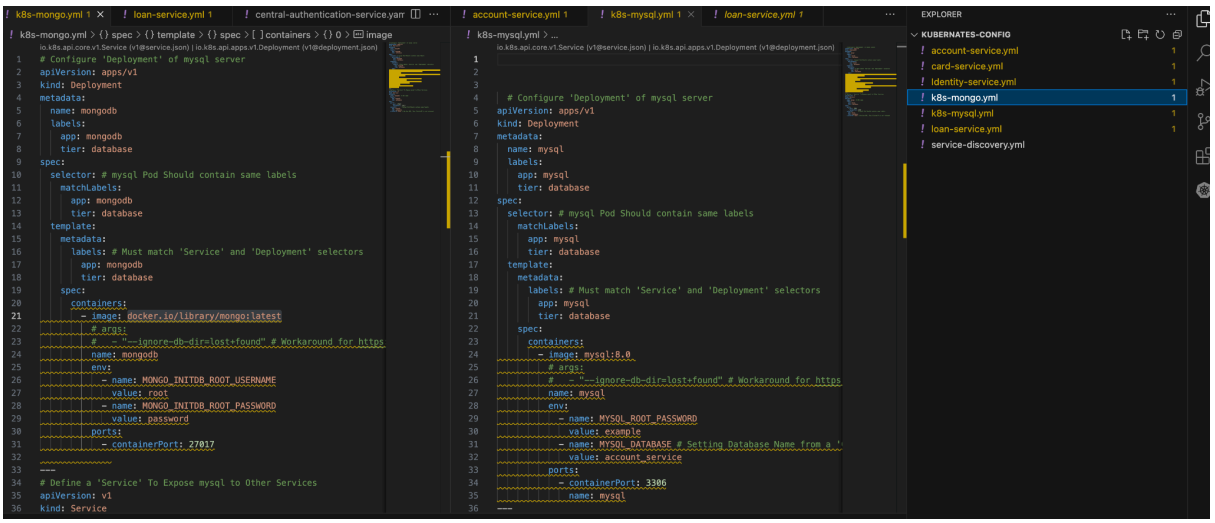
```
PROBLEMS 3   OUTPUT   TERMINAL   DEBUG CONSOLE   PORTS                                   zsh

service/service-discovery configured
(base) sulavdhakal@sulavs-MacBook-Pro KUBERNATES-CONFIG % kubectl get all
NAME                                    READY   STATUS             RESTARTS   AGE
pod/service-discovery-695bbb68b9-crxk4  0/1     ImagePullBackOff   0          30s

NAME                     TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/kubernetes       ClusterIP   10.96.0.1      <none>        443/TCP    4h8m
service/service-discovery ClusterIP  10.109.81.86   <none>        8761/TCP   71m

NAME                            READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/service-discovery  0/1   1            0           30s

NAME                                       DESIRED   CURRENT   READY   AGE
replicaset.apps/service-discovery-695bbb68b9  1       1         0       30s
(base) sulavdhakal@sulavs-MacBook-Pro KUBERNATES-CONFIG %
```



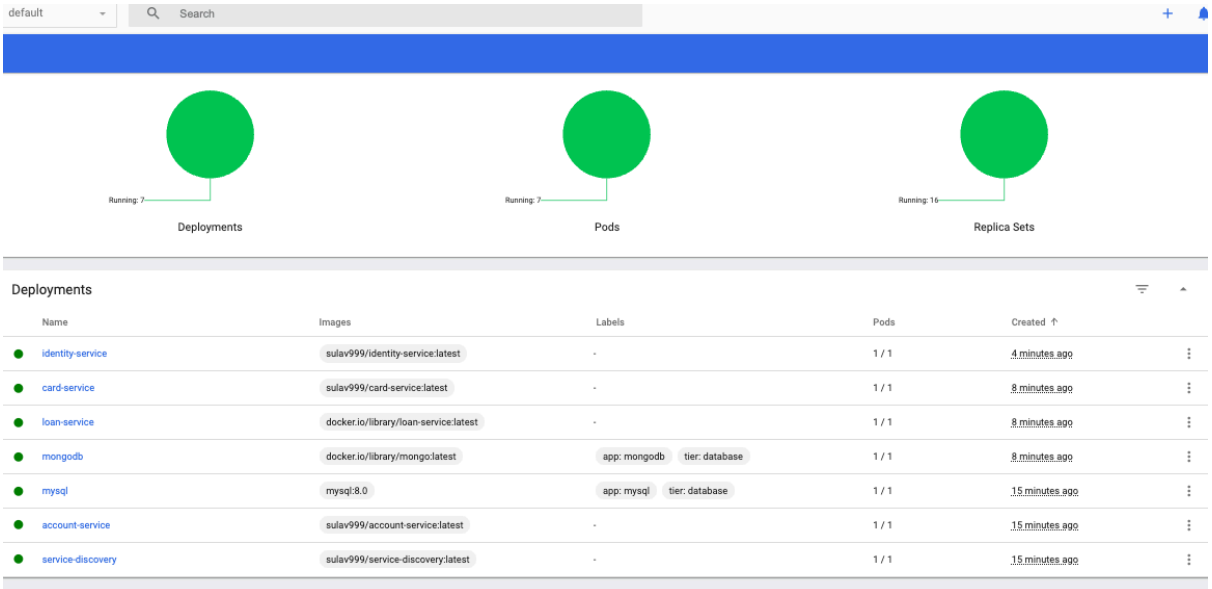
Thus, the pushed image has been pulled employing the configuration file and further, it has been set up on Kubernetes. The mongodb, mysql, and microservices use configuration files for the deployment of these and these can be obtained from the project at the 'Kubernetes' folder. The configuration files are as follows:







At last, all the microservices are deployed on the Kubernetes, and they are active, which is possible just with by command ‘kubectl apply –f Which <config file name>. Yaml’ (for instance ‘kubectl apply fiml<|reserved_special_token_272|> eureka-server. yaml’).



7. Testing

7.1 Current Approach:

In accordance to current scenario wa re calling account microservice from card microservice in the below way:

```java
@Override  1 usage  new *
public CardDTO getCardByAccountId(String accountId) {
    List<Card> cards = cardRepository.findByAccountId(accountId);

    return null;
}
```

```java
private CardDTO appendAccountDetail(Card card) {  4 usages  Sulav Dhakal *
    String url = "http://account-service/account/" + card.getAccountId();
    Account account = restTemplate.getForObject(url, Account.class);

    if (account == null) {
        System.out.println("Account not found for ID: " + card.getAccountId());

    }
```

Thus, in the current scenario, I was not able to address exception handling while calling the account service in the Card Service. Nevertheless, I noted that good thinking revealed that probably adding exception handling may enhance efficiency. Handling of exceptions therefore avails an enhanced opportunity of control over error situations so that any complications are effectively handled and do not hinder the flow of the application. If exceptions are not well handled, they bring in interventions that would otherwise alter performance and use of resources. With an error percentage of 19.11%.

Summary Report

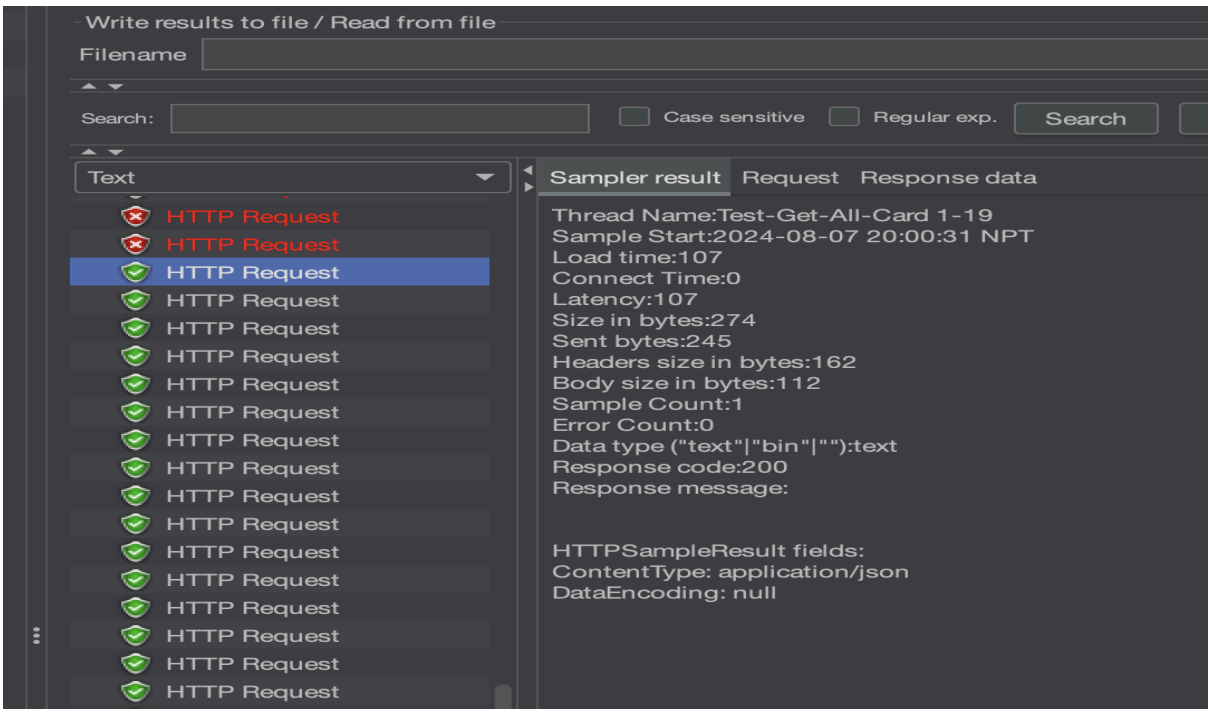
Name: Summary Report

Comments:

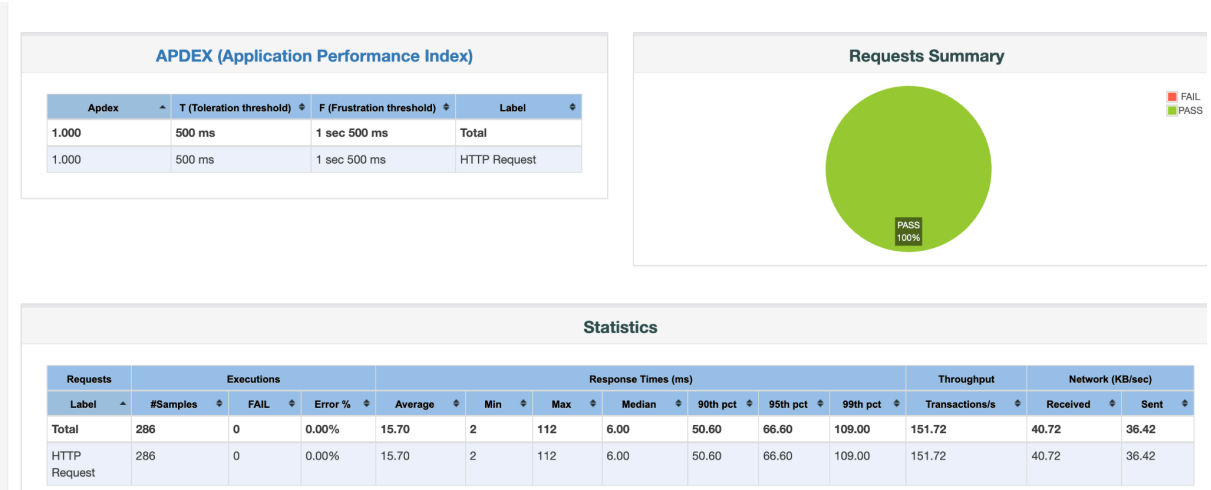
Write results to file / Read from file

Filename Browse... Log/Display Only: Errors Successes Configure

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	858	6	1	134	12.41	19.11%	3.6/sec	0.84	0.86	240.0
TOTAL	858	6	1	134	12.41	19.11%	3.6/sec	0.84	0.86	240.0



In particular, evaluation of this approach shows that it takes quite a lot of time and means to perform the necessary calculations significantly. It is noticeable that the API is being called repeatedly if the number of employees being evaluated is large which reduces efficiency due to multiple API requests. Also, although there are multiple CPU cores, the for-loop is executed on a single thread only. This facts show the general inefficient approach and the following performance statistics reveal the need for optimisation.



APDEX (Application Performance Index)

Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	500 ms	1 sec 500 ms	Total
1.000	500 ms	1 sec 500 ms	HTTP Request

Requests Summary

Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	286	0	0.00%	15.70	2	112	6.00	50.60	66.60	109.00	151.72	40.72	36.42
HTTP Request	286	0	0.00%	15.70	2	112	6.00	50.60	66.60	109.00	151.72	40.72	36.42

For another scenario: The above statistics show that the time for receiving the request is 15. 70 ms, although we are passing only a limited amount of data which has a value of 286.

Despite the fact that the available data is scanty, its value which has been quantified by the JMeter graph is not negligible.

7.2 Changed Approach:

In order to optimize the speed of the microservice, I used elements of exception handling. In so doing I was able to enhance efficiency as in place of interrupting other processes due to an error, it could gracefully move to the next task waiting in queue. This addition helped to get better in functions' workflow and cut down on the time requirements for error CaPs, which translated into tangible improvements of performance indicators.

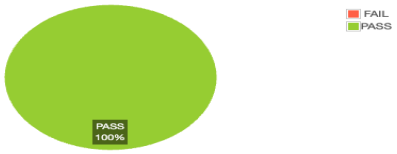
Sampler result | Request | Response data

```
Thread Name:Test-Get-All-Card 1-240
Sample Start:2024-08-08 01:11:34 NPT
Load time:4
Connect Time:0
Latency:4
Size in bytes:276
Sent bytes:247
Headers size in bytes:162
Body size in bytes:114
Sample Count:1
Error Count:0
Data type ("text"|"bin"|""):text
Response code:200
Response message:

HTTPSampleResult fields:
ContentType: application/json
DataEncoding: null
```

APDEX (Application Performance Index)

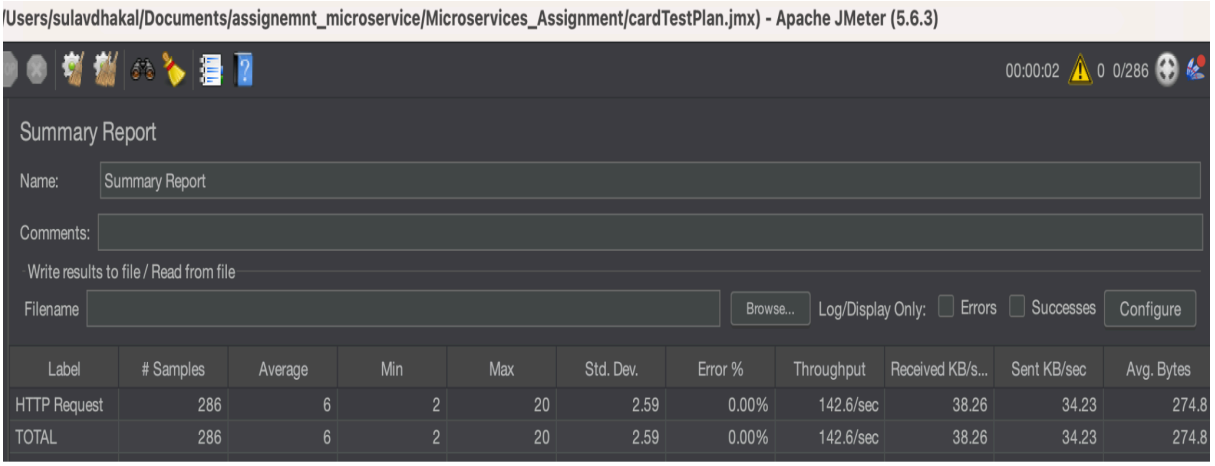
Apdex	T (Toleration threshold)	F (Frustration threshold)	Label
1.000	500 ms	1 sec 500 ms	Total
1.000	500 ms	1 sec 500 ms	HTTP Request

Requests Summary



Statistics

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	286	0	0.00%	10.95	2	69	7.00	11.30	47.65	69.00	152.94	41.05	36.72
HTTP Request	286	0	0.00%	10.95	2	69	7.00	11.30	47.65	69.00	152.94	41.05	36.72



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/s...	Sent KB/sec	Avg. Bytes
HTTP Request	286	6	2	20	2.59	0.00%	142.6/sec	38.26	34.23	274.8
TOTAL	286	6	2	20	2.59	0.00%	142.6/sec	38.26	34.23	274.8

In the revised code we can clearly see that the response latency is improved. And error percentage is zero. Here we have utilize multithreading concept which can help enhance the efficiency and address the errors in numerous ways, although it does not seize a specific mechanism of errors a lot as it passes them to an additional session.

8. Resource Utilization

For observing the usage of resources in the system, Prometheus and moreover Grafana were employed. Prometheus was configured by modifying the YML file in order to match the required needs of the service running in the cluster.

```
- job_name: 'card-service'
  metrics_path: '/actuator/prometheus'
  static_configs:
    - targets: ['localhost:8083']  #
```

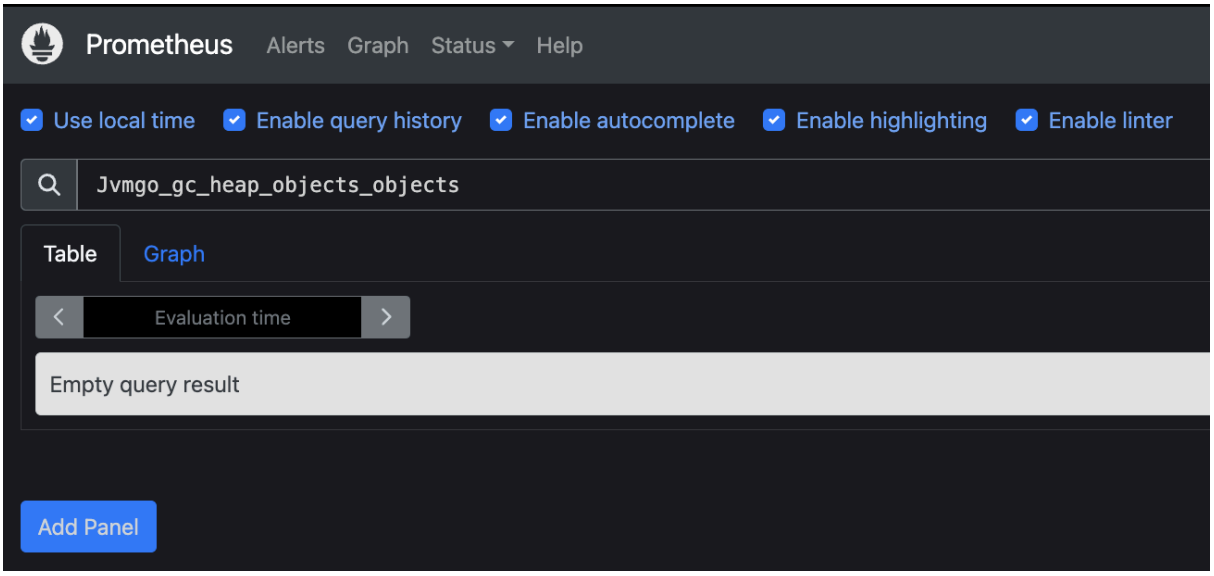
Prometheus has its own dependency which needs to be added to all service pom files after maven downloads its dependencies.

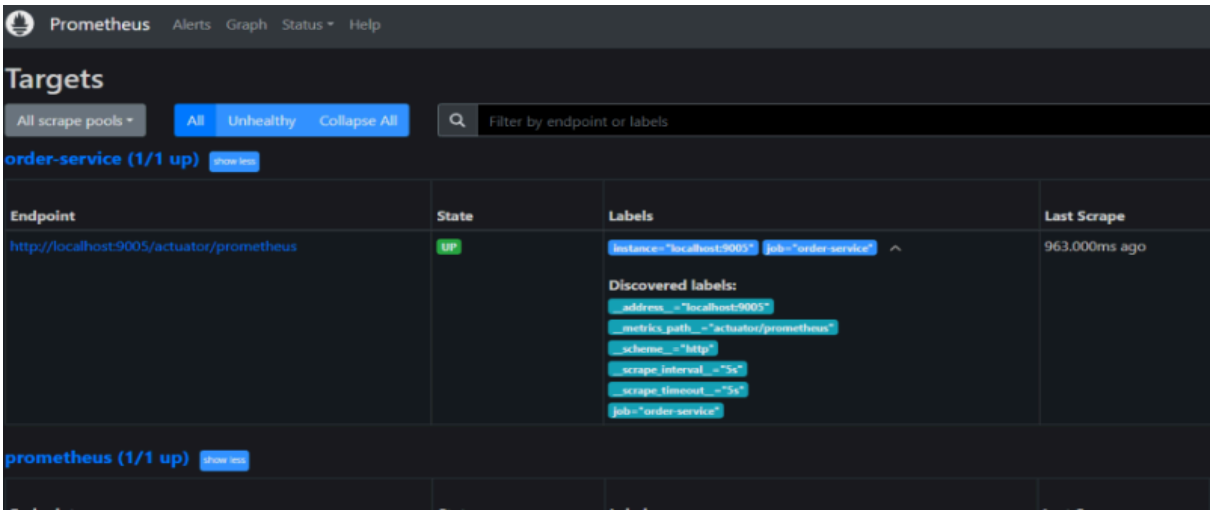
```xml
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
    <scope>runtime</scope>
</dependency>
```

8.1 Actuator Configuration for Prometheus in the dashboard

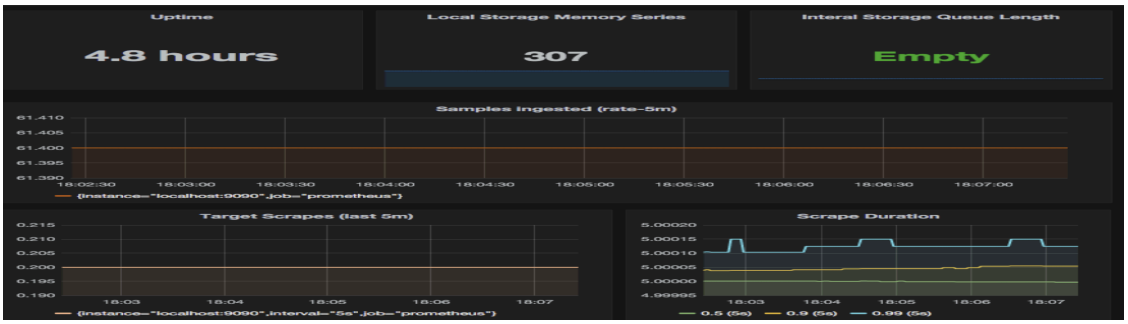
```
jvm_classes_loaded_classes 12873.0
# HELP jvm_classes_unloaded_classes_total The total number of classes unloaded since the Java virtual machine has started execution
# TYPE jvm_classes_unloaded_classes_total counter
jvm_classes_unloaded_classes_total 0.0
# HELP jvm_compilation_time_ms_total The approximate accumulated elapsed time spent in compilation
# TYPE jvm_compilation_time_ms_total counter
jvm_compilation_time_ms_total{compiler="HotSpot 64-Bit Tiered Compilers"} 1207.0
# HELP jvm_gc_live_data_size_bytes Size of long-lived heap memory pool after reclamation
# TYPE jvm_gc_live_data_size_bytes gauge
jvm_gc_live_data_size_bytes 0.0
# HELP jvm_gc_max_data_size_bytes Max size of long-lived heap memory pool
# TYPE jvm_gc_max_data_size_bytes gauge
jvm_gc_max_data_size_bytes 4.294967296E9
# HELP jvm_gc_memory_allocated_bytes_total Incremented for an increase in the size of the (young) heap memory pool after one GC to before the next
# TYPE jvm_gc_memory_allocated_bytes_total counter
jvm_gc_memory_allocated_bytes_total 9.6468992E7
# HELP jvm_gc_memory_promoted_bytes_total Count of positive increases in the size of the old generation memory pool before GC to after GC
# TYPE jvm_gc_memory_promoted_bytes_total counter
jvm_gc_memory_promoted_bytes_total 7531520.0
# HELP jvm_gc_overhead An approximation of the percent of CPU time used by GC activities over the last lookback period or since monitoring began, whichever
# TYPE jvm_gc_overhead gauge
jvm_gc_overhead 9.812584509875396E-5
# HELP jvm_gc_pause_seconds Time spent in GC pause
# TYPE jvm_gc_pause_seconds summary
jvm_gc_pause_seconds_count{action="end of minor GC",cause="G1 Evacuation Pause",gc="G1 Young Generation"} 2
jvm_gc_pause_seconds_sum{action="end of minor GC",cause="G1 Evacuation Pause",gc="G1 Young Generation"} 0.019
# HELP jvm_gc_pause_seconds_max Time spent in GC pause
# TYPE jvm_gc_pause_seconds_max gauge
jvm_gc_pause_seconds_max{action="end of minor GC",cause="G1 Evacuation Pause",gc="G1 Young Generation"} 0.014
# HELP jvm_memory_committed_bytes The amount of memory in bytes that is committed for the Java virtual machine to use
# TYPE jvm_memory_committed_bytes gauge
jvm_memory_committed_bytes{area="heap",id="G1 Eden Space"} 4.6137344E7
jvm_memory_committed_bytes{area="heap",id="G1 Old Gen"} 3.9845888E7
jvm_memory_committed_bytes{area="heap",id="G1 Survivor Space"} 6291456.0
jvm_memory_committed_bytes{area="nonheap",id="CodeCache"} 1.31072E7
jvm_memory_committed_bytes{area="nonheap",id="Compressed Class Space"} 8192000.0
jvm_memory_committed_bytes{area="nonheap",id="Metaspace"} 5.47256E7
# HELP jvm_memory_max_bytes The maximum amount of memory in bytes that can be used for memory management
# TYPE jvm_memory_max_bytes gauge
jvm_memory_max_bytes{area="heap",id="G1 Eden Space"} -1.0
jvm_memory_max_bytes{area="heap",id="G1 Old Gen"} 4.294967296E9
jvm_memory_max_bytes{area="heap",id="G1 Survivor Space"} -1.0
jvm_memory_max_bytes{area="nonheap",id="CodeCache"} 5.0331648E7
jvm_memory_max_bytes{area="nonheap",id="Compressed Class Space"} 1.073741824E9
jvm_memory_max_bytes{area="nonheap",id="Metaspace"} -1.0
# HELP jvm_memory_usage_after_gc The percentage of long-lived heap pool used after the last GC event, in the range [0..1]
# TYPE jvm_memory_usage_after_gc gauge
jvm_memory_usage_after_gc{area="heap",pool="long-lived"} 0.006368711590766907
# HELP jvm_memory_used_bytes The amount of used memory
# TYPE jvm_memory_used_bytes gauge
jvm_memory_used_bytes{area="heap",id="G1 Eden Space"} 1.2582912E7
jvm_memory_used_bytes{area="heap",id="G1 Old Gen"} 2.7353408E7
jvm_memory_used_bytes{area="heap",id="G1 Survivor Space"} 5730168.0
jvm_memory_used_bytes{area="nonheap",id="CodeCache"} 1.2057984E7
jvm_memory_used_bytes{area="nonheap",id="Compressed Class Space"} 7806584.0
jvm_memory_used_bytes{area="nonheap",id="Metaspace"} 5.379652E7
# HELP jvm_threads_daemon_threads The current number of live daemon threads
# TYPE jvm_threads_daemon_threads gauge
jvm_threads_daemon_threads 29.0
```

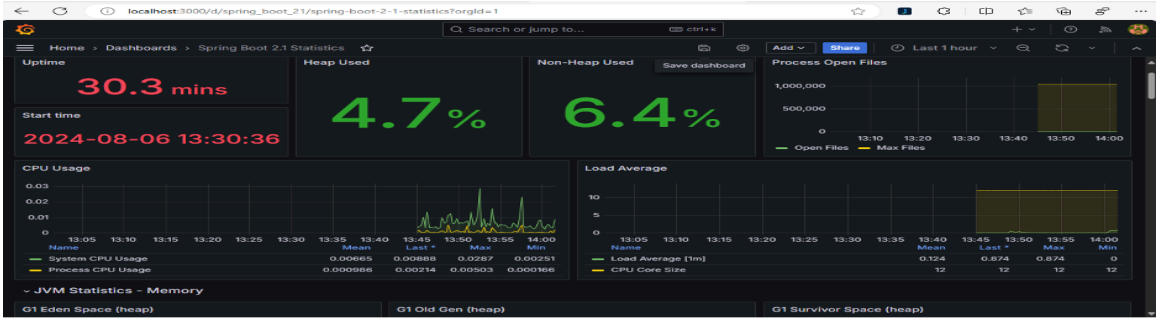
```
# TYPE disk_total_bytes gauge
disk_total_bytes{path="/Users/sulavdhakal/Documents/assignemnt_microservice/Microservices_Assignment/."} 4.94384795648E11
# HELP executor_active_threads The approximate number of threads that are actively executing tasks
# TYPE executor_active_threads gauge
executor_active_threads{name="applicationTaskExecutor"} 0.0
# HELP executor_completed_tasks_total The approximate total number of tasks that have completed execution
# TYPE executor_completed_tasks_total counter
executor_completed_tasks_total{name="applicationTaskExecutor"} 0.0
# HELP executor_pool_core_threads The core number of threads for the pool
# TYPE executor_pool_core_threads gauge
executor_pool_core_threads{name="applicationTaskExecutor"} 8.0
# HELP executor_pool_max_threads The maximum allowed number of threads in the pool
# TYPE executor_pool_max_threads gauge
executor_pool_max_threads{name="applicationTaskExecutor"} 2.147483647E9
# HELP executor_pool_size_threads The current number of threads in the pool
# TYPE executor_pool_size_threads gauge
executor_pool_size_threads{name="applicationTaskExecutor"} 0.0
# HELP executor_queue_remaining_tasks The number of additional elements that this queue can ideally accept without blocking
```


Prometheus
Alerts
Graph
Status
Help
Use local time
Enable query history
Enable autocomplete
Enable highlighting
Enable linter
Jvmgo_gc_heap_objects_objects
Table
Graph
Evaluation time
Empty query result
Add Panel


Prometheus
Alerts
Graph
Status
Help
Targets
All scrape pools
All
Unhealthy
Collapse All
Filter by endpoint or labels
order-service (1/1 up)
Endpoint
State
Labels
Last Scrape
http://localhost:9005/actuator/prometheus
UP
963.000ms ago
Discovered labels:
prometheus (1/1 up)

8.2 Grafana


Uptime
4.8 hours
Local Storage Memory Series
307
Interal Storage Queue Length
Empty
Samples ingested (rate-5m)
Target Scrapes (last 5m)
Scrape Duration



The Grafana shows several performance indicators of the application that includes a Spring Boot 2. 1 application. The key metrics are:

- Heap Used: 4. Heap total memory is now at 7% usage.
- Non-Heap Used: 6. Out of the total non-heap memory, 4% is occupied as of now.
- Process Open Files: This is depicted by a considerable Surge in the number of opened files to nearly one million at that point.
- CPU Usage: In regards to the system CPU usage it stays between 0 to 1 to provide explanation we have graph 1 below. 00665% and 0. 0287% with the logical CPU usage fluctuating between 0. 000896% and 0. 0053%.
- Load Average: The system run queue over the last one minute is 8. 74, this proves the level of load on the system is high.
- JVM Statistics - Memory: The JVM elaborated by the application has complete memory stats involving heap memory as follows: G1 Eden Space, G1 Old Gen, and G1 Survivor Space.

8.3 Microservice Scaling and Performance Monitoring

Some of the Spring created/followed microservices' architecture designs from the first week are the microservice architecture that was used to determine which of the microservices required to be scaled, in order to make the launching of multiple instances of these services,

and what can be done with it when limit of requests per second was increased. This process included architectural design factors, what we chose as scaling factors, and on our results of monitoring.

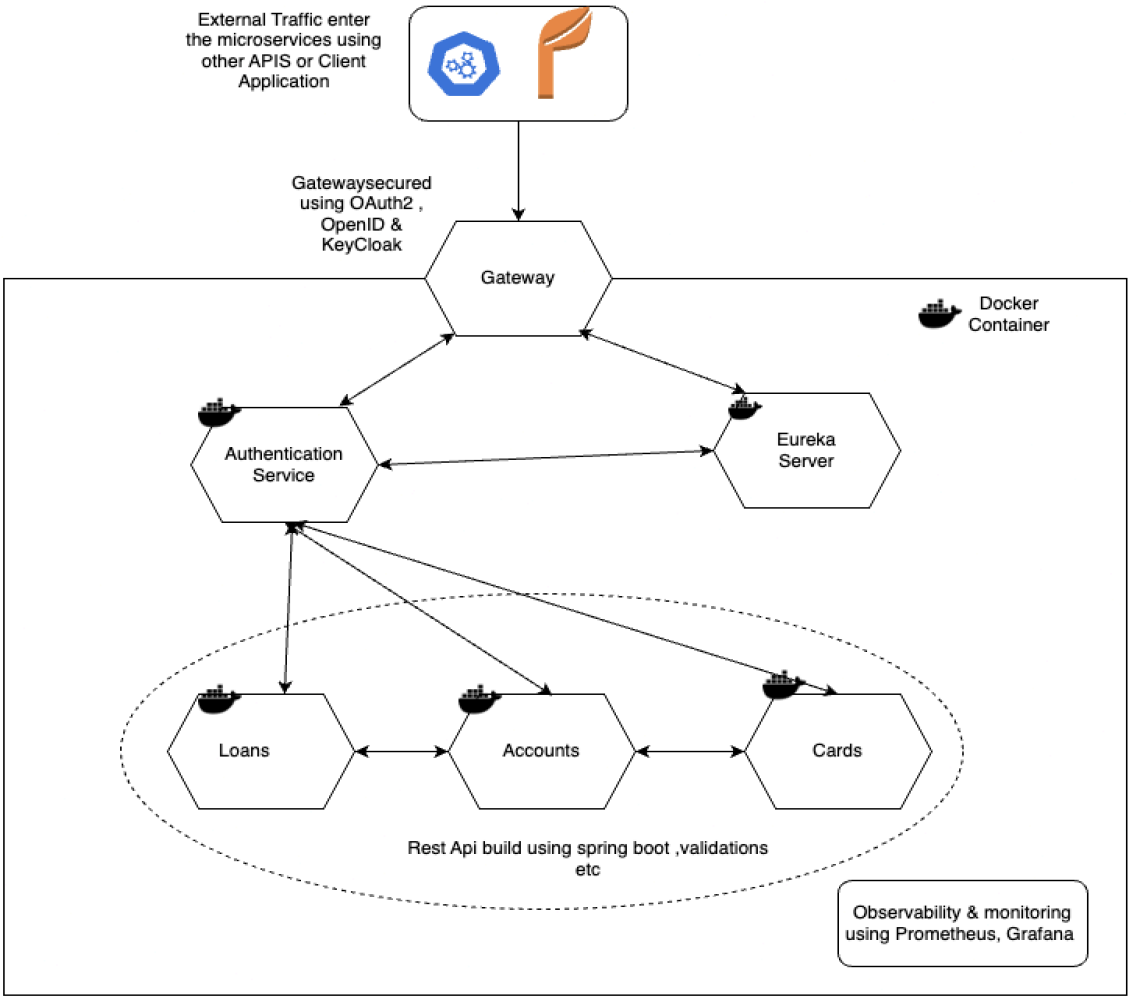
8.1 Architecture Design Consideration

Service Identification for Scaling: Not all microservices must be scaled in a similar way as one another either because they may not require similar capability. The issues, relations and forms of thinking are different and there must be a difference in a way they are scaled. Thus, we found the importance of the Account service much more to be scaled up as the main core base of data is placed in account db else in comparison to loan and card service which both interact with account service since they have been identified to be services with high load that would need to be scaled up in future. This services were selected in relation to their usage as well as the volume of data that any of the services is processing.

Securing Enterprise Architecture for the Banking Application

Security is the major concern for any application. Specially when it comes for banking applications various security measures must be implemented within the layered of a software architecture for the data control and transmission within the banking application microservices. This Banking microservice application utilize the OAuth 2.0 feature of authentication for controlling and securing the the interaction between the authentication server, and interconnected microservices.

Architecture Design with security implementation



Overview of Components

Authentication Service: This service has a very important function of processing OAuth 2. It shall ensure zero tokens by acting as the authenticator of them. It also checks them and is responsible for the generation of the access tokens. These access tokens are then used by the other services for the authentication as well as the authorization of the API operations for the secure interaction in the system.

API Gateway: API Gateway is designed to receive and manage incoming request from the clients and then forward it to the relevant microservices. Regarding the authentication of these sort of requests, it authenticates with the Security Service OAuth 2. 0 tokens. But, in the clients, such tokens are not consumed, or used, before the initiation of the sending of the requests and onward referral.

Discovery Server: This server will be used for the registration and discovery of services using Eureka. That is useful when your client microservices need to find other microservices at runtime, which is actually relatively frequent in the environment where microservices are added and removed and their IP addresses change. Through this discovery mechanism, the server guarantees that microservices can easily locate the required other services and can commission without hiccups frequently and effectively in an ever-shifting network environment.

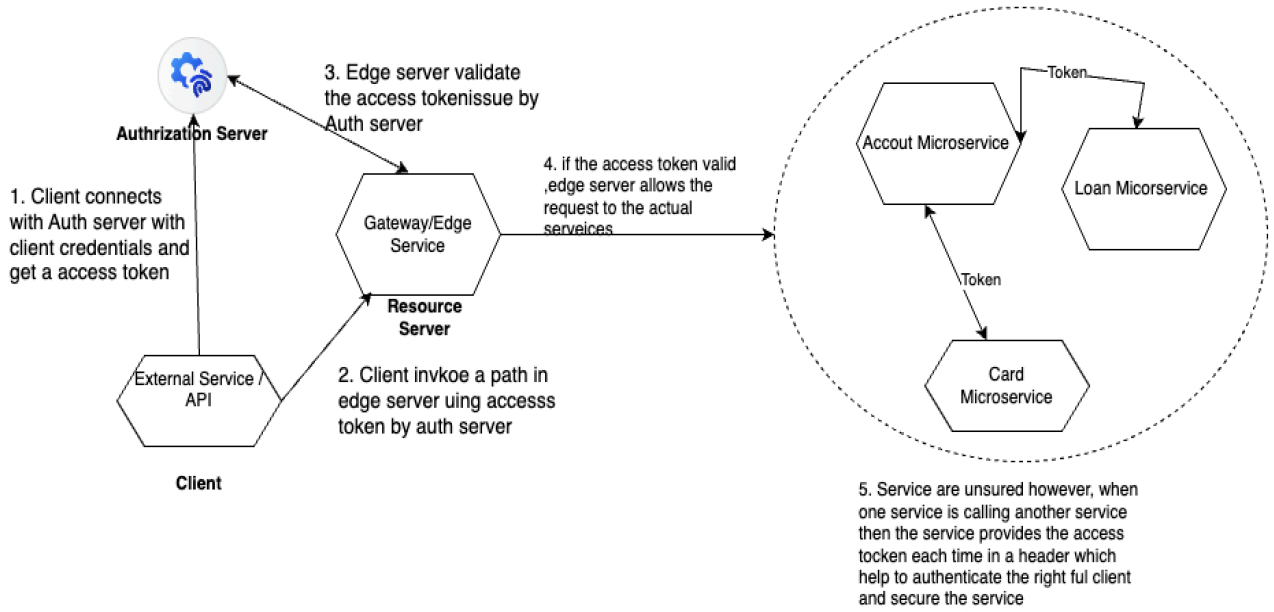
Microservices

- Account Service: Currently responsible for simple routines of the accounts in the context of the utilization of the core banking application.
- Card Service: Within the agency they oversee affairs to do with credit card.
- Loan Service: Although its role mainly deals with loans, it supervises the operation of the organization in the same aspect.

Database: Both MySQL and MongoDB databases are used for the different services. Here MySQL is used for account service, whereas MongoDB is used for loan and card services.

Docker: The project is deployed using containerization. For easy deployment and making the services Docker containers, use with in this project.

Architecture Flow Outline:



When applied to the above context of the code given herein, OAuth 2.0, Where a new Authentication Server is created, the practice known as OAuth 2.0 authentication is used. This server is also used for user login and creation of tokens for providing access to the data. Whenever the user or a service has authenticated itself with the assistance of an Authentication Server, the latter then creates an ID token, or an access token, as it is commonly called, which is further used by an altogether different service, for instance, Card service to access the protected resources in other services such as Account service.

In case where Card Service is extended to Account Service, it goes with the Token in the Header. On this, the Account Service verifies the validity of the token and also ensure that the collector has the permission to access the required assets. This validation is mostly conducted through checking on the sig, exp and permission of the users connected to token.

Those ensure only authorized and authenticated request are passed through this flow and that there is a reasonably safe way to inter-service communication. In this way, the code does not allow making all the possible calls able for any user, and, if it is necessary to protect some operations, then the code provides the ability to require authentication for those calls. It helps in negotiating security to cater for the unauthorized access and also the services in the microservices architecture.

```
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.11.5</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.11.5</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-jackson</artifactId>
        <version>0.11.5</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-oauth2-resource-server</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-oauth2-jose</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-oauth2-client</artifactId>
    </dependency>
</dependencies>
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
```

```
            .authorizeHttpRequests(requests -> requests.requestMatchers("/auth/token", "/auth/validate/**",
                    "/oauth2/authorize/github", "/oauth2/callback/github").permitAll())
            .oauth2ResourceServer(oauth2 -> oauth2.jwt()).build();
}

@Bean  new *
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    return http.csrf(AbstractHttpConfigurer::disable)
            .authorizeHttpRequests (requests -> requests.requestMatchers(HttpMethod.GET).permitAll()
                            .requestMatchers("/Microservices_Assignment/accounts/**").authenticated()
                            .requestMatchers("/Microservices_Assignment/card/**").authenticated()
                            .requestMatchers("/Microservices_Assignment/loan/**").authenticated()
                    )



                      requestMatchers( "/auth/validate/**",
                      "/oauth2/authorize/keycloak", "/oauth2/callback/github", "/actuator/**").permitAll())
            .oauth2Login(oauth2Login -> oauth2Login.successHandler(successHandler).failureHandler(failureHandler))
            .build();
```

```
spring.application.name=IDENTITY-SERVICE
eureka.client.service-url.defaultZone=http://localhost:8761/eureka/
server.port=9191


spring.security.oauth2.authorizationserver.google.client-id= 674519707866-kfddh69fklekg7n4ii0
spring.security.oauth2.client.registration.goolge.client-secret= GOCSPX-ETk0DhFZFY_tTJJ7Ty-eG
spring.security.oauth2.client.registration.google.scope=profile,email
spring.security.oauth2.client.registration.google.redirect-uri=http://localhost8097/login/out
spring.security.oauth2.client.registration.google.authorization-grant-type=authorization_code
spring.security.oauth2.client.registration.google.client-name=Google


spring.security.oauth2.client.provider.google.authorization-uri=http://account.google.com/o/o
spring.security.oauth2.client.provider.google.token-uri=https://oauth.googleapis.com/token
spring.security.oauth2.client.povider.gooogle.uer-infor-uri=https//www.googleleapis.com/oauth2/
spring.security.oauth2.client.provider.google.user-name-attribute=sub
```

The OAuth 2. flow the use of Google as an external authenticating/authorising third party is applied. Thus, the user comes to the authorization page of Google where he or she enters the account data and provides the necessary access. In Google, after a request authentication processes, an access token will be issued to the application. This token is then used to get authorized to get into secured areas or to get hold of services in the application so that only authorized users are allowed by the application to access particular resources.

Authentication Service

In the OAuth 2. flow for the application, Google is used as a third-party authorization server. The process starts when a user tries to use a resource that is protected by a WebID

system. It sends the user to the Google's OAuth 2. 0 authorization endpoint, through which Google asks for the access rights that the user wants to grant. Google then checks the user's credentials and thereupon provides an authorization code to the application's authentication server. This authorization code is then used to request for an access token from Google, using the authentication server's token endpoint. It will be so useful to mention here that the server put the client credentials in this request. Google returns an access token while the authentication server may also receive a refresh token as shall be discussed later in the paper.

Inserting Filter in the Api Gateway

```
public class OncePerRequestFilter extends AbstractGatewayFilterFactory<OncePerRequestFilter.Config> {

    public OncePerRequestFilter() { super(Config.class); }

    @Override
    public GatewayFilter apply(Config config) {
        return new OrderedGatewayFilter((exchange, chain) -> {
            if (exchange.getAttribute("onceperrequest") == null) {
                exchange.getAttributes().put("onceperrequest", true);
                ServerHttpRequest request = exchange.getRequest();
                request.getHeaders().get("Authorization ");
                return chain.filter(exchange);
            } else {

                // Logic to handle repeated execution
                return Mono.empty();
            }
        }, Ordered.LOWEST_PRECEDENCE);
    }

    public static class Config {
        // Put the configuration properties for your filter here
    }
}
```

```java
@Service
public class ValidationServiceImpl {
    @Autowired
    private RestTemplate restTemplate;

    public int validateToken(String token) {
        String uri = "http://SECURITY-SERVICE-HMS/auth/validate/" + token;
        ResponseEntity<Integer> resp = restTemplate.getForEntity(uri, Integer.class);
        return resp.getBody();

    }

}
```

Validating the service token from the another service

Token Access

If the card service requires information from account service, it sends the access token in the Authorization header of HTTP requests. The account service as the resource server checks the token as its authenticity and whether it has expired or not using Google's public keys. If the token is valid, the account service provides an access to the data that has been requested.

Token lifetime and expiry

Access tokens are small and their lifetime is limited to the defined time. At some point, the access token expires, and the authentication server will use the refresh token to manage to get a new access token from Google's token endpoint. This process helps the card service to remain always connected to resources and does not demand the user to re-authenticate. The token management works to guarantee that in the case where a token is being used to access data within the application only the authenticated and authorized request can be permitted to access the sensitive data.

Pretty-print


```
{
  "respCode": 200,
  "respDesc": "Success",
  "jwtToken": "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJhaXJlZXNvbml5YTdAZ21haWwuY29tIiwiaWF0IjoxNzIzMzgwNzM1LCJleHAiOjE3MjMzODI1MzV9.E7r_BMEgn1h3ixn493W7vhvv-bk_pjmySK8uwC-u9H0"
}
```


Output

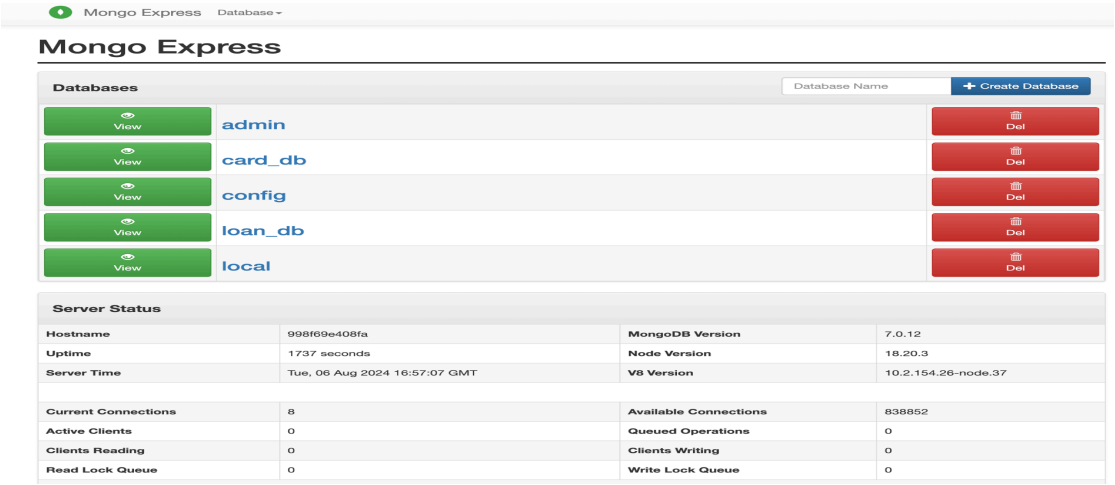

```
[
  {
    "id": "66b260953abede249915862d",
    "cardNumber": "12456469495",
    "account": {
      "id": "1",
      "accountNumber": "1233",
      "accountHolderName": "Ram Bahadur",
      "balance": 50000
    },
    "cardType": "CREDIT",
    "accountId": 1
  },
  {
    "id": "66b2e828f446407b35371d35",
    "cardNumber": "3",
    "account": null,
    "cardType": "DEBIT",
    "accountId": 3
  },
  {
    "id": "66b2e841f446407b35371d36",
    "cardNumber": "3",
    "account": null,
    "cardType": "DEBIT",
    "accountId": 3
  },
  {
    "id": "66b2e841f446407b35371d37",
    "cardNumber": "4",
    "account": null,
    "cardType": "CREDIT",
    "accountId": 4
  },
  {
    "id": "66b2e841f446407b35371d38",
    "cardNumber": "5",
    "account": null,
    "cardType": "CREDIT",
    "accountId": 5
  },
  {
    "id": "66b2e841f446407b35371d39",
    "cardNumber": "6",
    "account": null,
    "cardType": "DEBIT",
    "accountId": 6
  },
  {
    "id": "66b2e841f446407b35371d3a",
    "cardNumber": "7",
    "account": null,
    "cardType": "CREDIT",
    "accountId": 7
  },
  {
    "id": "66b2e841f446407b35371d3b",
    "cardNumber": "8",
    "account": null,
    "cardType": "DEBIT",
    "accountId": 8
  },
```


localhost:8083/card

Close | Use a mixing cons... | (156) YouTube | Download Django |... | IPL 2017 Livestrea... | study | HTML elements re... | Adding Boostra

Pretty-print


```
{
  "id": "66b260953abede249915862d",
  "cardNumber": "12456469495",
  "account": {
    "id": "1",
    "accountNumber": "1233",
    "accountHolderName": "Ram Bahadur",
    "balance": 50000
  },
  "cardType": "CREDIT",
  "accountId": 1
},
{
  "id": "66b2e828f446407b35371d35",
  "cardNumber": "3",
  "account": null,
  "cardType": "DEBIT",
  "accountId": 3
},
{
  "id": "66b2e841f446407b35371d36",
  "cardNumber": "3",
  "account": null,
  "cardType": "DEBIT",
  "accountId": 3
},
{
  "id": "66b2e841f446407b35371d37",
  "cardNumber": "4",
  "account": null,
  "cardType": "CREDIT",
  "accountId": 4
},
{
  "id": "66b2e841f446407b35371d38",
  "cardNumber": "5",
  "account": null,
  "cardType": "CREDIT",
  "accountId": 5
},
```



Access token (received JWT token)

Here the scenario is like client application is trying to invoke auth server with the help of postman

Encoded PASTE A TOKEN HERE


```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiw
ia2lkIiA6ICJMdUlCU1FiOUZQdmZSR3dFMUhMbU
s4aHFKOWZkVGlKQlcxRExzMmlQX3FFIn0.eyJle
HAiOjE3MjMzNzUxNDYsImlhdCI6MTcyMzM3NTA4
NiwianRpIjoiZDBlYzZiOGItNGVkNi00OWQ5LWE
xN2MtZTAwZjgyZDY3YTJmIiwiaXNzIjoiaHR0cD
ovL2xvY2FsaG9zdDo4MDgwL3JlYWxtcy9tYXN0Z
XIiLCJhdWQiOiJwcm9qZWN0X21pY3Jvc2Vydmlj
ZXMtY2FsbGNlbnRlci1iYyIsInN1YiI6IjJmOTE
yNjA2LTQ3ZWItNGM3[illegible]1MzliOW
E0OSIsInR5cCI6IklEIiwiYXpwIjoicHJvamVjd
F9taWNyb3NlcnZpY2VzLWNhbGxjZW50ZXItY2Mi
LCJhdF9oYXNoIjoiQWdCN3lXTzc4ZzBMOEJOckF
WejFIQSIsImFjciI6IjEiLCJlbWFpbF92ZXJpZm
llZCI6ZmFsc2UsImNsaWVudEhvc3QiOiIwOjA6M
DowOjA6MDowOjEiLCJwcmVmZXJyZWRfdXNlcm5h
bWUiOiJzZXJ2aWNlLWFjY291bnQtcHJvamVjdF9
taWNyb3NlcnZpY2VzLWNhbGxjZW50ZXItY2MiLC
JjbGllbnRBZGRyZXNzIjoiMDowOjA6MDowOjA6M
DoxIiwiY2xpZW50X2lkIjoicHJvamVjdF9taWNy
b3NlcnZpY2VzLWNhbGxjZW50ZXItY2MifQ.ckgU
g_mUwI2O_jvs9wuFHXr4pyZ1AqUkMPoCWzlddMo
7j0MOgVQtk7oVgxJLs1AmvBnezLutozdWvaq1L9
FlkywX2URaGooMO4ntGTFuyhPU7Z0vxoi4h1sZB
sF-ZgX7tcp07t5yhow_JOln6wdxT4iMxI-
SKwJnfHh36q5--
nwPnSAK0HeVBZkby0pzzpPotsyr7QISofOYtuJp
7Fm8kCKfeslyXGodoviSvwPAkN8vfJPJDrQlZ1m
LkqmjgESqj6tlVwFRkFo05_80lhQi9PkLXJ8FUy
llJ-
0PEDCROEz0p0Hq1rAA9jSbJsQ0tmvYHF62glgEh
K0x-EC1etjR_A
```


Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE


```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "LuIBSQb9FPvfRGwE1HLmK8hqJ9fdTiJBW1DLs2iP_qE"
}
```


PAYLOAD: DATA


```
{
  "exp": 1723375146,
  "iat": 1723375086,
  "jti": "d0ec6b8b-4ed6-49d9-a17c-e00f82d67a2f",
  "iss": "http://localhost:8080/realms/master",
  "aud": "project_microservices-callcenter-cc",
  "sub": "2f912606-47eb-4c77-994a-36d0539b9a49",
  "typ": "ID",
  "azp": "project_microservices-callcenter-cc",
  "at_hash": "AgB7yWO78g0L8BNrAVz1HA",
  "acr": "1",
  "email_verified": false,
  "clientHost": "0:0:0:0:0:0:0:1",
  "preferred_username": "service-account-
project_microservices-callcenter-cc",
  "clientAddress": "0:0:0:0:0:0:0:1",
  "client_id": "project_microservices-callcenter-cc"
}
```


VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  Public Key in SPKI, PKCS #1,
  X.509 Certificate, or JWK stri
  ng format.
  ,
  Private Key in PKCS #8, PKCS #
  1, or JWK string format. The k
  ey never leaves your browser.
)
```

Client ID token

Encoded PASTE A TOKEN HERE


```
eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiw
ia2lkIiA6ICJMdUlCU1FiOUZQdmZSR3dFMUhMbU
s4aHFKOWZkVGlKQlcxRExzMmlQX3FFIn0.eyJle
HAiOjE3MjMzNzUxNDYsImlhdCI6MTcyMzM3NTA4
NiwianRpIjoiYzY3NjEzYzEtYjgyMC00NjdjLWE
4N2MtYjgwZmY2NjA0ZmEzIiwiaXNzIjoiaHR0cD
ovL2xvY2FsaG9zdDo4MDgwL3JlYWxtcy9tYXN0Z
XIiLCJhdWQiOiJhY2NvdW50Iiwic3ViIjoiMmY5
MTI2MDYtNDdlYi00Yzc3LTk5NGEtMzZkMDUzOWI
5YTQ5IiwidHlwIjoiQmVhcmVyIiwiYXpwIjoicH
JvamVjdF9taWNyb3NlcnZpY2VzLWNhbGxjZW50Z
XItY2MiLCJhY3IiOiIxIiwiYWxsb3dlZC1vcmln
aW5zIjpbIi8qIl0sInJlYWxtX2FjY2VzcyI6eyJ
yb2xlcyI6WyJkZWZhdWx0LXJvbGVzLWNhc3RlcI
IsIm9mZmxpbmVfYWNjZXNzIiwidW1hX2F1dGhvc
ml6YXRpb24iXX0sInJlc291cmNlX2FjY2VzcyI6
eyJhY2NvdW50Ijp7InJvbGVzIjpbIm1hbmFnZS1
hY2NvdW50IiwibWFuYWdlLWFjY291bnQtbGlua3
MiLCJ2aWV3LXByb2ZpbGUiXX19LCJzY29wZSI6I
m9wZW5pZCBwcm9maWxlIGVtYWlsIiwiZW1haWxf
dmVyaWZpZWQiOmZhbHNlLCJjbGllbnRIb3N0Ijo
iMDowOjA6MDowOjA6MDoxIiwicHJlZmVycmVkX3
VzZXJuYW1lIjoic2VydmljZS1hY2NvdW50LXByb
2plY3RfbWljcm9zZXJ2aWNlcy1jYWxsY2VudGVy
LWNjIiwiY2xpZW50QWRkcmVzcyI6IjA6MDowOjA
6MDowOjA6MSIsImNsaWVudF9pZCI6InByb2plY3
RfbWljcm9zZXJ2aWNlcy1jYWxsY2VudGVyLWNjI
n0.LnHZb2cMXZyOmhuBl4Z7YrLI62QNMtW1pK5q
FO7UWG_S0sDHaaHJYiW9bMOVV4W7i8ENJIPncED
6e9uQ9YMvLbizmAviTus43ANQK4qykQ7kGXP-
lgJhyUv1cXZ4xIJ3-
uqoKN_y5kVlbrJ9mbRQdEJqSv6gc7R2RtGVhbVT
dmZga404ShmwQnnQg7YnKsj73xU3ODGed6g9cGH
xc5MPrl3P7WUGxxICc7rP1YAcSCIbKZR_u2UuaY
IvsUpj3QSviGOLWI_Lq8F6AjnuJO1tiDOPc0pg5
6swZ5W5CU5dNcrau_fUma1lylsVYL6fYMnEJFvi
zDuUN6RTAVwAA7fnKQ
```


Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE


```
  "typ": "JWT",
  "kid": "LuIBSQb9FPvfRGwE1HLmK8hqJ9fdTiJBW1DLs2iP_qE"
}
```


PAYLOAD: DATA


```
{
  "exp": 1723375146,
  "iat": 1723375086,
  "jti": "c67613c1-b820-467c-a87c-b80ff6604fa3",
  "iss": "http://localhost:8080/realms/master",
  "aud": "account",
  "sub": "2f912606-47eb-4c77-994a-36d0539b9a49",
  "typ": "Bearer",
  "azp": "project_microservices-callcenter-cc",
  "acr": "1",
  "allowed-origins": [
    "/*"
  ],
  "realm_access": {
    "roles": [
      "default-roles-master",
      "offline_access",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "openid profile email",
  "email_verified": false,
  "clientHost": "0:0:0:0:0:0:0:1",
  "preferred_username": "service-account-
project_microservices-callcenter-cc",
  "clientAddress": "0:0:0:0:0:0:0:1",
  "client_id": "project_microservices-callcenter-cc"
}
```


VERIFY SIGNATURE

```
RSASHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  Public Key in SPKI, PKCS #1,
  X.509 Certificate, or JWK stri
  ng format.
  ,
  Private Key in PKCS #8, PKCS #
  1, or JWK string format. The k
  ey never leaves your browser.
)
```

Decoded details of access token

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

Adding dependencies into gateway server

GET http://localhost:8086/account

Params Authorization Headers (8) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
{

    "accountHolderName": "gaman",
    "accountNumber": 100199257832,
    "balance" : 10000
}

```

Body Cookies Headers (10) Test Results

Pretty Raw Preview Visualize JSON

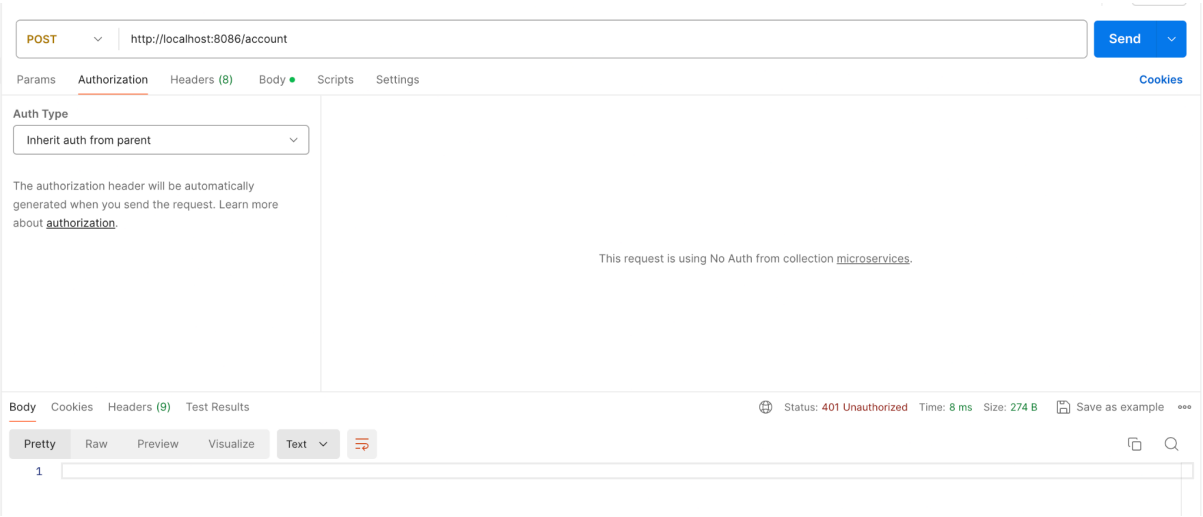
```
[
    {
        "id": 1,
        "accountNumber": "100199282912",
        "accountHolderName": "sulav Dhakal",
        "balance": 3000
    },
    {
        "id": 2,
        "accountNumber": "100199282912",
        "accountHolderName": "sulav Dhakal",
        "balance": 3000
    }
]
```

Trying to invoke account service get account created info,, we are successfully able to get account info.

```
{

    "accountHolderName": "gaman",
    "accountNumber": 100199257832,
    "balance" : 10000
}

```

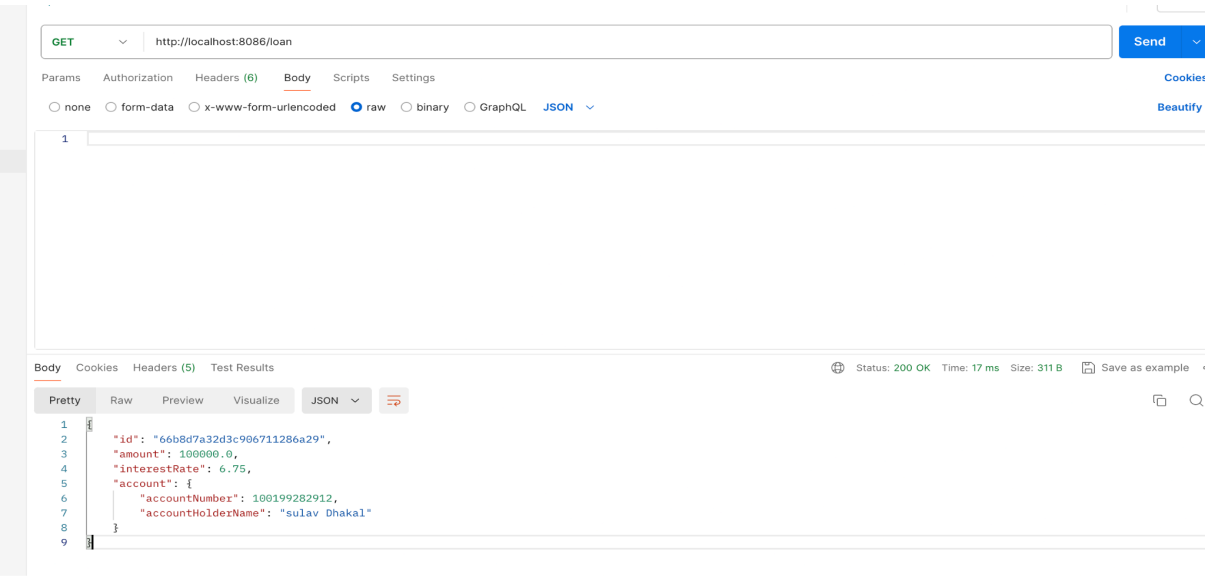
Body Cookies Headers (9) Test Results Status: 401 Unauthorized Time: 8 ms Size: 274 B Save as example

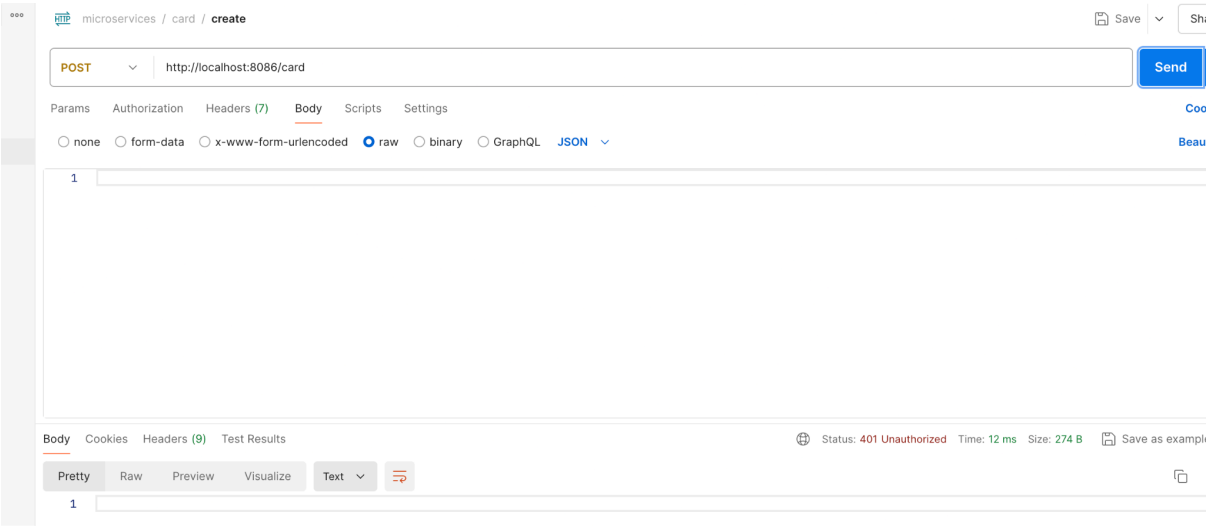
Pretty Raw Preview Visualize Text

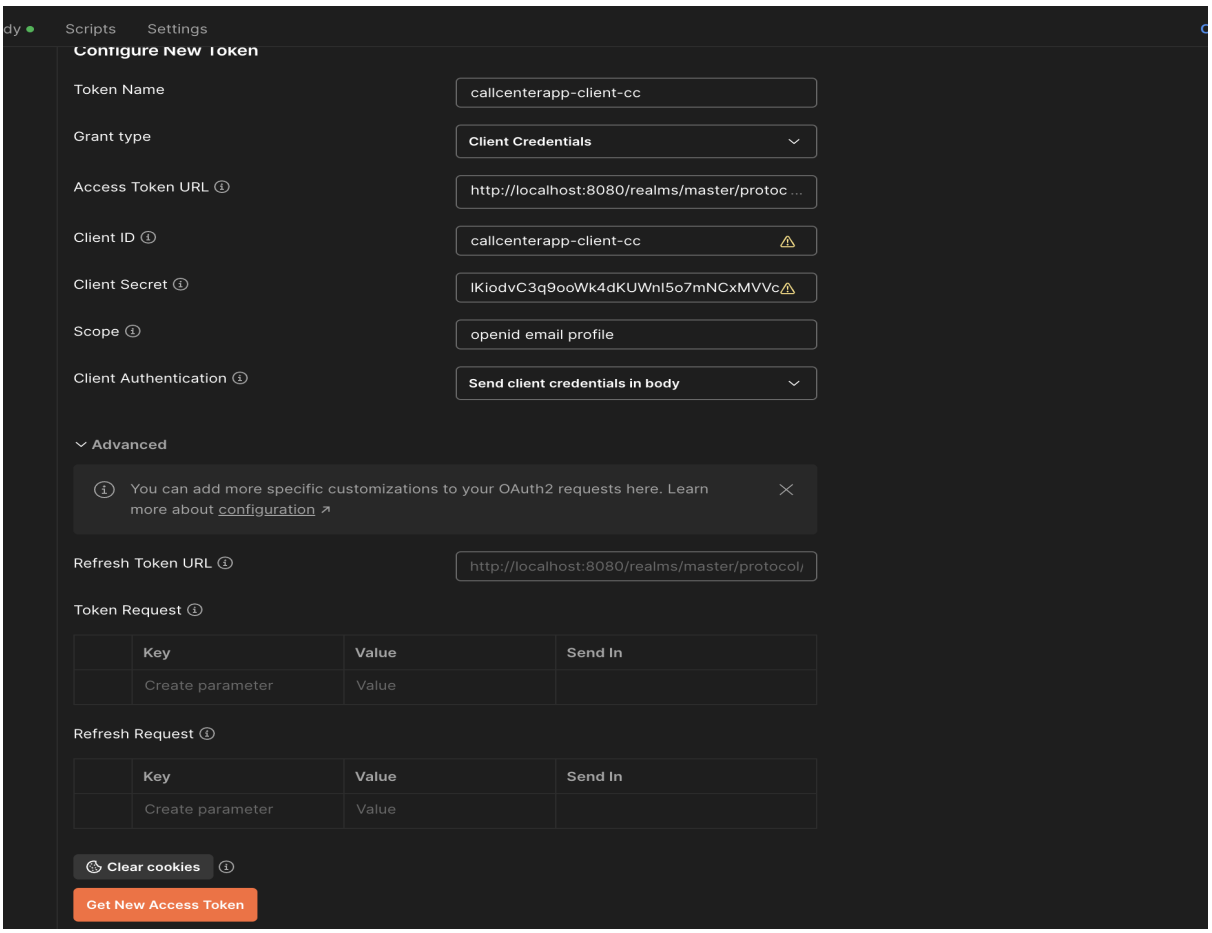


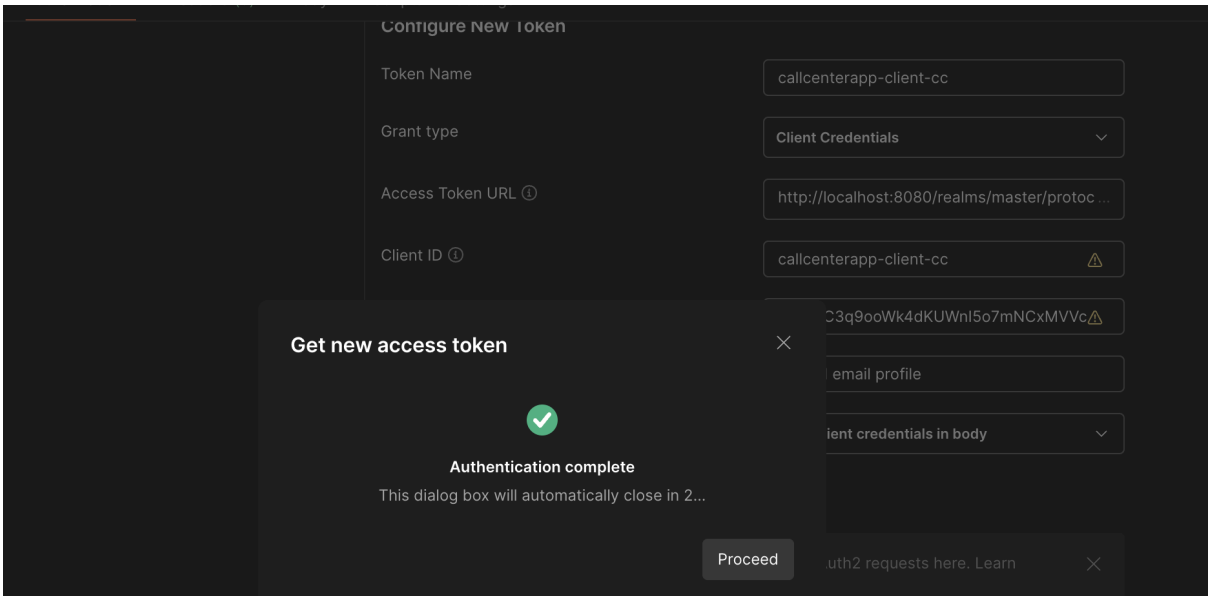
Here, create account using postman post, we have not provided any authorization access token due to security already implementation as the result it is showing 402 unauthorized access.

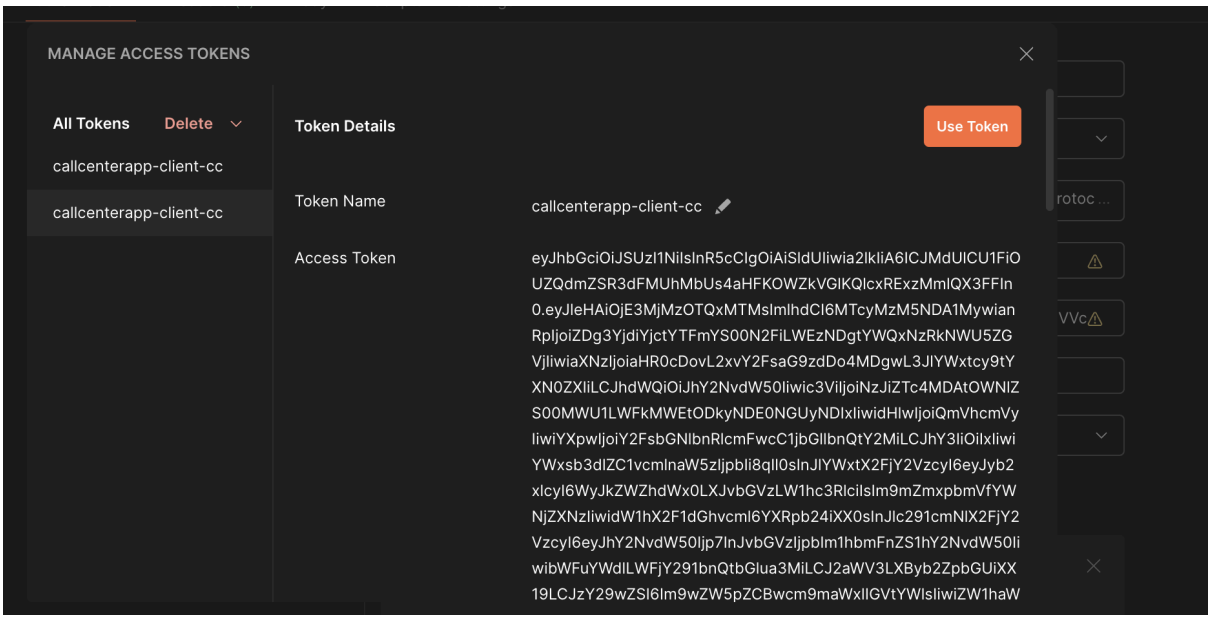
Similarly with card and loan

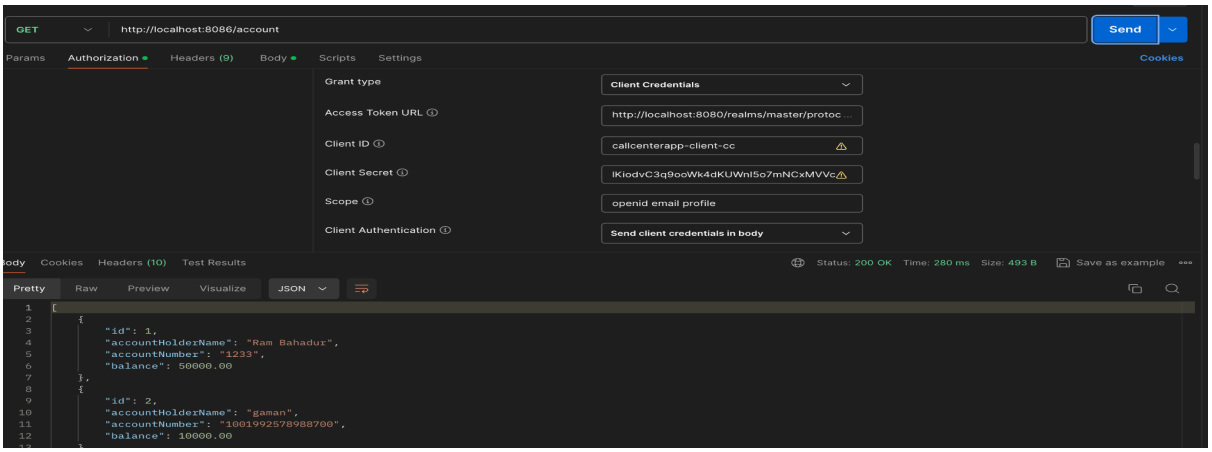



microservices / card / create
Save
POST
http://localhost:8086/card
Send
Params
Authorization
Headers (7)
Body
Scripts
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
1
Body
Cookies
Headers (9)
Test Results
Status: 401 Unauthorized
Time: 12 ms
Size: 274 B
Pretty
Raw
Preview
Visualize
Text
1


Scripts
Settings
Configure New Token
Token Name
callcenterapp-client-cc
Grant type
Client Credentials
Access Token URL
http://localhost:8080/realms/master/protoc ...
Client ID
callcenterapp-client-cc
Client Secret
lKiodvC3q9ooWk4dKUWnI5o7mNCxMVVc
Scope
openid email profile
Client Authentication
Send client credentials in body
Advanced
You can add more specific customizations to your OAuth2 requests here. Learn more about configuration
Refresh Token URL
http://localhost:8080/realms/master/protocol/
Token Request
Key
Value
Send In
Create parameter
Value
Refresh Request
Key
Value
Send In
Create parameter
Value
Clear cookies
Get New Access Token


Configure New Token
Token Name
callcenterapp-client-cc
Grant type
Client Credentials
Access Token URL
http://localhost:8080/realms/master/protoc ...
Client ID
callcenterapp-client-cc
Get new access token
Authentication complete
This dialog box will automatically close in 2...
Proceed





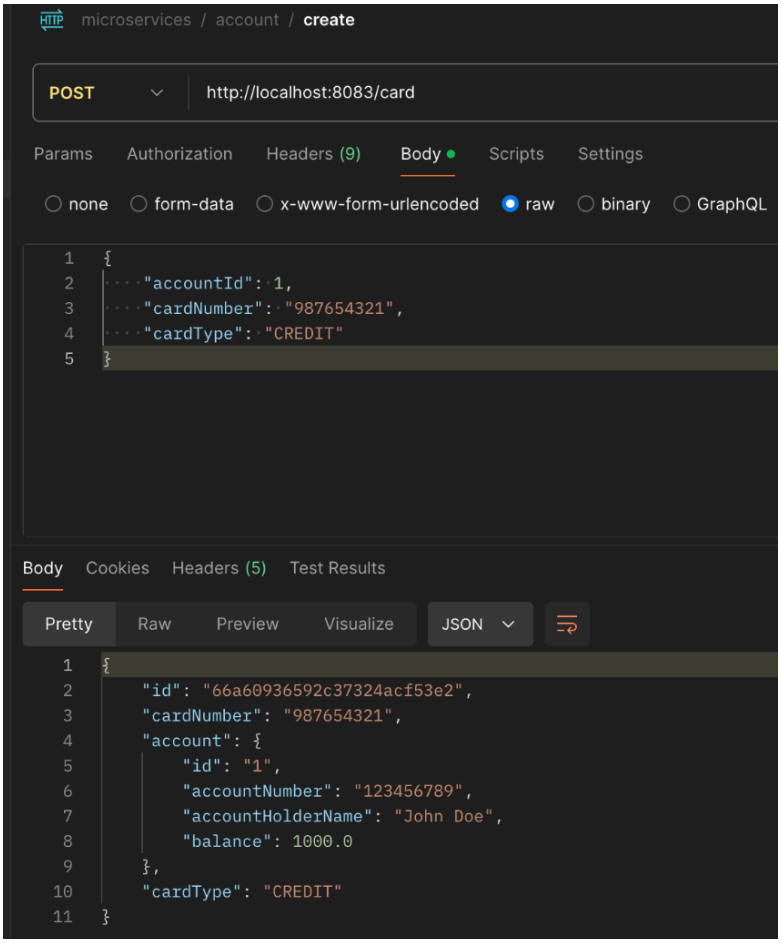
Enough to validate

Source Code:

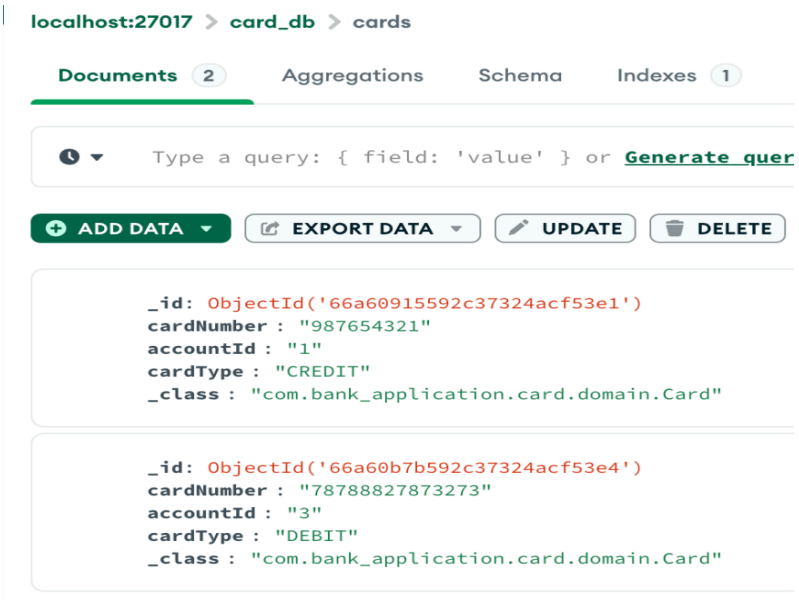
The Source Code link is :

https://drive.google.com/drive/folders/1t2HjoV1BCbudZLkEnbaAp7qrIZKkbZnK?usp=sharing

ScreenShot



Interaction between services Card and account services



Database connection for Card

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
ACCOUNT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.18.13:account-service:8081
BANK-API-GATEWAY	n/a (1)	(1)	UP (1) - 192.168.18.13:bank-api-gateway:8080
CARD	n/a (1)	(1)	UP (1) - 192.168.18.13:card:8083
LOAN	n/a (1)	(1)	UP (1) - 192.168.18.13:loan:8082

Registry of Eureka server

GET http://localhost:8080/account

Params Authorization Headers (9) Body Scripts Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

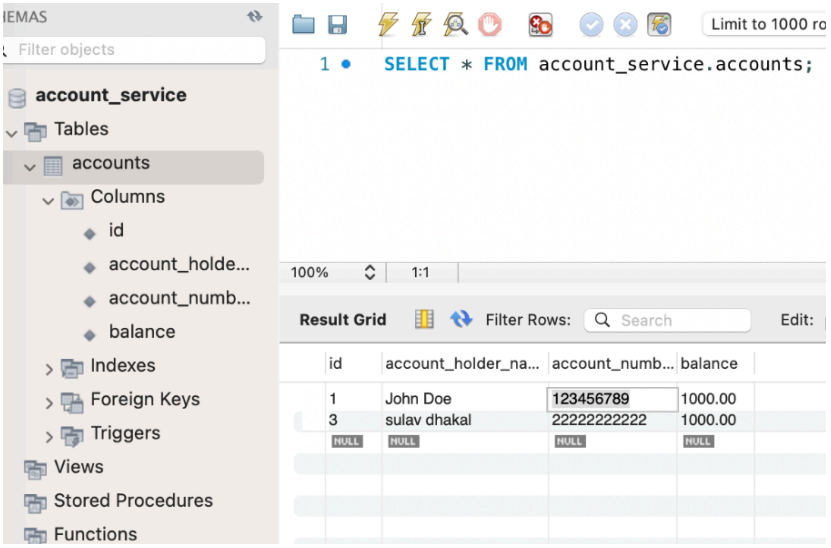
```
{
    "accountId": 3,
    "cardNumber": "78788827873273",
    "cardType": "DEBIT"
}
```

Body Cookies Headers (3) Test Results

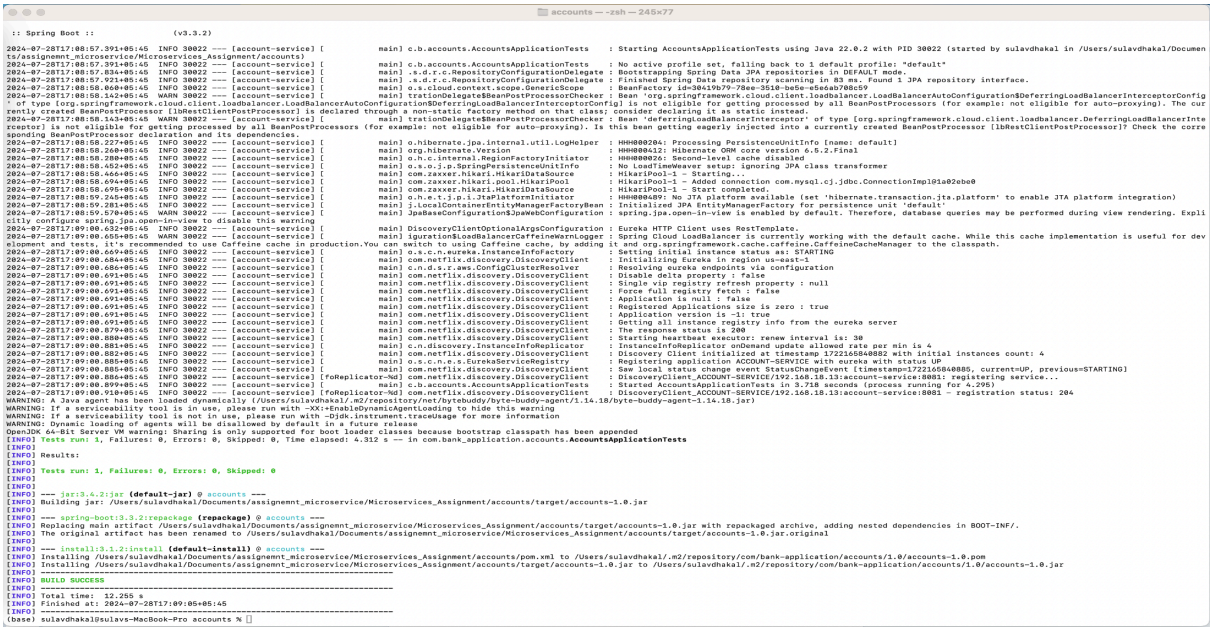
Pretty Raw Preview Visualize JSON

```
[
    {
        "id": 1,
        "accountHolderName": "John Doe",
        "accountNumber": "123456789",
        "balance": 1000.00
    },
    {
        "id": 3,
        "accountHolderName": "sulav dhakal",
        "accountNumber": "22222222222",
        "balance": 1000.00
    }
```

Account service testing for MySql DB



Data stored in MySql Database

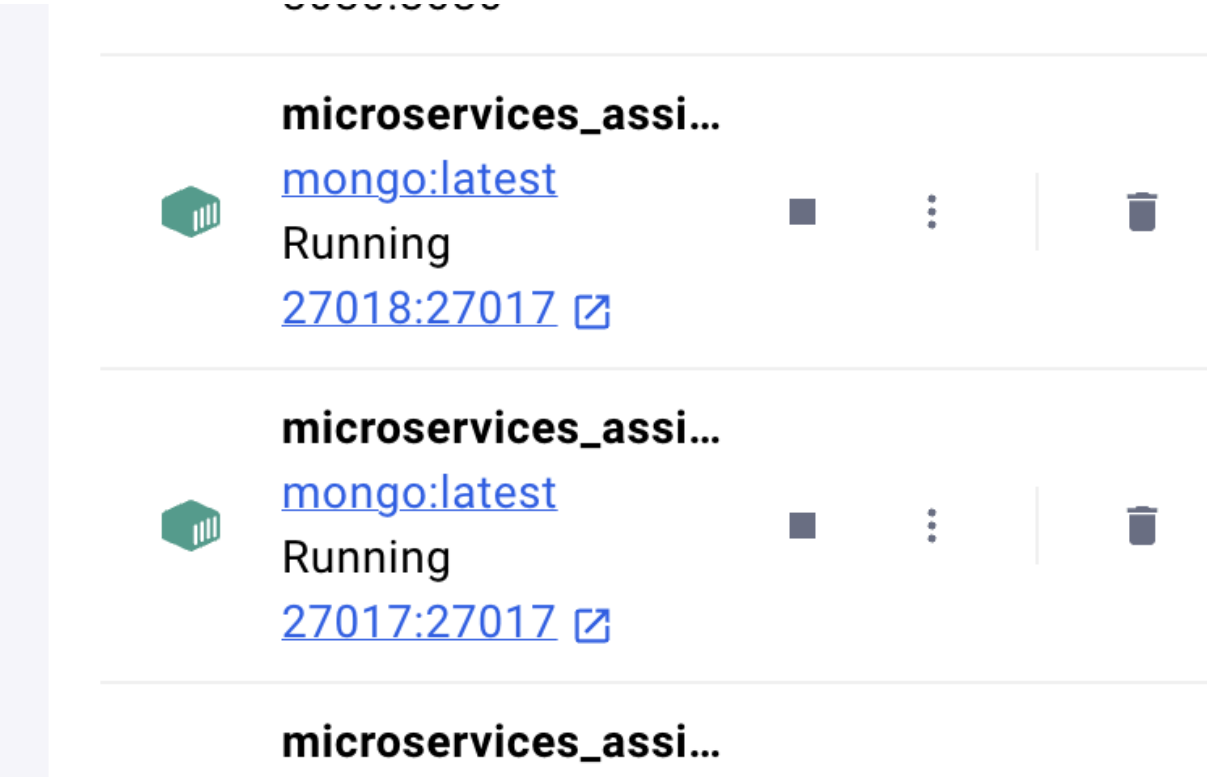


Docker Initialization and Creation of Jar File

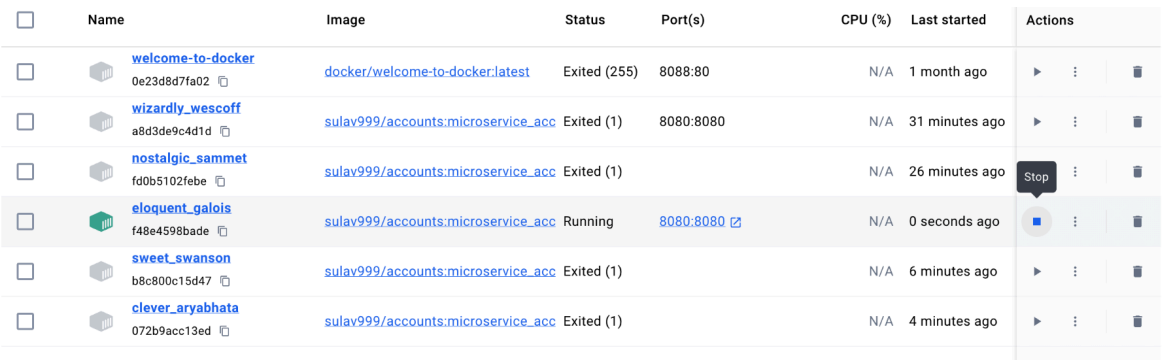
```
(base) sulavdhakal@sulavs-MacBook-Pro accounts % docker images
REPOSITORY                  TAG                 IMAGE ID       CREATED              SIZE
sulav999/accounts           microservice_account  d2da5d7990ce   About a minute ago   481MB
docker/welcome-to-docker    latest              648f93a1ba7d   8 months ago         19MB
(base) sulavdhakal@sulavs-MacBook-Pro accounts %
```

```
        at com.bank_application.accounts.AccountsApplication.main(AccountsApplication.java:11) ~[!/:1.0]
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method) ~[na:na]
        at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77) ~[na:na]
        at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43) ~[na:na]
        at java.base/java.lang.reflect.Method.invoke(Method.java:568) ~[na:na]
        at org.springframework.boot.loader.launch.Launcher.launch(Launcher.java:91) ~[accounts-1.0.jar:1.0]
        at org.springframework.boot.loader.launch.Launcher.launch(Launcher.java:53) ~[accounts-1.0.jar:1.0]
        at org.springframework.boot.loader.launch.JarLauncher.main(JarLauncher.java:58) ~[accounts-1.0.jar:1.0]
```

Account service image deployed in Docker



Microsevice deployed and running in Docker container



Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
welcome-to-docker 0e23d8d7fa02	docker/welcome-to-docker:latest	Exited (255)	8088:80	N/A	1 month ago	
wizardly_wescoff a8d3de9c4d1d	sulav999/accounts:microservice_acc	Exited (1)	8080:8080	N/A	31 minutes ago	
nostalgic_sammet fd0b5102febe	sulav999/accounts:microservice_acc	Exited (1)		N/A	26 minutes ago	Stop
eloquent_galois f48e4598bade	sulav999/accounts:microservice_acc	Running	8080:8080	N/A	0 seconds ago	
sweet_swanson b8c800c15d47	sulav999/accounts:microservice_acc	Exited (1)		N/A	6 minutes ago	
clever_aryabhata 072b9acc13ed	sulav999/accounts:microservice_acc	Exited (1)		N/A	4 minutes ago	

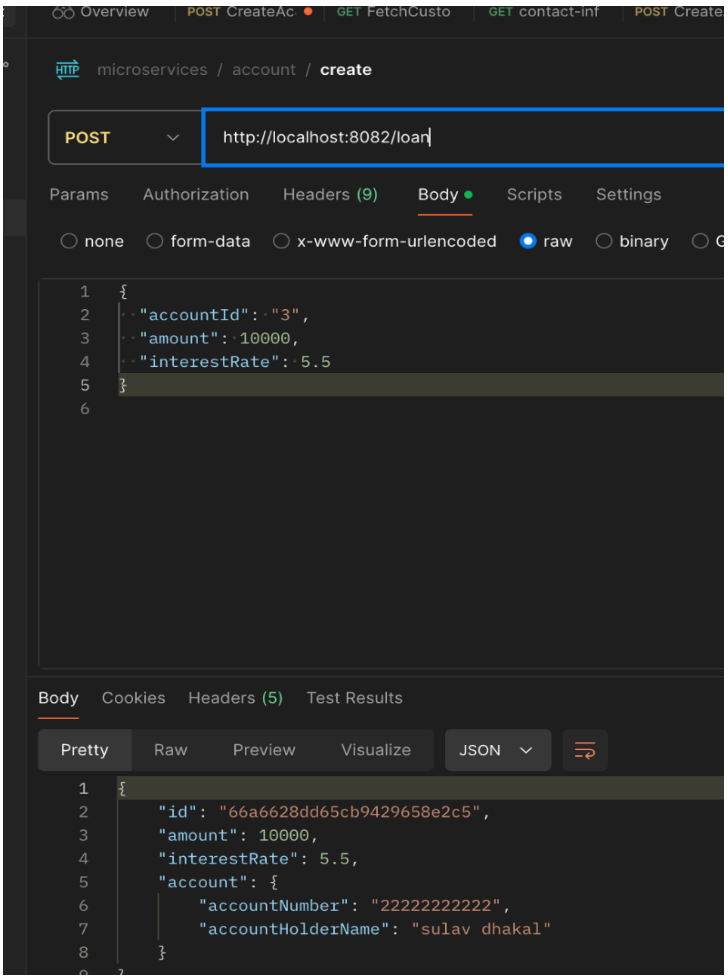
Service Running in Docker

Name	Tag	Status	Created	Size
microservices_assignment-card-service 23c96731a75d	latest	In use	16 minutes ag	454.58 MB
microservices_assignment-service-discovery bf0c6b96b748	latest	In use	16 minutes ag	458.95 MB
microservices_assignment-loan-service e9daac476adb	latest	In use	16 minutes ag	454.58 MB
microservices_assignment-account-service f1b16cceda5a	latest	In use	16 minutes ag	481.24 MB
microservices_assignment-bank-api-gateway 22cfc798fc96	latest	In use	16 minutes ag	451.42 MB
mysql 2010e93e56ca	latest			608.44 MB
mongo 4b372541400c	latest			760.57 MB

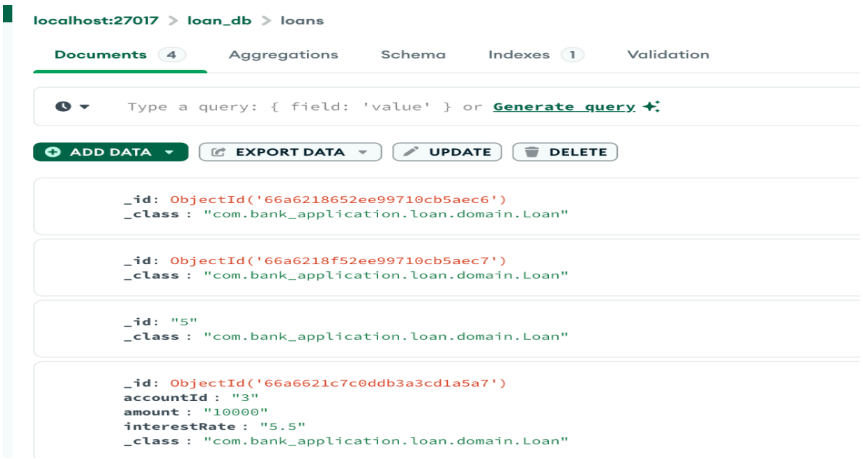
Microservices_Assignment — -zsh — 80×24

```
 ✔ Container microservices_assignment-loan-db-1            Created        0.1s
 ✔ Container microservices_assignment-account-db-1         Created        0.1s
 ✔ Container microservices_assignment-account-service-1    Created        0.1s
 ✔ Container microservices_assignment-loan-service-1       Created        0.1s
 ✔ Container microservices_assignment-card-service-1       Created        0.1s
Attaching to account-db-1, account-service-1, bank-api-gateway-1, card-db-1, car
d-service-1, loan-db-1, loan-service-1, service-discovery-1
Gracefully stopping... (press Ctrl+C again to force)
Error response from daemon: Ports are not available: exposing port TCP 0.0.0.0:3
306 -> 0.0.0.0:0: listen tcp 0.0.0.0:3306: bind: address already in use
(base) sulavdhakal@sulavs-MacBook-Pro Microservices_Assignment % docker-compose
up

[+] Running 4/0
 ✔ Container microservices_assignment-card-db-1            Running        0.0s
 ✔ Container microservices_assignment-bank-api-gateway-1   Created        0.0s
 ✔ Container microservices_assignment-service-discovery-1  Created        0.0s
 ✔ Container microservices_assignment-loan-db-1            Running        0.0s
Attaching to account-db-1, account-service-1, bank-api-gateway-1, card-db-1, car
d-service-1, loan-db-1, loan-service-1, service-discovery-1
Gracefully stopping... (press Ctrl+C again to force)
Error response from daemon: Ports are not available: exposing port TCP 0.0.0.0:3
306 -> 0.0.0.0:0: listen tcp 0.0.0.0:3306: bind: address already in use
(base) sulavdhakal@sulavs-MacBook-Pro Microservices_Assignment %
```

Microservice Deployed in Docker Container



Data posted in the loan service after the service runs from a Docker container



Database connection for Card

POST http://localhost:8086/loan

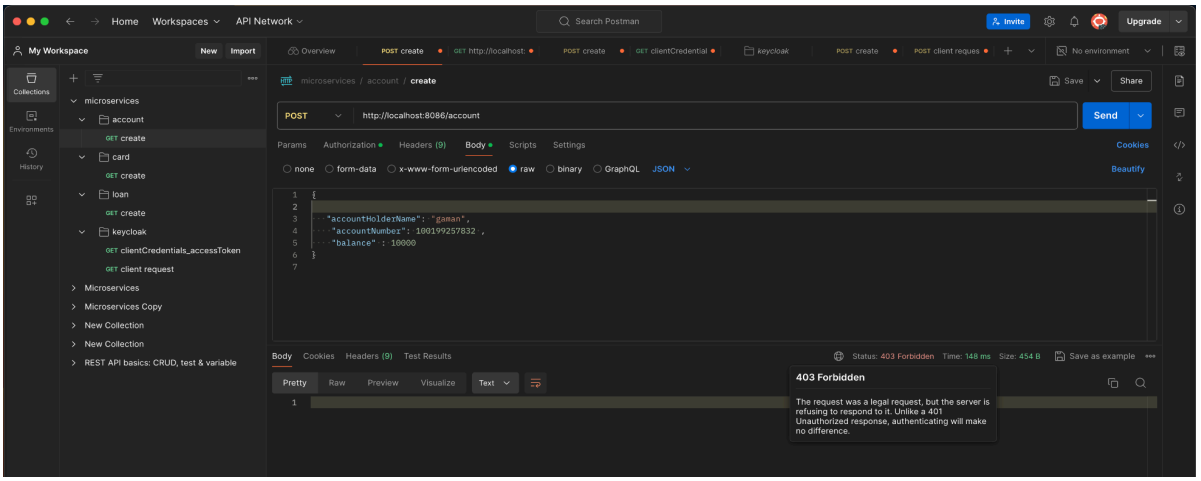
Send

Params Authorization Headers (7) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

Body Cookies Headers (9) Test Results Status: 401 Unauthorized Time: 21 ms Size: 274 B Save as example

Pretty Raw Preview Visualize Text



GET http://localhost:8086/card

Send

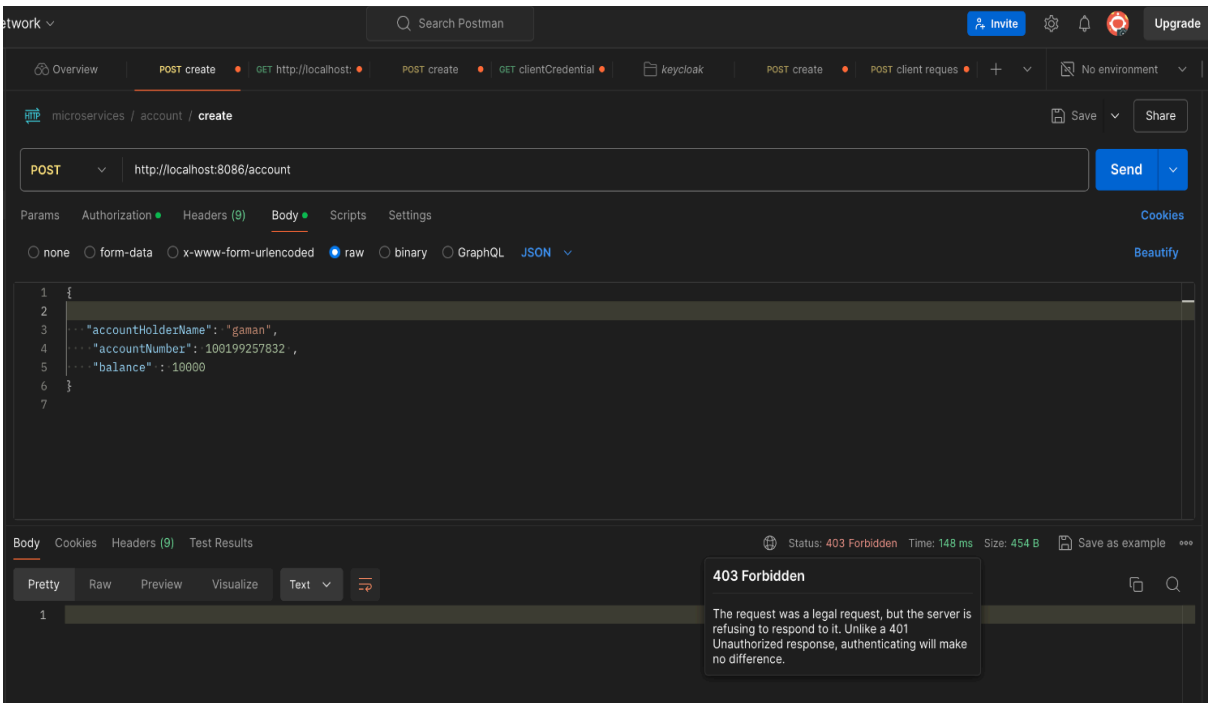
Params Authorization Headers (6) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

Body Cookies Headers (5) Test Results Status: 200 OK Time: 240 ms Size: 368 B Save as example

Pretty Raw Preview Visualize JSON


```
{
    "id": "66b8d815b1274503010fb7b1",
    "cardNumber": "173-445-393-20220",
    "account": {
        "id": "2",
        "accountNumber": "100199282912",
        "accountHolderName": "sulav Dhakal",
        "balance": 3000.0
    },
    "cardType": "CREDIT",
    "accountId": 2
}
```



Conclusion

A real-life example of how microservices architecture benefits the sphere of accounts, cards, and loans is demonstrated by the “banking application” project. If the application is broken down into separate services where each is focused on carrying out an individual task, the entire system becomes more efficient and easy to manage. Thus, for the interaction between the microservices we used Spring Cloud components namely Eureka for service registry/deregistration and API Gateway which helps in routing the requests between the address components. Services were deployed in a very minimal and compact way by Containerization using Docker and Docker Compose. Spring Cloud allows for the implementation of the microservice architecture, as such, it is possible to create a scalable,

easily expandable, and maintainable banking system. The services do not necessarily depend on one another and hence can be built and released consecutively which adheres to the concepts of continuous integration and continuous deployment. Spring Cloud Gateway has excellent routing features which come in handy, and Eureka Server enables efficient microservices discovery, laying a solid groundwork for the microservices architecture.

OAuth 2. 0 and token validation are some of the main advantages when it comes to protection of applications as well as the control of access to the resource for users. OAuth 2. 0 means that it becomes more convenient to permit the application to utilize the users' resources without necessarily providing their passwords which leads to high security measures. This has adopted the usage of access tokens to govern user sessions thus regulates the abilities of the user on what he lays his hands on hence minimizing the cases of unauthorized access.

Token validation helps to make sure that these tokens are genuine and have not been tampered with thus will ensure the authentication process is secure. It complements the function that validates tokens to make certain that they originate from an accredited entity and are still active; this assists to safeguard confidential data and restrict entree to protected resources merely to those who are authorized. This approach improves security and simplifies the task of organizing user access on a large scale and with minimal probability of error.

References

Camunda. (2024). *How to Write Microservices in Java*. Camunda. Retrieved July 29, 2024, from https://camunda.com/resources/microservices/java/

Docker. (2013). *Docker overview*. Docker Docs. Retrieved July 29, 2024, from https://docs.docker.com/guides/docker-overview/

Guide, S., & Jain, S. (2024, May 21). *Java Spring Boot Microservices Example - Step by Step Guide*. GeeksforGeeks. Retrieved July 15, 2024, from https://www.geeksforgeeks.org/java-spring-boot-microservices-example-step-by-step-guide/

Reddy, M., & Eazy Bytes. (2023). *MASTER MICROSERVICES WITH SPRINGBOOT, DOCKER,KUBERNETES*. Eazy Bytes. 1

Stackify. (2024). *What are Microservices? Code Examples, Tutorials & More*. Stackify. Retrieved July 29, 2024, from https://stackify.com/what-are-microservices/

Stine, M. (2015, July 14). *Microservices with Spring*. Spring. Retrieved July 15, 2024, from https://spring.io/blog/2015/07/14/microservices-with-spring