CS328 · Distributed and Cloud Computing Fall 2024

Assignment 2

THA SREYNY / 12113053

December 1, 2024

1 Introduction

This document describes the architecture and design of the *SUSTech Merch Store*, an online store where customers can purchase a limited selection of three products. The store will offer its services through a RESTful API that allows customers to interact with the system, view product information, maintain user profiles, and submit orders. The system is designed to be scalable, extensible, and modular, enabling seamless communication with various microservices via gRPC.

2 System Architecture

The architecture of the SUSTech Merch Store is shown in Figure 1. The system is designed to integrate multiple components, each responsible for different tasks within the overall operation of the store.

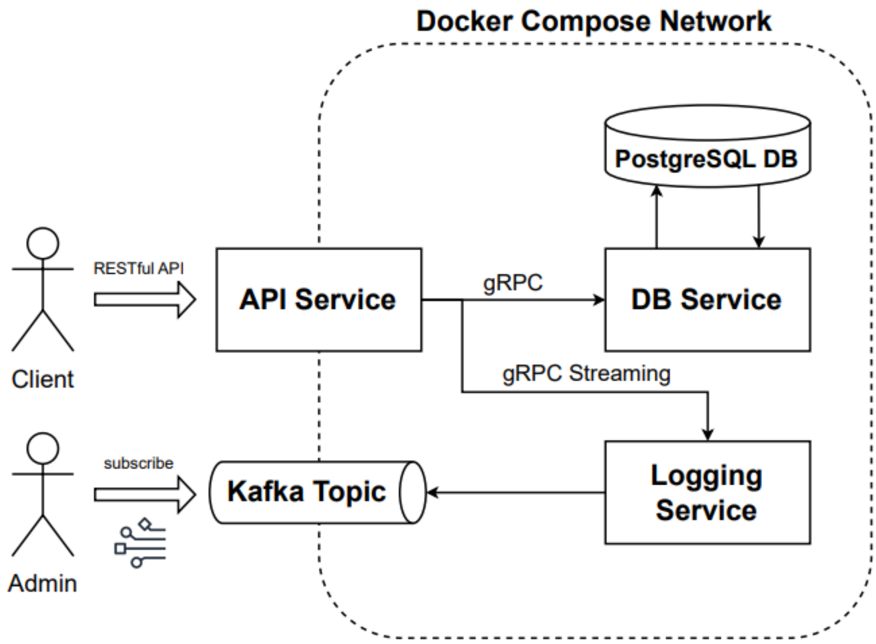


Figure 1: Architecture design of SUSTech Merch Store

- **RESTful API Service:** This is the main entry point for customers to interact with the system. It exposes several endpoints that allow customers to view products, manage their user profiles, and place orders. The API service communicates with other backend services to fetch product data, process user information, and manage orders.
- **DB service:** Receive request from REST API and interact with database and implement all business logic and return data back to REST API.
- **Logging Service:** Log messages from REST API and send those messages to kakfa consumer.
- **Authentication Feature:** use JWT(JSON WEB TOKEN) for usr authentication and authorization.
- **Deployment:** After testing on local, test services on docker environment.

3 Implementation

3.1 Q1. The procedures of your implementation for each component (set up the environment, generate code, and implement the business logic)

REST API SERVICE:

The REST API is implemented using Flask to handle various CRUD operations for products, users, and orders in a goods store system. This API interacts with two essential services:

- **gRPC DB Service**: This service handles interactions with the database, such as retrieving products, managing users, and processing orders.
- **Logging Service**: This service handles logging for various operations performed by the API, ensuring that all significant events are recorded for audit or debugging purposes.

Detailed Explanation of the REST API Implementation

A. Initial Setup

- **Flask App**: A Flask application is initialized to handle incoming HTTP requests.
- **gRPC Communication**: The Flask app communicates with the `DBService` (for CRUD operations) and `LoggingService` (for logging events) via gRPC.
  - The `DBService` is connected using `goods_store_pb2_grpc.DBServiceStub`.
  - The `LoggingService` is connected using `logging_service_pb2_grpc.LoggingServiceStub`.

B. Logging Functionality

- The `send_log()` function is responsible for sending log messages to the Logging Service.
- It constructs a log message that includes the log content, the service name (`"REST API"`), and the timestamp (in UTC format).
- The log message is sent to the Logging Service using a `StreamLogs` RPC call.
- If the log submission fails (due to an exception or a failed response), an error message is printed on the console.

DB Service

Database Connection Pool

The connection pool is initialized using the `psycopg2 SimpleConnectionPool` with the following configuration:

- `minconn = 1`, `maxconn = 10` (pool size)
- Database credentials stored in the `DB_CONFIG` dictionary.

Connections are fetched from and returned to the pool using

`connection_pool.getconn()` and `connection_pool.putconn()`.

JWT Authentication

The `jwt_required` decorator ensures that all gRPC methods are secured with JWT. It checks for a token in the `Authorization` header, decodes it using a `SECRET_KEY`, and adds the user details to the request context:

- `jwt.decode()` verifies the token.

- Invalid or expired tokens raise `Unauthorized` exceptions.

gRPC Service: `DBService`

The `DBService` class implements all operations with database: For example connect to database, get desired data from database through SQL query and do any operation such as create, update, or delete. The error handling ensures that any exception is caught, the status code is set to `INTERNAL`, and error details are provided.

Logging Service

receives log messages from clients and produces them to a Kafka topic named `log-channel`. Using the Confluent Kafka library, the service establishes a Kafka producer configured to connect to a Kafka broker at `kafka:9092`. When the `StreamLogs` method is called, it processes incoming log messages, formats them with a timestamp and service name, and sends them to Kafka. The service runs in a multi-threaded environment, allowing it to handle multiple log streams concurrently. It provides a simple response indicating the success or failure of the message production process.

3.2 Q2. APIs require authentication and the implement the authentication logic

All endpoints, except for the following, require authentication to access the resources:

- `/api/v1`
- `/api/v1/users/create`
- `/api/v1/users/login`

These endpoints are public and accessible to all users without the need for authentication. However, all other endpoints require users to be authenticated in order to access the resources.

The authentication process is implemented as follows:

1. **User Login:** When a user attempts to log in, they provide their credentials (username and password). The server validates the credentials and, if valid, generates a JSON Web Token (JWT).
2. **JWT Token Generation:** Upon successful login, the server generates a JWT token using a secret key and a specific algorithm (`HS256`).
3. **Token Usage:** After receiving the token, the user must include it in the `Authorization` header of subsequent requests. The token should be prefixed with `Bearer`:

   Authorization: Bearer <JWT_TOKEN>

4. **Token Validation:** For all endpoints that require authentication, the server verifies the validity of the token. If the token is valid (correctly signed, not expired), the server grants access to the requested resource. If the token is missing or invalid, an `Unauthorized` error is returned.
5. **Protected Endpoints:** All endpoints that require authentication are protected by a middleware or a decorator (`@jwt_required`) that ensures only requests with a valid JWT can access the resource. Any request without a valid token or with an expired token will be rejected.

3.3 Q3. SQL to Code Data Type Mapping

- **SQL: DECIMAL(10, 2)** → `double`
- **SQL: INT** → `int32`
- **SQL: VARCHAR(n)** → `String`
- **SQL: TEXT** → `String`

- **SQL: BOOLEAN** → `boolean`
- **SQL: TIMESTAMP** → `datetime`
- **SQL: SERIAL / INT PRIMARY KEY** → `int32`
- **SQL: REFERENCES (Foreign Keys)** → `int32`

3.4 Q4. select an arbitrary Proto message from your definition and analyze how it is encoded into binary format. Use Protobuf to programmatically verify the encoding result

Let's analyze the encoding of the `UpdateUserRequest` message using Protocol Buffers (Protobuf). The message definition is as follows:

```
message UpdateUserRequest {
  string sid = 1;
  string username = 2;
}
```

To manually calculate the Protobuf encoding, we follow these steps:

Field 1: `sid`

- Field tag number is `1`. - Wire type for strings is `2`. - The tag is calculated as:

$$\text{Tag} = 00001010 = 0x0a$$

- The string `sid` has the value `"12113053"` with a length of 8 bytes. The length is encoded as a varint:

$$\text{Length} = 8 = 0x08$$

- The UTF-8 string `"12113053"` is: `0x31 0x32 0x31 0x31 0x33 0x30 0x35 0x33`.

Thus, the encoding for the `sid` field is:

$$0\text{xoa } 0\text{x}08\ 0\text{x}31\ 0\text{x}32\ 0\text{x}31\ 0\text{x}31\ 0\text{x}33\ 0\text{x}30\ 0\text{x}35\ 0\text{x}33$$

Field 2: `username`

- Field tag number is `2`. - The tag is calculated as:

$$\text{Tag} = (2 << 3)|2 = 0x12$$

- The string `username` has the value `"sreyny"` with a length of 6 bytes. The length is encoded as a varint:

$$\text{Length} = 6 = 0x06$$

- The UTF-8 byte sequence for the string `"sreyny"` is: `0x73 0x72 0x65 0x79 0x6e 0x79`.

Thus, the encoding for the `username` field is:

$$0\text{x}12\ 0\text{x}06\ 0\text{x}73\ 0\text{x}72\ 0\text{x}65\ 0\text{x}79\ 0\text{x}6\text{e}\ 0\text{x}79$$

Therefore result in hexa is **0a083132313133303533120673726579 6e79**

Comparing the manually calculated encoding with the programmatically generated encoding:

```
C:\Users\Admin\AppData\Local\Programs\Python\Python311\python.exe D:\SUSTech\
Encoded binary data (hexadecimal): 0a0831323131333035331206737265796e79

Decoded UpdateUserRequest message:
SID: 12113053
Username: sreyny

Process finished with exit code 0
```

Figure 2: protobuf encoding result

we can see that both result are the same

3.5 Q5. Server-side streaming RPC process of logging service

The server-side streaming RPC in gRPC allows the server to process multiple client messages in a continuous stream, with the server sending multiple responses over the lifetime of a single RPC call. In the LoggingService, the StreamLogs method receives a stream of log messages, processes them, and produces each log to a Kafka topic. The service continues to process incoming logs and sends a final response indicating success or failure after all logs are handled. The server runs in a Docker container alongside Kafka and Zookeeper, configured using Docker Compose. This setup ensures the logging service can handle streams of log data efficiently while producing messages to Kafka for persistent storage.

3.6 Q6. Communiction between each service inside docker environment

API Service (`api-service`)

- **Build:** Built from the `./src/api_service` directory.
- **Environment Variables:** Configures Flask and connects to PostgreSQL using `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`.
- **Ports:** Exposes the API on port `8081`.
- **Depends_on:** Waits for PostgreSQL and Kafka services to be available before starting.
- **Network:** Connected to `my_network`.

DB Service (`db-service`)

- **Build:** Built from the `./src/db_service` directory.
- **Environment Variables:** Configures Flask and connects to PostgreSQL using `POSTGRES_USER`, `POSTGRES_PASSWORD`, and `POSTGRES_DB`.
- **Ports:** Exposes the DB service on port `50051`.
- **Depends_on:** Waits for PostgreSQL to be available before starting.
- **Network:** Connected to `my_network`.

Logging Service (`logging-service`)

- **Build:** Built from the `./src/logging_service` directory.
- **Environment Variables:** Configures Flask settings for logging, with port `50052`.
- **Ports:** Exposes the logging service on port `50052`.

- **Depends_on:** Waits for the API service, Kafka, and Kafka Topic Creator to be ready before starting.
- **Network:** Connected to `my_network`.

Network (`my_network`)

- All services are connected to the custom bridge network `my_network`, allowing seamless inter-container communication.
- Services like Kafka, API service, and Logging service communicate using container names within this network.

3.7 Q7. Testing

I use postman to test the REST API which is the client of db-service gRPC service. Here the process and result of my experiment:

Postman Environment and variable set up

I have setup environment which goods-store and one token variable to store token when user login. There are two environments, one if for local testing and another one is for docker testing. The purpose is just to avoid type the entire endpoint over and over by just using environment that all endpoints have in common. Token variable store jwt token user got after login, when user just use this variable with having copy and paste the token for every endpoint.

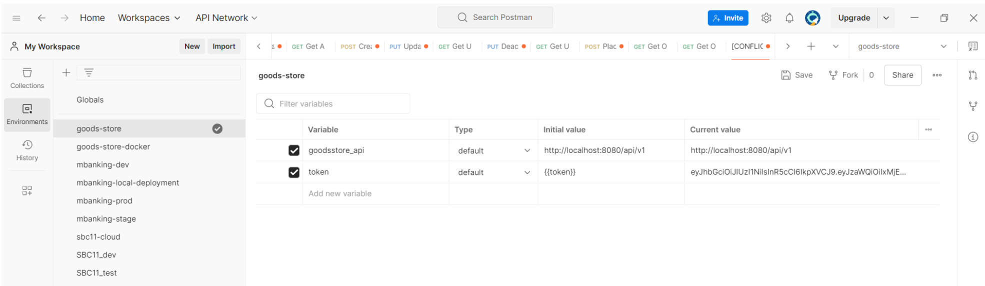


Figure 3: local environment

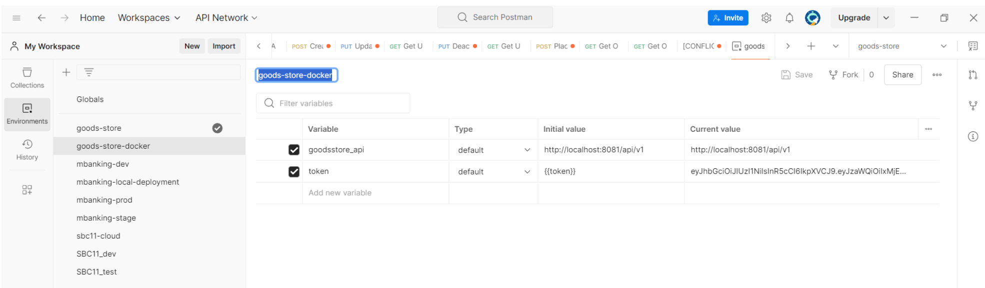


Figure 4: local environment

Testing Result

CRUD users

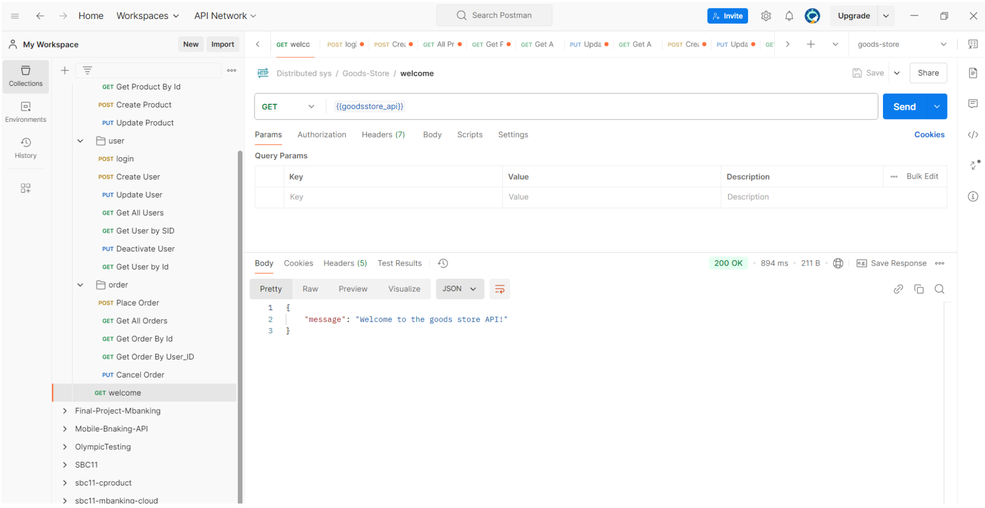


Figure 5: entry endpoint

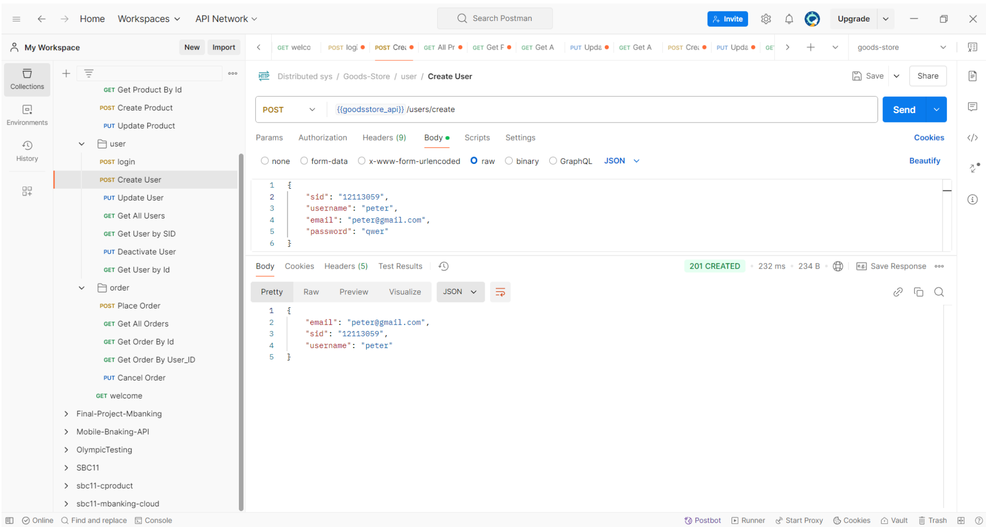


Figure 6: Create User

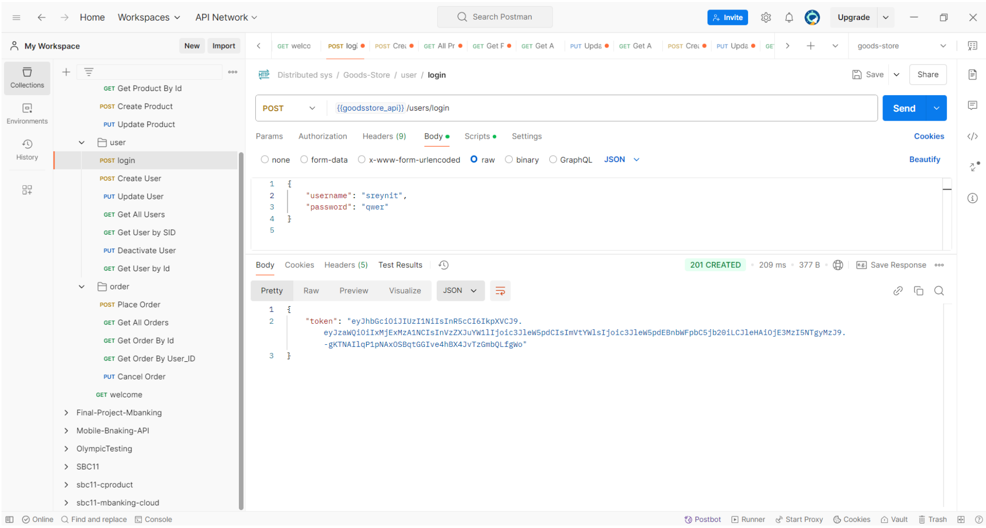


Figure 7: Login

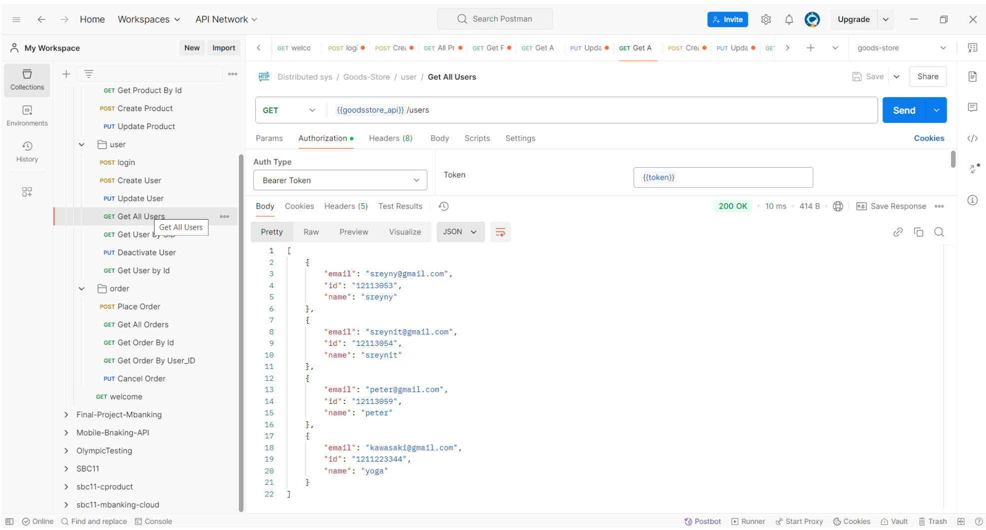


Figure 8: Get All Users

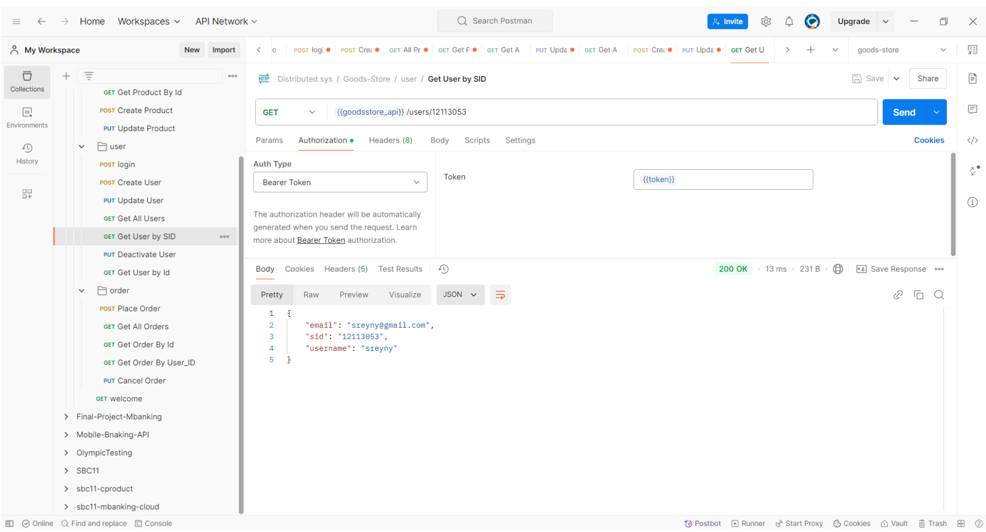


Figure 9: Get User by SID

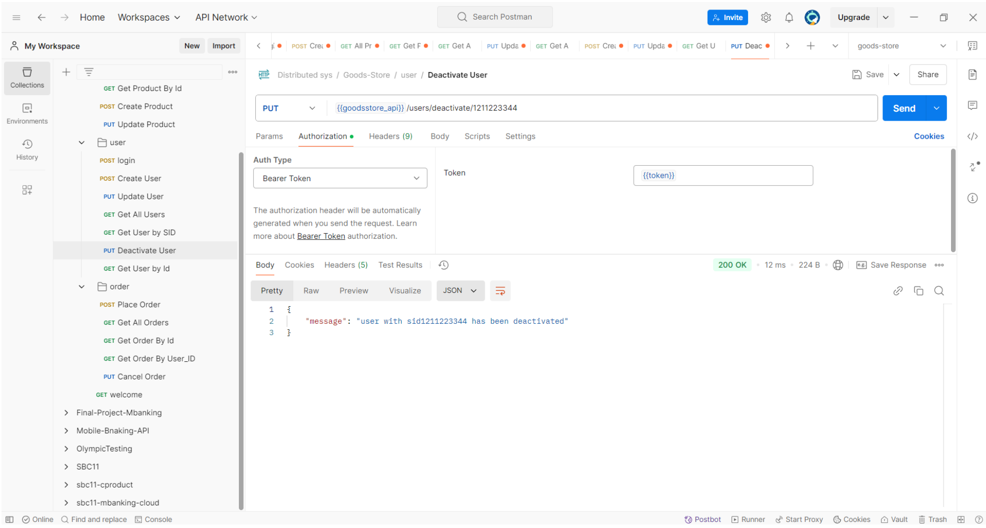


Figure 10: Deactivate User

CRUD Product

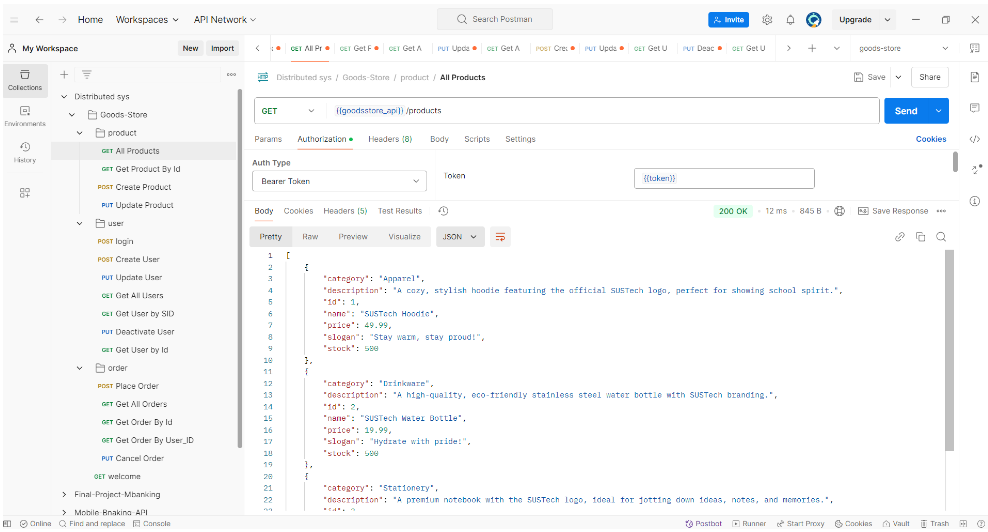


Figure 11: Get All Products

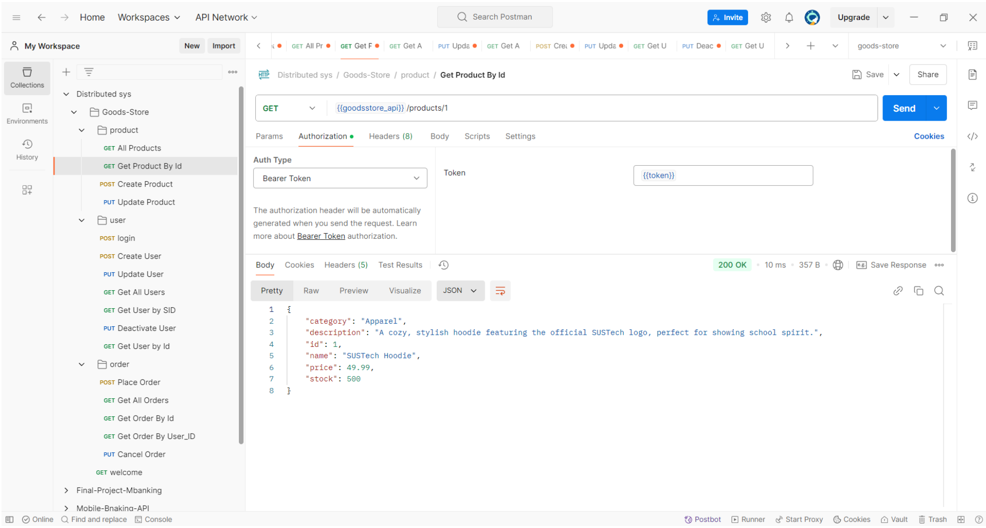


Figure 12: Get Product By Id

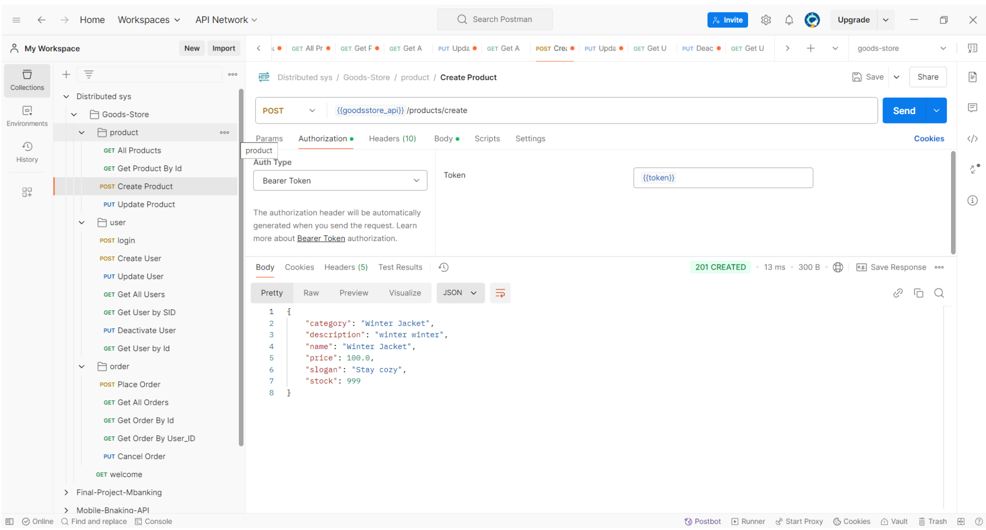


Figure 13: Create Product

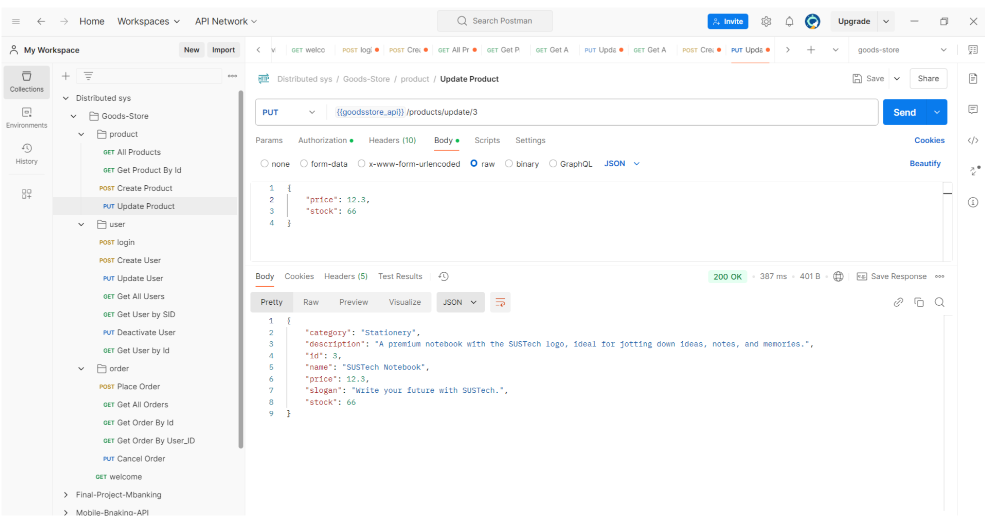


Figure 14: Update Product

CRUD Orders

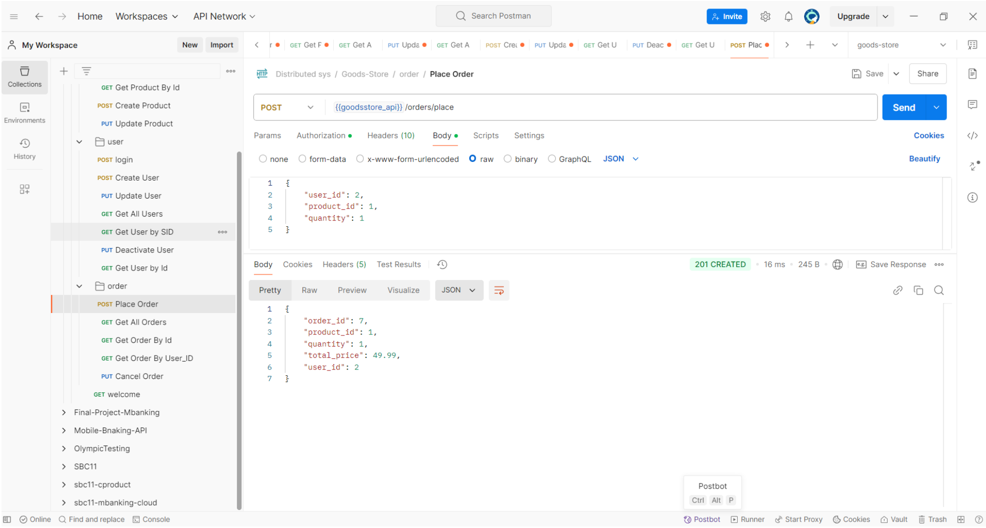


Figure 15: Place Order

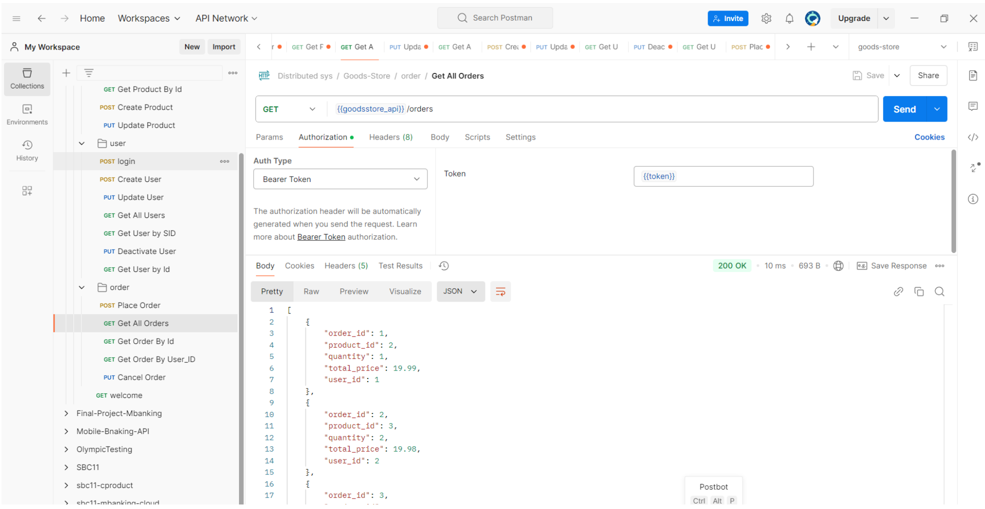


Figure 16: Get All Orders

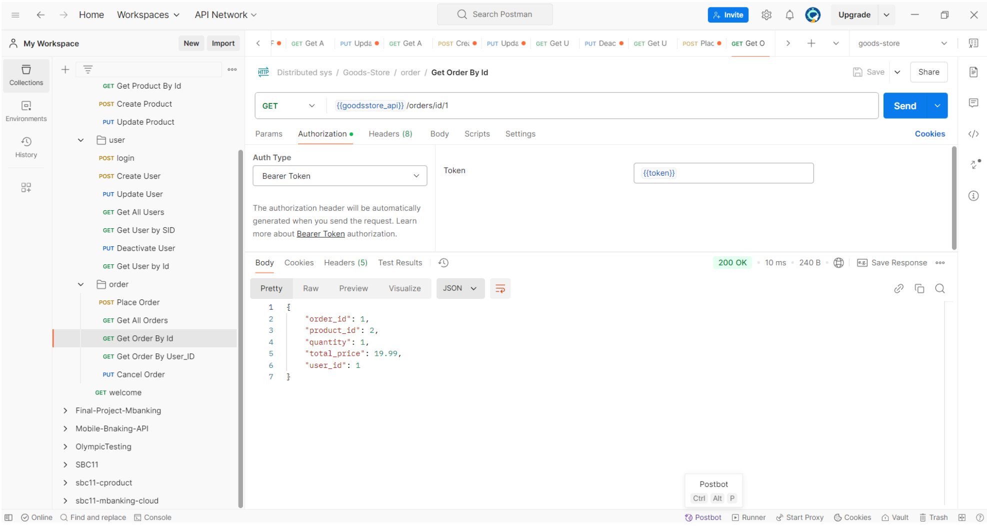


Figure 17: Ger Order by Id

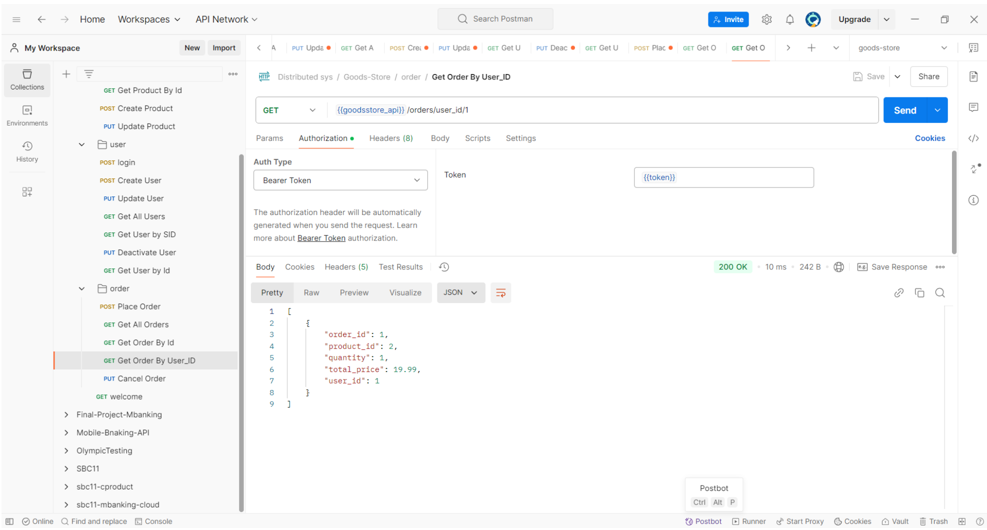


Figure 18: Get Order by User

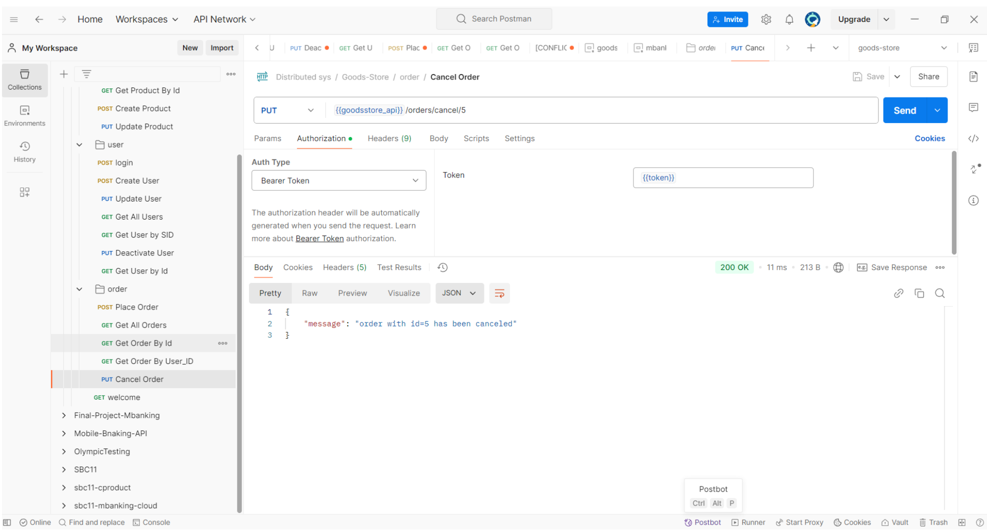
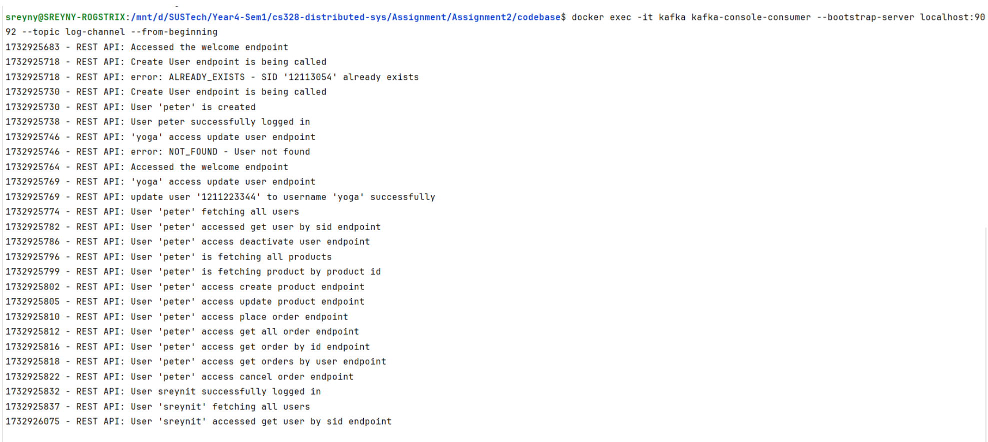


Figure 19: Cancel Order

monitoring the log messages from the Kafka topic



```
sreyny@SREYNY-ROGSTRIX:/mnt/d/SUSTech/Year4-Sem1/cs328-distributed-sys/Assignment/Assignment2/codebase$ docker exec -it kafka kafka-console-consumer --bootstrap-server localhost:90
92 --topic log-channel --from-beginning
1732925683 - REST API: Accessed the welcome endpoint
1732925718 - REST API: Create User endpoint is being called
1732925718 - REST API: error: ALREADY_EXISTS - SID '12113054' already exists
1732925730 - REST API: Create User endpoint is being called
1732925730 - REST API: User 'peter' is created
1732925738 - REST API: User peter successfully logged in
1732925746 - REST API: 'yoga' access update user endpoint
1732925746 - REST API: error: NOT_FOUND - User not found
1732925764 - REST API: Accessed the welcome endpoint
1732925769 - REST API: 'yoga' access update user endpoint
1732925769 - REST API: update user '1211223344' to username 'yoga' successfully
1732925774 - REST API: User 'peter' fetching all users
1732925782 - REST API: User 'peter' accessed get user by sid endpoint
1732925786 - REST API: User 'peter' access deactivate user endpoint
1732925796 - REST API: User 'peter' is fetching all products
1732925799 - REST API: User 'peter' is fetching product by product id
1732925802 - REST API: User 'peter' access create product endpoint
1732925805 - REST API: User 'peter' access update product endpoint
1732925810 - REST API: User 'peter' access place order endpoint
1732925812 - REST API: User 'peter' access get all order endpoint
1732925816 - REST API: User 'peter' access get order by id endpoint
1732925818 - REST API: User 'peter' access get orders by user endpoint
1732925822 - REST API: User 'peter' access cancel order endpoint
1732925832 - REST API: User sreynit successfully logged in
1732925837 - REST API: User 'sreynit' fetching all users
1732926075 - REST API: User 'sreynit' accessed get user by sid endpoint
```

Figure 20: Stream Log Messages

4 Deployment

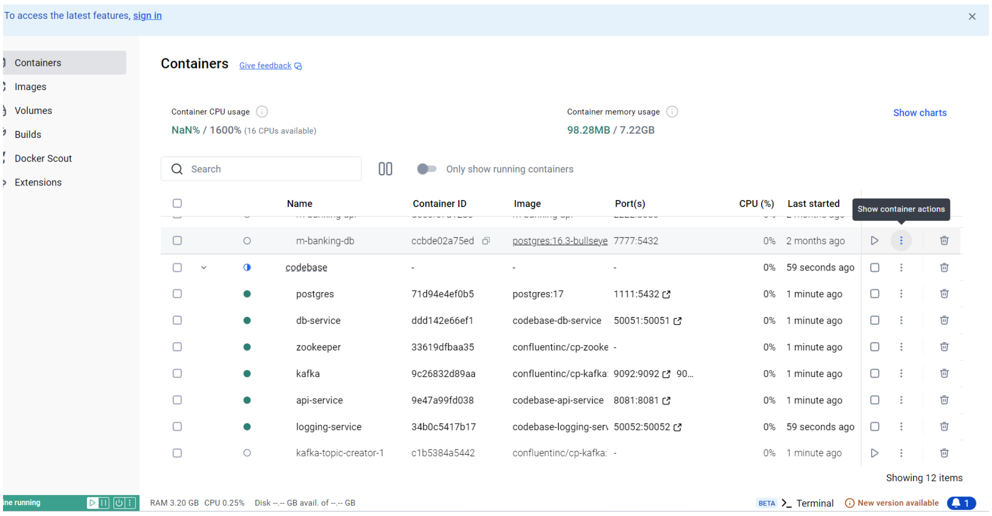


Figure 21: Docker Containers

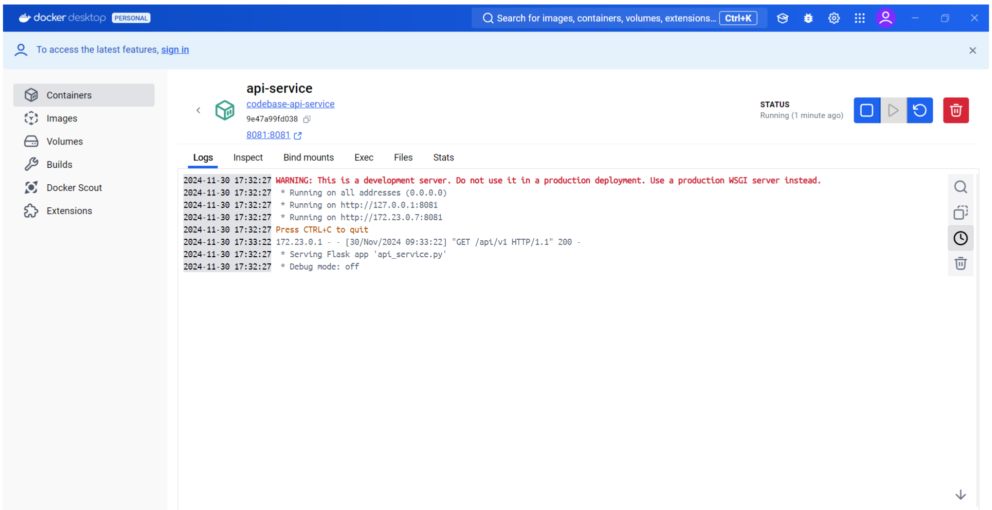


Figure 22: API Service Container

5 Summary

I have successfully implemented all necessary operations for managing **users**, **products**, and **orders** in the system. These operations include creating, updating, retrieving, and deleting users and products, as well as placing and managing orders. Additionally, I have integrated Kafka for log management and PostgreSQL for data storage.

After completing the implementation, I deployed all the services on **Docker** using Docker Compose. The services include `postgres`, `zookeeper`, `kafka`, `api-service`, `db-service`, and `logging-service`, all connected through a custom Docker network to enable seamless communication. The deployment was successfully tested, and all services are running as expected, ensuring a robust and scalable system for managing users, products, and orders.