Cloud Computing - Mini Project Report

Microservice communication with RabbitMQ

April 2023

Submitted By:

Names: Anuraag Mallinamadugu, Arun Kumar Rath, Chetan Gurram, Appini Akhil

SRNs: PES1UG20CS067, PES1UG20CS076, PES1UG20CS112, PES1UG20CS074

VI Semester Section B

PES University

Short Description and Scope of the Project

Microservice architecture is a software development approach that structures an application as a collection of small, independent, and loosely coupled services that communicate with each other through APIs or message brokers. Each service is responsible for performing a single, well-defined task, and can be developed, deployed, and scaled independently of other services in the system. This approach allows for greater flexibility, scalability, and resilience than traditional monolithic architectures, as well as enabling organizations to more easily adopt and integrate new technologies and functionality into their systems.

This project is about building and deploying a microservices architecture where multiple components communicate with each other using RabbitMQ, a message broker. The architecture will consist of four microservices: an HTTP server that handles incoming requests to perform CRUD operations on a Student Management Database, a microservice that acts as the health check endpoint, a microservice that inserts a single student record, a microservice that retrieves student records, and a microservice that deletes a student record given the SRN.

The project will also involve creating a Docker network that hosts the RabbitMQ image, starting a RabbitMQ container on the network created, and accessing this network through its gateway IP address to connect to RabbitMQ from the producer and consumers. Additionally, an HTTP server (using Flask for Python or Express for NodeJS) will be created to listen to health checks, insert records, read databases, delete record requests and distribute them to the respective consumers. The consumers will use RabbitMQ clients to listen to incoming requests and process them accordingly.

Finally, the project will involve Dockerizing the application and using a MySQL database for the project. The scope of the project includes creating the necessary files and code for the front end as well.

Methodology

Define requirements and architecture: Clearly define the requirements for the microservices, including their functionality and how they will communicate with each other via RabbitMQ. Determine the overall architecture for the system, including the number of microservices and their responsibilities. In this case the producer communicates with 4 consumers using a direct connection through rabbitmq.

Setup development environment: Set up a development environment for the project, including any necessary tools, libraries, and frameworks. Python virtual environment is used to store the necessary pip installations. Hence, they can be easily imported from other users as well.

Develop and test each microservice: Develop each microservice individually, following the defined requirements and architecture. Test each microservice to ensure that it performs its intended function correctly. All four of our consumers wait for messages from the query parameters transferred by the producer which gets triggered when it received an HTTP request.

Setup RabbitMQ network: Manually create a Docker network to host the RabbitMQ image, and start a RabbitMQ container on the network. Access this network through its gateway IP address to connect to RabbitMQ from producers/consumers. The producers and consumers must be stored on a separate network which communicates with the former. Ensure that all the microservices are configured to connect to the RabbitMQ container.

Integrate microservices: Integrate the microservices by configuring the RabbitMQ exchanges and queues to allow for message passing between the services. Test the entire system to ensure that each microservice is correctly communicating with each other via RabbitMQ.

Dockerize the application: Dockerize the producer and consumer microservices by creating Dockerfiles and a docker-compose file that runs the entire system, including the RabbitMQ container and any necessary database containers.

Test: Test the overall project consisting of the two networks by using a tool such as Postman or Curl which can send HTTP requests to the flask server and see the values that get returned.

Deploy and scale: Deploy the Dockerized application to a production environment, and scale the microservices as necessary to handle increased traffic or load.

Monitor and maintain: Monitor the system to ensure that it is functioning correctly and efficiently, and maintain it by addressing any issues or bugs that arise over time.

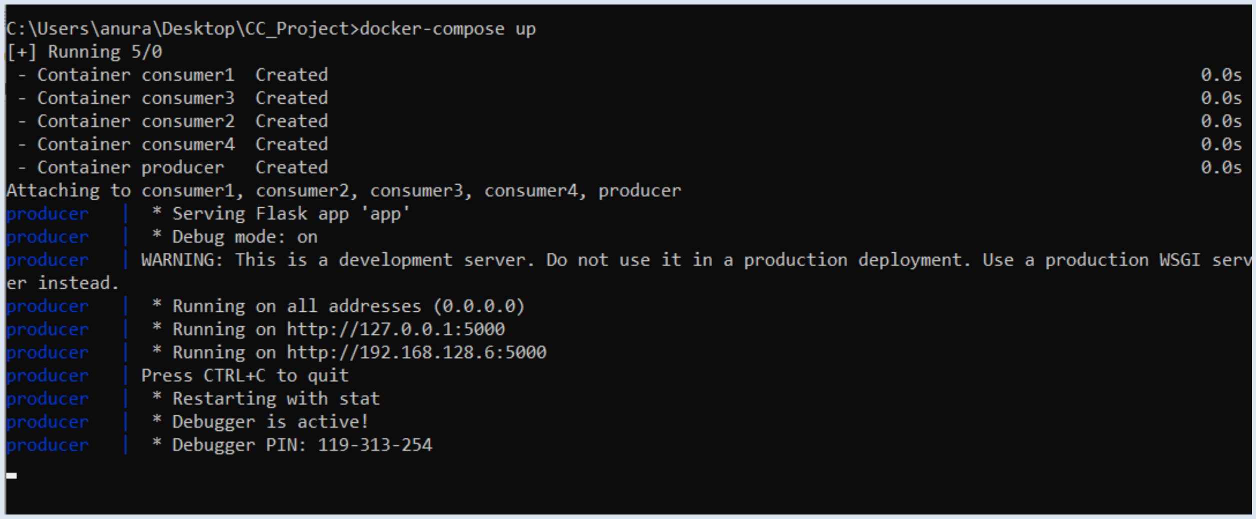
Enhance and update: As the needs of the system evolve, enhance and update the microservices to meet these needs, while continuing to maintain and monitor the system.

Testing

Postman is a popular API testing tool that can be used to test the microservices architecture built using RabbitMQ. Once the microservices are running, the HTTP endpoints exposed by the producer microservice can be tested using Postman. For example, to test the health_check endpoint, a GET request can be sent to the URL of the endpoint using Postman. Similarly, to test the insert_record endpoint, a POST request with the necessary fields can be sent to the URL of the endpoint. Postman can also be used to test the retrieval and deletion of records from the database by sending GET and DELETE requests to the respective endpoints. By testing the microservices using Postman, it is possible to ensure that the endpoints are working correctly and that data is being transferred between the producer and consumers via RabbitMQ.

Steps taken for testing:

- Running docker-compose up:

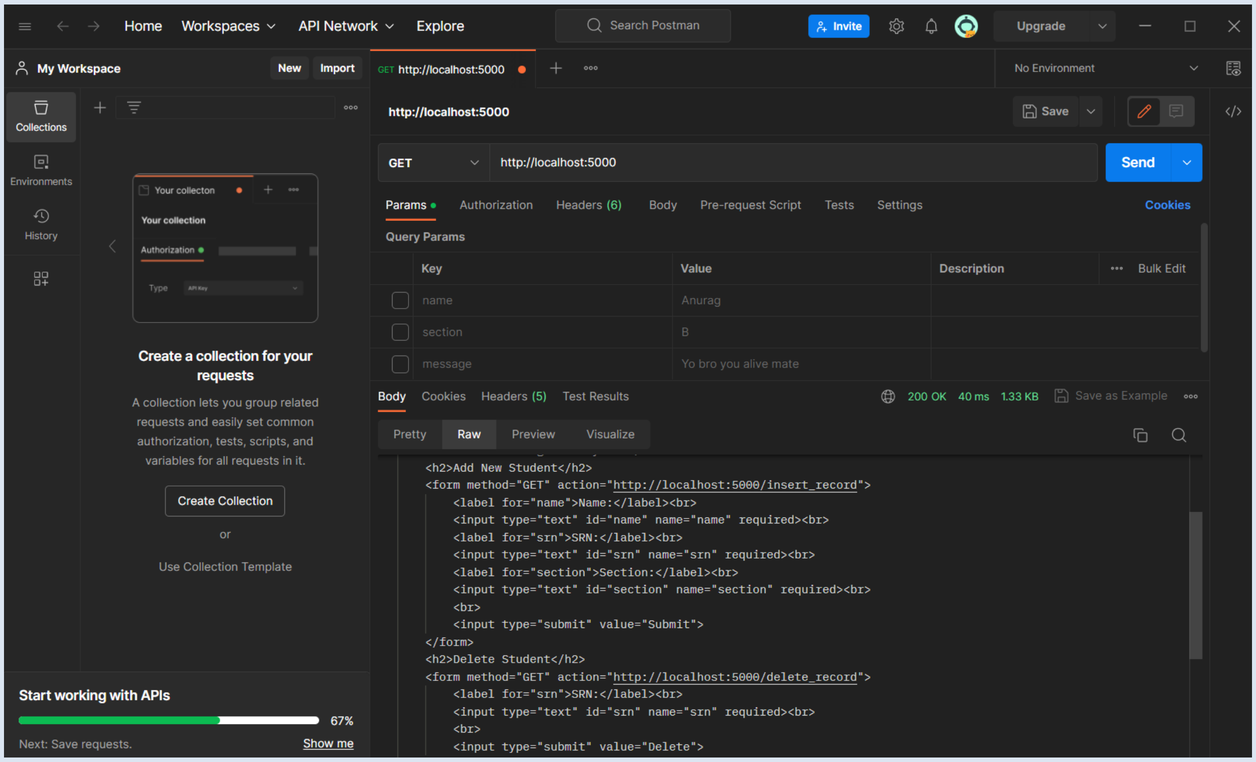


```
C:\Users\anura\Desktop\CC_Project>docker-compose up
[+] Running 5/0
 - Container consumer1  Created                                                          0.0s
 - Container consumer3  Created                                                          0.0s
 - Container consumer2  Created                                                          0.0s
 - Container consumer4  Created                                                          0.0s
 - Container producer   Created                                                          0.0s
Attaching to consumer1, consumer2, consumer3, consumer4, producer
producer   |  * Serving Flask app 'app'
producer   |  * Debug mode: on
producer   | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI serv
er instead.
producer   |  * Running on all addresses (0.0.0.0)
producer   |  * Running on http://127.0.0.1:5000
producer   |  * Running on http://192.168.128.6:5000
producer   | Press CTRL+C to quit
producer   |  * Restarting with stat
producer   |  * Debugger is active!
producer   |  * Debugger PIN: 119-313-254
```

The producer and consumers are shown to have started immediately

- Checking the main page by giving the URL of the index.html file to Postman. Here we can get the output in two forms: in raw form and in preview form which shows how the web page looks like on deployment.

In raw form



GET http://localhost:5000

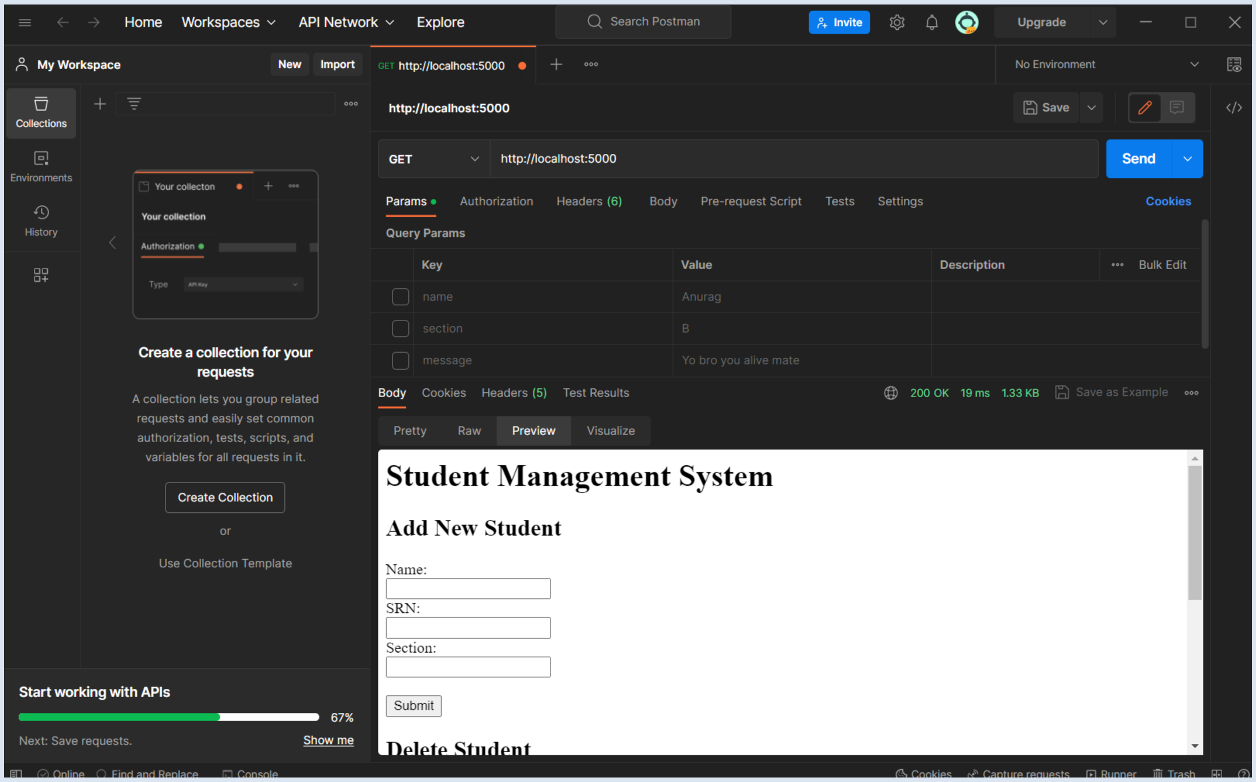
Query Params

Key	Value	Description
name	Anurag	
section	B	
message	Yo bro you alive mate	

200 OK 40 ms 1.33 KB

```
<h2>Add New Student</h2>
<form method="GET" action="http://localhost:5000/insert_record">
    <label for="name">Name:</label><br>
    <input type="text" id="name" name="name" required><br>
    <label for="srn">SRN:</label><br>
    <input type="text" id="srn" name="srn" required><br>
    <label for="section">Section:</label><br>
    <input type="text" id="section" name="section" required><br>
    <br>
    <input type="submit" value="Submit">
</form>
<h2>Delete Student</h2>
<form method="GET" action="http://localhost:5000/delete_record">
    <label for="srn">SRN:</label><br>
    <input type="text" id="srn" name="srn" required><br>
    <br>
    <input type="submit" value="Delete">
```

In preview form



GET http://localhost:5000

Query Params

Key	Value	Description
name	Anurag	
section	B	
message	Yo bro you alive mate	

200 OK 19 ms 1.33 KB

Student Management System

Add New Student

Name:

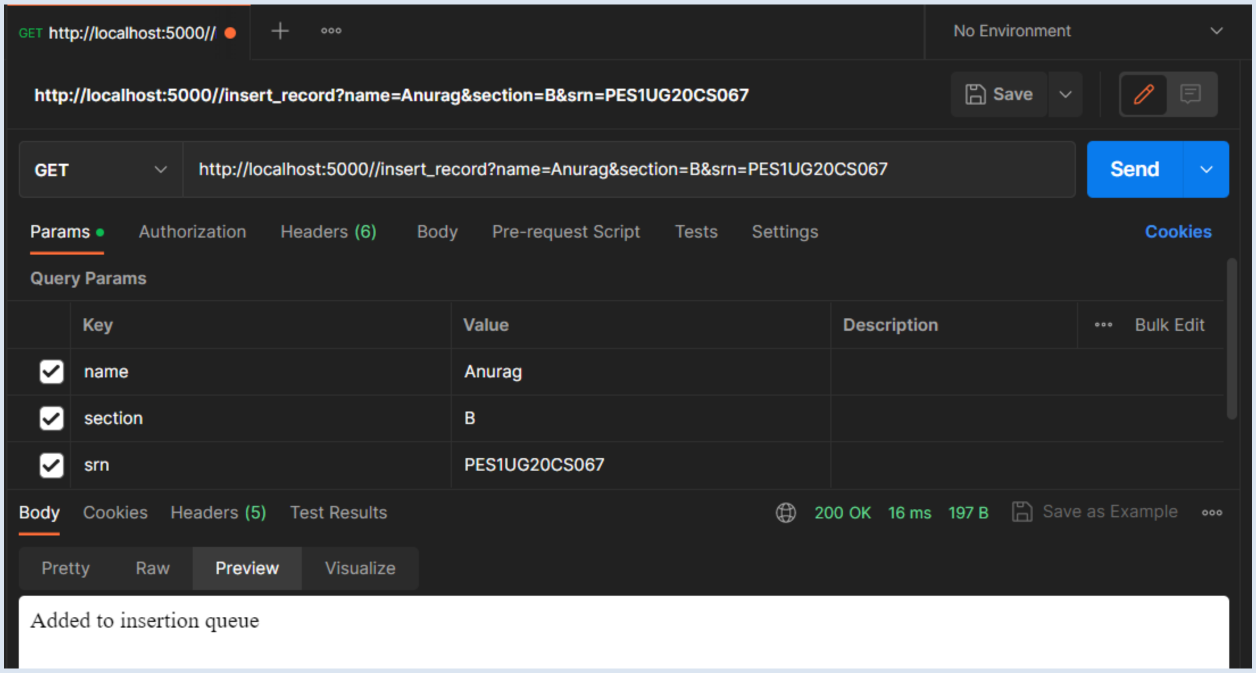
SRN:

Section:

Submit

Delete Student

- Now we add /insert_record to the URL and add some query parameters:



GET http://localhost:5000//

No Environment

http://localhost:5000//insert_record?name=Anurag&section=B&srn=PES1UG20CS067

GET http://localhost:5000//insert_record?name=Anurag&section=B&srn=PES1UG20CS067

Query Params

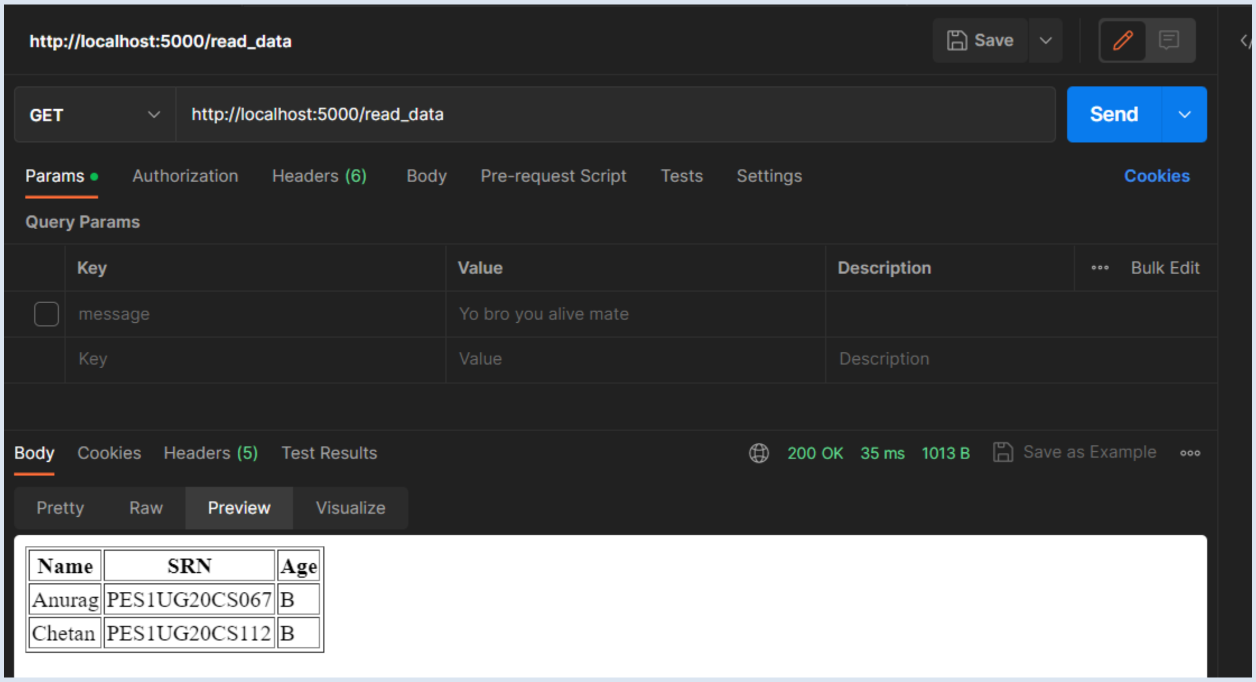
	Key	Value	Description
✓	name	Anurag	
✓	section	B	
✓	srn	PES1UG20CS067	

200 OK 16 ms 197 B

```
Added to insertion queue
```

The given data has been added to the database.

- Now we add /read_data to the URL to see the database



http://localhost:5000/read_data

GET http://localhost:5000/read_data

Query Params

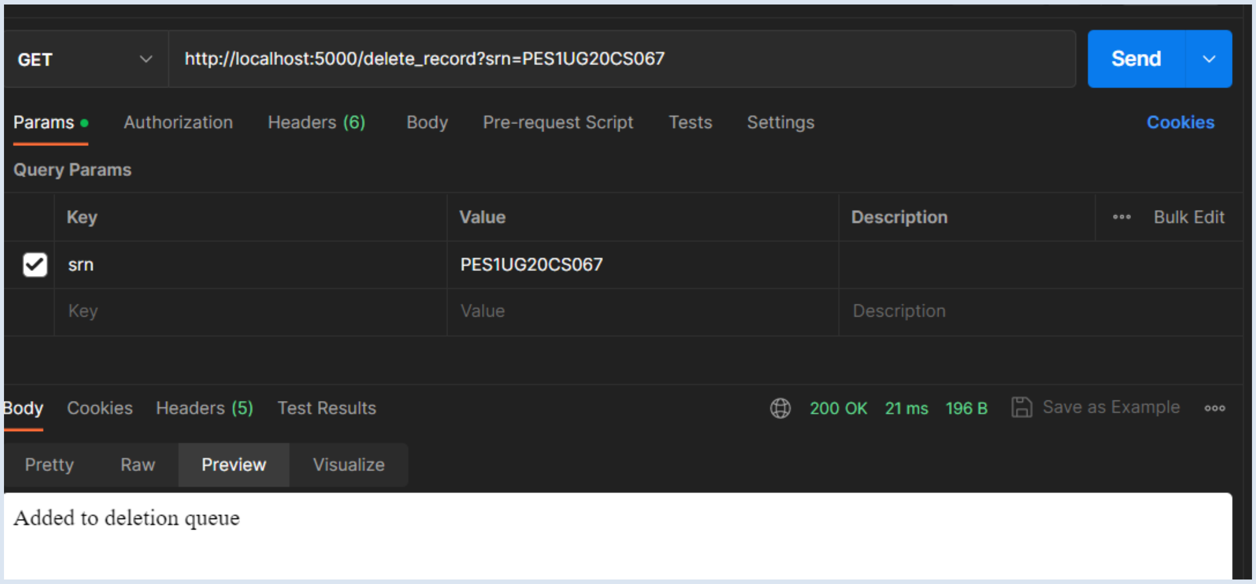
	Key	Value	Description
☐	message	Yo bro you alive mate	
	Key	Value	Description

200 OK 35 ms 1013 B

Name	SRN	Age
Anurag	PES1UG20CS067	B
Chetan	PES1UG20CS112	B

As it can be seen, the required value has been inserted

- Now, we add delete_record to remove the person with SRN PES1UG20CS067:



http://localhost:5000/delete_record?srn=PES1UG20CS067

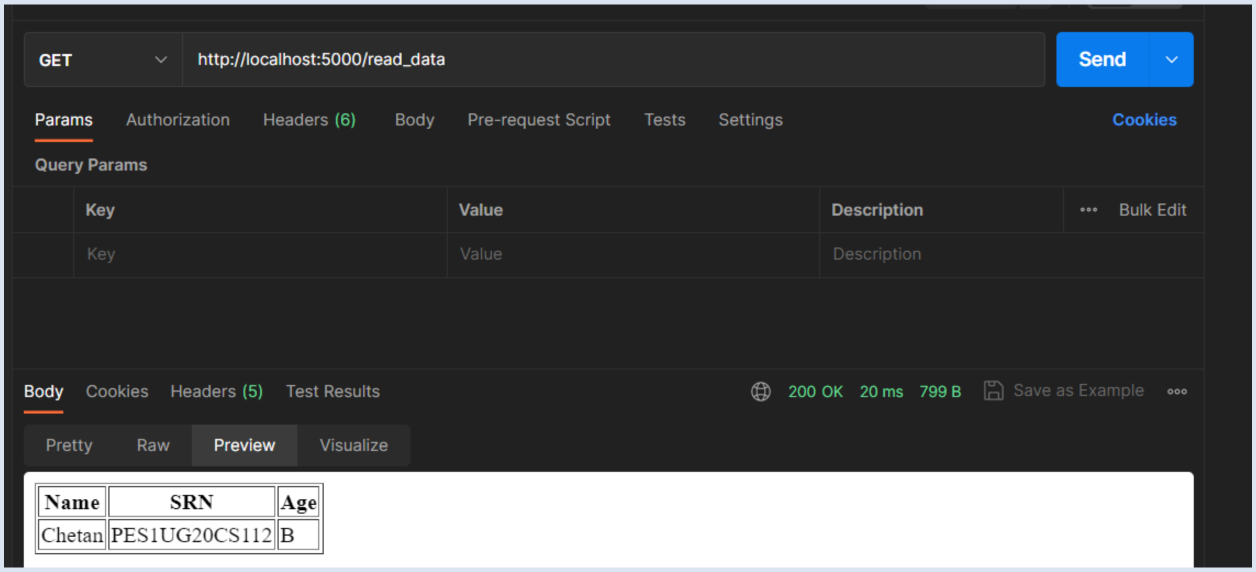
Query Params

Key	Value	Description
srn	PES1UG20CS067	
Key	Value	Description

200 OK 21 ms 196 B

Added to deletion queue

Now if we read data again:



http://localhost:5000/read_data

Query Params

Key	Value	Description
Key	Value	Description

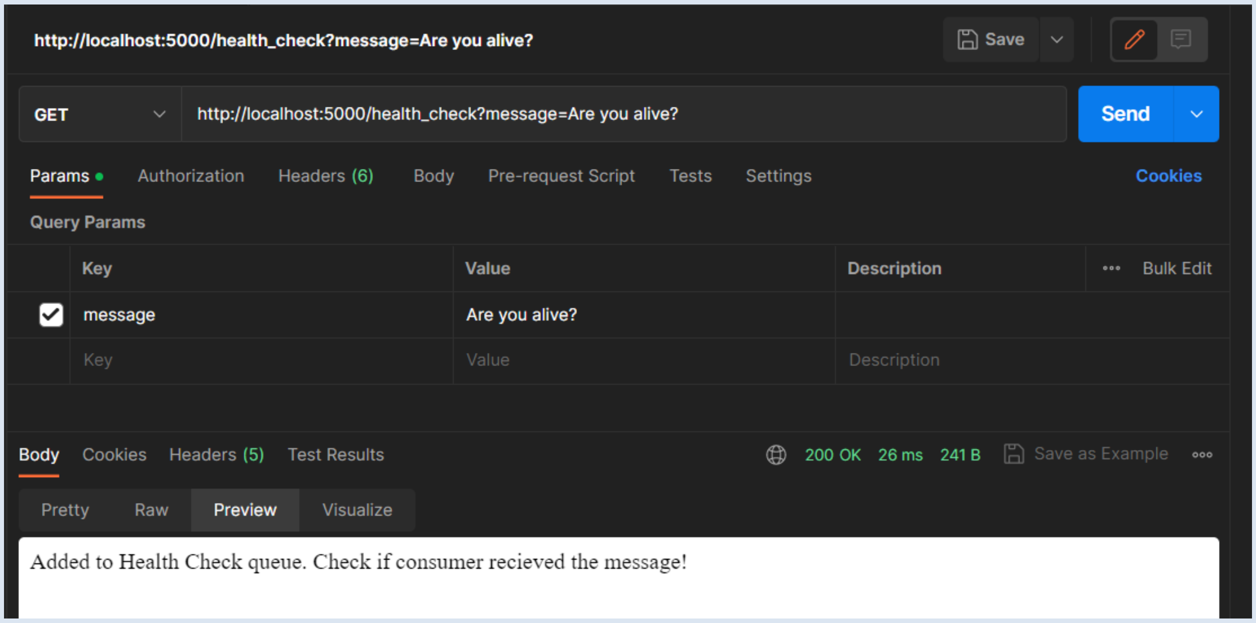
200 OK 20 ms 799 B

Name	SRN	Age
Chetan	PES1UG20CS112	B

Value has been successfully deleted

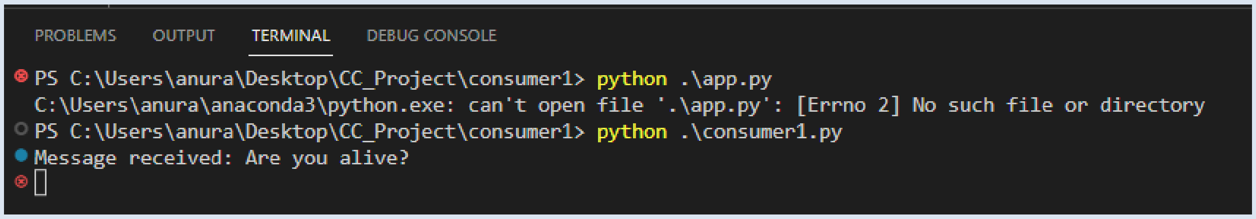
- Sending health check message:

Here the producer sends a customized message and checks if the consumer has received it. This shows that the connection is established.



Key	Value	Description
message	Are you alive?	
Key	Value	Description

Added to Health Check queue. Check if consumer recieved the message!



```
PS C:\Users\anura\Desktop\CC_Project\consumer1> python .\app.py
C:\Users\anura\anaconda3\python.exe: can't open file '.\app.py': [Errno 2] No such file or directory
PS C:\Users\anura\Desktop\CC_Project\consumer1> python .\consumer1.py
Message received: Are you alive?
```

As it can be seen, the consumer has received the required message

Results and Conclusions

In this project, we have successfully implemented a microservices architecture using RabbitMQ as a message broker to handle communication between the different components. We built four microservices for performing CRUD operations on a student management database using RabbitMQ as a messaging system.

We also containerized the application using Docker, and we created a docker-compose file that runs all the microservices along with the MySQL database.

To test the application, we used Postman, a popular tool for API testing. We tested each microservice's API endpoints to ensure that they were working correctly and communicating with the message broker as intended.

Overall, this project demonstrates the power and flexibility of microservices architecture, which allows us to build complex, distributed systems that can scale and evolve over time. By using RabbitMQ as a message broker, we can ensure that messages are delivered reliably and efficiently between the different microservices, improving the overall performance and reliability of the system.

In conclusion, this project provides a solid foundation for building similar microservices-based applications using RabbitMQ and other message brokers. By following the methodology outlined in this project, developers can create scalable and efficient applications that can meet the demands of modern, data-driven businesses.