Cloud Computing (UE20CS351)
Mini Project Report

Microservice Communication with RabbitMQ

Submitted By:
Lagan T G (PES1UG20CS555)
Pratik M Katti (PES1UG20CS577)
Rajat Rayaraddi (PES1UG20CS578)
Sacheth N S (PES1UG20CS585)

Short Description and Scope of the Project

The microservice architecture is one of the most popular forms of deployment, especially in larger organizations where there are multiple components that can be loosely coupled together. Not only does this make it easier to work on separate components independently, but ensures that issues in one component do not bring down the rest of the service. A microservices architecture consists of a collection of small, autonomous services where each service is self-contained and should implement a single business capability within a bounded context. This also comes with the advantage that a single system can scale thereby limiting the resources to required components.

This project is about building and deploying a microservices architecture where multiple components communicate with each other using RabbitMQ, a message broker.
The project comprises of 4 microservices:
A HTTP server that handles incoming requests to perform CRUD operations on a Student Management Database + Check the health of the RabbitMQ connection, a microservice that acts as the health check endpoint, a microservice that inserts a single student record, a microservice that retrieves student records, a microservice that deletes a student record given the SRN.

The project will also involve creating a Docker network that hosts the RabbitMQ image, running RabbitMQ 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) 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.

Methodology

- Define the architecture and the requirements: Define the functional needs for the microservices, as well as how they will communicate with one another using RabbitMQ. Establish the system's overall design, including the number of microservices and their roles. In this instance, the producer uses a direct RabbitMQ connection to speak with 4 consumers.

- Establish a development environment: Create the project's development environment, including any tools, libraries, and frameworks that are required. Pip installations are stored in a Python virtual environment. As a result, it is simple to import them from other users as well.

- Create and test each microservice: Create each microservice separately, adhering to the specified architecture and specifications. To guarantee that each microservice fulfils its intended purpose, test it. The producer triggers when it receives an HTTP request, and all four of our consumers wait for messages from the query parameters that are transferred by the producer.

- RabbitMQ network configuration to host the RabbitMQ image: Manually construct a Docker network and launch a RabbitMQ container on it. To connect to RabbitMQ from producers or consumers, use the network's gateway IP address. It is necessary to store the producers and consumers on a separate network that connects to the former. Make sure that each microservice is set up to communicate with the RabbitMQ container.

- Microservice integration: To integrate the services, set up the RabbitMQ exchanges and queues so that messages may be passed between them. To confirm that each microservice is correctly interacting with one another over RabbitMQ, test the complete system.

- Application Dockerization: By developing Dockerfiles and a docker-compose file that runs the complete system, including the RabbitMQ container and any required database containers, you may dockerize the producer and consumer microservices.

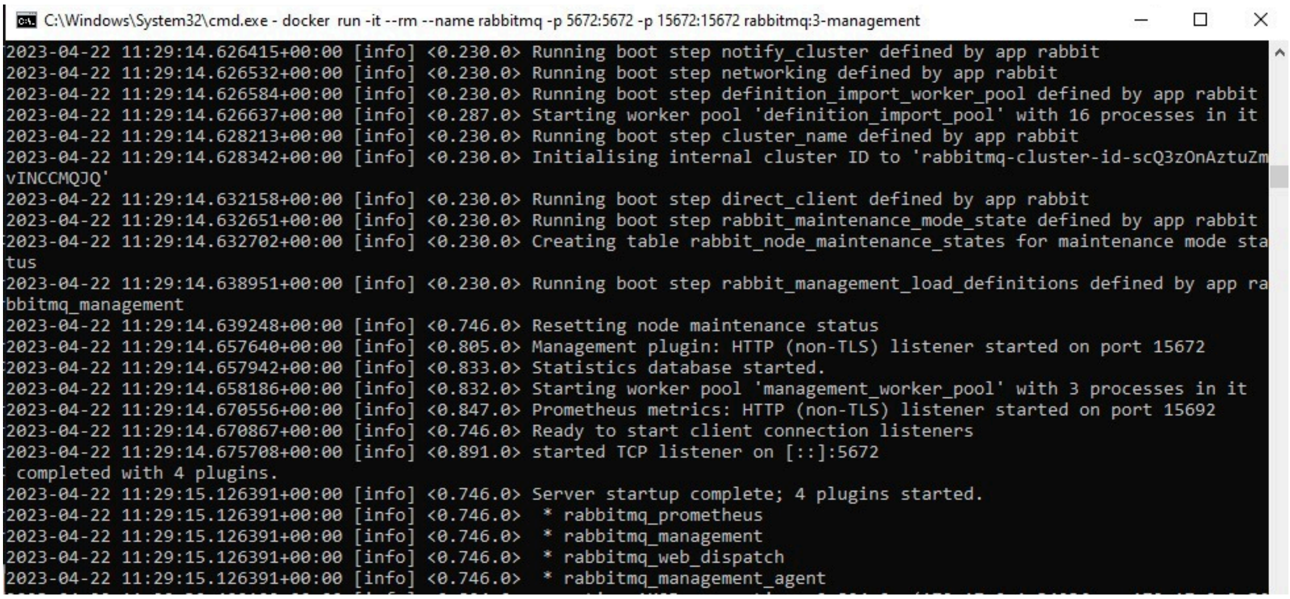
- Test: Use a tool like Postman or Curl to submit HTTP requests to the flask server and view the values that are returned to test the complete project, which consists of the two networks.

Testing

After the microservices have started, Postman may be used to test the HTTP endpoints that the producer microservice has exposed. For instance, Postman can be used to send a GET request to the health_check endpoint's URL in order to test it. Similarly, a POST request with the required information can be sent to the endpoint's URL to test the insert_record endpoint. By making GET and DELETE requests to the appropriate endpoints, Postman can also be used to test the retrieval and deletion of records from the database. It is possible to validate the functionality of the endpoints and confirm that data is being sent between the producer and consumers over RabbitMQ by testing the microservices using Postman.

Steps:

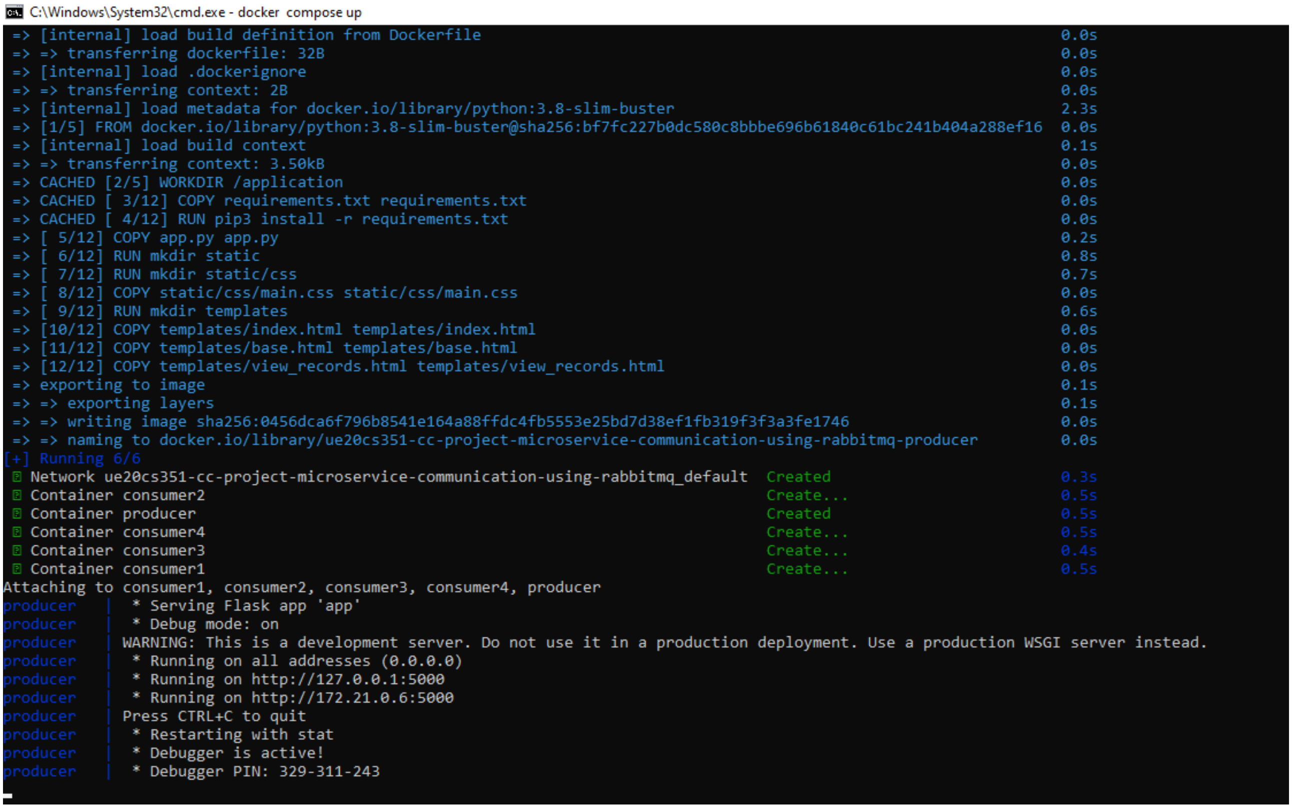
1. Running ‘docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management’ to run RabbitMQ.



C:\Windows\System32\cmd.exe - docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3-management

```
2023-04-22 11:29:14.626415+00:00 [info] <0.230.0> Running boot step notify_cluster defined by app rabbit
2023-04-22 11:29:14.626532+00:00 [info] <0.230.0> Running boot step networking defined by app rabbit
2023-04-22 11:29:14.626584+00:00 [info] <0.230.0> Running boot step definition_import_worker_pool defined by app rabbit
2023-04-22 11:29:14.626637+00:00 [info] <0.287.0> Starting worker pool 'definition_import_pool' with 16 processes in it
2023-04-22 11:29:14.628213+00:00 [info] <0.230.0> Running boot step cluster_name defined by app rabbit
2023-04-22 11:29:14.628342+00:00 [info] <0.230.0> Initialising internal cluster ID to 'rabbitmq-cluster-id-scQ3zOnAztuZm
vINCCMQJQ'
2023-04-22 11:29:14.632158+00:00 [info] <0.230.0> Running boot step direct_client defined by app rabbit
2023-04-22 11:29:14.632651+00:00 [info] <0.230.0> Running boot step rabbit_maintenance_mode_state defined by app rabbit
2023-04-22 11:29:14.632702+00:00 [info] <0.230.0> Creating table rabbit_node_maintenance_states for maintenance mode sta
tus
2023-04-22 11:29:14.638951+00:00 [info] <0.230.0> Running boot step rabbit_management_load_definitions defined by app ra
bbitmq_management
2023-04-22 11:29:14.639248+00:00 [info] <0.746.0> Resetting node maintenance status
2023-04-22 11:29:14.657640+00:00 [info] <0.805.0> Management plugin: HTTP (non-TLS) listener started on port 15672
2023-04-22 11:29:14.657942+00:00 [info] <0.833.0> Statistics database started.
2023-04-22 11:29:14.658186+00:00 [info] <0.832.0> Starting worker pool 'management_worker_pool' with 3 processes in it
2023-04-22 11:29:14.670556+00:00 [info] <0.847.0> Prometheus metrics: HTTP (non-TLS) listener started on port 15692
2023-04-22 11:29:14.670867+00:00 [info] <0.746.0> Ready to start client connection listeners
2023-04-22 11:29:14.675708+00:00 [info] <0.891.0> started TCP listener on [::]:5672
 completed with 4 plugins.
2023-04-22 11:29:15.126391+00:00 [info] <0.746.0> Server startup complete; 4 plugins started.
2023-04-22 11:29:15.126391+00:00 [info] <0.746.0>  * rabbitmq_prometheus
2023-04-22 11:29:15.126391+00:00 [info] <0.746.0>  * rabbitmq_management
2023-04-22 11:29:15.126391+00:00 [info] <0.746.0>  * rabbitmq_web_dispatch
2023-04-22 11:29:15.126391+00:00 [info] <0.746.0>  * rabbitmq_management_agent
```

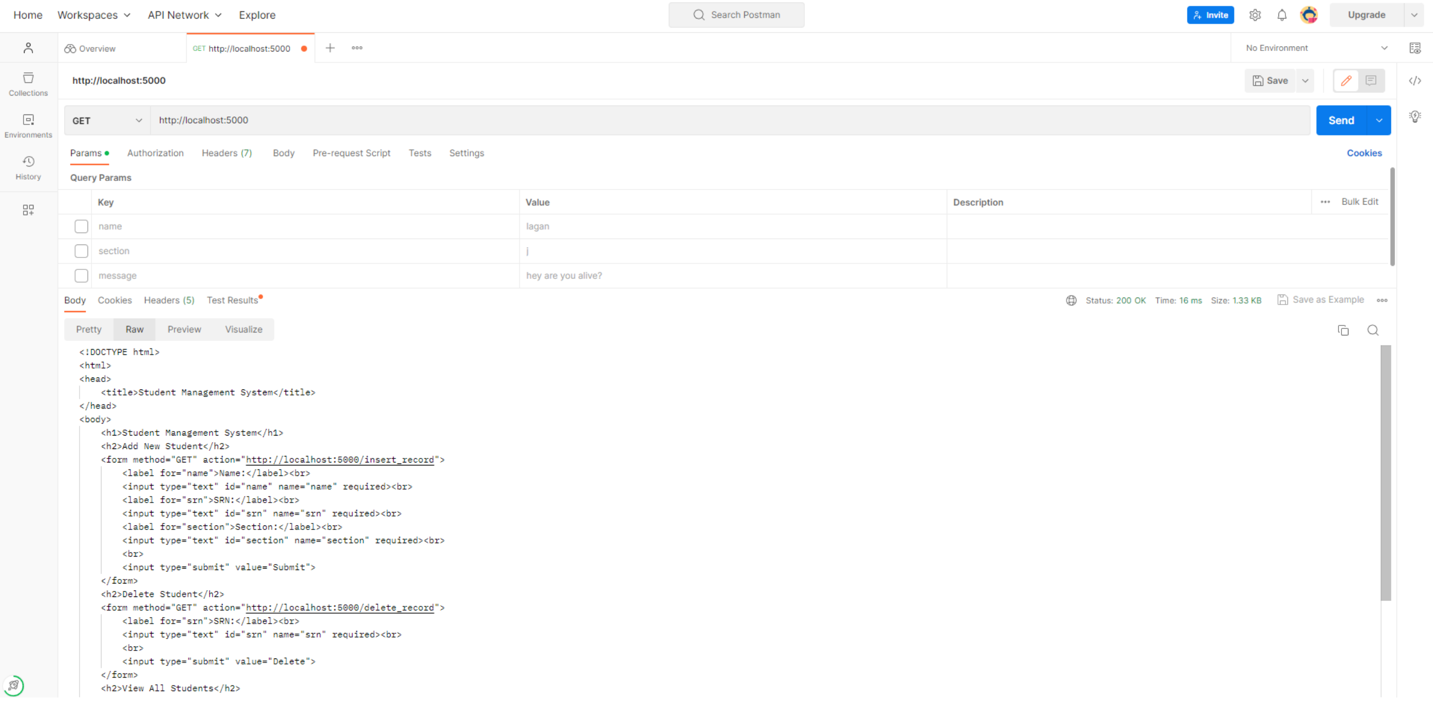
2. Running ‘docker compose up’ to build and run the producer and consumers.



C:\Windows\System32\cmd.exe - docker compose up

```
=> [internal] load build definition from Dockerfile                                                                0.0s
=> => transferring dockerfile: 32B                                                                                 0.0s
=> [internal] load .dockerignore                                                                                   0.0s
=> => transferring context: 2B                                                                                     0.0s
=> [internal] load metadata for docker.io/library/python:3.8-slim-buster                                           2.3s
=> [1/5] FROM docker.io/library/python:3.8-slim-buster@sha256:bf7fc227b0dc580c8bbbe696b61840c61bc241b404a288ef16  0.0s
=> [internal] load build context                                                                                   0.1s
=> => transferring context: 3.50kB                                                                                 0.0s
=> CACHED [2/5] WORKDIR /application                                                                               0.0s
=> CACHED [ 3/12] COPY requirements.txt requirements.txt                                                           0.0s
=> CACHED [ 4/12] RUN pip3 install -r requirements.txt                                                             0.0s
=> [ 5/12] COPY app.py app.py                                                                                      0.2s
=> [ 6/12] RUN mkdir static                                                                                        0.8s
=> [ 7/12] RUN mkdir static/css                                                                                    0.7s
=> [ 8/12] COPY static/css/main.css static/css/main.css                                                            0.0s
=> [ 9/12] RUN mkdir templates                                                                                     0.6s
=> [10/12] COPY templates/index.html templates/index.html                                                          0.0s
=> [11/12] COPY templates/base.html templates/base.html                                                            0.0s
=> [12/12] COPY templates/view_records.html templates/view_records.html                                            0.0s
=> exporting to image                                                                                              0.1s
=> => exporting layers                                                                                             0.1s
=> => writing image sha256:0456dca6f796b8541e164a88ffdc4fb5553e25bd7d38ef1fb319f3f3a3fe1746                        0.0s
=> => naming to docker.io/library/ue20cs351-cc-project-microservice-communication-using-rabbitmq-producer           0.0s
[+] Running 6/6
 ✓ Network ue20cs351-cc-project-microservice-communication-using-rabbitmq_default  Created                         0.3s
 ✓ Container consumer2                                                             Create...                       0.5s
 ✓ Container producer                                                              Created                         0.5s
 ✓ Container consumer4                                                             Create...                       0.5s
 ✓ Container consumer3                                                             Create...                       0.4s
 ✓ Container consumer1                                                             Create...                       0.5s
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 server instead.
producer   |  * Running on all addresses (0.0.0.0)
producer   |  * Running on http://127.0.0.1:5000
producer   |  * Running on http://172.21.0.6:5000
producer   | Press CTRL+C to quit
producer   |  * Restarting with stat
producer   |  * Debugger is active!
producer   |  * Debugger PIN: 329-311-243
```

3. Providing Postman the URL of the index.html file will allow you to check the home page. Here, we can obtain the result in two different formats: raw form and preview form, which displays how the web page will appear when it is deployed.



GET http://localhost:5000

http://localhost:5000

GET http://localhost:5000

Params ● | Authorization | Headers (7) | Body | Pre-request Script | Tests | Settings

Query Params

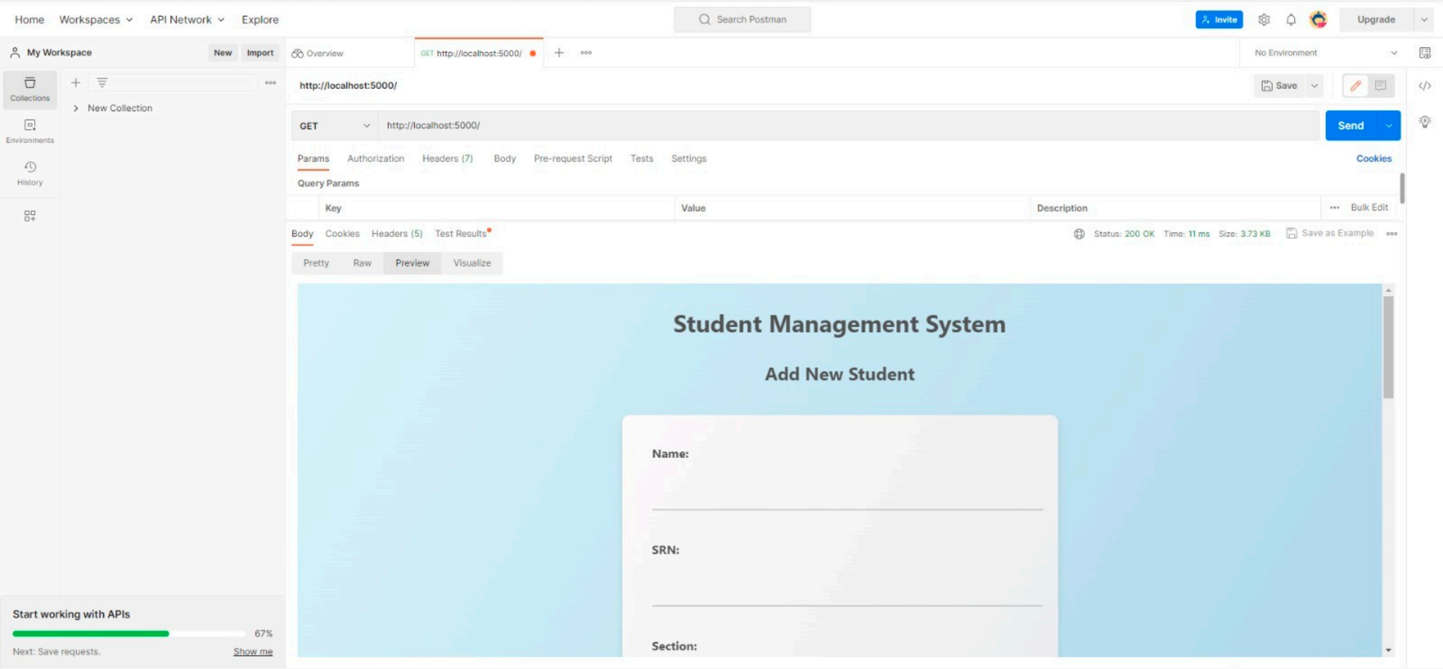
	Key	Value	Description
□	name	lagan	
□	section	j	
□	message	hey are you alive?	

Body | Cookies | Headers (5) | Test Results

Status: 200 OK Time: 16 ms Size: 1.33 KB

Pretty | Raw | Preview | Visualize

```
<!DOCTYPE html>
<html>
<head>
    <title>Student Management System</title>
</head>
<body>
    <h1>Student Management System</h1>
    <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">
    </form>
    <h2>View All Students</h2>
```



http://localhost:5000/

GET http://localhost:5000/

Status: 200 OK Time: 11 ms Size: 3.73 KB

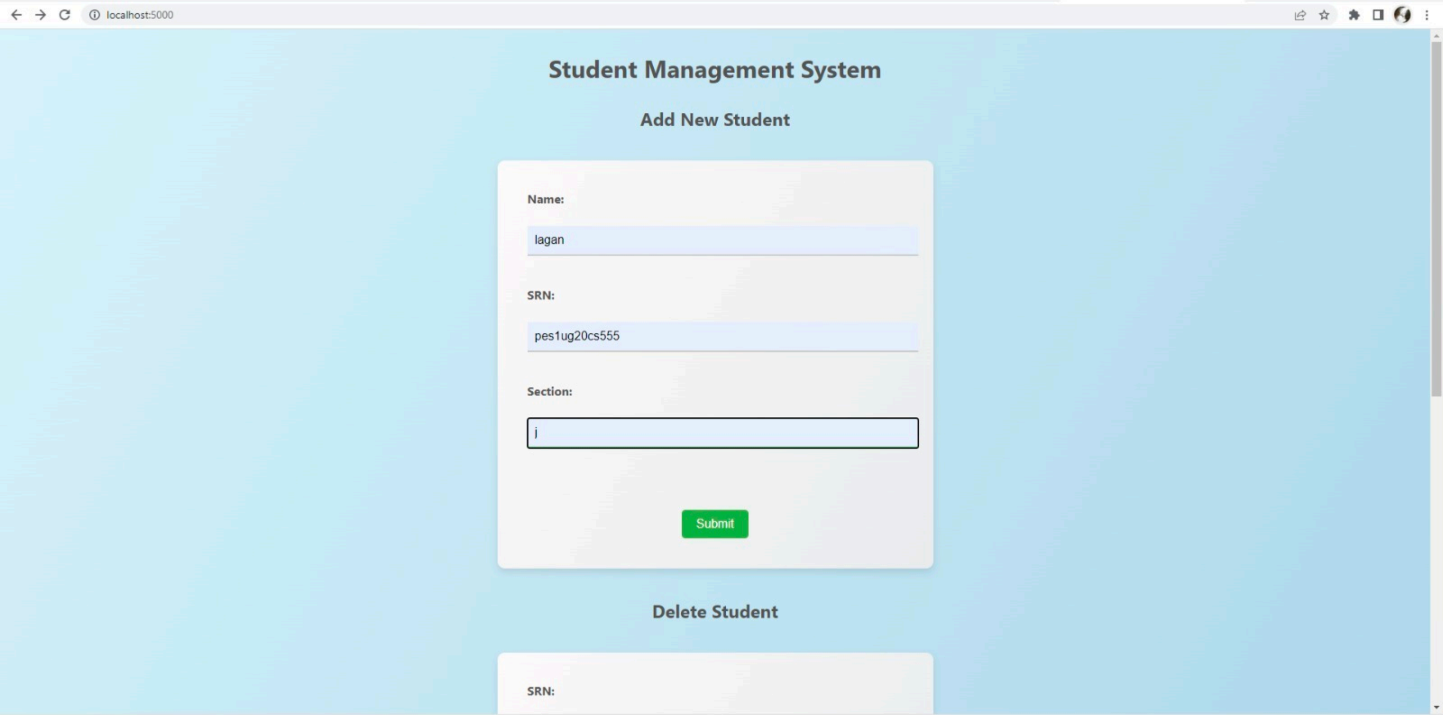
Student Management System

Add New Student

Name:

SRN:

Section:



localhost:5000

Student Management System

Add New Student

Name: lagan

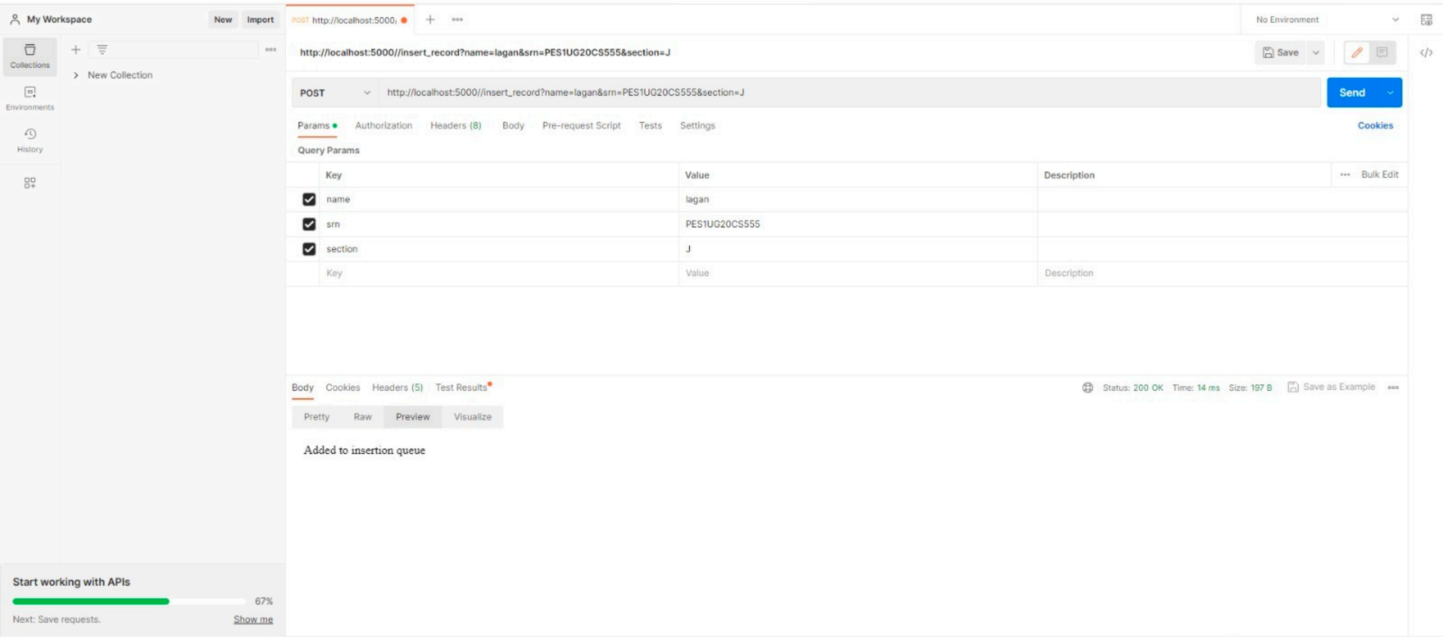
SRN: pes1ug20cs555

Section: j

Delete Student

SRN:

4. Adding a record using ‘insert_record’.



http://localhost:5000//insert_record?name=lagan&srn=PES1UG20CS555&section=J

POST http://localhost:5000//insert_record?name=lagan&srn=PES1UG20CS555&section=J

Params ● Authorization Headers (8) Body Pre-request Script Tests Settings

Query Params

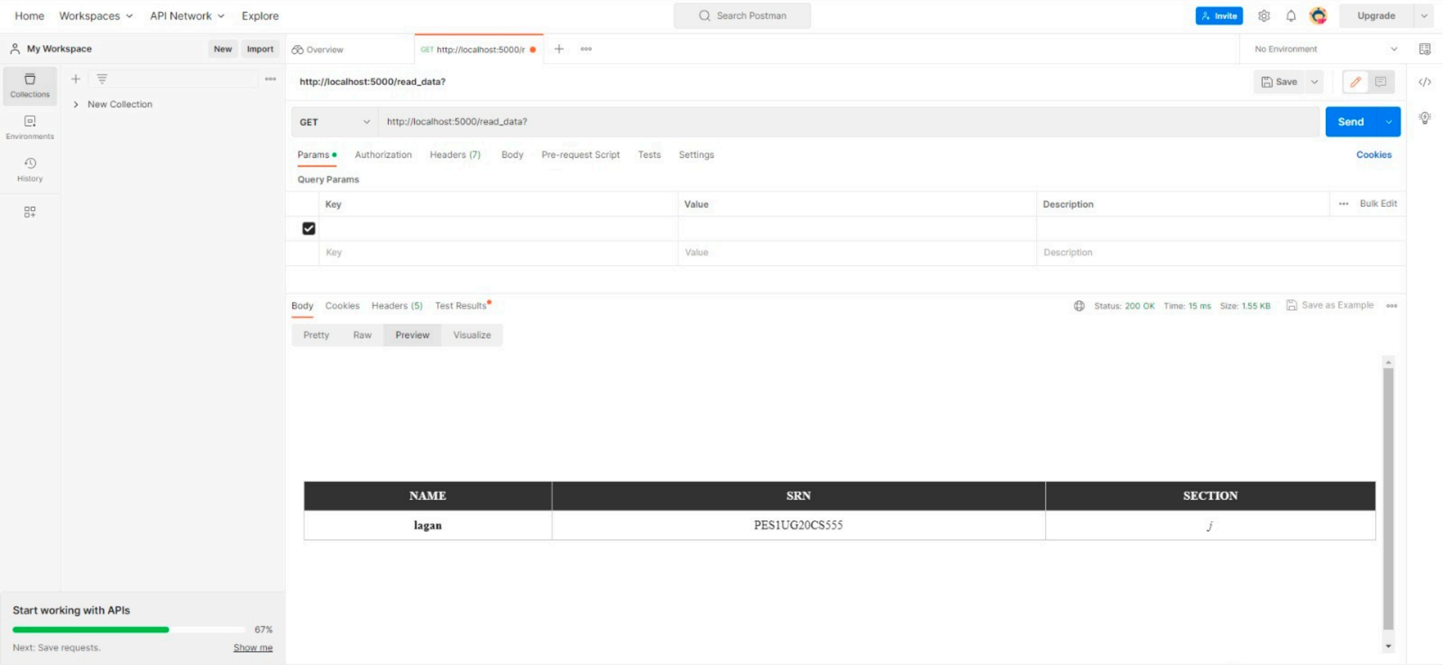
	Key	Value	Description
☑	name	lagan	
☑	srn	PES1UG20CS555	
☑	section	J	
	Key	Value	Description

Body Cookies Headers (5) Test Results

Status: 200 OK Time: 14 ms Size: 197 B

Pretty Raw Preview Visualize

Added to insertion queue



http://localhost:5000/read_data?

GET http://localhost:5000/read_data?

Params ● Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	Key	Value	Description
☑			
	Key	Value	Description

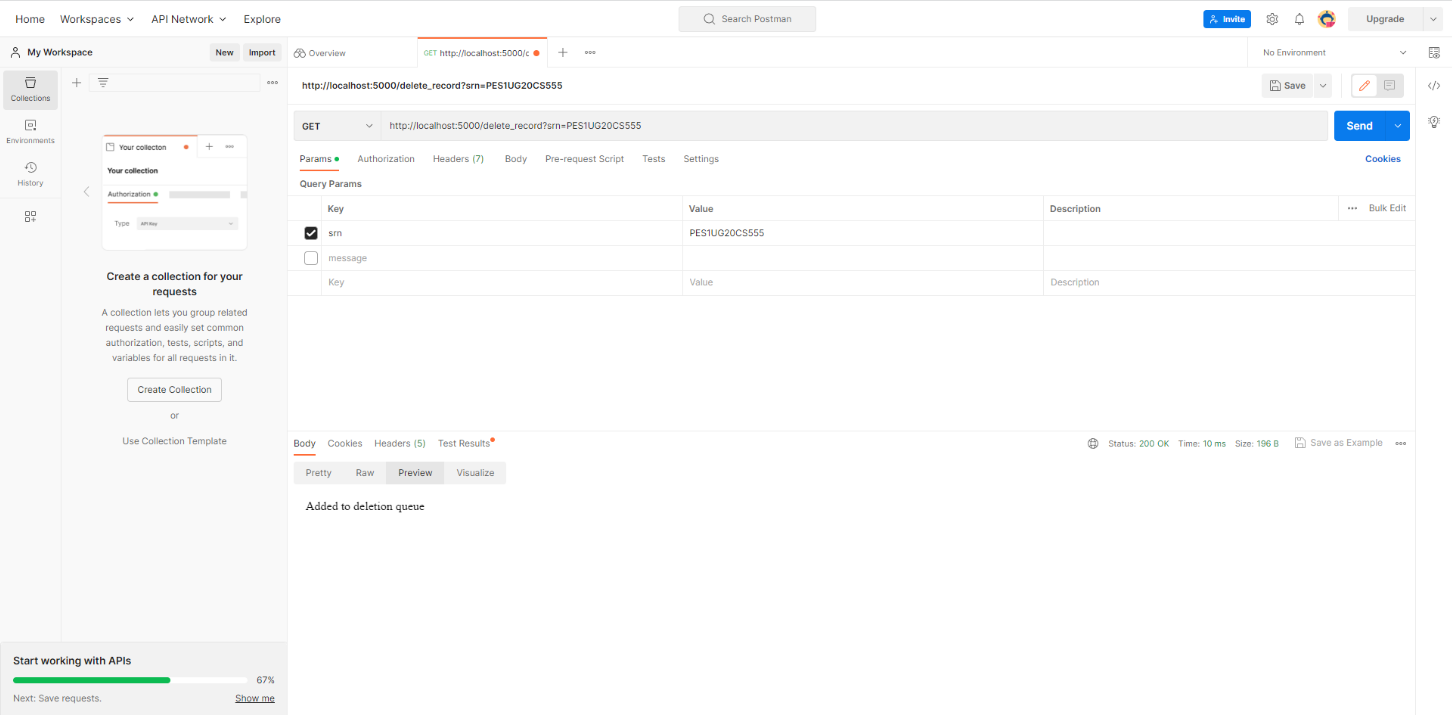
Body Cookies Headers (5) Test Results

Status: 200 OK Time: 15 ms Size: 1.55 KB

Pretty Raw Preview Visualize

NAME	SRN	SECTION
lagan	PES1UG20CS555	j

5. Deleting a record with the specified SRN using ‘delete_record’.



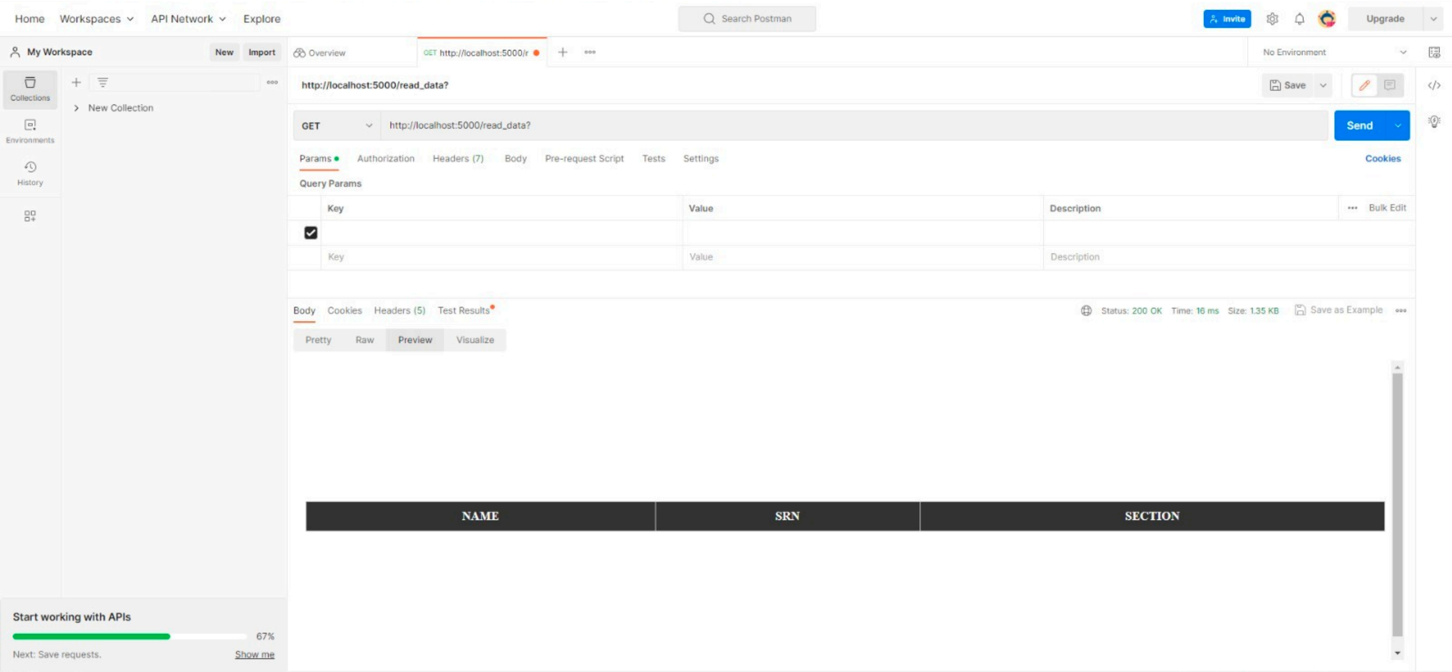
GET http://localhost:5000/delete_record?srn=PES1UG20CS555

Query Params

	Key	Value	Description
☑	srn	PES1UG20CS555	
□	message		
	Key	Value	Description

Status: 200 OK Time: 10 ms Size: 196 B

```
Added to deletion queue
```



GET http://localhost:5000/read_data?

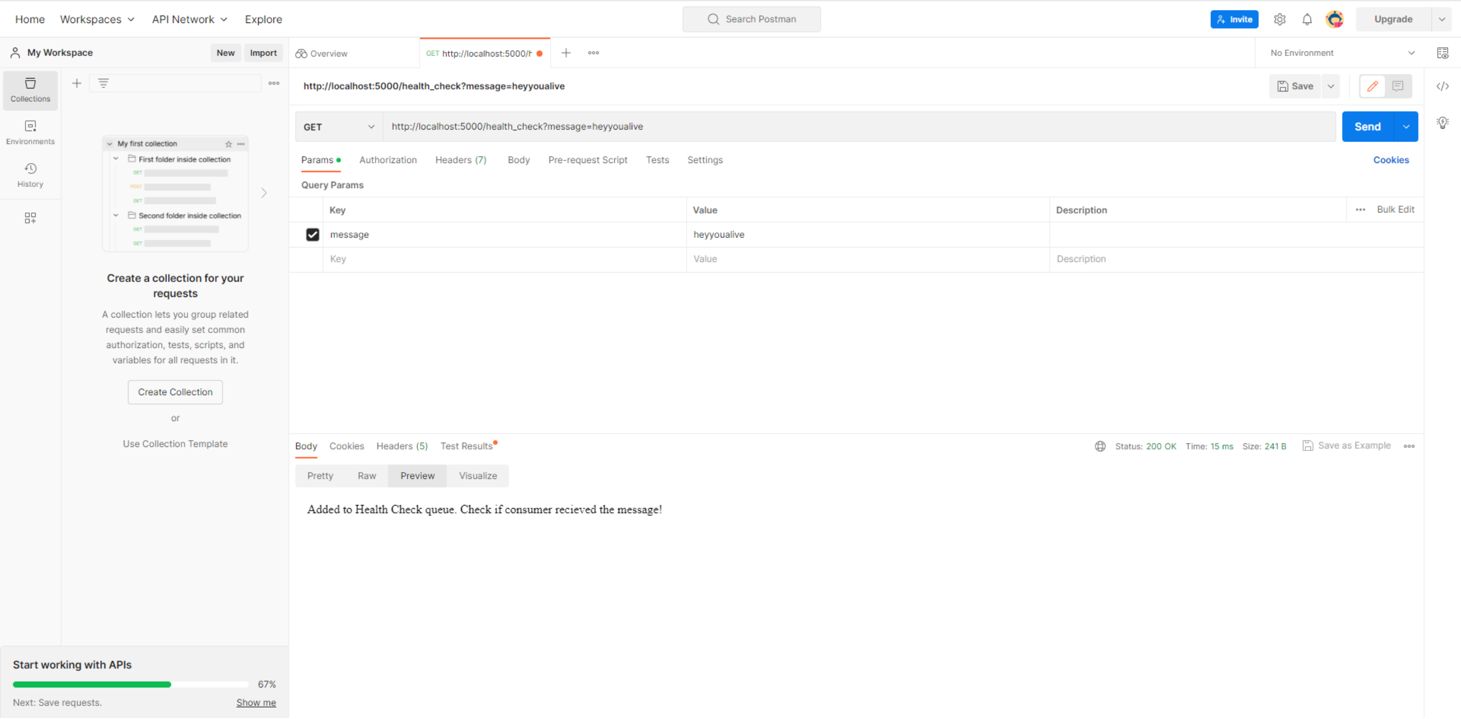
Query Params

	Key	Value	Description
☑			
	Key	Value	Description

Status: 200 OK Time: 16 ms Size: 1.35 KB

NAME	SRN	SECTION

6. Sending health check messages to consumer.



http://localhost:5000/health_check?message=heyyoualive

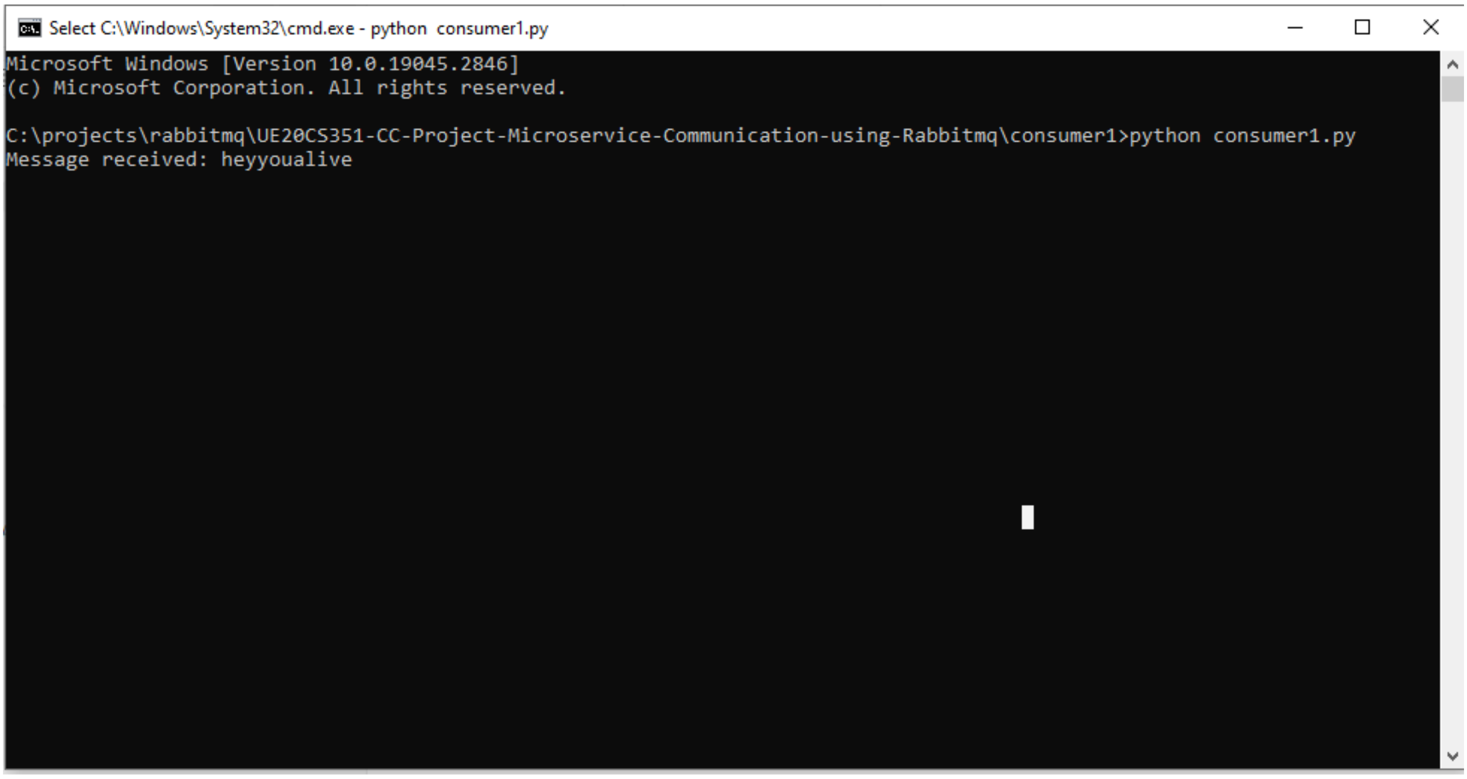
GET http://localhost:5000/health_check?message=heyyoualive

Query Params

	Key	Value	Description
☑	message	heyyoualive	
	Key	Value	Description

Status: 200 OK Time: 15 ms Size: 241 B

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



```
Microsoft Windows [Version 10.0.19045.2846]
(c) Microsoft Corporation. All rights reserved.

C:\projects\rabbitmq\UE20CS351-CC-Project-Microservice-Communication-using-Rabbitmq\consumer1>python consumer1.py
Message received: heyyoualive
```

Results and Conclusions

To handle communication between the various components, we successfully created a microservices architecture in this project utilising RabbitMQ as a message broker. Using RabbitMQ as a messaging system, we developed four microservices to carry out CRUD operations on a student management database.

Additionally, we used Docker to containerize the application, and we made a docker- compose file that runs all the microservices in addition to the MySQL database.

We utilised Postman, a well-liked API testing tool, to test the application. Each microservice's API endpoints were examined to make sure they were operational and engaged in proper communication with the message broker.

Overall, this project serves as an example of the versatility and strength of the microservices design, which enables us to create intricate, distributed systems that can grow and change over time. By utilising RabbitMQ as a message broker, we can make sure that messages are sent swiftly and dependably amongst the various microservices, enhancing the system's overall performance.

In conclusion, leveraging RabbitMQ, this project offers a strong basis for creating comparable microservices-based applications.