COEN 6313 Assignment 2

1st Nishant Kumar Barua
40267821
barua.nishant97@gmail.com

***Abstract*—This assignment implements a microservices-based architecture for a user and order management system, utilizing MongoDB for data storage and Apache Kafka for event-driven synchronization. It consists of two main microservices: the User Microservice, which handles user account creation and updates, and the Order Microservice, which manages order creation and status updates. When a user's email or delivery address is updated, an event is triggered through Kafka to synchronize the user and order databases. The API gateway routes requests to different versions of the user microservice (v1 and v2) using the Strangler Pattern, enabling gradual migration between versions. The system is deployed using Docker on a cloud VM, with configurable endpoints for API access. Optionally, GitHub integration is used for Continuous Integration and Continuous Deployment (CI/CD), automating the deployment process on code commits. This solution demonstrates scalable microservices, event-driven architecture with Kafka, and modern cloud deployment practices.**

***Index Terms*—Microservice, Event-Driven Architecture, MongoDB, Message Queue, Apache Kafka, Java, SpringBoot, Strangler Pattern.**

I. Implementation Details

We will discuss the tasks under this section, with relevant diagrams and explanations.

A. Task 1: Creating Data Models

We have used MongoDB cloud to store our databases for the user and order data collections.

User Data Model We have the user data collection with fields:

- id
- firstname
- lastname
- emails
- delivery address

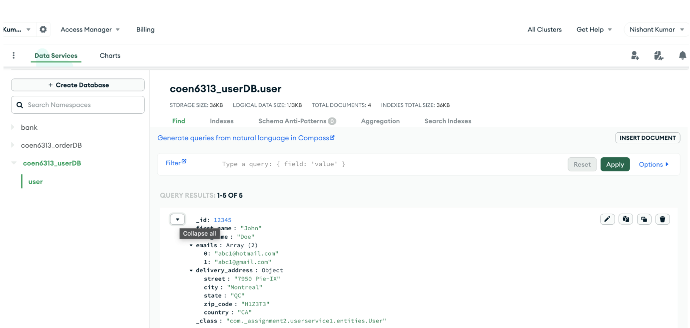

Fig. 1. User Data Model.

Order Data Model We have the order data collection. It has the unique orderId field. It takes the userEmails (list object) and the deliveryAddress object are of same format as the user Data model and will be used to later synchronise across the two databases. The complete list of fields are as follows:

- id
- userId
- userEmails
- DeliveryAddress
- orderStatus
- itemList

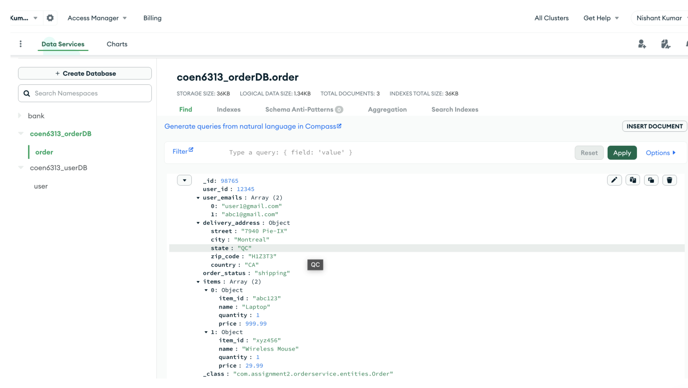

Fig. 2. Order Data Model.

B. Task 2: Creating The Microservices

we have designed four microservices here. For the development, we have used SpringBoot to expose REST APIs for the instructed operations as detailed below.

- **User Service V1:** This microservice interacts with the User Data Model in MongoDB, using the User Repository class, which extends the Mongo Repository for this. It has endpoints to get user based on id (GET), create user (POST), update address (PUT) and update email (PUT), as exposed in the User Controller class. It operates on Port 8080. The source code is folder userserviceV1.
- **Order Service:** The order microservice interacts with the MongoDB order data base collection. It has endpoints to fetch orders by shipping status (GET), create new order(POST, update order status (PUT), delivery address (PUT) and email address (PUT). The update email address and delivery address services will be leveraged by the messaging queue to synchronise between the user and

order databases. It operates on Port 8081. The source code is folder orderservice.

- **API Gateway:** This is a SpringBoot cloud gateway which routes HTTP requests to user and order microservices. It operates on Port 80 and is the only API endpoint exposed externally for interaction with client requests. The microservice endpoints are not directly exposed for client interaction and can only be accessed by the cloud gateway. The source code is folder apigateway.

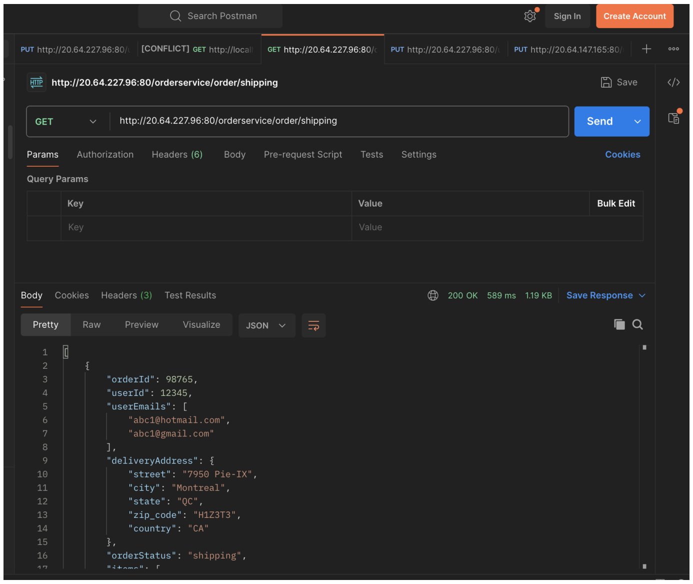

Fig. 3. Get Order EndPoint.

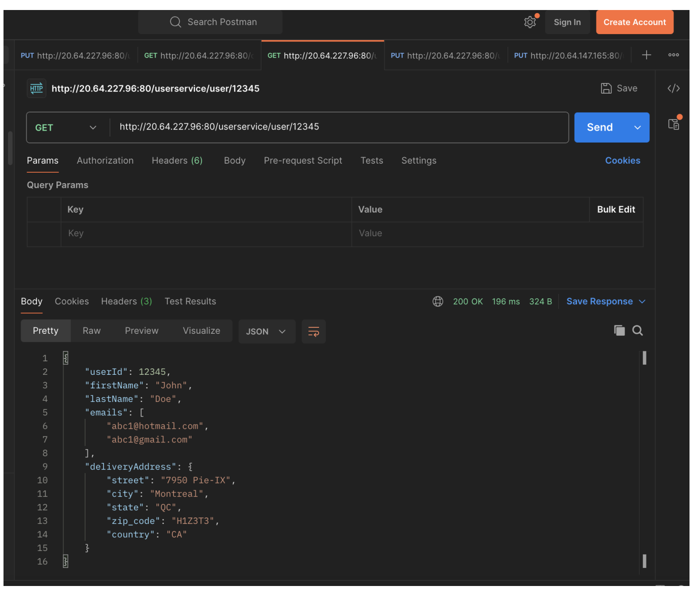

Fig. 4. Get User EndPoint.

C. Task 3: Event Driven Architecture

We have leveraged Apache Kafka for this implementation. The Apache Kafka setup consists of a Kafka broker (Port

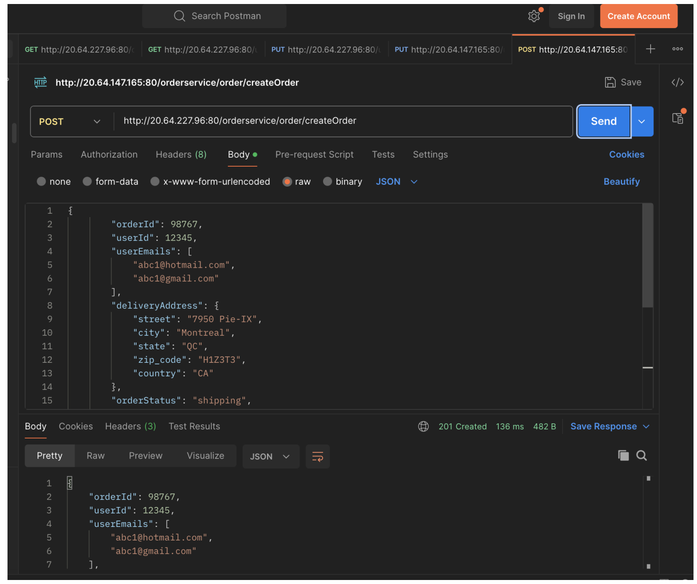

Fig. 5. Create Order EndPoint.

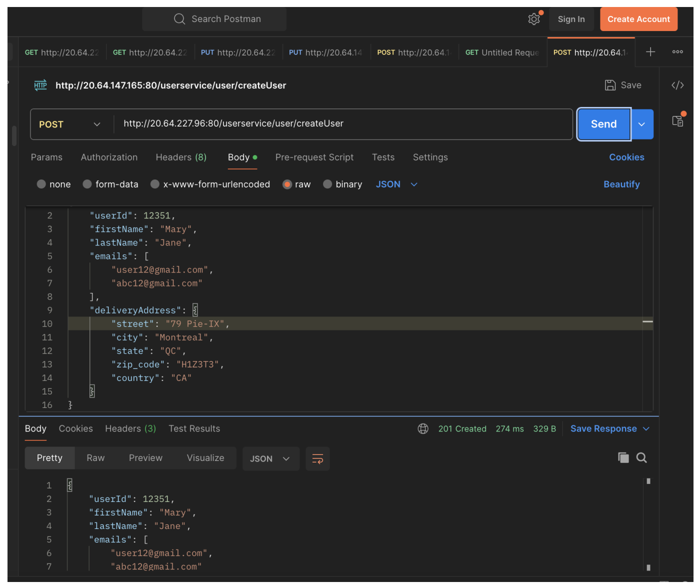

Fig. 6. Create User EndPoint.

9092) and a Zookeeper instance (Port 2181) for cluster coordination. Two Kafka topics (queues) are created: one for address updates and another for email updates. The steps are detailed below:

- The Producer microservice (both UserService V1 and V2) publish events to the respective topics when user details (address or emails) are updated via PUT requests.
- The Consumer microservice (OrderService) listen to these topics, processing the updates to synchronize the order database with the user database.

Kafka ensures reliable message delivery and event-driven communication between services, enabling decoupled data synchronization across systems.

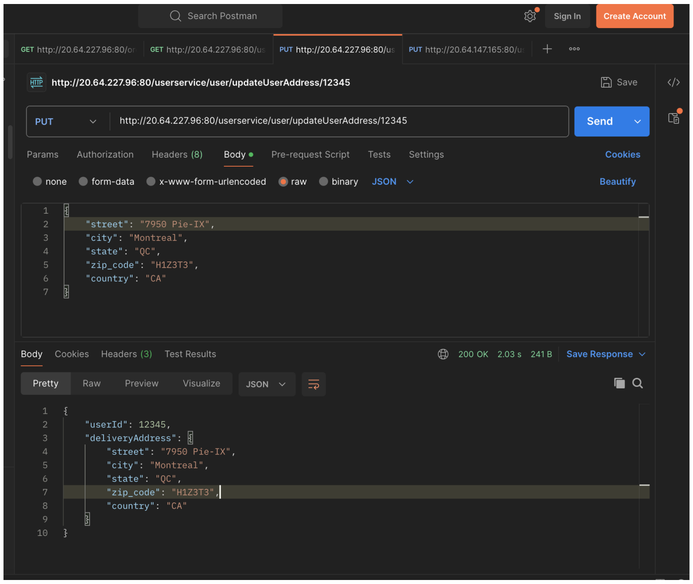

Fig. 7. Update User Address EndPoint.

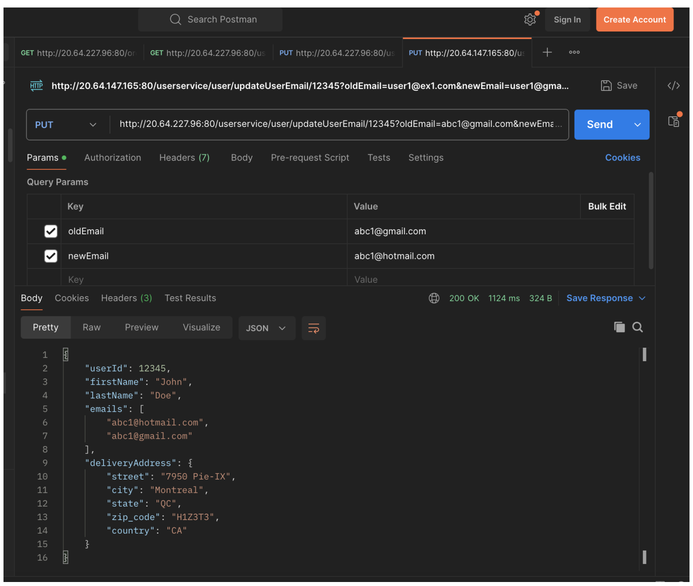

Fig. 8. Update User Email EndPoint.

D. Task 4: User Microservice V2

This microservice is also similar to v1, the change refactored here is the nature of returned response to the update user address and update email address API requests. The service V1 gives entire user object as the response, while the service V2 gives a modifies data object, with only the user id and updated address. Thus defining a new version of the existing V1 API.It operates on Port 8082. The source code is folder userserviceV2.

E. Task 5: Strangler Pattern

The Strangler Pattern involves gradually replacing an old system with a new one by routing a percentage of traffic to

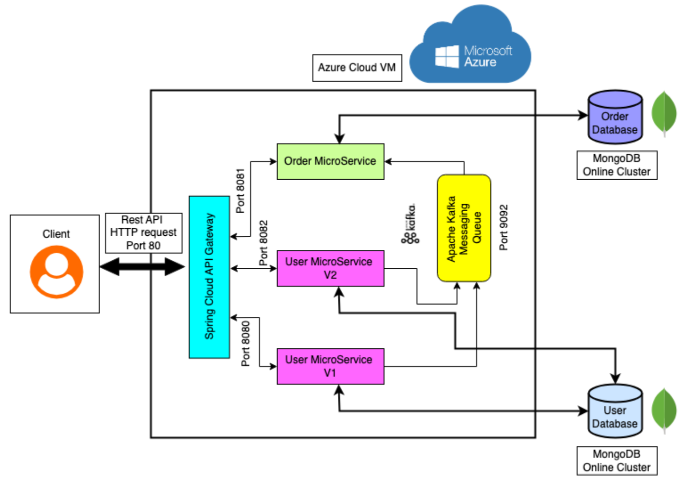

Fig. 9. Architecture Diagram (Task 2,3,4).

the new version while the old version remains operational. In this project, the API gateway routes requests to UserService v1 and v2 based on a configurable percentage, enabling smooth migration. The Strangler Pattern is implemented in the API Gateway and configured into the application properties file of the Spring Cloud Gateway. Screenshot below for reference.

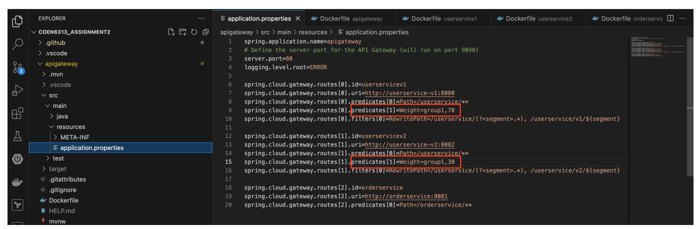

Fig. 10. Strangler Pattern on Gateway.

F. Task 6: MongoDB and Apache Kafka

We use MongoDB online cluster for our data base solution and Apache Kafka for messaging system.

G. Task 7: Docker Deployment

Each microservice, the api gateway, the Kafka Broker and Zookeeper, have been deployed as individual deployments using Docker containers. The configuration is available in the docker compose file, which accesses the individual docker file for each microservice to build up. We additionally create a Docker network and add all our containers to it so they can communicate seamlessly with each other.

H. Task 8: Cloud Deployment

We have used a Azure VM to deploy our solution into the cloud. We launch the Ubuntu container, with size Standard B2s (2 vcpus, 4 GiB memory), as Apache Kafka needs more memory to run. Now, we can access our endpoints on the cloud, which are enabled by the api gateway, running on Port 80. Our Azure VM cloud instance is 20.64.227.96

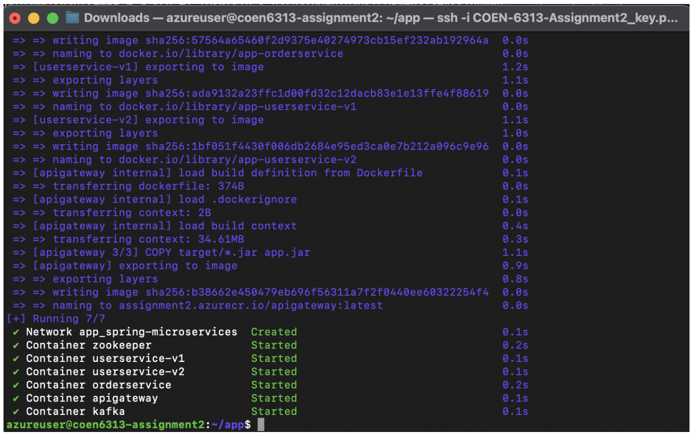

Fig. 11. Azure Cloud VM Deployment.

I. Task 9: CI/CD Deployment

We have used GitHub Actions to automate the CI/CD deployment on the cloud. Every time, an update is pushed to the main branch, it triggers the CI/CD pipeline to rebuild the jars and deploy the fresh instance of the app. Thus achieving end to end automation of the CI/CD pipeline. We have followed the following steps:

- First step is to build the Jars for each microservice and api gateway using maven build context.
- The, ssh into the Azure VM, build the docker containers with the Jar files.
- Final step is to deploy the containers, including Apache Kafka and Zookeeper, create the messaging queues (via Docker compose file) and run the containers.

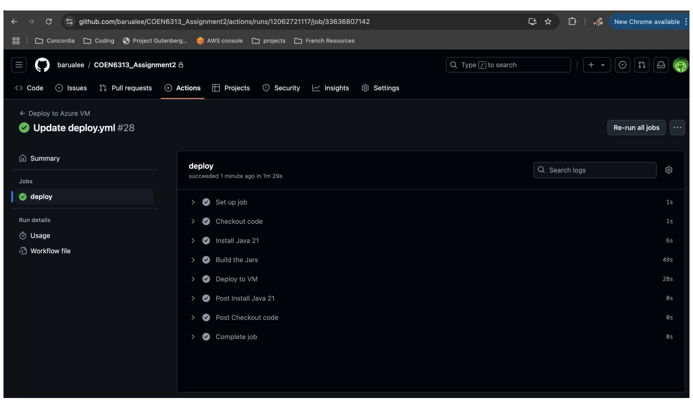

Fig. 12. CI/CD Pipeline via GitHub Actions.

References

[1] Azure Documentation.
[2] MongoDB Documentation.
[3] SpringBoot documentation.
[4] Docker Documentation.
[5] GitHub Actions Documentation.

Name (SID) and Signature	Task List		Contribution Role and Percentage (X %) (If Y for the task list, write the contribution role and percentage counted for the completeness of this task)
40267821	Task 1	Y	100%
	Task 2	Y	100%
	Task 3	Y	100%
	Task 4	Y	100%
	Task 5	Y	100%
	Task 6	Y	100%
	Task 7	Y	100%
	Task 8	Y	100%
	Task 9	Y	100%

Fig. 13. Contribution.