Cloud Computing - Mini Project Report
Microservice communication with RabbitMQ

April 2023

Submitted By:

NAME	SRN
HARSHINI BS	PES2UG20CS135
HIMA SR	PES2UG20CS142
VAMSI KRISHNA	PES2UG20CS149
MAHITHA SRI ADDANKI	PES2UG20CS183

VI Semester Section : C

PES University

Short Description and Scope of the Project

This project aims to design an architecture for performing simple CRUD operations as microservices. Microservice architecture is extensively used in industrial and large-scale deployments of various applications and software's. This is because it allows loose coupling and also installs the necessary dependencies automatically from the system libraries. Microservice architecture also allows easy compartmentalisation of responsibilities, by isolating the functions of every compartment in the system.

It involves a group of small, independent services. Each of these services must perform a single business logic. Additional logic can be added to the architecture by creating additional independent services that cater that logic specifically. Thus, microservices enable scalability of applications with ease.

RabbitMQ is a message-queueing software. It is also known as Message Broker and it has the responsibility of queueing messages and requests in their respective channels/queues so as to send them to their respective recipients.

For our project, we have used the following tools and software:

- Docker
- Pika - RabbitMQ's Python implementation using AMQP Protocol
- Python - Language used
- Testing suites - cURL & Insomnia
- Database - MongoDB Atlas (MongoDB Cloud Services)

Methodology

This project has 6 components to it:

1. **Producer** - Producers routes all API requests and enqueues them correctly in their respective queues for their respective recipients.
2. **Consumer_one** - This component is used to verify the health of the container. It responds to GET requests and consumes messages from the queue ‘health_check’.
3. **Consumer_two** - This component consumes messages from the ‘insert_record’ queue. Its messages are POST requests that include a record to be inserted into a database. We have used mongoDB Atlas(which is mongoDB’s Cloud Service) as our database. A collection called “*student*” is defined from the program under the database “*studentdb*” in our Cluster.
4. **Consumer_three** - This component consumes messages from the “delete_record” queue. Its messages are GET requests that only include the SRN of the student to be deleted. The mongoDB function *db.collection.delete_one()* is used for this purpose.
5. **Consumer_four** - This component consumes messages from the “read_database” queue. It receives GET requests from the producer and performs the function *db.collection.find({})* to retrieve all records from the database.
6. **Docker_compose.yml** - This file is used to dockerize all the above services as individual components of a system. It includes the commands necessary to build every service, along with its dependencies installed within the container. It includes 2 parts to it:

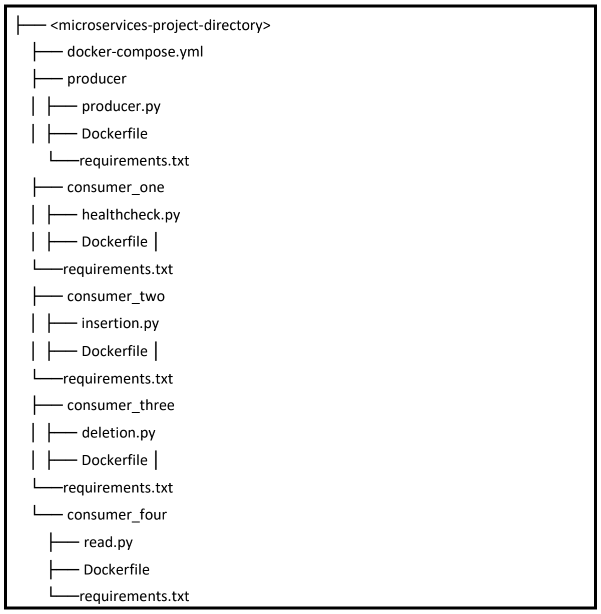
   Version

   Services

   Services include all the abovementioned services and includes details on their volumes, builds and ports, etc.

   Every component has its respective requirements.txt file and Dockerfile. Requirements.txt includes all the python modules that are needed for that particular service. The Dockerfile is needed to run the necessary commands to build and start the container.

These components have been arranged according to the specified directory structure:



```
├── <microservices-project-directory>
  ├── docker-compose.yml
  ├── producer
  │ ├── producer.py
  │ ├── Dockerfile
    └──requirements.txt
  ├── consumer_one
  │ ├── healthcheck.py
  │ ├── Dockerfile │
  └──requirements.txt
  ├── consumer_two
  │ ├── insertion.py
  │ ├── Dockerfile │
  └──requirements.txt
  ├── consumer_three
  │ ├── deletion.py
  │ ├── Dockerfile │
  └──requirements.txt
  └── consumer_four
    ├── read.py
    ├── Dockerfile
    └──requirements.txt
```

Every component is connected to Pika Connection defined on host=’rabbitmq’. All channels and queues are declared with this host. For each consumer, a prefetch_count = 1 is defined which indicates that at any point in time, the number of queued requests can only be 1. This avoids the need to purge the queue after processing requests.

The Flask app is defined in the producer and it is what routes requests to their respective queues. We have defined delivery_mode = 2 while publishing messages in the producer. This indicates that the message must be made persistent. Messages marked as 'persistent' that are delivered to 'durable' queues will be logged to disk. Durable queues are recovered in the event of a crash, along with any persistent messages they stored prior to the crash.

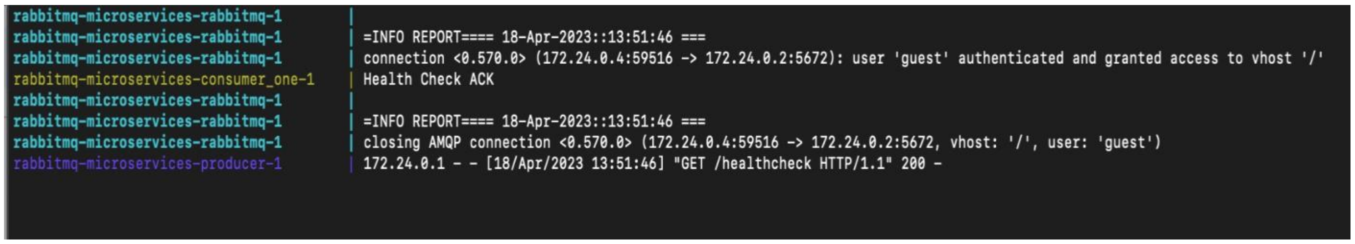
Testing

This project has been tested using Curl and Insomnia.

Postman did not allow requests to be sent to localhost, and hence we decided to use Insomnia.

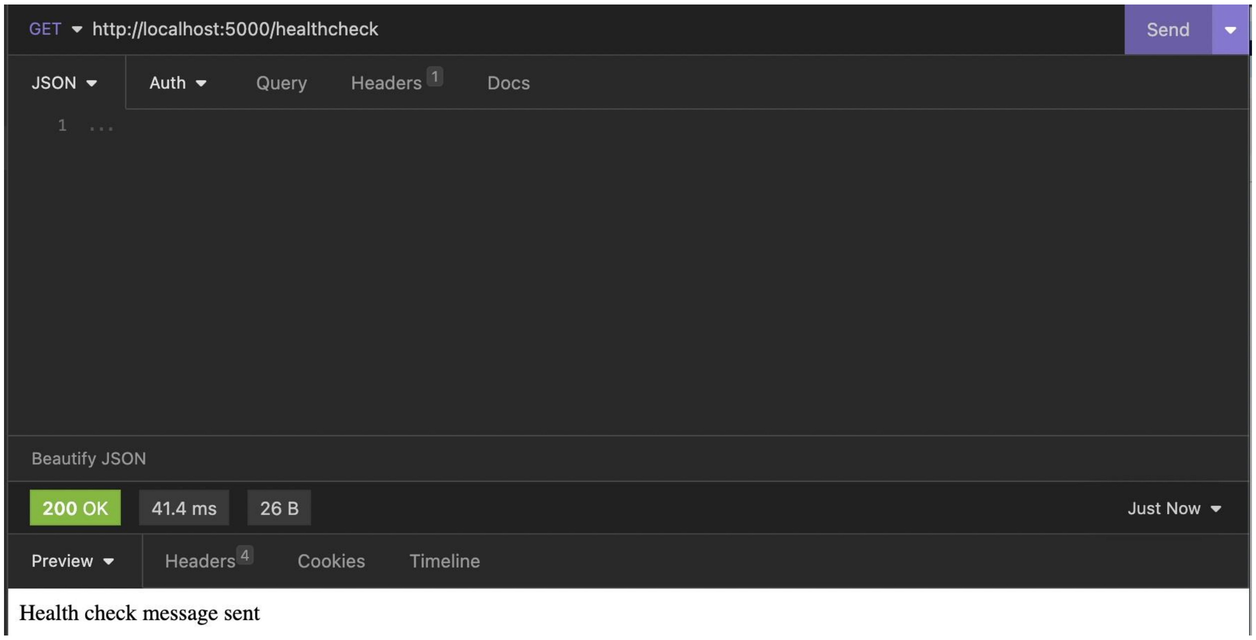
1). **Consumer_one:**

-Using cURL:



```
rabbitmq-microservices-rabbitmq-1         |
rabbitmq-microservices-rabbitmq-1         | =INFO REPORT==== 18-Apr-2023::13:51:46 ===
rabbitmq-microservices-rabbitmq-1         | connection <0.570.0> (172.24.0.4:59516 -> 172.24.0.2:5672): user 'guest' authenticated and granted access to vhost '/'
rabbitmq-microservices-consumer_one-1     | Health Check ACK
rabbitmq-microservices-rabbitmq-1         |
rabbitmq-microservices-rabbitmq-1         | =INFO REPORT==== 18-Apr-2023::13:51:46 ===
rabbitmq-microservices-rabbitmq-1         | closing AMQP connection <0.570.0> (172.24.0.4:59516 -> 172.24.0.2:5672, vhost: '/', user: 'guest')
rabbitmq-microservices-producer-1         | 172.24.0.1 - - [18/Apr/2023 13:51:46] "GET /healthcheck HTTP/1.1" 200 -
```

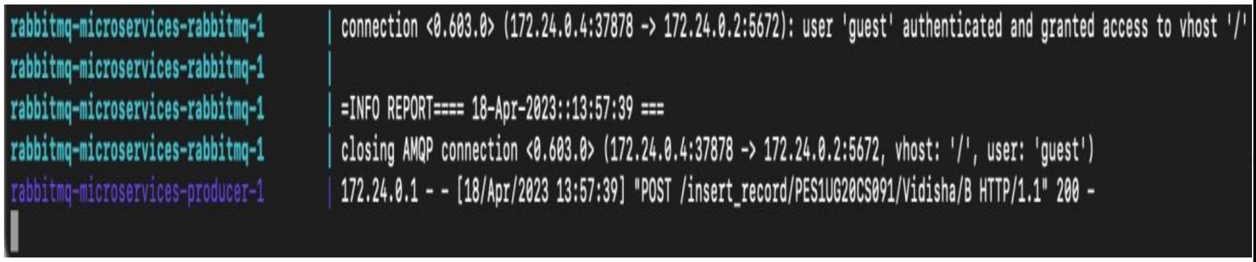
-Using Insomnia:



```
GET http://localhost:5000/healthcheck
200 OK   41.4 ms   26 B   Just Now
Health check message sent
```

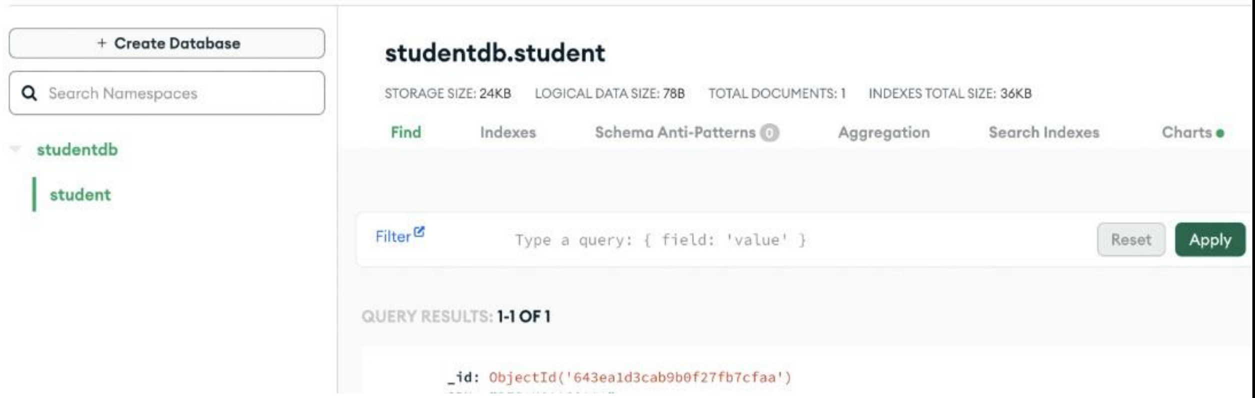
2). **Consumer_two:**

-Using cURL:



```
rabbitmq-microservices-rabbitmq-1      | connection <0.603.0> (172.24.0.4:37878 -> 172.24.0.2:5672): user 'guest' authenticated and granted access to vhost '/'
rabbitmq-microservices-rabbitmq-1      |
rabbitmq-microservices-rabbitmq-1      | =INFO REPORT==== 18-Apr-2023::13:57:39 ===
rabbitmq-microservices-rabbitmq-1      | closing AMQP connection <0.603.0> (172.24.0.4:37878 -> 172.24.0.2:5672, vhost: '/', user: 'guest')
rabbitmq-microservices-producer-1      | 172.24.0.1 - - [18/Apr/2023 13:57:39] "POST /insert_record/PES1UG20CS091/Vidisha/B HTTP/1.1" 200 -
```

Database:



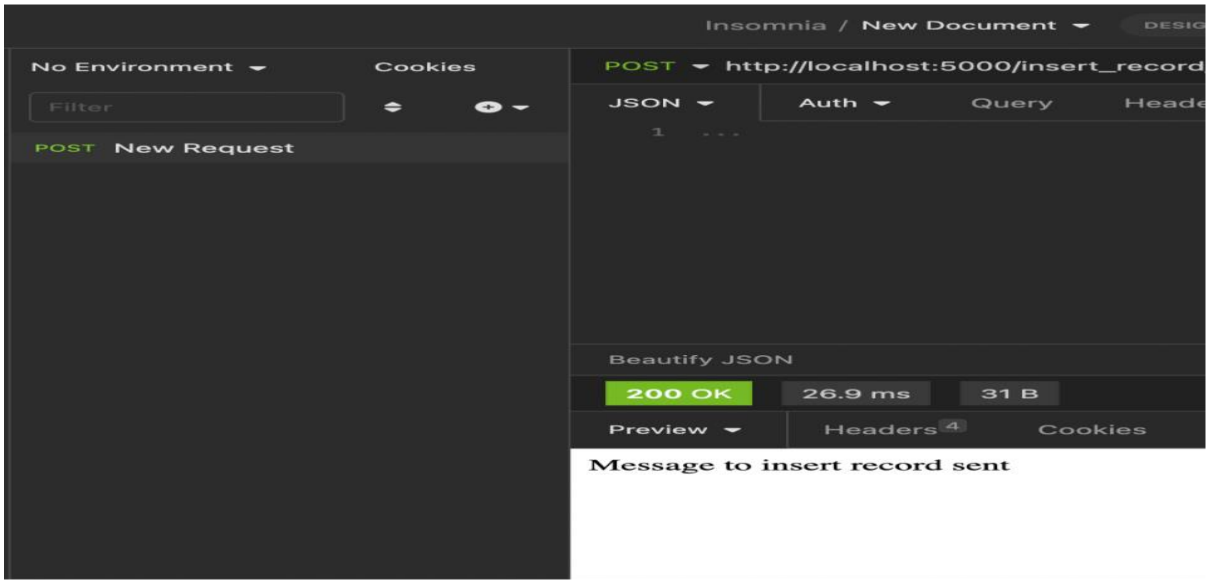
studentdb.student

STORAGE SIZE: 24KB LOGICAL DATA SIZE: 78B TOTAL DOCUMENTS: 1 INDEXES TOTAL SIZE: 36KB

QUERY RESULTS: 1-1 OF 1

```
_id: ObjectId('643ea1d3cab9b0f27fb7cfaa')
```

-Using Insomnia:


Insomnia / New Document

No Environment Cookies

POST New Request

POST http://localhost:5000/insert_record

JSON Auth Query Heade

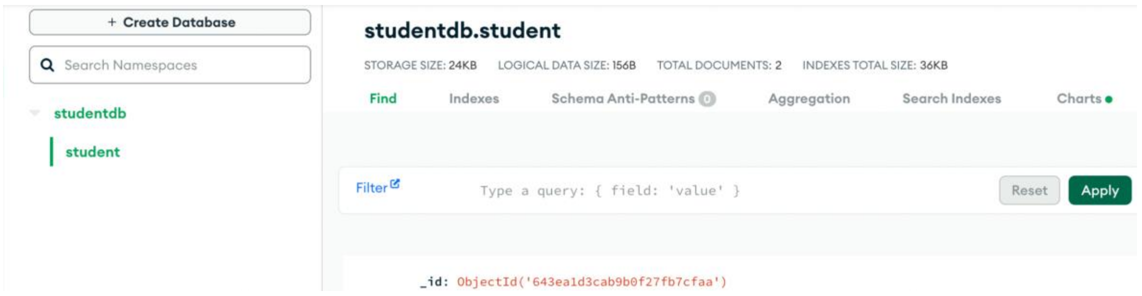
Beautify JSON

200 OK 26.9 ms 31 B

Preview Headers 4 Cookies

Message to insert record sent

Database:


studentdb.student

STORAGE SIZE: 24KB LOGICAL DATA SIZE: 156B TOTAL DOCUMENTS: 2 INDEXES TOTAL SIZE: 36KB

studentdb

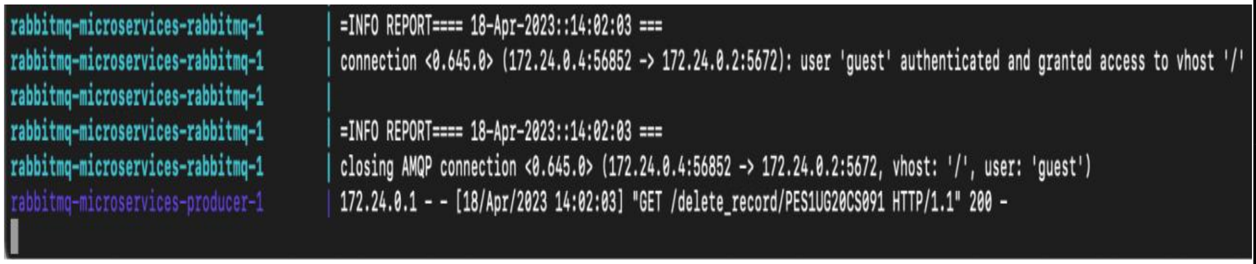
student

Filter Type a query: { field: 'value' } Reset Apply

```
_id: ObjectId('643ea1d3cab9b0f27fb7cfaa')
```

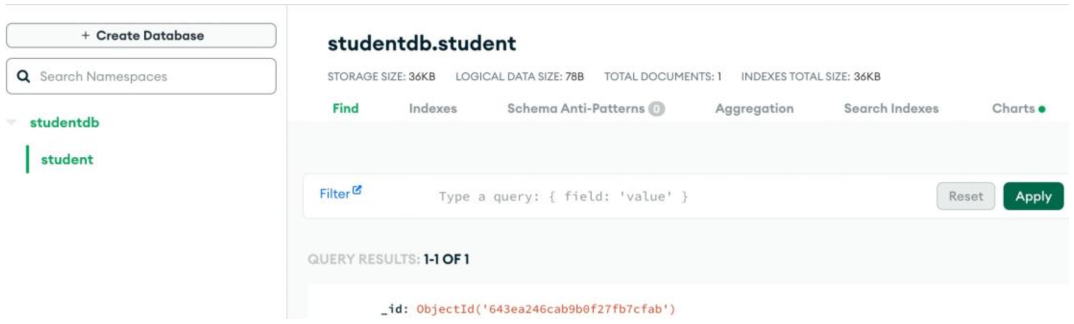
3). **Consumer_three:**

-Using cURL:



```
rabbitmq-microservices-rabbitmq-1     | =INFO REPORT==== 18-Apr-2023::14:02:03 ===
rabbitmq-microservices-rabbitmq-1     | connection <0.645.0> (172.24.0.4:56852 -> 172.24.0.2:5672): user 'guest' authenticated and granted access to vhost '/'
rabbitmq-microservices-rabbitmq-1     |
rabbitmq-microservices-rabbitmq-1     | =INFO REPORT==== 18-Apr-2023::14:02:03 ===
rabbitmq-microservices-rabbitmq-1     | closing AMQP connection <0.645.0> (172.24.0.4:56852 -> 172.24.0.2:5672, vhost: '/', user: 'guest')
rabbitmq-microservices-producer-1     | 172.24.0.1 - - [18/Apr/2023 14:02:03] "GET /delete_record/PES1UG20CS091 HTTP/1.1" 200 -
```

Database:



studentdb.student

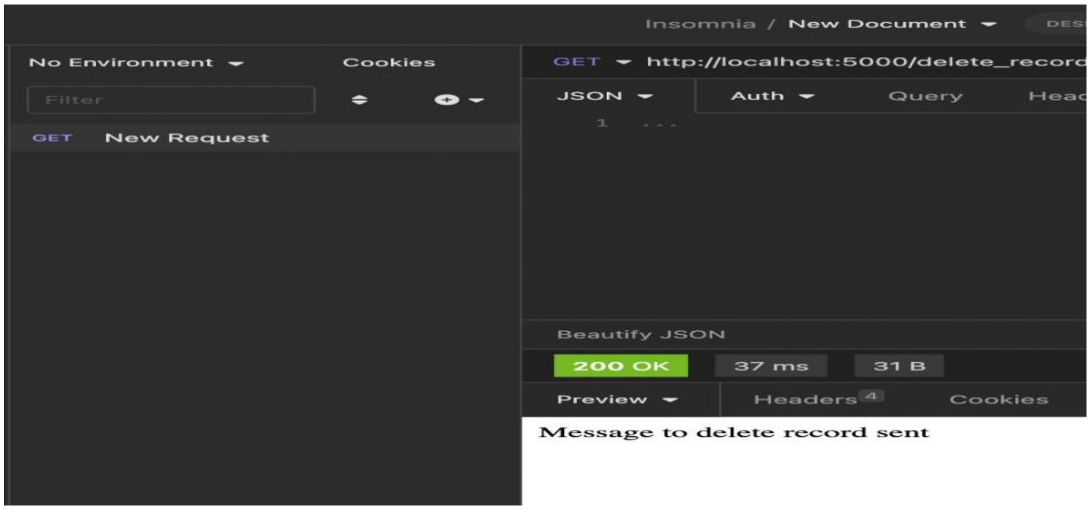
STORAGE SIZE: 36KB   LOGICAL DATA SIZE: 78B   TOTAL DOCUMENTS: 1   INDEXES TOTAL SIZE: 36KB

Filter   Type a query: { field: 'value' }

QUERY RESULTS: 1-1 OF 1

_id: ObjectId('643ea246cab9b0f27fb7cfab')

-Using Insomnia:

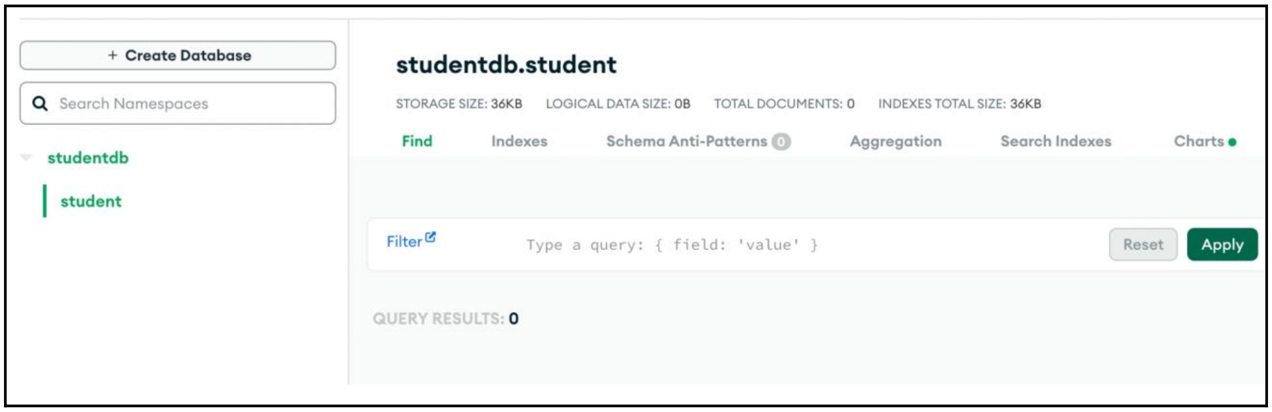


GET http://localhost:5000/delete_record

200 OK   37 ms   31 B

Message to delete record sent

Database:



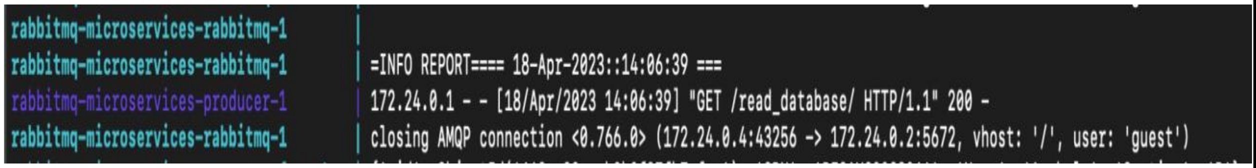
studentdb.student

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 0B TOTAL DOCUMENTS: 0 INDEXES TOTAL SIZE: 36KB

QUERY RESULTS: 0

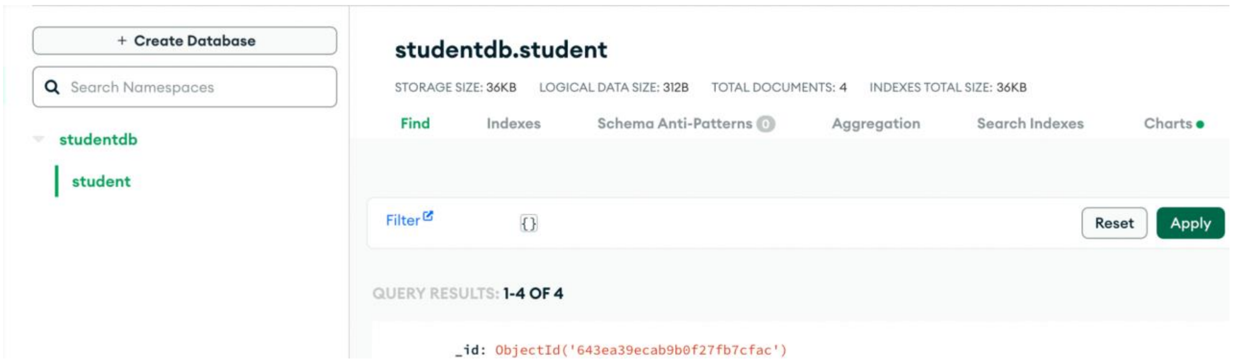
4).**Consumer_four**:

-Using cURL:



```
rabbitmq-microservices-rabbitmq-1       |
rabbitmq-microservices-rabbitmq-1       | =INFO REPORT==== 18-Apr-2023::14:06:39 ===
rabbitmq-microservices-producer-1       | 172.24.0.1 - - [18/Apr/2023 14:06:39] "GET /read_database/ HTTP/1.1" 200 -
rabbitmq-microservices-rabbitmq-1       | closing AMQP connection <0.766.0> (172.24.0.4:43256 -> 172.24.0.2:5672, vhost: '/', user: 'guest')
```

-Database:

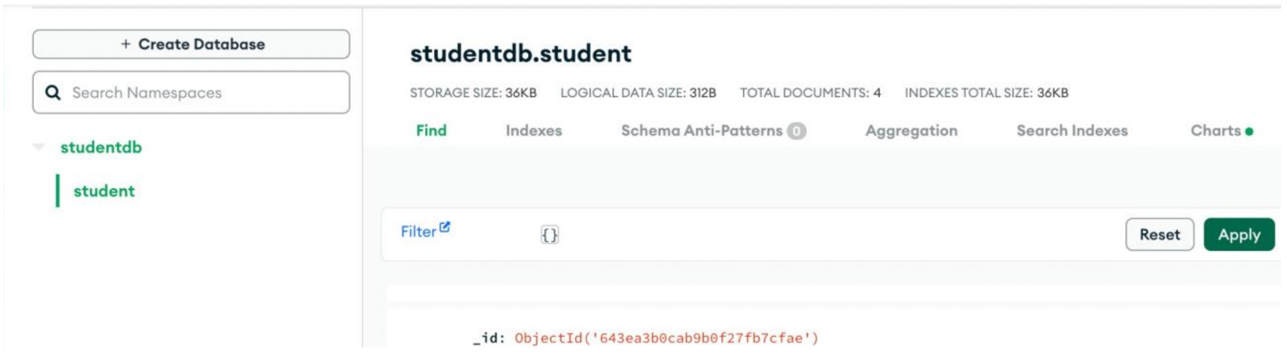


studentdb.student

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 312B TOTAL DOCUMENTS: 4 INDEXES TOTAL SIZE: 36KB

QUERY RESULTS: 1-4 OF 4

```
_id: ObjectId('643ea39ecab9b0f27fb7cfac')
```



studentdb.student

STORAGE SIZE: 36KB LOGICAL DATA SIZE: 312B TOTAL DOCUMENTS: 4 INDEXES TOTAL SIZE: 36KB

```
_id: ObjectId('643ea3b0cab9b0f27fb7cfae')
```

Results and Conclusions

This project allowed us to study and understand microservices architecture using Docker. It taught us how to establish coupling and interaction between the various modules/services of the container without affecting the host system.

Docker was used to manage all the different services that were set up. The docker-compose.yml was used to build and manage all the various modules. Each module was individually built using its Dockerfile. The Dockerfile necessitated that all the python libraries be defined in a separate file, for ease of installation and hence, all of these modules were listed in requirements.txt.

RabbitMQ can be implemented using many protocols. But we have chosen

Pika, because it was the most widely distributed RabbitMQ protocol for Python. The implementation of Pika and the importance of queues, hosts, routing and APIs was made significant during the course of this project.

MongoDB Atlas was used as the database. This taught us the ease with which cloud services can handle Data-Intensive CRUD operations without having to install the specific software or build a new container for it in our host systems. Now this database can be accessed by the users from anywhere without worrying about its system requirements, configuration , memory management, CPU architecture, etc.