Work Integrated Learning Programmes

M.Tech Software Engineering

Scalable Services – S1-24_SEZG583

Assignment

Book Management Microservices Application

Submitted By,

Saurabh Vashishat-2023TM93654

Nivash-2023TM93523

Pooja Raj-2023TM93573

SivaShankari-2023TM93585

Introduction :

The Book Management microservices application is developed using Springboot and comprise 4 distinct services –

- User Service (Saurabh Vashishat)
- Book Service (Nivash)
- Transaction Service (Pooja)
- Review Service. (Sivashankari)

We hosted our backend services individually on localhost using Spring Boot, uploaded our codes to git repo shared among us and cloned the individual repo's agreed on consistent configuration for ports, database connections, and dependencies to avoid conflicts

Used shared configurations (e.g., .properties or .yaml files) for common settings

Used **RabbitMQ** to services communication asynchronously.

Each microservice operates with its own dedicated database, adhering to the "Database per Service" pattern within the microservices architectural approach. This design ensures data independence and enhances the autonomy of each service by providing a database schema tailored specifically to its functionality.

It is worth mentioning that the development of this application was carried out individually due to challenges in group collaboration. Despite this, the resulting application demonstrates a strong grasp of microservices principles and their practical implementation.

Explanation Video

The explanation video for the application is available in the google drive line below.

https://drive.google.com/file/d/1asBnywO-L6enkKMRcyaARlVyGslSWvIK/view?usp=sharing

Book Management Application

The Book Management application is designed to simplify and streamline the management of books, authors, and users. The application involves various independent services making it the most apt choice for building a microservices based application.

Let's explore the services included in the Book Management application and how they are integrated.

Prerequisites

Before starting the development of the Book exchange microservices application, it is required to set up the necessary tools and dependencies. The application was built on a Windows machine.

Below are the key tools and installations required for the development environment:

1. Visual Studio Code

Visual Studio Code (VS Code) serves as the integrated development environment (IDE) for building and managing the application code.

Downloaded and installed VS Code from https://code.visualstudio.com/.

2. IntelliJ IDEA

IntelliJ IDEA is a powerful Integrated Development Environment (IDE) used for developing Java applications

Downloaded and installed **IntelliJ IDEA** from IntelliJ IDEA download page.

3. Eclipse IDE

Eclipse IDE serves as the integrated development environment (IDE) for building and managing Java application code.

Downloaded and installed **Eclipse** from **https://www.eclipse.org/**

4. MongoDB

MongoDB is a popular, open-source, NoSQL database designed for modern application development

Downloaded and installed MongoDb from https://www.mongodb.com/try/download/community

Ensured that the MongoDB service is running after installation.

5. MySQLDB

MySQLDB is a popular relational database management system (RDBMS), widely used for web applications and data storage.

Downloaded and installed MySQLDB from MySQL Community Downloads page.

Ensured that the MySQLDB service is running after installation.

6. Maven

Maven is a build automation tool primarily used for Java projects. It simplifies the build process by managing dependencies, compiling code, running tests, packaging the application, and deploying it.

Downloaded and installed Maven from Apache Maven Download

7. Postman Desktop

Postman is a powerful tool for testing APIs and streamlining the development workflow.

Downloaded and installed Postman Desktop from https://www.postman.com/downloads/.

Postman is very helpful for testing the functionalities of our microservices and ensuring smooth API interactions.

8. RabbitMQ

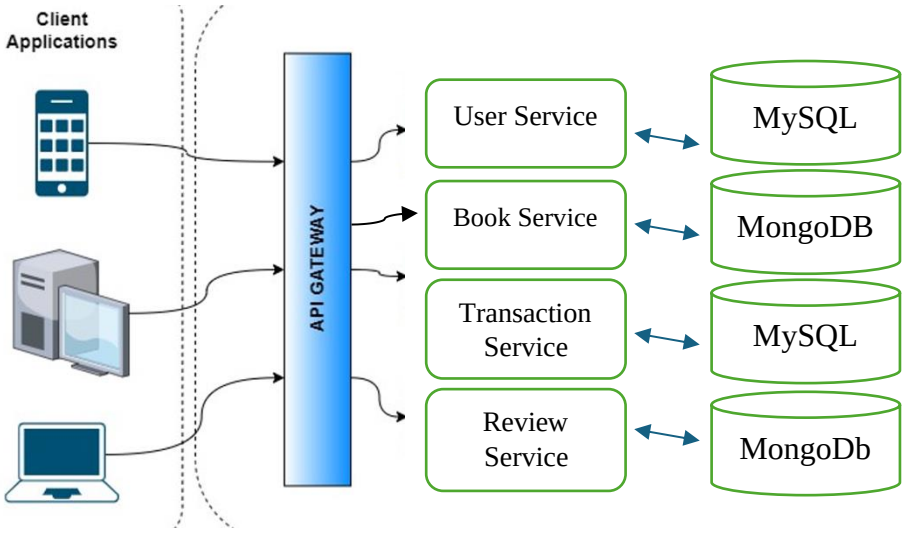
RabbitMQ is a robust messaging broker for handling communication between microservices and ensuring reliable message delivery.

Downloaded and installed RabbitMQ from https://www.rabbitmq.com/download.html.

RabbitMQ is essential for enabling asynchronous communication between services and improving the scalability and fault tolerance of our system.

Once these tools are successfully installed, all the setup required for the development of the book exchange microservices application is done.

System Architecture



Database Schema:

The database schema for our BookManagement microservices application is structured to align with the microservices architecture, particularly adhering to the "Database per Service" pattern. This approach ensures that each microservice has its own dedicated database, managing its data storage independently, rather than sharing a common database among multiple services

Key Benefits of Independent Databases in Microservices

1. **Decoupled Architecture**: Reduces interdependencies among microservices, enhancing flexibility and making it easier to modify or replace a service without affecting others.
2. **Autonomous Operations**: Each microservice has its own database, supporting independent development, deployment, and scaling.
3. **Resilient Failure Isolation**: Ensures that issues in one microservice or its database do not impact the functionality of others, improving overall system reliability.
4. **Granular Scalability**: Allows each microservice to scale independently, optimizing resource allocation based on specific performance demands.
5. **Tailored Security and Compliance**: Enables microservices dealing with sensitive data to implement customized security measures and comply with specific regulations at the database level.
6. **Technology Diversity**: Supports the use of different database technologies to meet the unique requirements of individual microservices.
7. **Data Integrity**: Ensures transactions remain confined to a single microservice's database, maintaining consistency and reliability within that service.
8. **Simplified Maintenance and Evolution**: Facilitates independent updates, such as schema changes, for each microservice's database, making maintenance and upgrades more manageable.

Microservices Communication

In our Book Management microservices architecture, the communication between the individual services is done through APIs, adhering to the **API Gateway Pattern**.

This pattern includes a centralized API Gateway application to manage and streamline communication between client applications and the diverse microservices within the system.

The API Gateway application acts as the main entry point for external clients, providing a unified interface to interact with the microservices ecosystem. It routes incoming requests to the appropriate microservice based on the requested functionality.

User Service [Saurabh Vashishat]

Overview

The User Microservice offers a range of endpoints to facilitate user account management, including user creation, authentication, and the management of user Details.

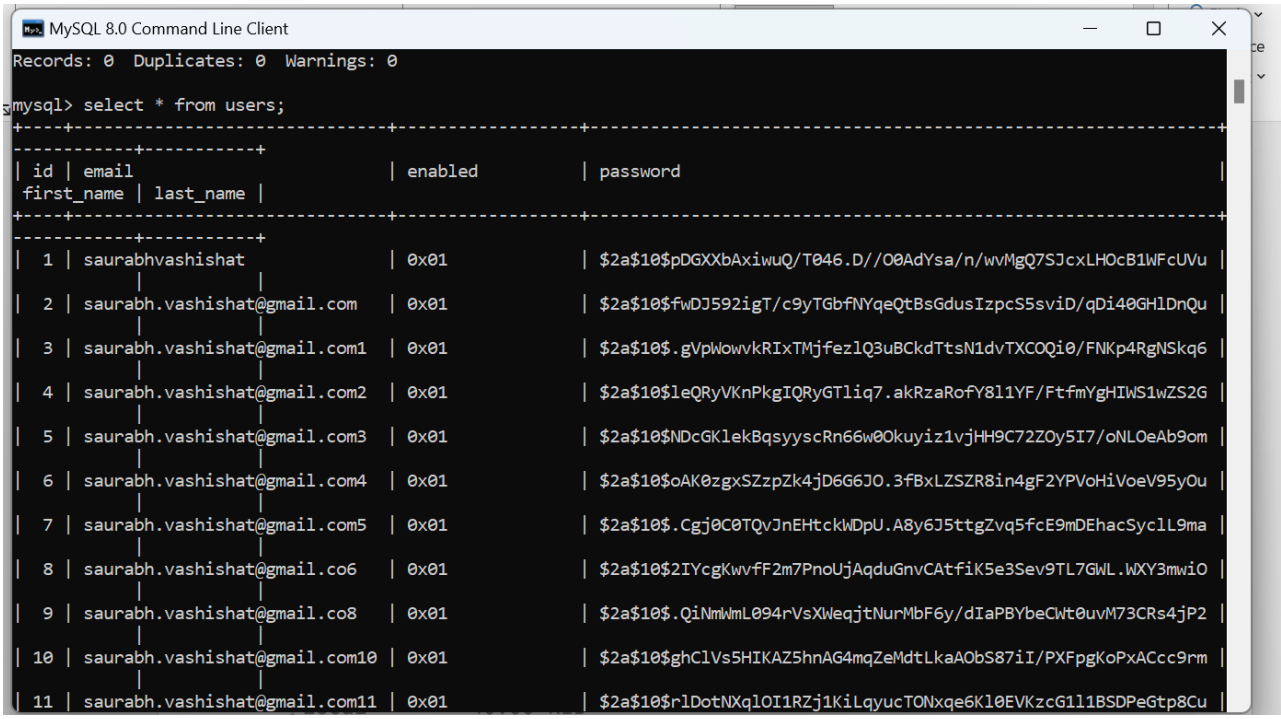
API Endpoints

User Account Endpoints

This group of endpoints provides functionalities to manage user accounts, including creation, authentication, retrieval and deletion. The "/validate" endpoint allows the validation of the JWT token generated post user login.
API Endpoints:

#	Endpoint	Method	Description
1	/api/users/register	POST	Creates a new user account.
2	/api/users	GET	Retrieves a list of all user accounts.
3	/api/auth/login	POST	Authenticates a user based on provided credentials.
4	/api/auth/validate	POST	Validate the JWT token is valid or not.
5	/api/users/reset-password	POST	Updates user details by ID.

Database :Mysql



Code Repository

The complete source code for the User Microservice is available on GitHub. You can access and explore the code repository at the following link:

https://github.com/saurabhvashishat89/SC-Library-Management-System-Auth

Review Service :[SivaShankari]

Product Review Service in a microservices architecture is a standalone component designed to handle all operations related to product reviews. It operates independently, allowing seamless integration and communication with other services while maintaining its own database for review-related data

API Endpoints:

- **GET** http://localhost:8080/books: Retrieve all reviews of all books.
- **GET** http://localhost:8080/books/1: Retrieve all reviews of all books for specific userid .
- **POST** http://localhost:8080/books: Add a new review for the book.
- **PUT** / http://localhost:8080/books: Update an existing review.
- **DELETE** http://localhost:8080/books/1 Delete a reviews of specific userid.

Database: MongoDb

```
test> use bookreview
switched to db bookreview
bookreview> show dbs
admin        40.00 KiB
bookreview   40.00 KiB
config      108.00 KiB
local        40.00 KiB
test         72.00 KiB
bookreview> show collections
books
bookreview> db.books.find().pretty()
[
  {
    _id: ObjectId('6739f812e48b424c4eb86c78'),
    title: 'Oracle_Database',
    author: 'BOB'
  }
]
```

Code Repository

The complete source code for the Product Microservice is available on GitHub. You can access and explore the code repository at the following link:

https://github.com/sivashankarisampathkumar/scalable_bookreview

Transaction Microservice [Pooja Raj]

Overview

The Transaction Microservice is a key component of our system responsible for handling all operations related to transactions, such as processing, storing, and managing transaction data. Here is an overview of its features and functionality:

Database Design

The transaction table in the book_exchange database is as below:

```
mysql> DESCRIBE transaction;
+----------+--------------+------+-----+---------+----------------+
| Field    | Type         | Null | Key | Default | Extra          |
+----------+--------------+------+-----+---------+----------------+
| id       | bigint       | NO   | PRI | NULL    | auto_increment |
| book_id  | bigint       | YES  |     | NULL    |                |
| receiver | varchar(255) | YES  |     | NULL    |                |
| sender   | varchar(255) | YES  |     | NULL    |                |
| status   | varchar(255) | YES  |     | NULL    |                |
+----------+--------------+------+-----+---------+----------------+
```

API Endpoints

Transaction EndPoints

This group of endpoints provides functionalities to manage the transactions, including creation, retrieval, updating, and deletion.

Endpoint	Method	Description	Request Body	Response
/transactions	POST	Creates a new transaction.	json { "bookId": 123, "sender": "user1@example.com", "receiver": "user2@example.com", "status": "PENDING" }	**201 Created**: Returns created transaction. **400 Bad Request**: Invalid input data.
/transactions	GET	Retrieves all transactions.	None	**200 OK**: Returns list of transactions
/transactions/{id}	GET	Retrieves details of a specific transaction by ID.	None	**200 OK**: Returns transaction details. **404 Not Found**: Transaction does not exist.
/transactions/{id}	PUT	Updates the status of a transaction by ID.	json { "status": "COMPLETED" }	**200 OK**: Returns updated transaction. **404 Not Found**: Transaction does not exist.
/transactions/{id}	DELETE	Deletes a transaction by ID.	None	**204 No Content**: Transaction deleted. **404 Not Found**: Transaction does not exist.

Code Repository

The complete source code for the Transaction Microservice is available on GitHub. You can access and explore the code repository at the following link:

https://github.com/PoojaRaj-2023tm93573/ScalableServices

Book services: [NIVASH]

The Book Service in a microservices architecture is a dedicated component responsible for managing all book-related operations. It is designed to function independently, facilitating seamless communication and integration with other services while maintaining its own database to ensure data isolation and scalability. The service provides a range of API endpoints to manage book data effectively. These include retrieving details of all books or books associated with a specific user ID, adding new book entries, updating existing book information, and deleting a book by its user ID

API Endpoints:

- **GET** http://localhost:8080/api/books: Retrieve all book details of all books.
- **GET** http://localhost:8080/api/books/{id}: Retrieve all book details from specific user-id.
- **POST** http://localhost:8080/api/books: Add new book details for the book.
- **PUT** / http://localhost:8080/api/books: Update an existing book details.
- **DELETE** http://localhost:8080/api/books/1 Delete a book of a specific user-id.

Database: MongoDB

Stores book data. The schema might include:

id: A unique identifier (ObjectId) for each document.

title: The title of the book (e.g., "book").

author: The author of the book .

genre: A genre identifier represented as a string .

description: A brief description of the book's content.

_class: A field indicating the Java class associated with this document in a Spring Boot application (e.g., "com.example.SpringBoot.model.Book").

CODE REPOSITORY:

The complete source code for the Book services is available on GitHub. You can access and explore the code repository at the following link:

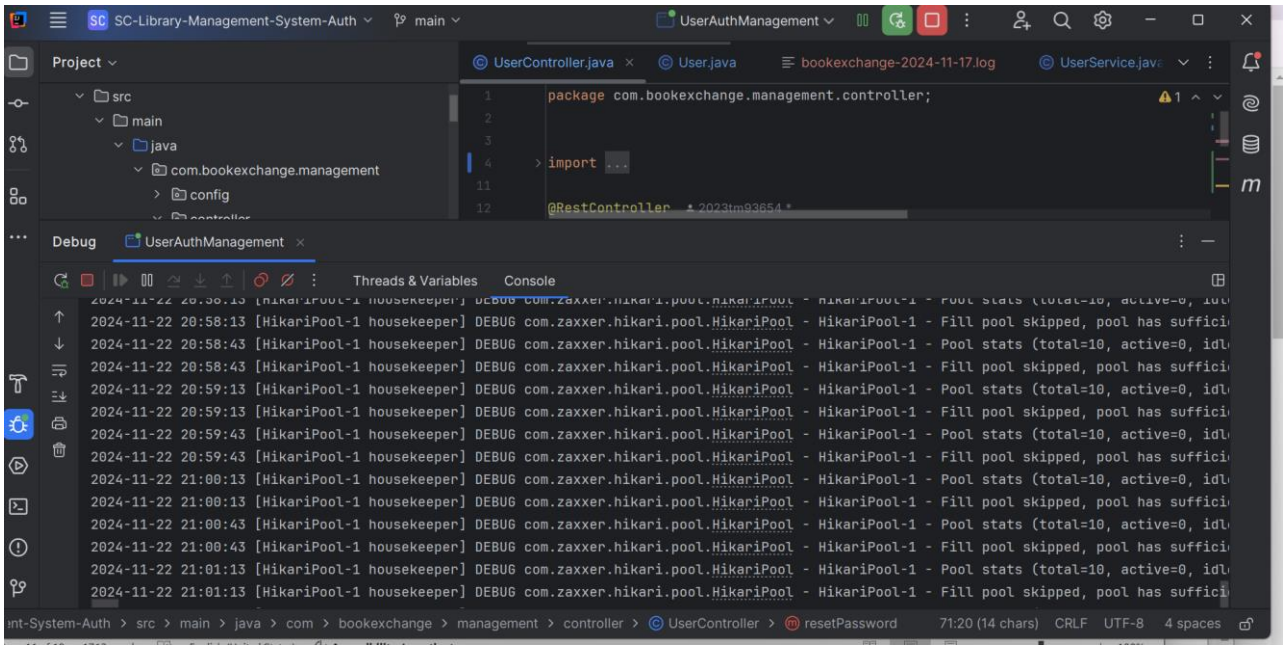
https://github.com/nivashsivagnanam/SpringBoot

Testing using Postman

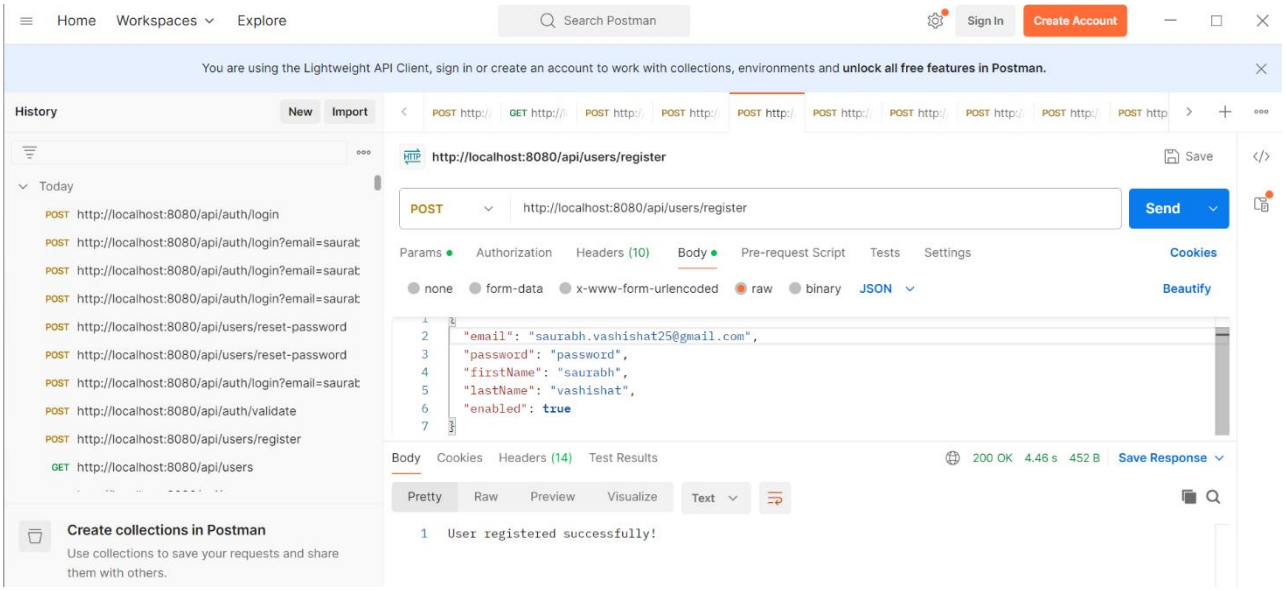
Postman provides a powerful and user-friendly platform for testing microservices. It helps in identifying potential issues early in the development lifecycle.

User Service: POSTMAN Test for User Microservice:

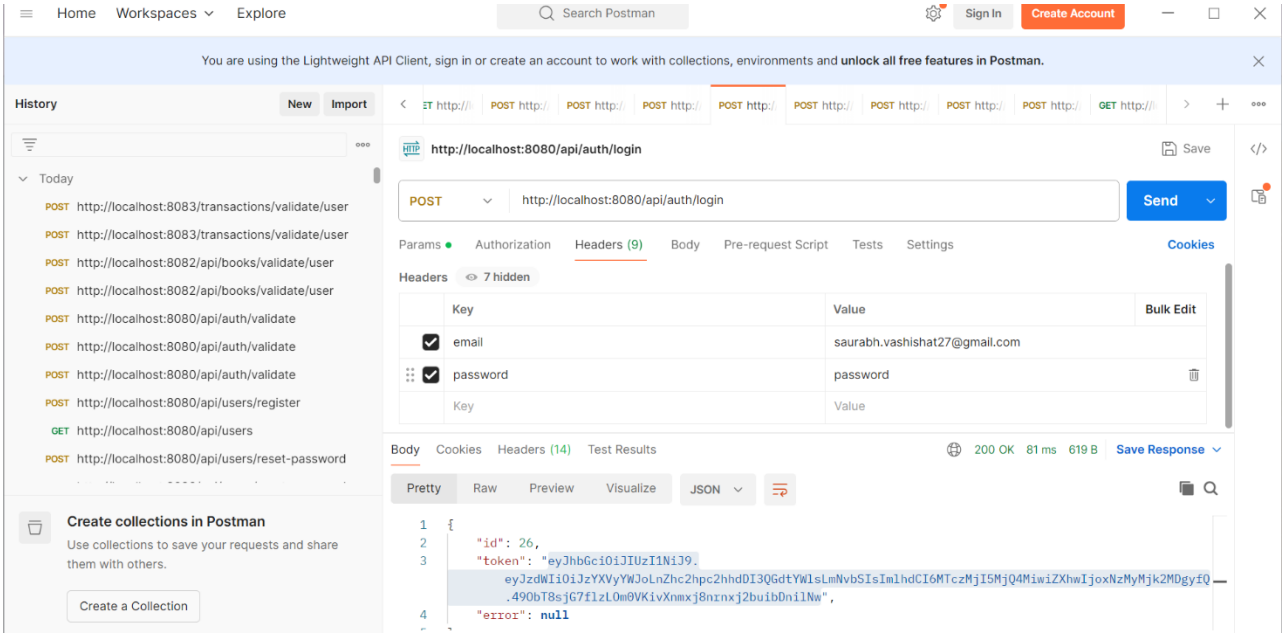
1. Run the transaction microservice:



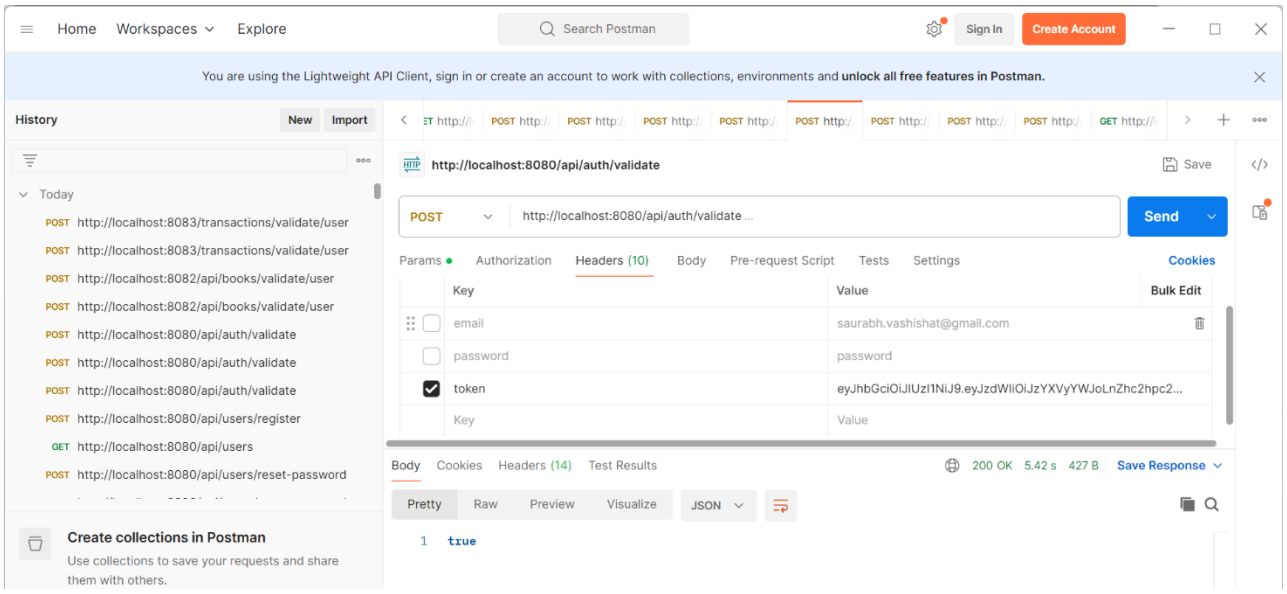
2. Register the user:



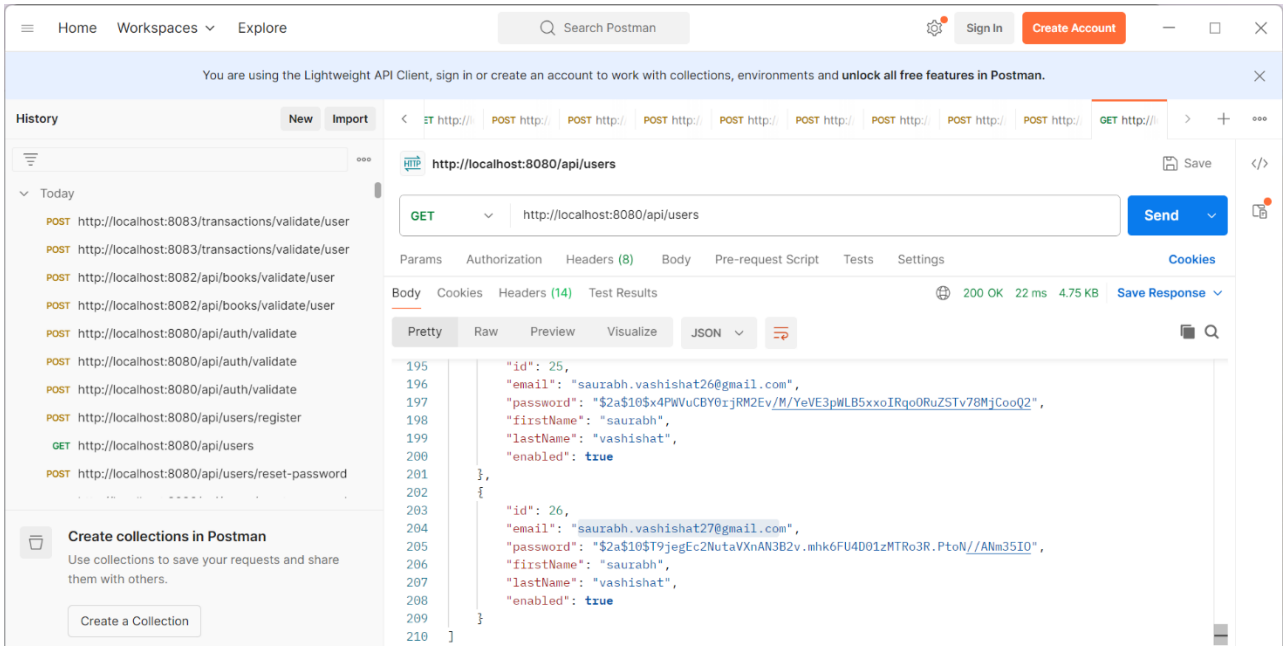
3. Login a user :



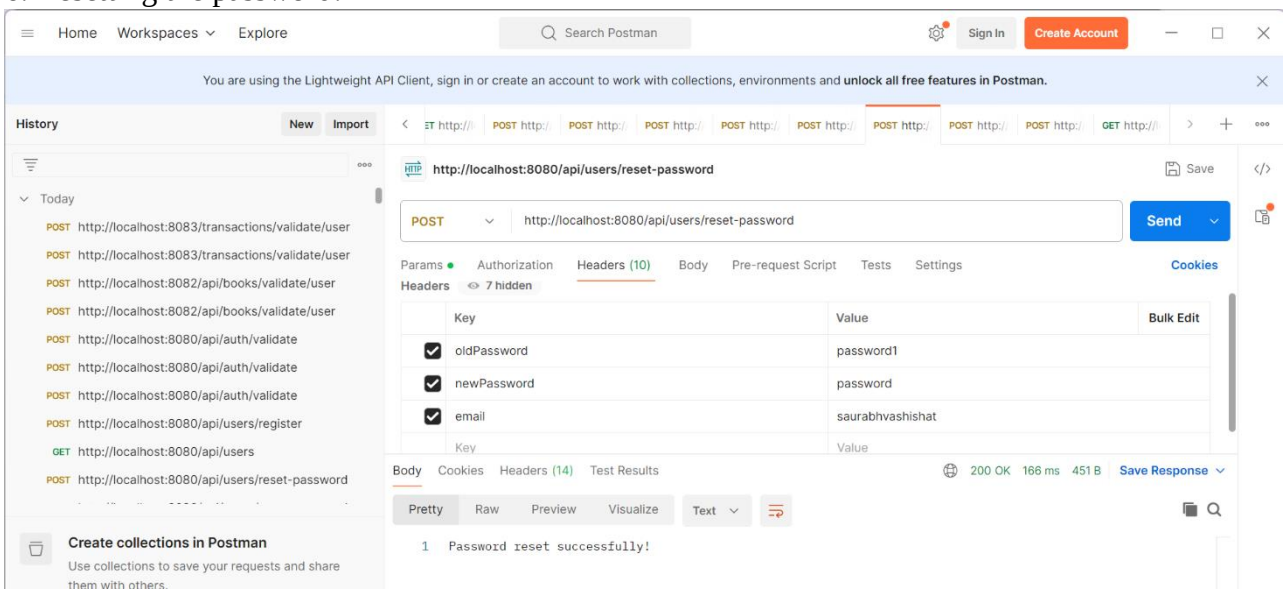
4. Validate JWT token :



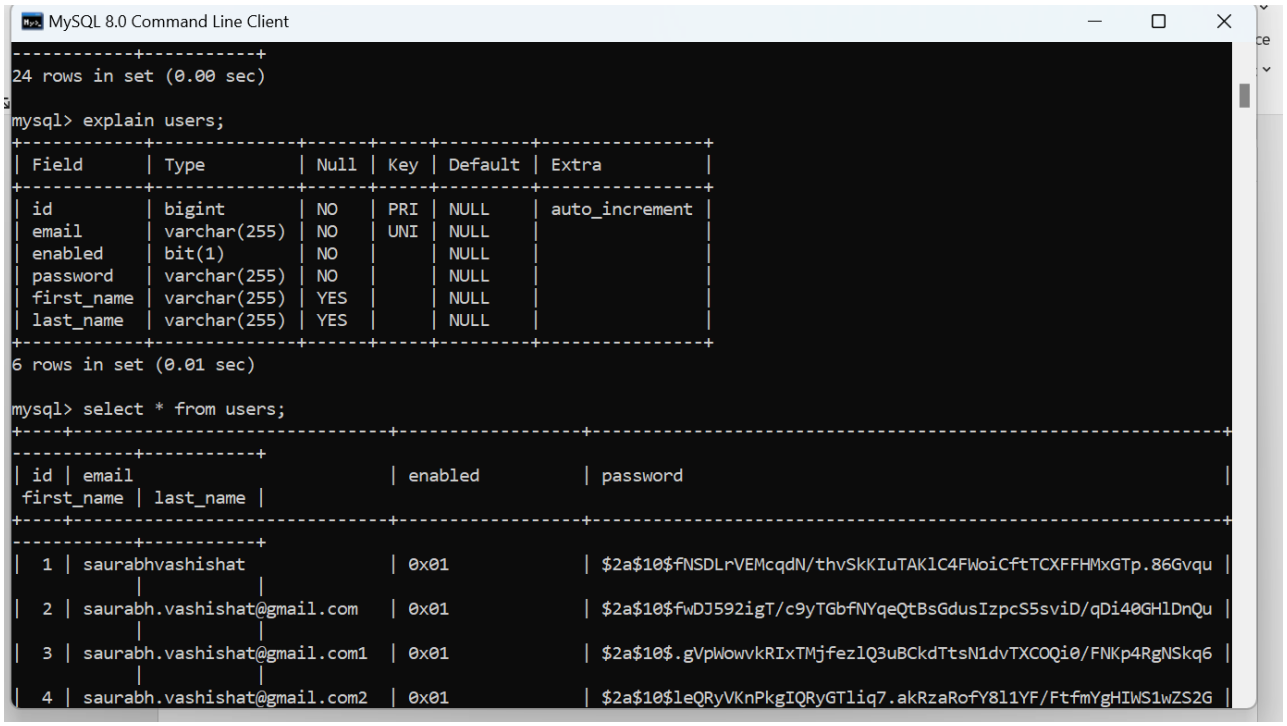
5. Get all User details :



6. Resetting the password:

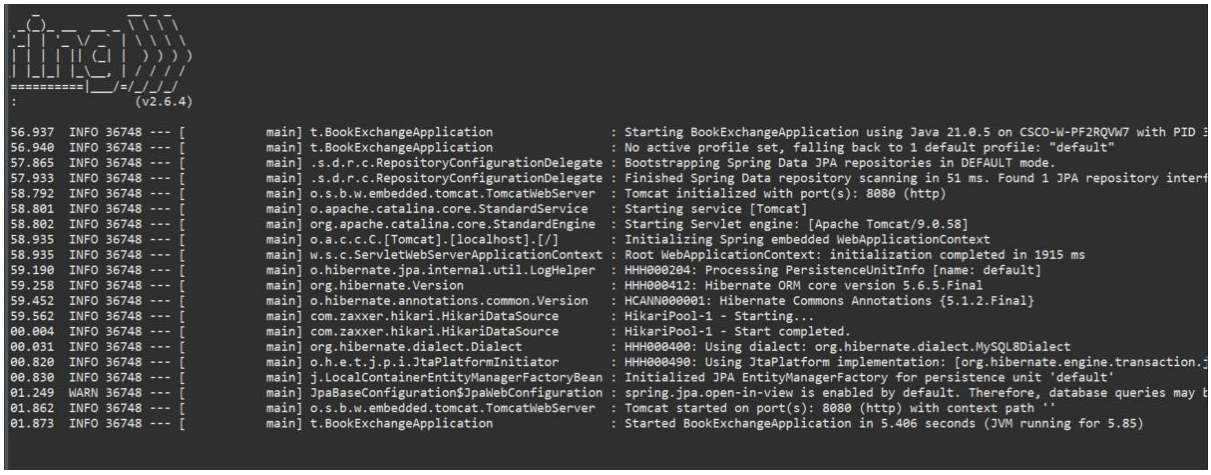


7. Database content:

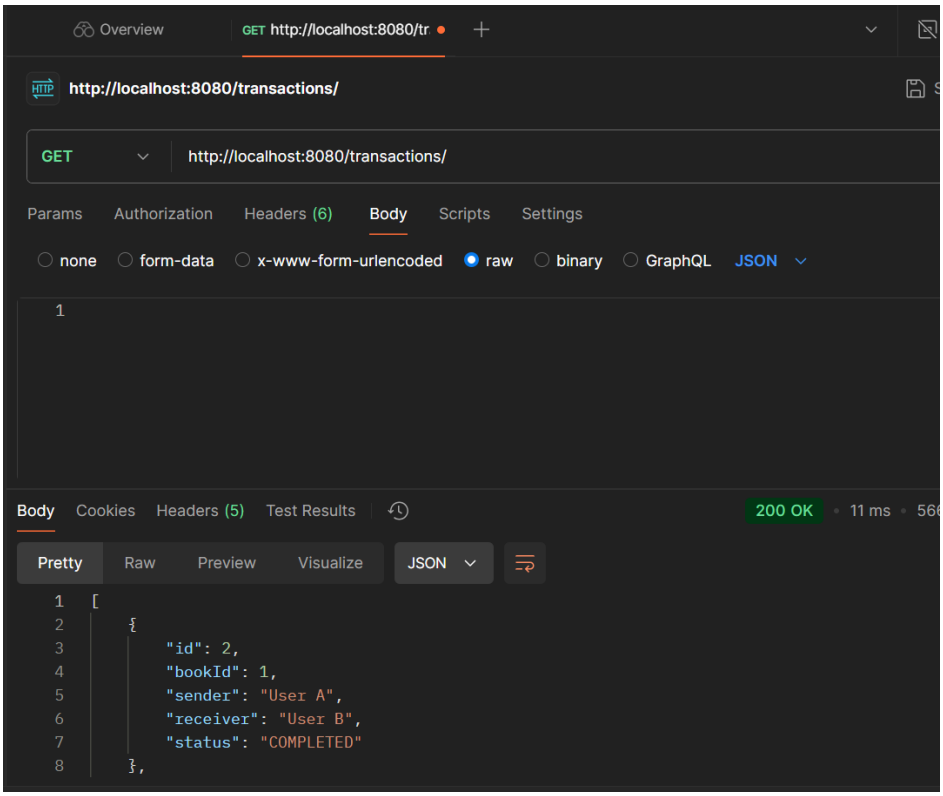


Transaction Service: POSTMAN Test for Transaction Microservice:

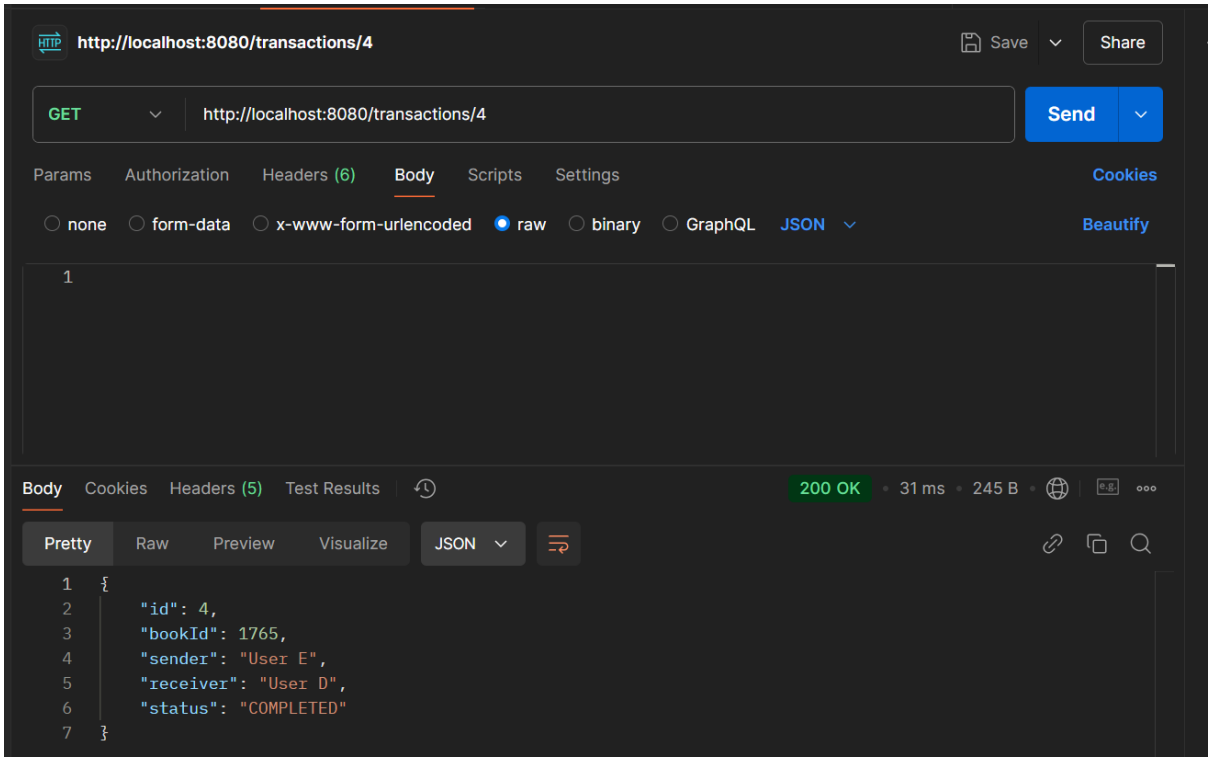
1.We run the transaction microservice:



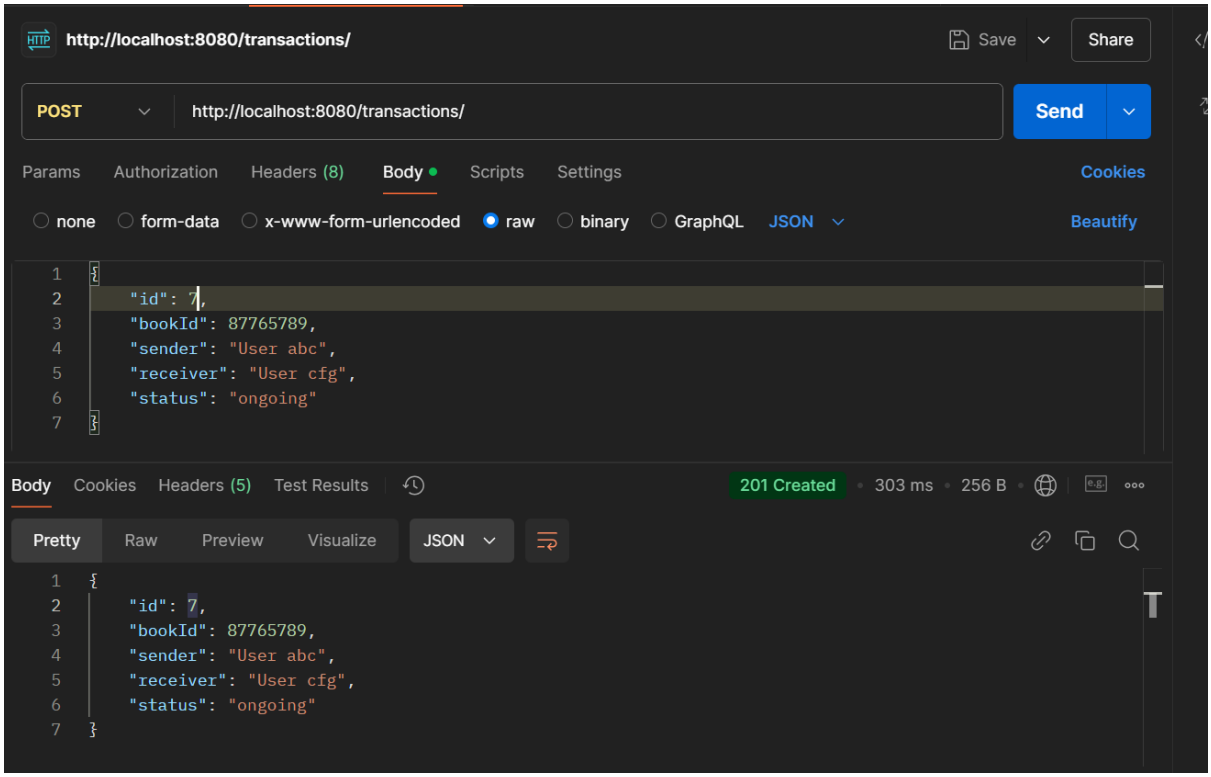
2. Get all transactions:



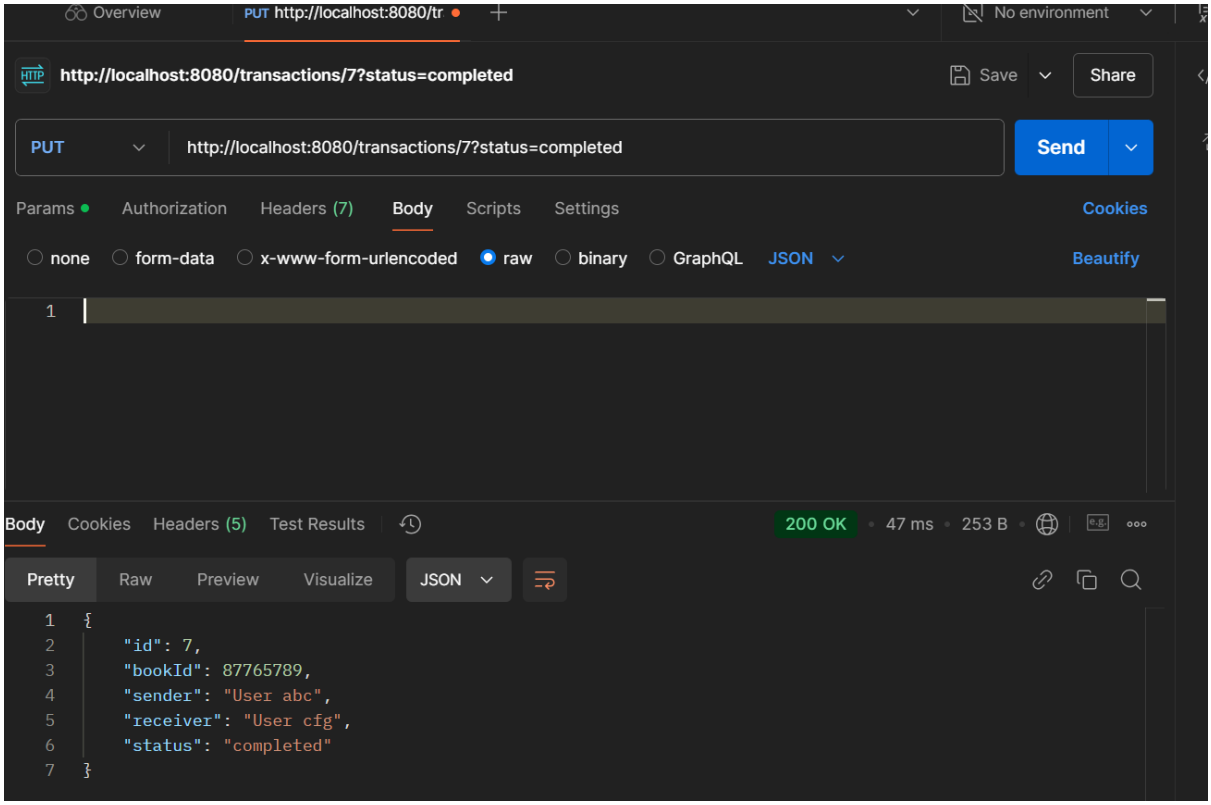
3. Get individual transactions identified by ID:



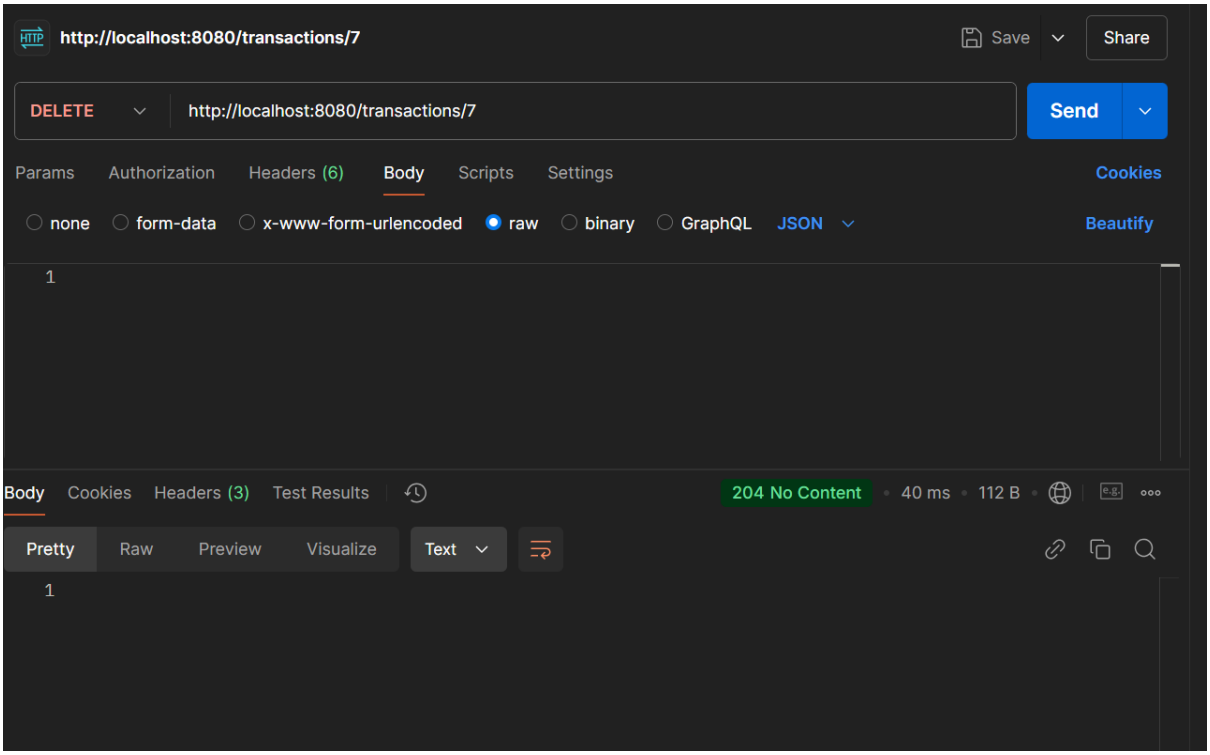
4. Post a new transaction:



5.Patch/Update (PUT) transaction:



6. Delete Transaction:

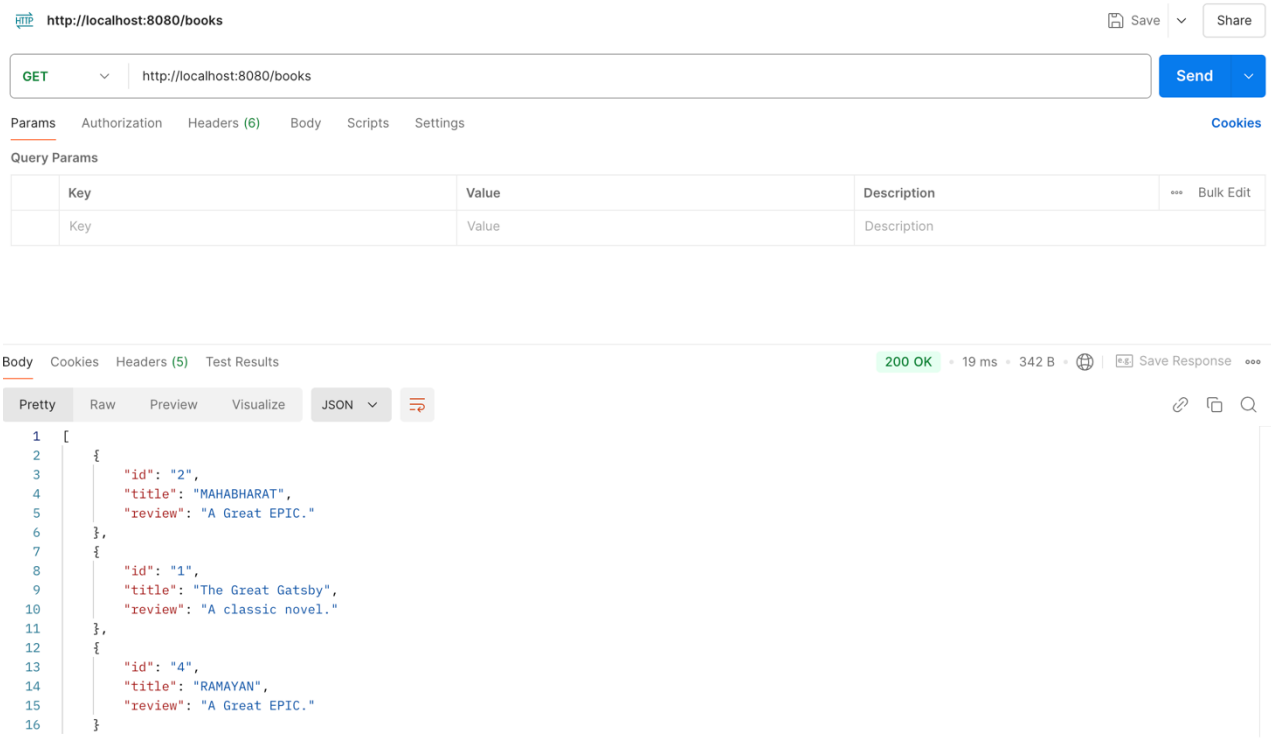


DataBase Content:

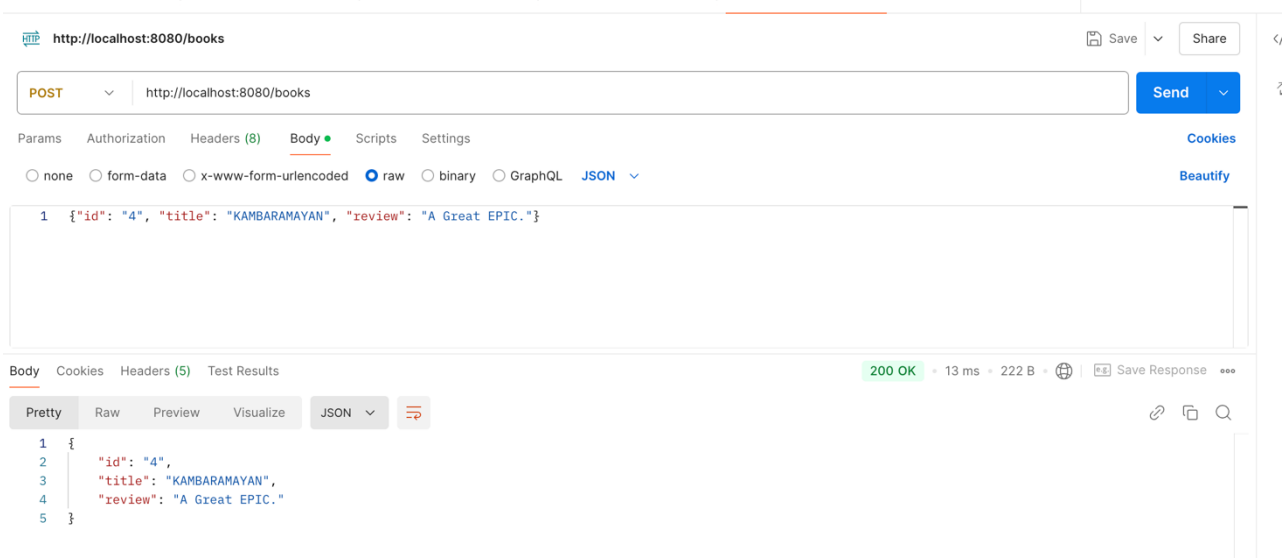
```
mysql> SELECT * FROM transaction LIMIT 5;
+----+---------+----------+--------+-----------+
| id | book_id | receiver | sender | status    |
+----+---------+----------+--------+-----------+
|  2 |       1 | User B   | User A | COMPLETED |
|  3 |    1345 | User D   | User E | ongoing   |
|  4 |    1765 | User D   | User E | COMPLETED |
|  5 |    1765 | User D   | User E | COMPLETED |
|  6 |      78 | User D   | User E | ongoing   |
+----+---------+----------+--------+-----------+
5 rows in set (0.01 sec)
```

Review Service:

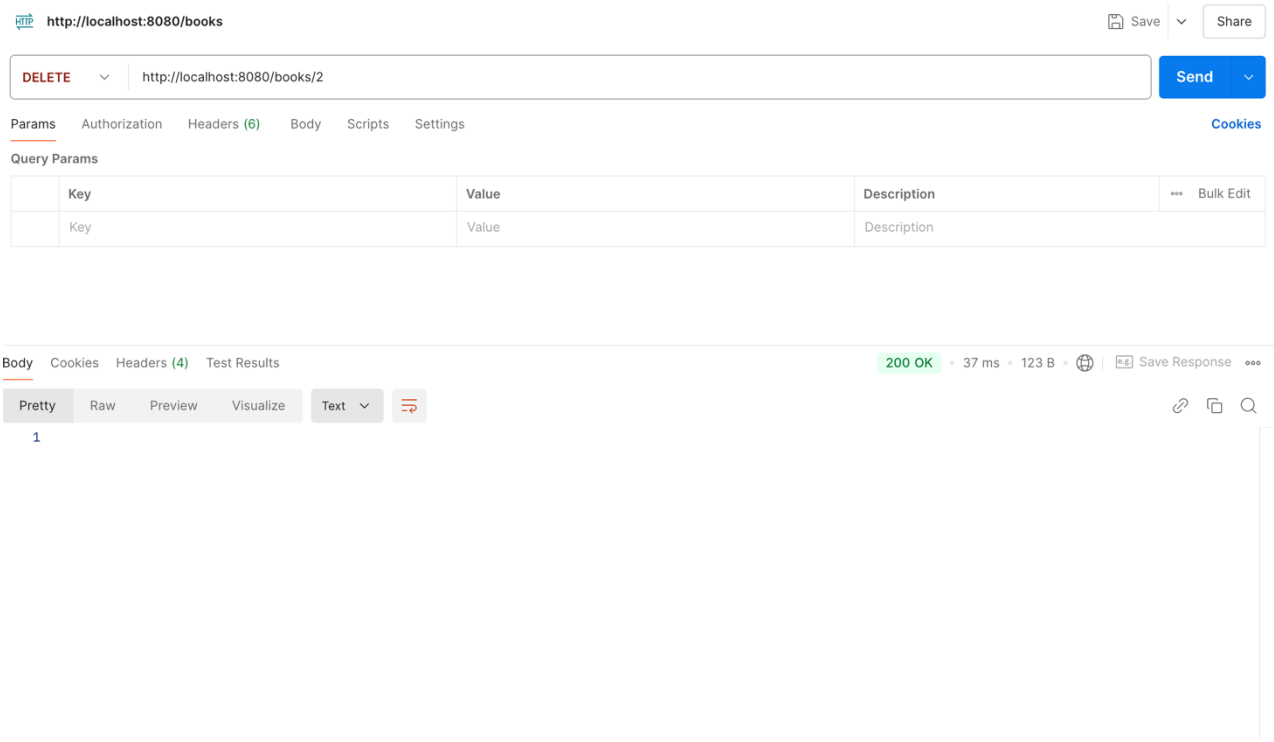
1) GET all Reviews



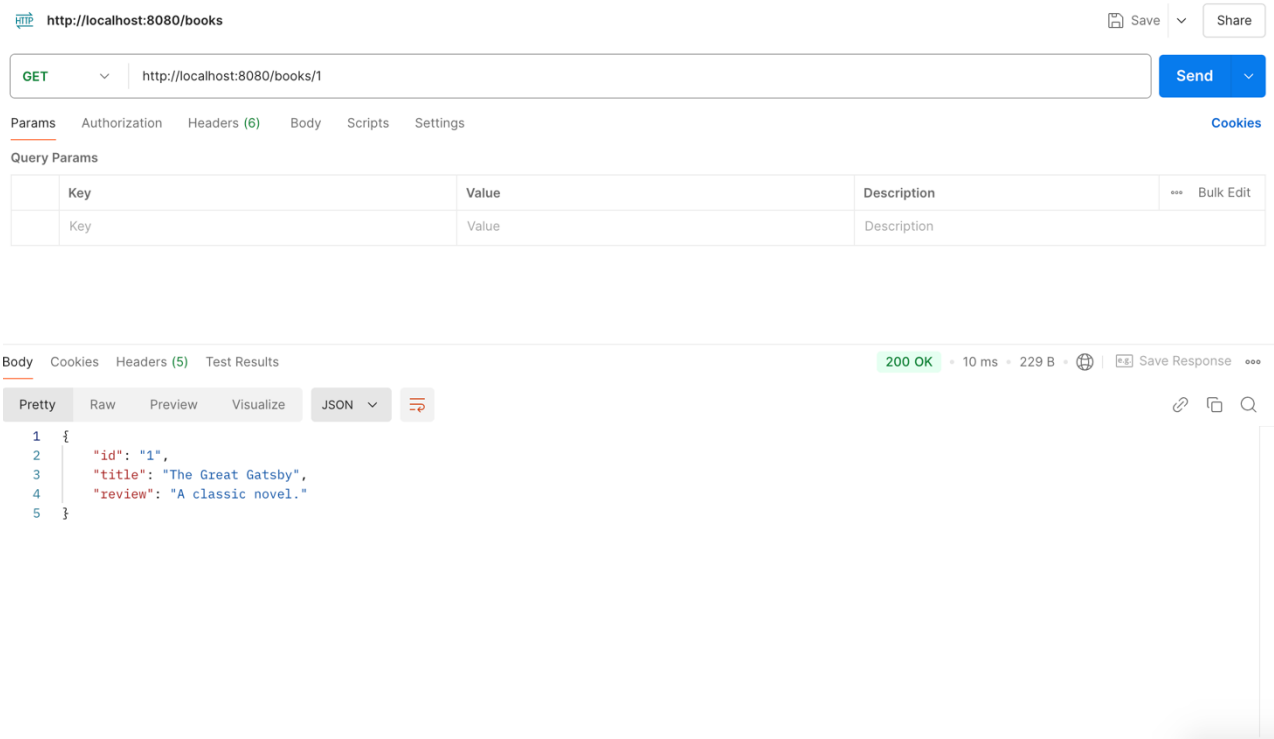
2) POST a review



3) DELETE a review

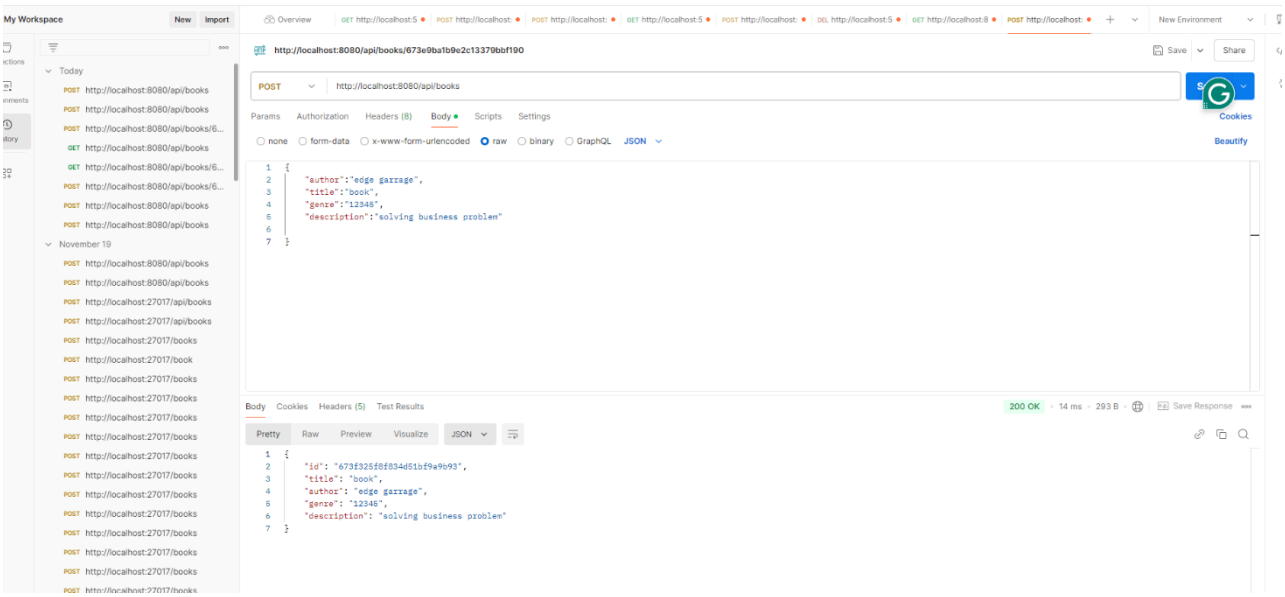


4) GET a reviews from particular USERID

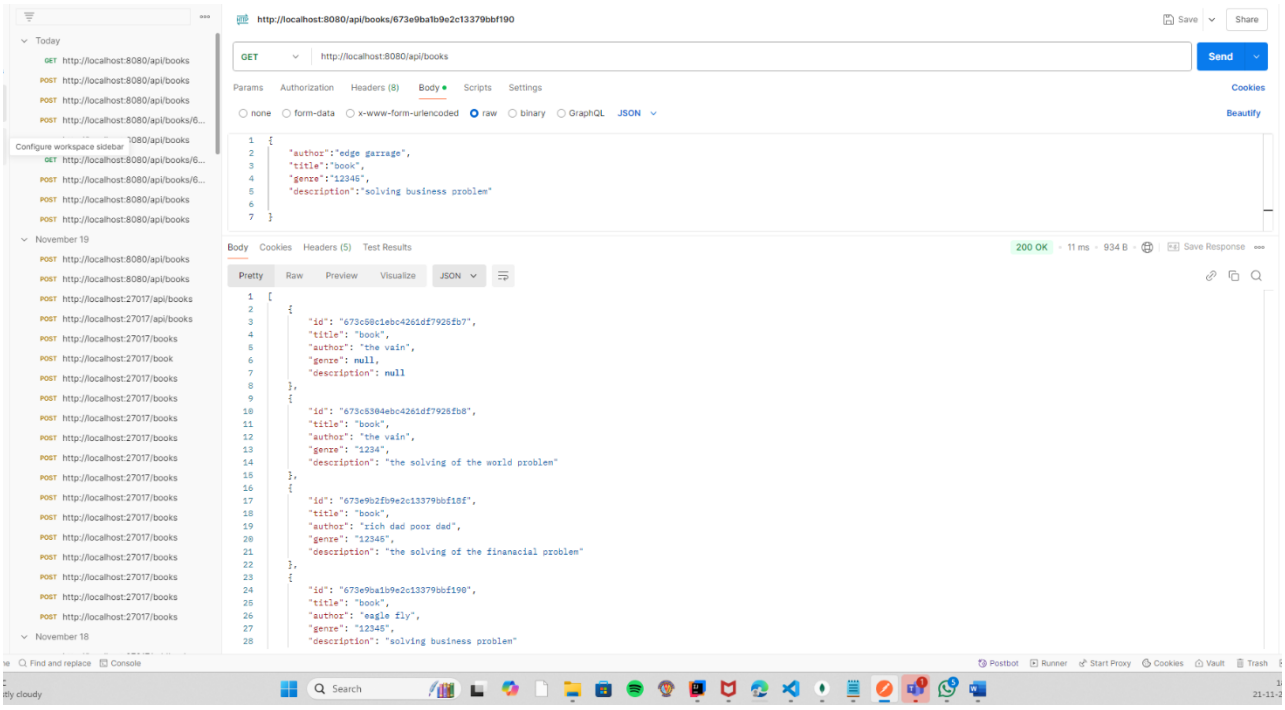


Book Services:

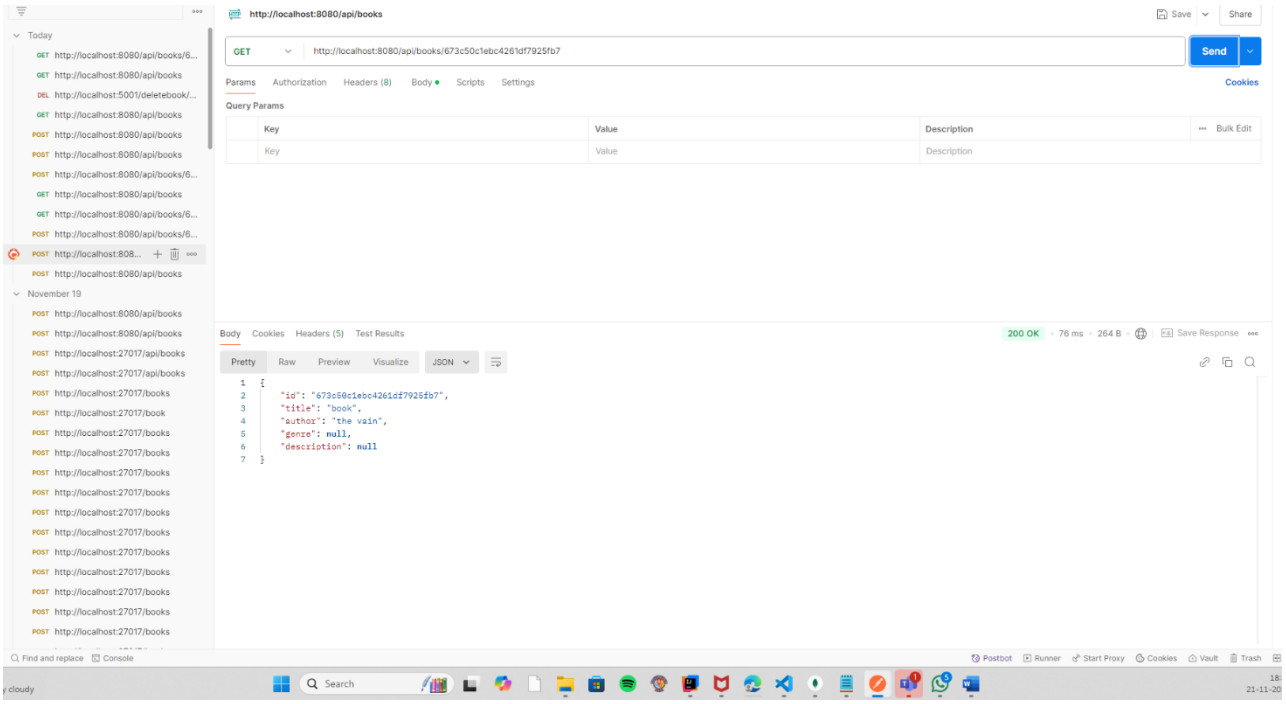
POST a Books



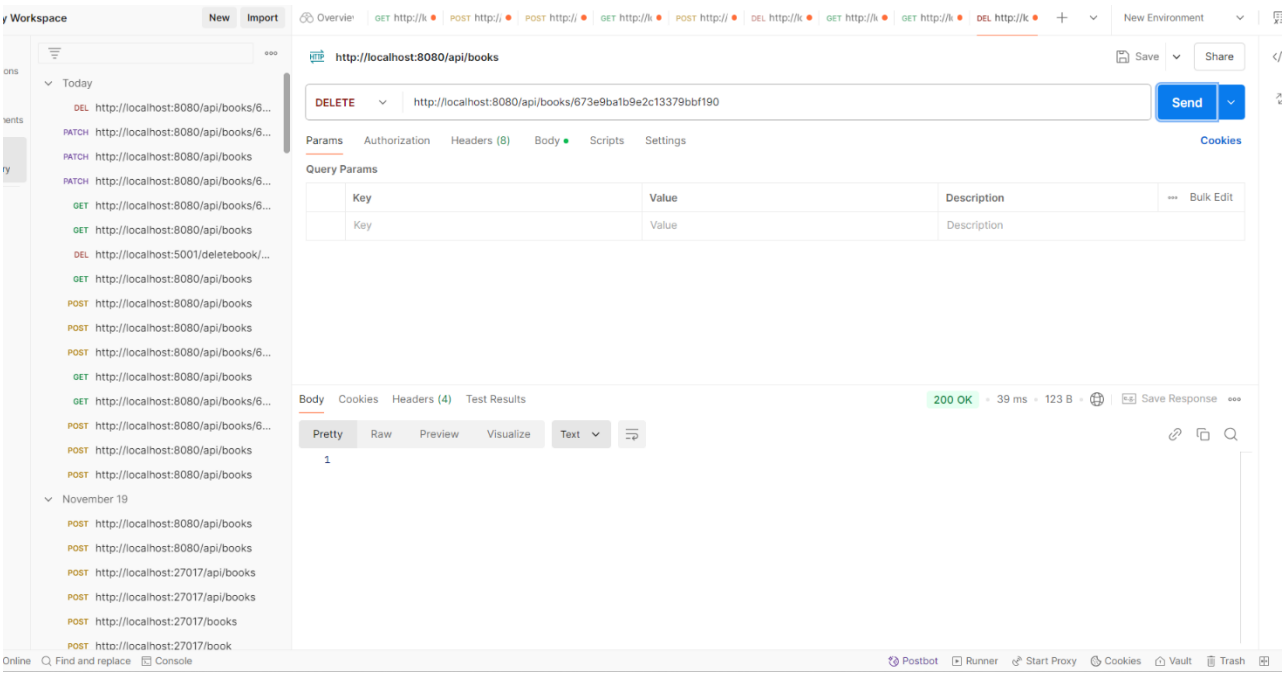
GET all Books



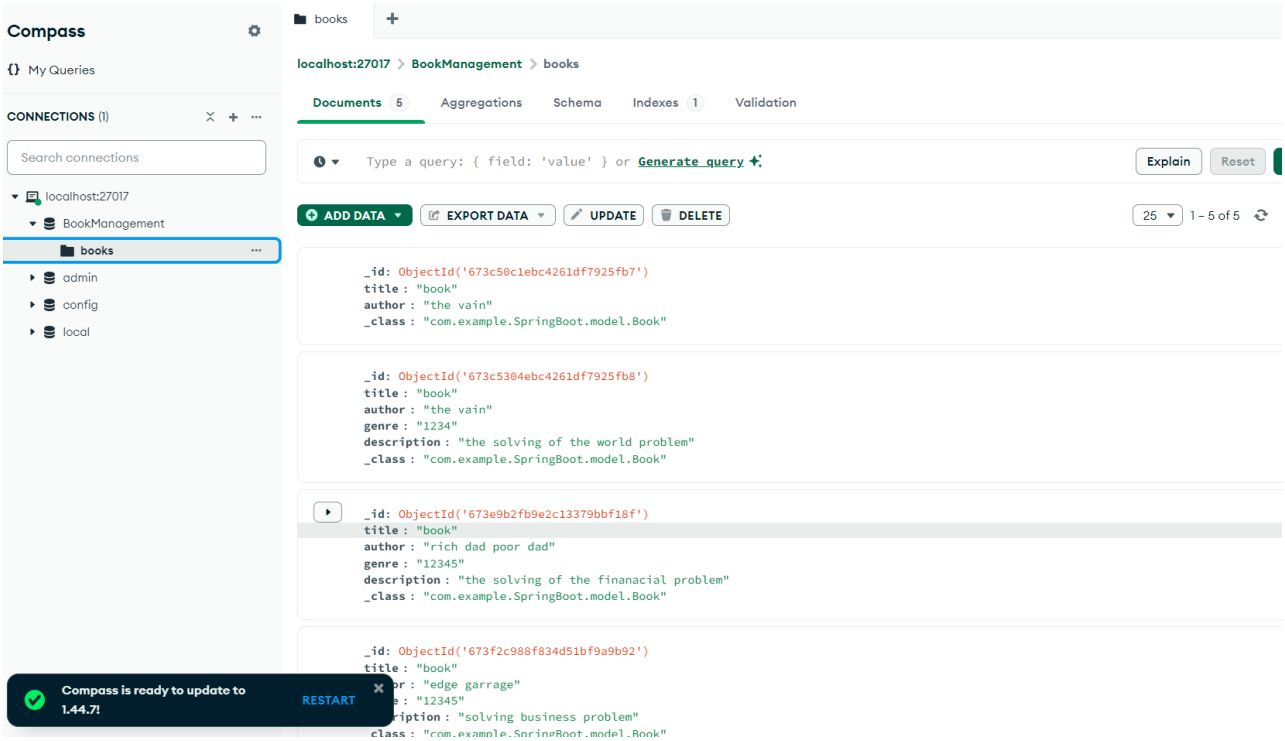
GET book by particular{id}



DELETE a book details

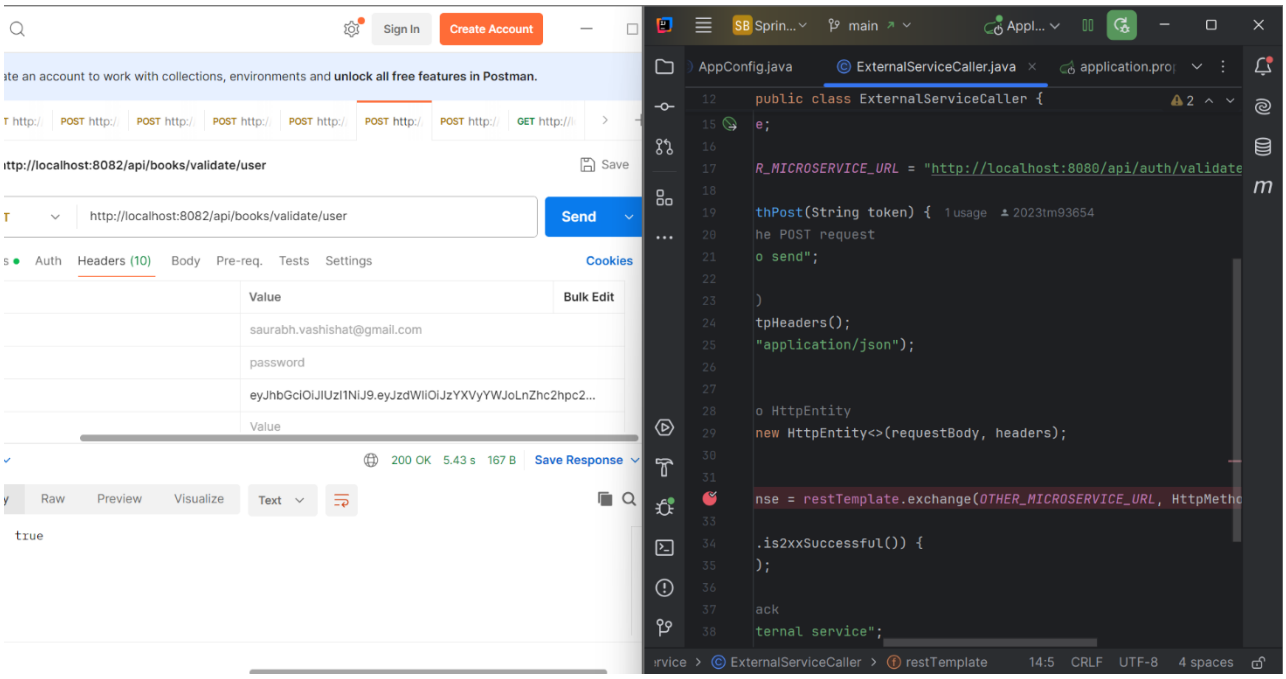


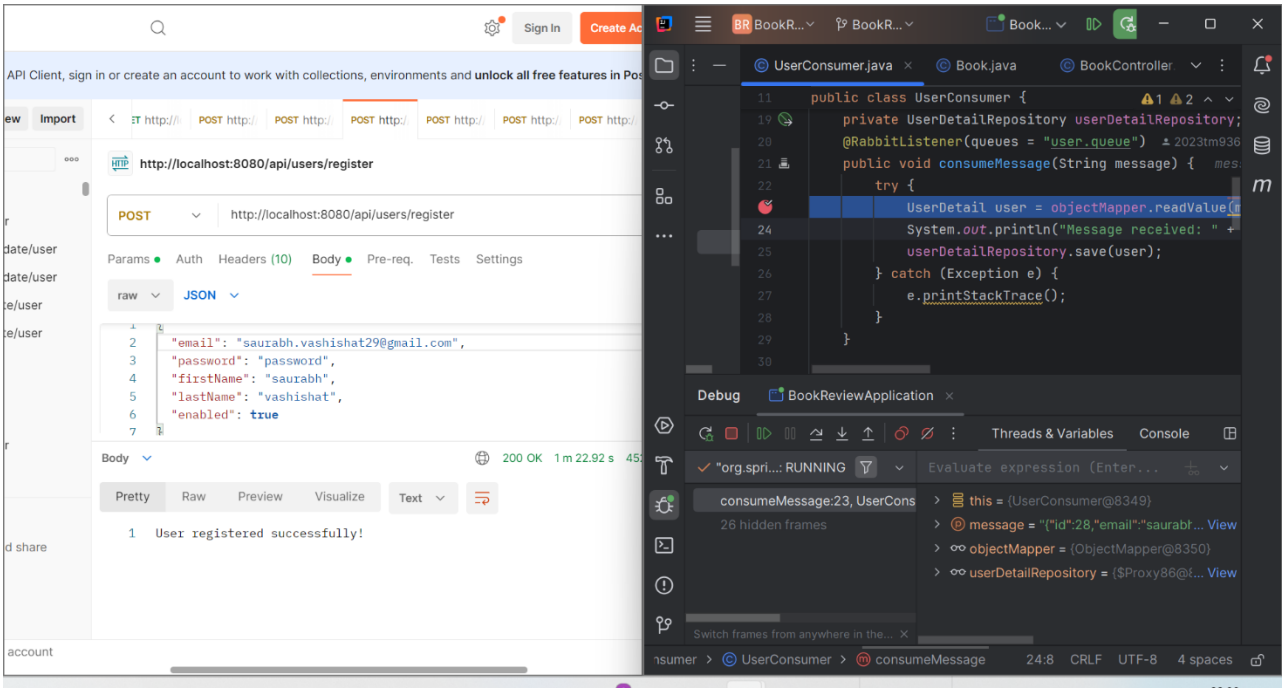
database Content:



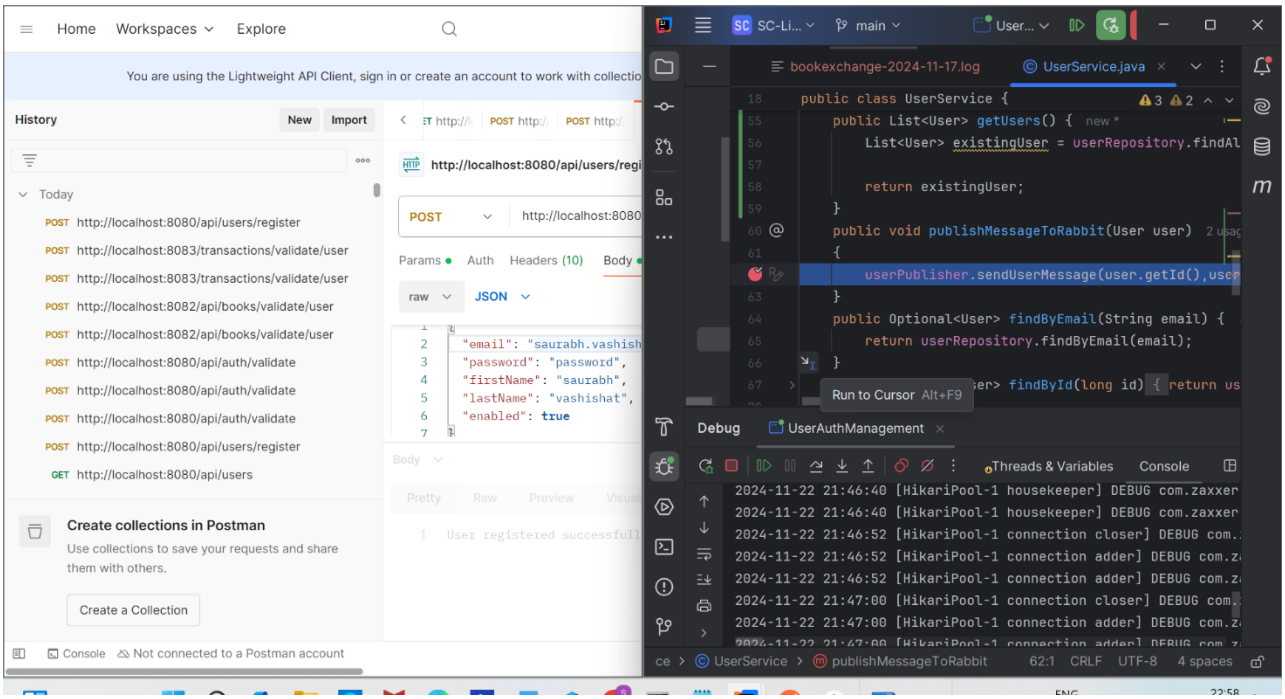
API Integration (Sync and Async)

Sync Integration : Here one of the Book management and transaction management service is calling the user management service that to validate the JWT token that is valid or not ,On the basis of this we can allow/block the user to access the microservice .

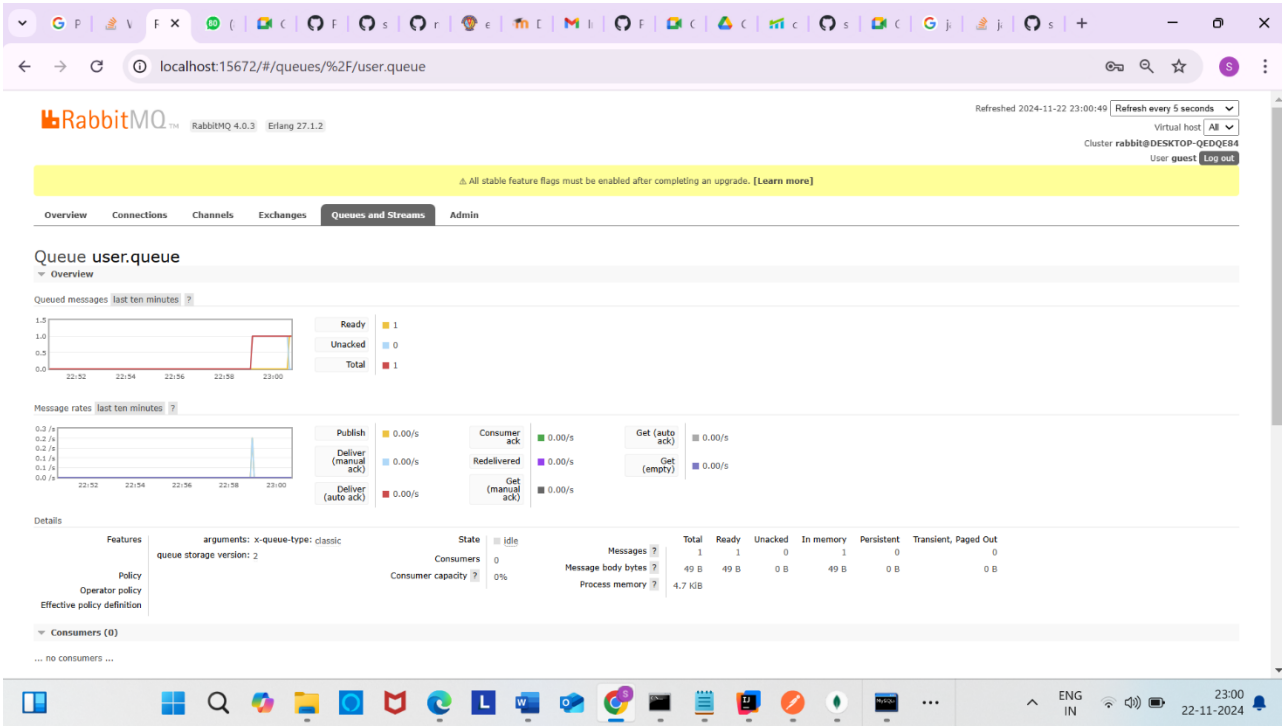




Async API integration through RabbitMq:



Rabbit mq



Conclusion:

The Book Management Microservices application demonstrates a robust implementation of microservices principles. Each service operates independently, with clear separation of concerns, dedicated databases, and well-defined API endpoints. The system integrates both synchronous (JWT validation via User Service) and asynchronous (RabbitMQ for message communication) communication patterns, ensuring flexibility and scalability.

Independent Development:
Each team member implemented their microservice independently, showcasing expertise in their respective areas. The approach highlighted the feasibility of decentralized development in a microservices architecture.

Asynchronous Communication with RabbitMQ:
RabbitMQ enabled seamless, event-driven communication between services, enhancing system scalability and fault tolerance.

Comprehensive Testing:
Postman was extensively used to test each service. The team ensured API functionality, data integrity, and proper service integration.

Challenges Overcome:
Despite challenges in group collaboration, the final product aligns with microservices best practices and delivers a functional Book Management application.

This project exemplifies the potential of microservices in simplifying complex systems. By leveraging Spring Boot, RabbitMQ, and modern database solutions, the team successfully built a scalable and maintainable application. Continued iterations on this foundation can lead to a production-ready solution for Book Management.