Distributed Systems (521290S)
Course Project Report
Team Tesla

Member	Student ID	Email	Track
Md Mobusshar Islam	2305578	mislam23@student.oulu.fi	Corporate
Muhammad Ahmed	2304796	mahmed23@student.oulu.fi	Corporate
Muhammad Talha Arshad	2304797	Muhammad.arshad@student.oulu.fi	Corporate

Github：https://github.com/Mobusshar/distributed-systems-project-msc-oulu-2024.git

Course project overview

Overview

The project aims to process the request to scrape the data from websites according to the instructions of the requesting nodes (users) and extract useful/required information from the data. The following are the major components of the system.

Distributed System Topics Covered

- **Message Queuing System**:
  - Round Robin Queuing:

    Distributes incoming requests evenly among available server resources in a circular manner.
- **Load Balancing:**
  - Round Robin Load Balancing:

    Distributes incoming network traffic or requests evenly across multiple servers.
  - Weighted Round Robin:

Assigns different weights to servers based on their weights assigned to the servers, allowing for more efficient resource utilization.

- Concurrent call:

  Based on the high requests, assigns values to the servers to overcome load of the client requests.

- **Priority Queue Management:**
  - RabbitMQ Queue System:

    Based on the type of request i.e. type 1, type 2, type 3, all the requests are sorted in their respective queues for processing
- **Resource Management:**
  - Queue system to handle requests sequentially

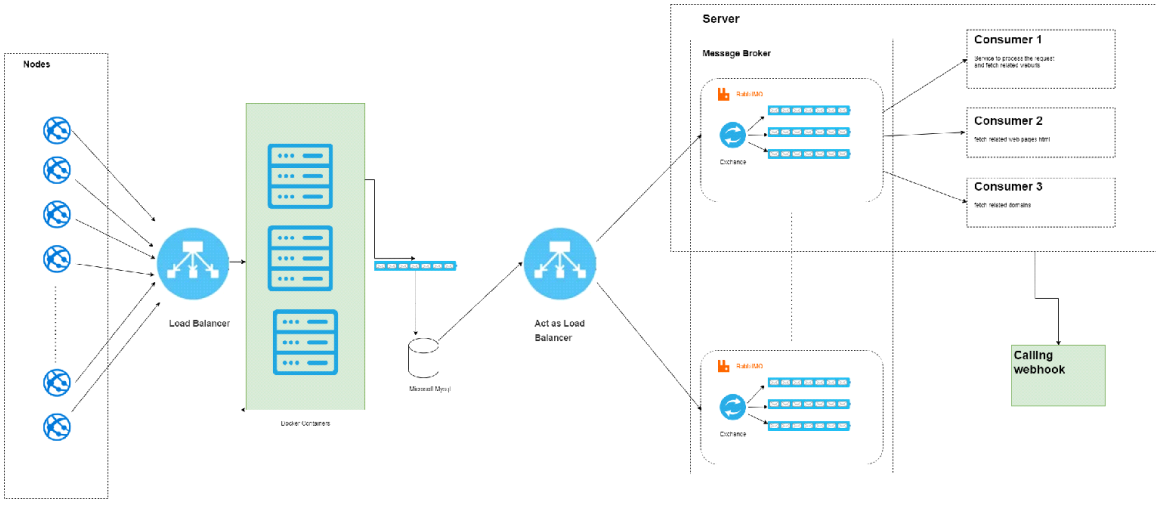
    Controls the rate at which requests are processed to prevent resources.

    Smooth outbursts of incoming requests, preventing sudden spikes in resource usage.

- **Fault Tolerance and Redundancy:**

  In the above-mentioned basic topics covered, we will be trying different approaches to find out the best possible solutions for a specific problem.

Architecture Diagram

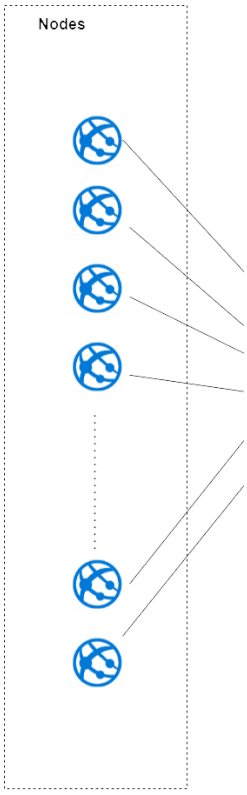


In our proposed **Event-Driven Architecture (EDA) style**, Nodes will be sending the requests to the load balancer and load balancer will send the client's requests to the servers based on the different algorithm like "Weighted round robin". We are making limited connections to the Database since database connections are limited and servers will be publishing the data through RabbitMQ queue system that will be consumed and stored in the database. Next, the logical **load balancer** will act as publisher and different consumers will subscribe to the respective queue maintained by the **RabbitMQ message broker**. Components in the system are **loosely coupled** as they interact with others through events rather than direct connections. Nodes having a connection with the central server are making client-server architecture in the system.

Following are **the roles of the components** of the system.

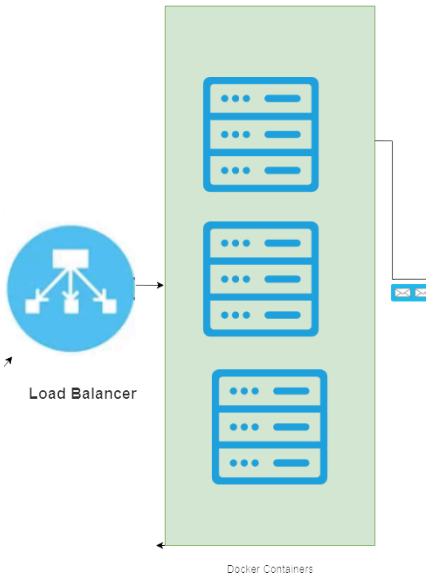
- **Requesting Nodes:**

Different nodes in the system will request the data to extract the data and, in this project, we will be mocking the role of nodes by some utility/script to generate a good number of requests.



- **Load Balancer:**

This component of the system will receive the requests from the nodes and here load balancer will route the traffic on different servers. Here different algorithm will be applied to test the load from clients.

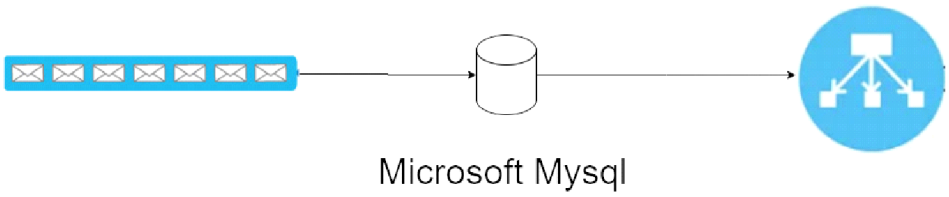


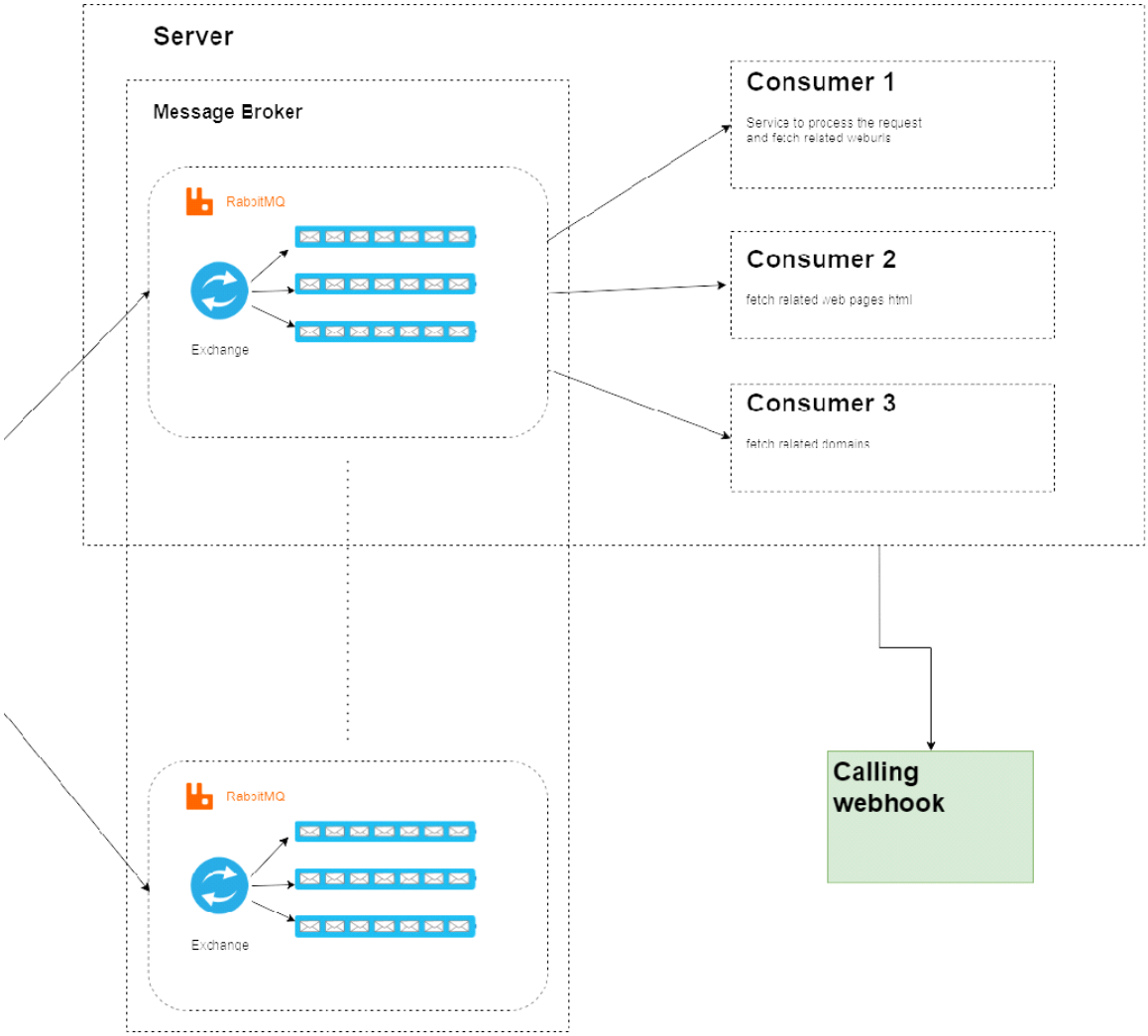
- **Central Server:**

It will put all the receiving messages in a message queue and all the requests will be registered in the database.

- **Message Broker**

We are implementing RabbitMQ as a message broker. This will use exchange to assign the requests inside a server to different queues based on the different types of consumers.


Microsoft Mysql


Server
Message Broker
RabbitMQ
Exchange
Consumer 1
Service to process the request and fetch related weburls
Consumer 2
fetch related web pages html
Consumer 3
fetch related domains
RabbitMQ
Exchange
Calling webhook

- **Consumers**

Based on different jobs to perform in our case, we have different consumers as follows

- Fetching related text data from the website
- Fetch related web pages HTML.
- Fetch related domains.

- **Callback_ WebHook**

  As the request is completed, the system will be sending the “OK” response to the client.

2. Implementation

For this system, we are using **python, C# as programming languages** and **MSSQL as a database**. **Docker** is used to containerize and run different servers on the same machine for demo purposes. **Nginx** is used to make load balancer to route the traffic to different servers.
Certainly, here's a more concise version:

1. Client-side Implementation:

- Use Python (Flast API) for backend APIs.
- Utilize C# (ASP.NET Core) for backend development.
- Ensure RESTful APIs for communication.

2. Server-side Implementation:

- Implement Python microservices and C# for main backend services.
- Use MSSQL for the database.
- Employ pyodbc for Python and Entity Framework Core for C# for database interaction.
- Implement robust error handling, authentication, and authorization.

3. Infrastructure Setup:

- Containerize components with Docker.
- Utilize Nginx as a reverse proxy and load balancer.

4. Hardware Considerations:

- Ensure nodes have sufficient resources.

- Consider cloud-based services for scalability.
- Implement redundancy.

Server Infrastructure: We operate our system on Docker containers distributed across three nodes. Each node functions as a server, providing the necessary computing resources for our application.

3. Communication

Design:

Components in our distributed system interact as follows:

Interaction pattern: Publish/Subscribe

Protocol: JSON messages

Interfaces: REST API

Communication protocol: HTTP over TCP/IP

User requests, initiated via **HTTP**, are routed through **Nginx** for load balancing. Requests then move through **RabbitMQ** using a **publisher/subscriber** model for seamless communication.

Processing occurs on **Dockerized servers** with **MS SQL** for data storage. Upon completion, a response is sent to the user. Callback notifications are sent via provided URLs.

4. Naming

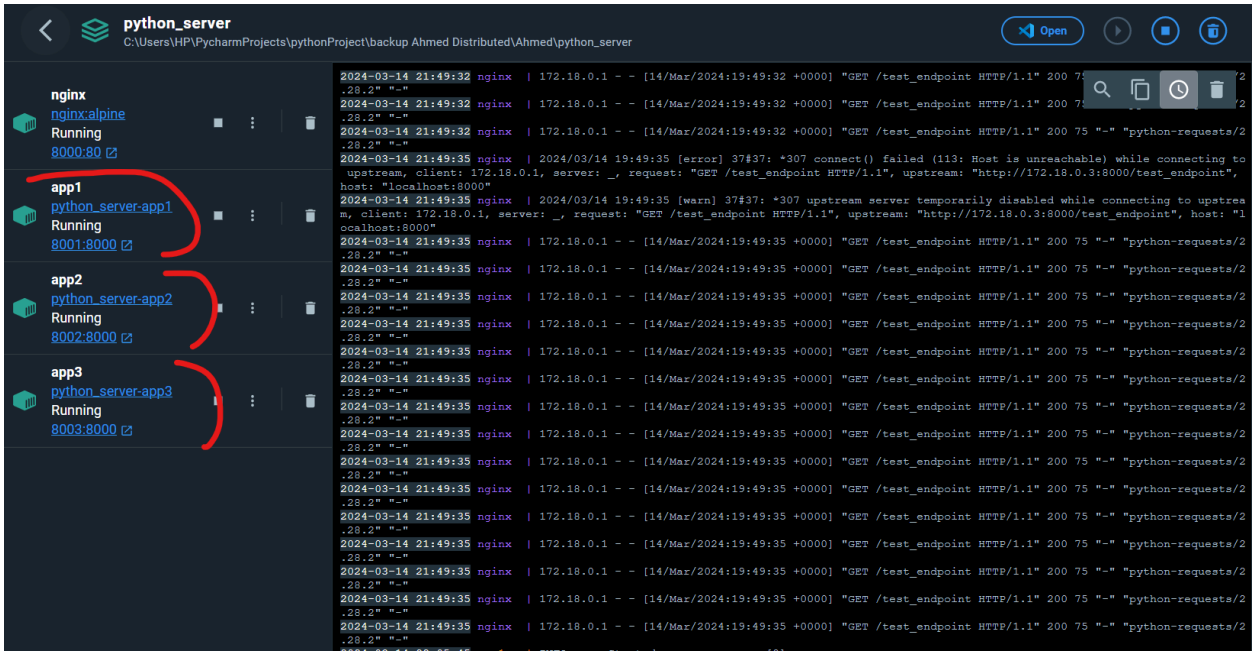
For HTTP requests, we are following structured naming conventions where users are sending requests to the servers. In the pub-sub model, attribute based naming is followed where we are assigning queues and exchange names in the for RabbitMQ.

6. Consistency and replication

We have developed the system in a data centric approach where Database ensures the flow of the system and in case of failure at any stage, data from the database is handling the data consistency and replication of processes where needed.

In our project, we've replicated our server into three instances for horizontal scaling. Each instance handles client requests, distributing them evenly using a round-robin technique for

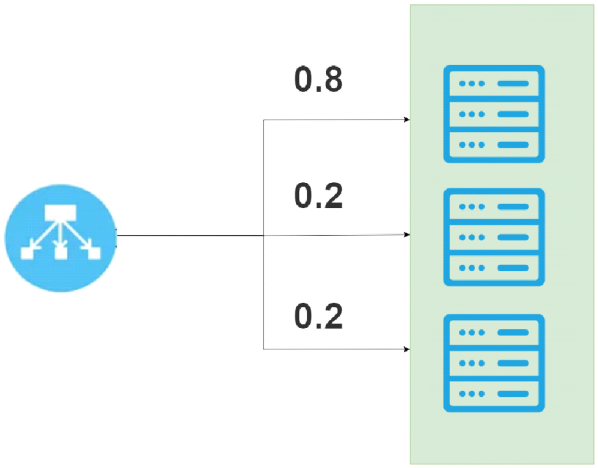
efficient traffic management. This setup enhances reliability and performance while enabling seamless scalability.



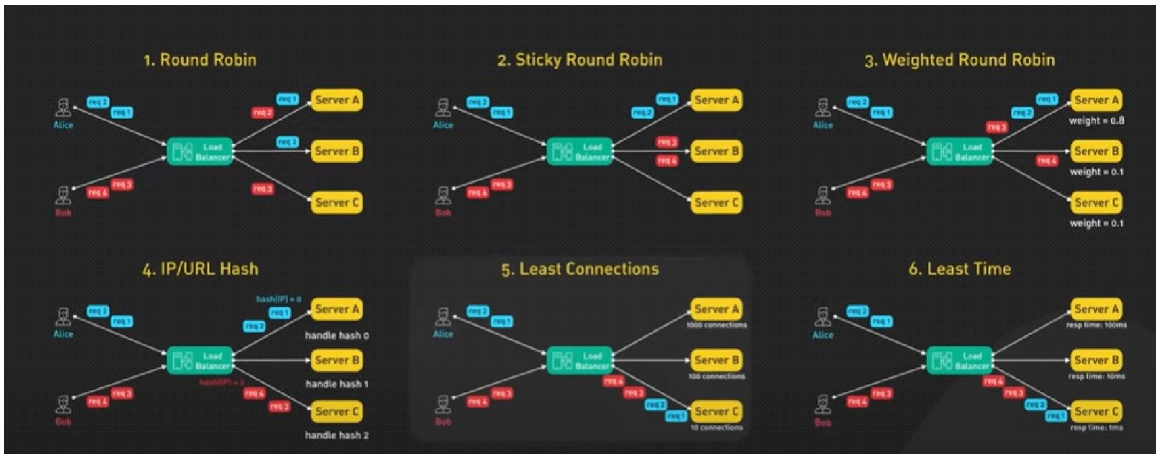
7. Fault tolerance

We have designed the system in such a way that at every step of communication, we are making the system fault tolerance, Lets go through the system step by step

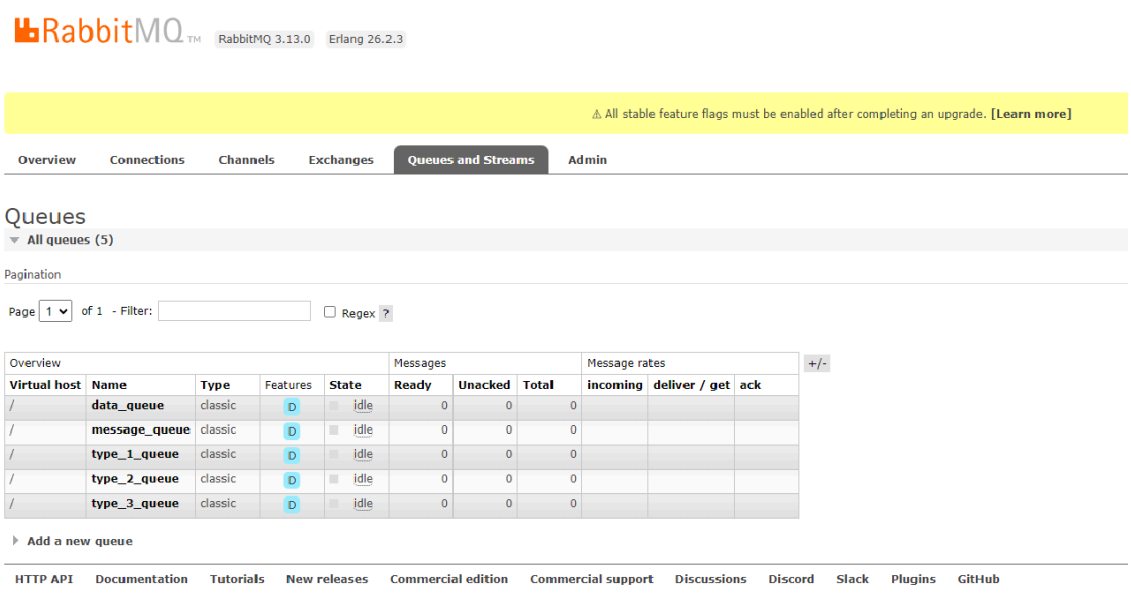
Clients send request to the system for processing the data, here we are using load balancer to handle and send requests to the servers. Three servers are up with load balancer to cater the user request and we have applied some load balancing algorithms like **"weighted round robin"**,



with 0.8, 0.2 and 0.2 weights. Here are some famous algorithms commonly used for load balancing



If any server goes down, still the system will be live and okay since requests will be sent to live two servers remaining. For example, If server with 0.8 is down, requests will be sent to servers 0.2 and 0.2 and in that case 50 percents of requests will be handled by each server. Moving forward, If in any case, DB connection gown down, request in the message queue will remain in the queue unless until consumed by the DB to store. For this purpose we are using RabbitMQ to handle message queue mechanism



Since DB connections are limited, so we are not exposing the clients directly to DB or servers directly to DB in case of async data handling, we are using RabbitMQ message queue system to make only one connection with DB to submit the requests gracefully.

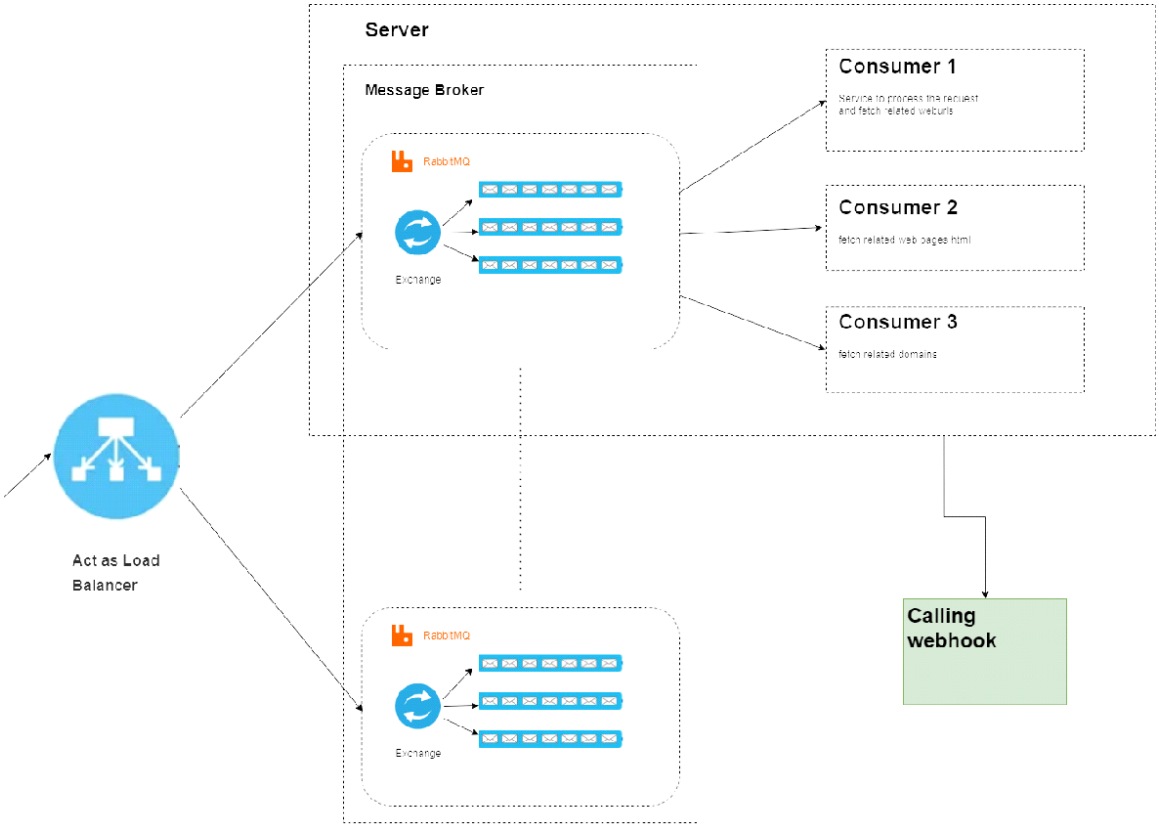
After storing data in DB, we are using a logical load balancer that publishes requests to all available consumers and in this case consumers are RabbitMQ message queues that use "exchange" of the RabbitMQ to assign the requests to different queues based on the type. Lets analyze the data stored in the DB for insights about the operation

Here is the RabbitMQ internal Logs and message mechanism that can used to visualize the system

Results Messages

	Id	user_Id	request_url	type	Is_processed	callback_url	date_added	date_process_complete	Is_processing
1	1073	92b149df-7f14-4846-8993-f587c085f450	varma.fi	1	1	NULL	2024-03-13 08:04:35.163	NULL	1

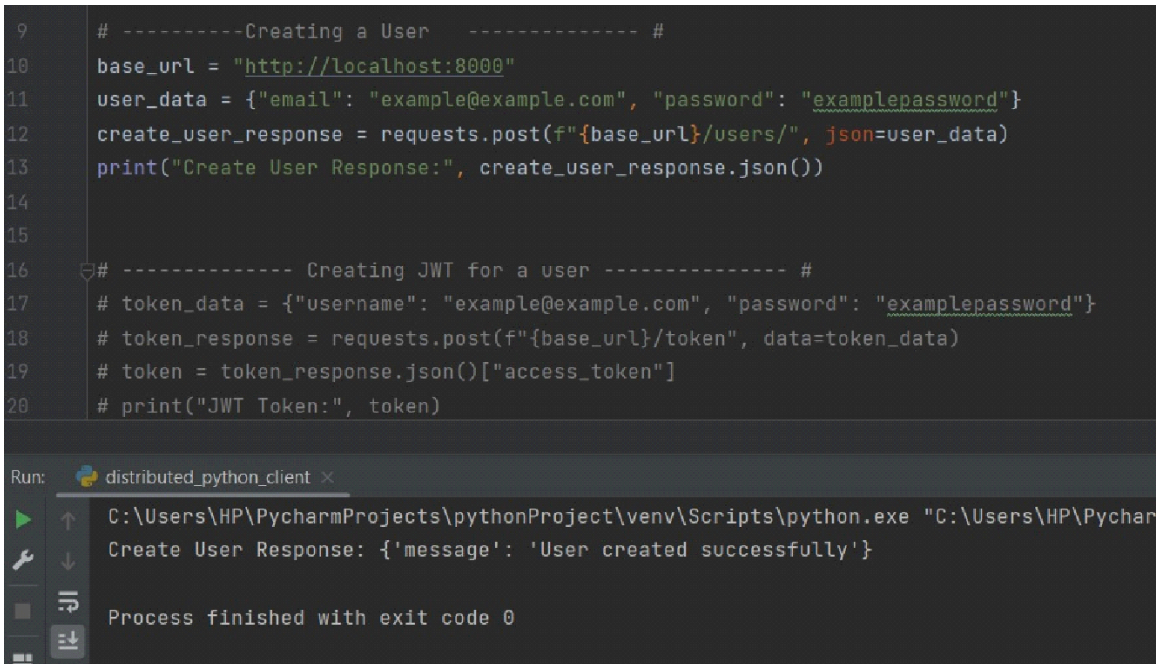
after storing the data in DB, data is published with **is_processing = true**, so logical load balancer after DB will only pick where **is_processing = false** and **is_processes = false.**



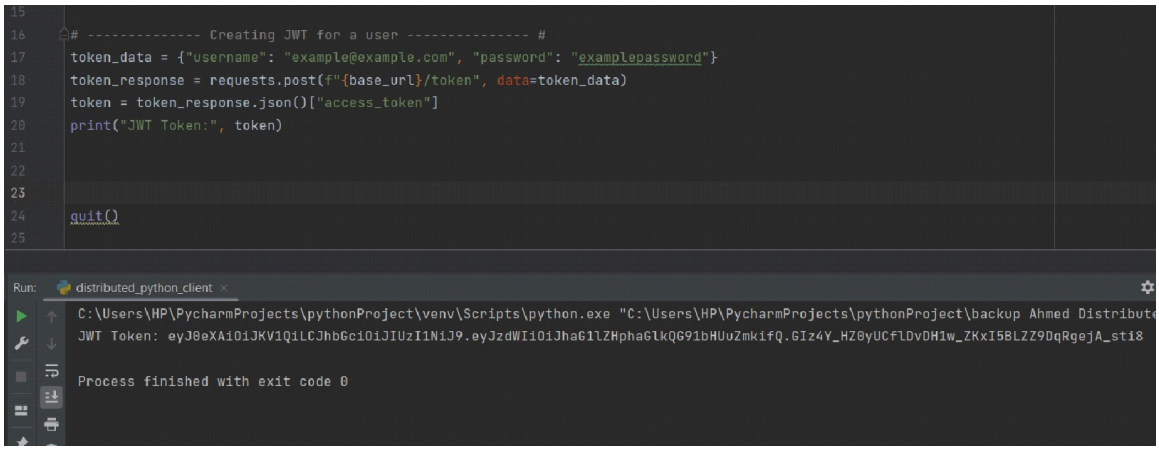
Let's assume, somehow all the RabbitMQ consumers broke, message queues will remain the queue until consumed so no data will be lost. Now, consider an other scenario where consumer has consumed the request but just before sending "OK" to call back url, system goes down, even in that case our is_processed bit will be 0 and we can re-initialize the request to scrap the data since scraping the data again is not a potential failure in our system.

8. Security

1. **Registration**: Users register via email and password securely hashes the password using **SHA56** and stores the user's data in a MySQL database.



2. **Authentication**: Once registered, users can log in using their credentials. Upon successful authentication, the system returns JWT (JSON Web Token) for authorization.



3. **Authorization**: Endpoints now called by including the token in the Authorization header of their requests.

9. Evaluation

For the analysis we have used different techniques how our traffic routes between our backend applications:

1- Round Robin

2- Weighted Round Robin

3- Least Connection

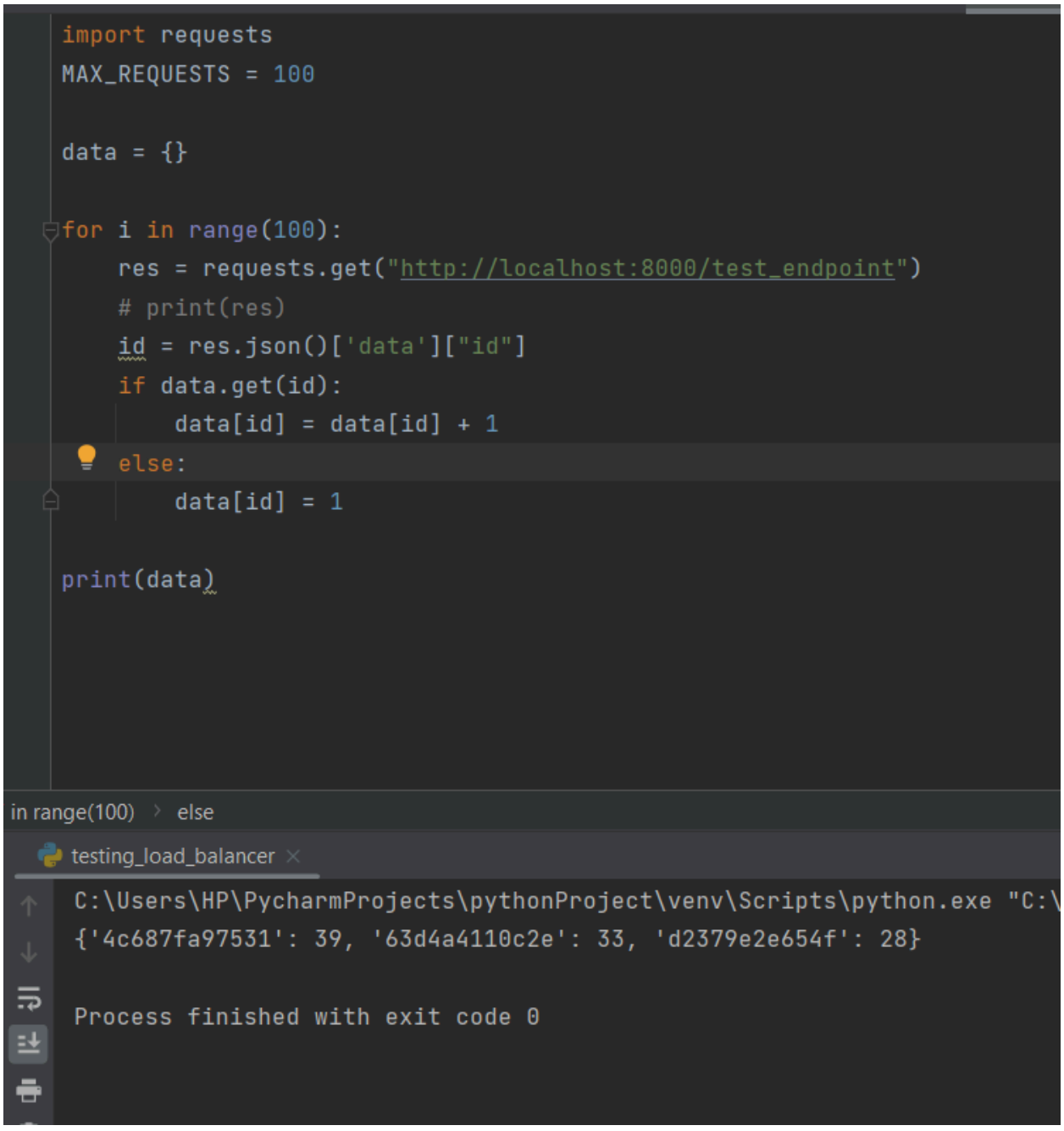
4- Parallel Calls

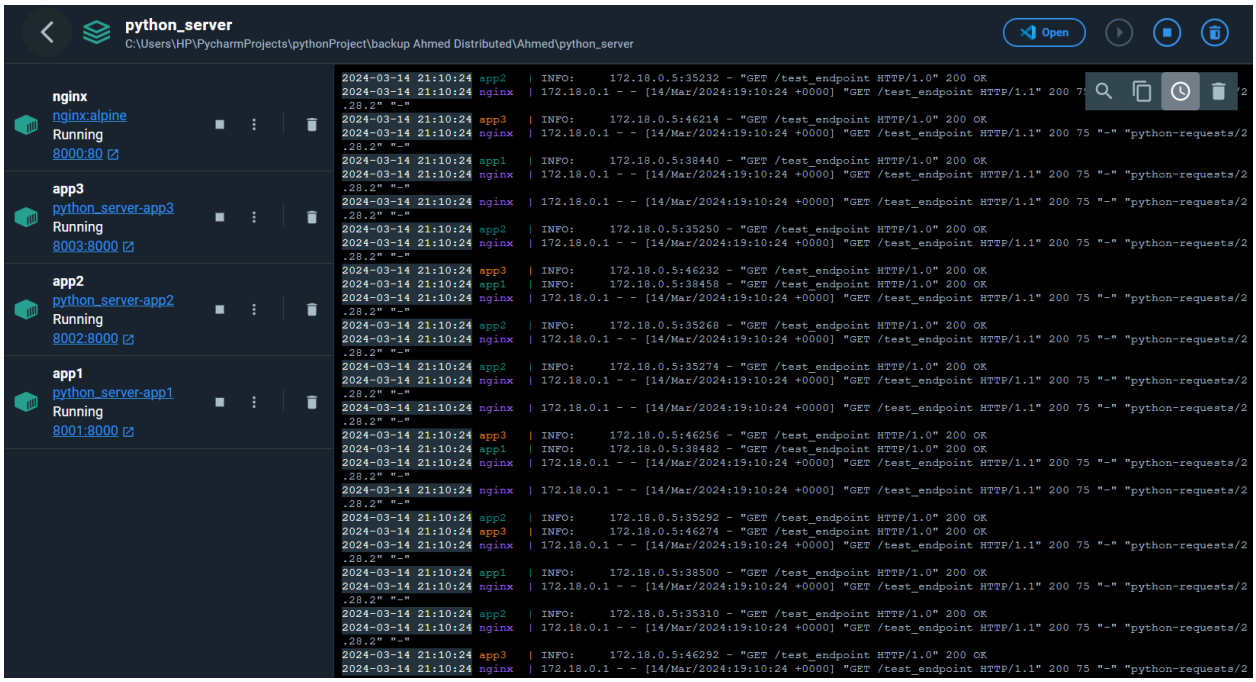
5- Making one container down to check how the traffic effects.

6- Temporary shutting down our Queue system and when the queue is up it will process the messages we have tested this out as well

Below are the screenshots how as we are testing out different test cases.

Round Robin:


import requests
MAX_REQUESTS = 100
data = {}
for i in range(100):
res = requests.get("http://localhost:8000/test_endpoint")
# print(res)
id = res.json()['data']["id"]
if data.get(id):
data[id] = data[id] + 1
else:
data[id] = 1
print(data)
in range(100) › else
testing_load_balancer
C:\Users\HP\PycharmProjects\pythonProject\venv\Scripts\python.exe "C:\
{'4c687fa97531': 39, '63d4a4110c2e': 33, 'd2379e2e654f': 28}
Process finished with exit code 0



Weighted Round Robin:

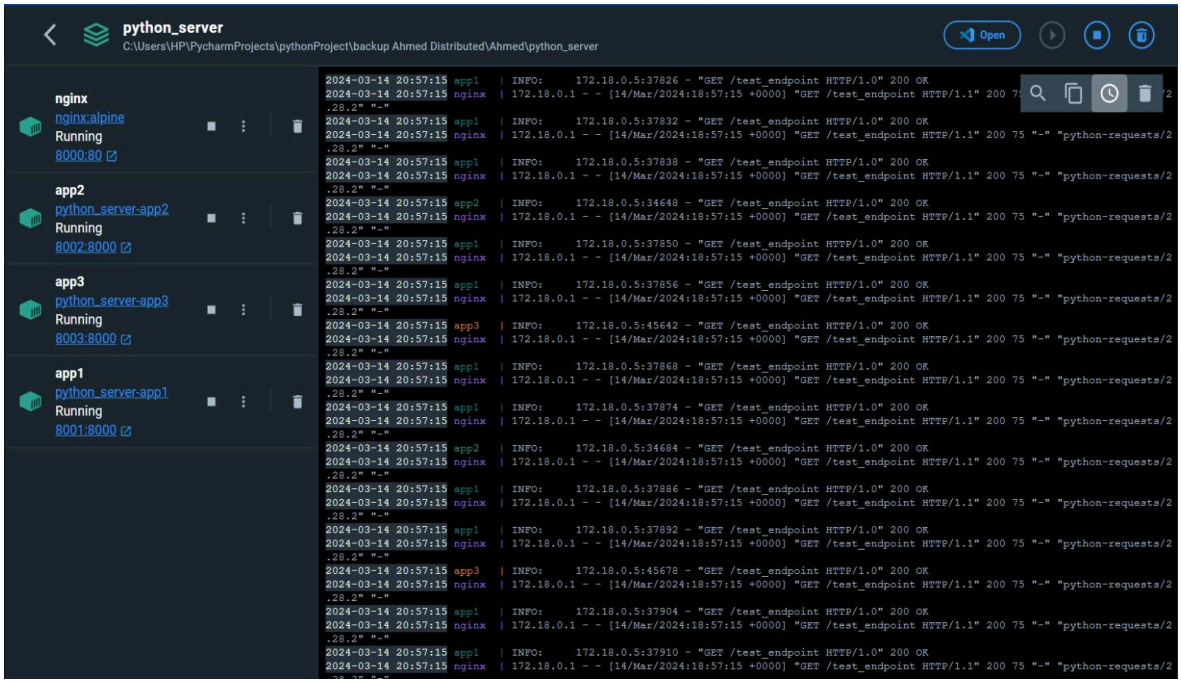
We are making 100 calls and you can see in the image 80 percent calls went on 1st docker container 10 percent on other and 10 percent.

distributed_python_client.py | testing_load_balancer.py | distributed_python_server.py | requirements.txt

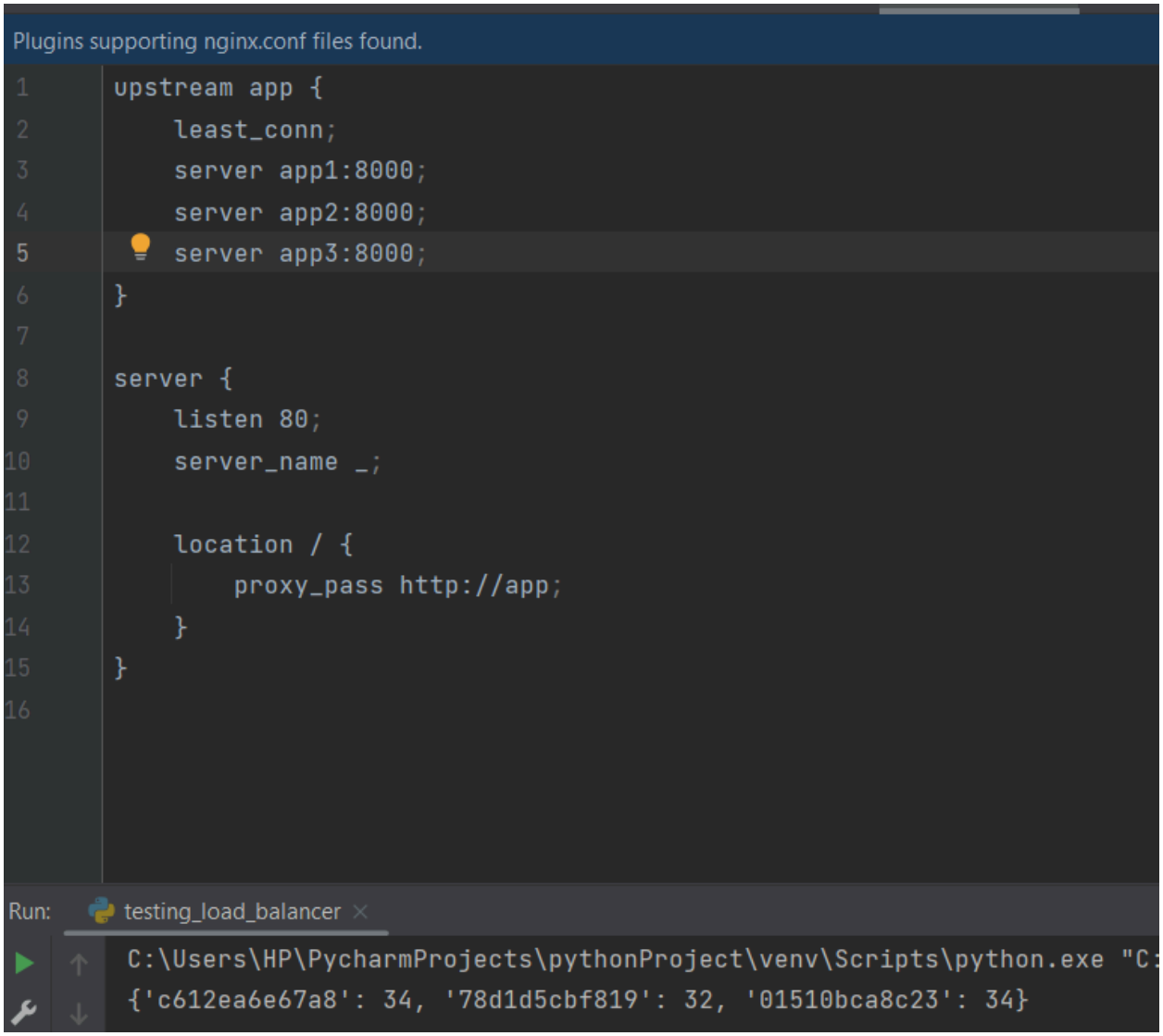
```python
import requests
MAX_REQUESTS = 100

data = {}

for i in range(100):
    res = requests.get("http://localhost:8000/test_endpoint")
    # print(res)
    id = res.json()['data']["id"]
    if data.get(id):
        data[id] = data[id] + 1
    else:
        data[id] = 1

print(data)
```

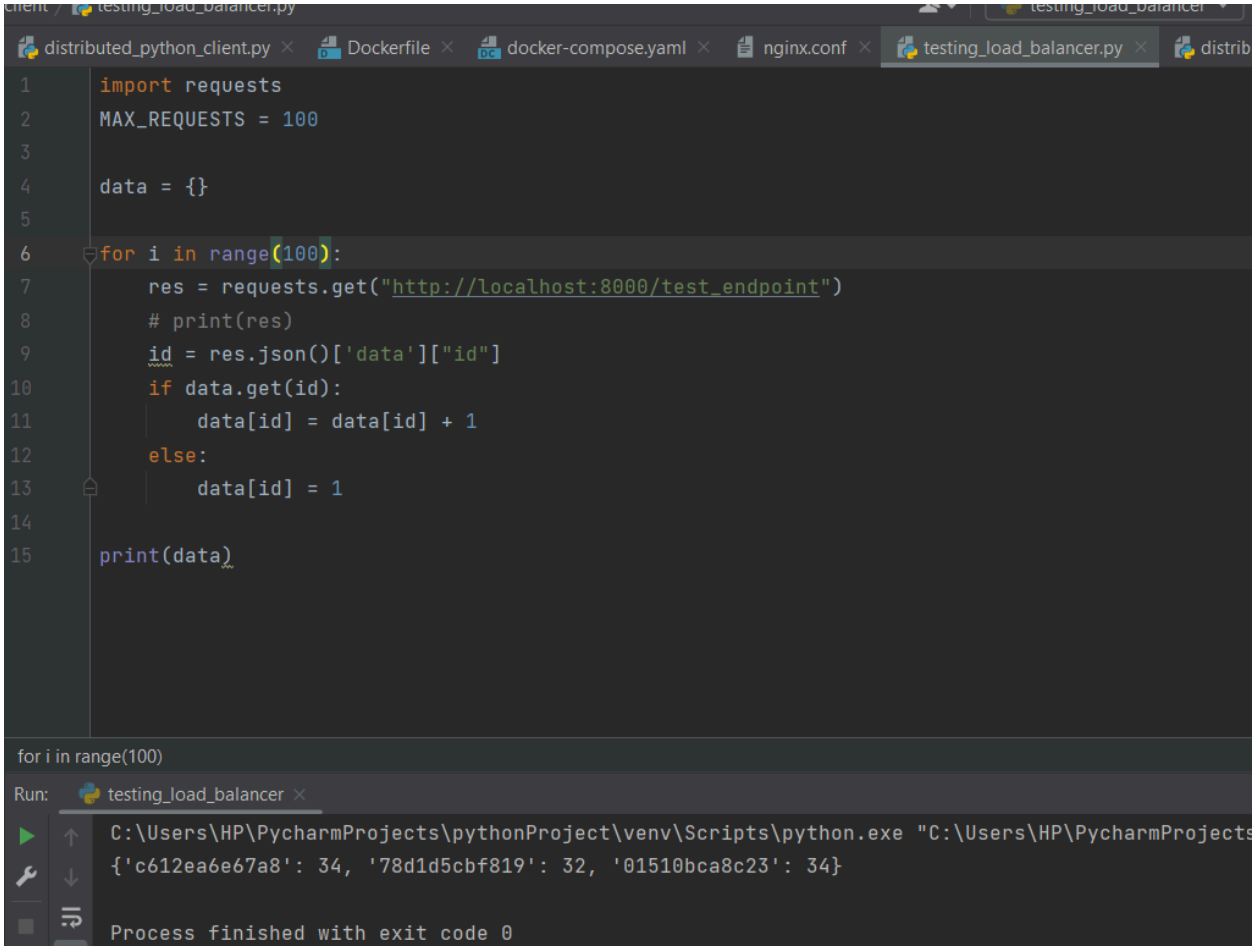
Run: testing_load_balancer

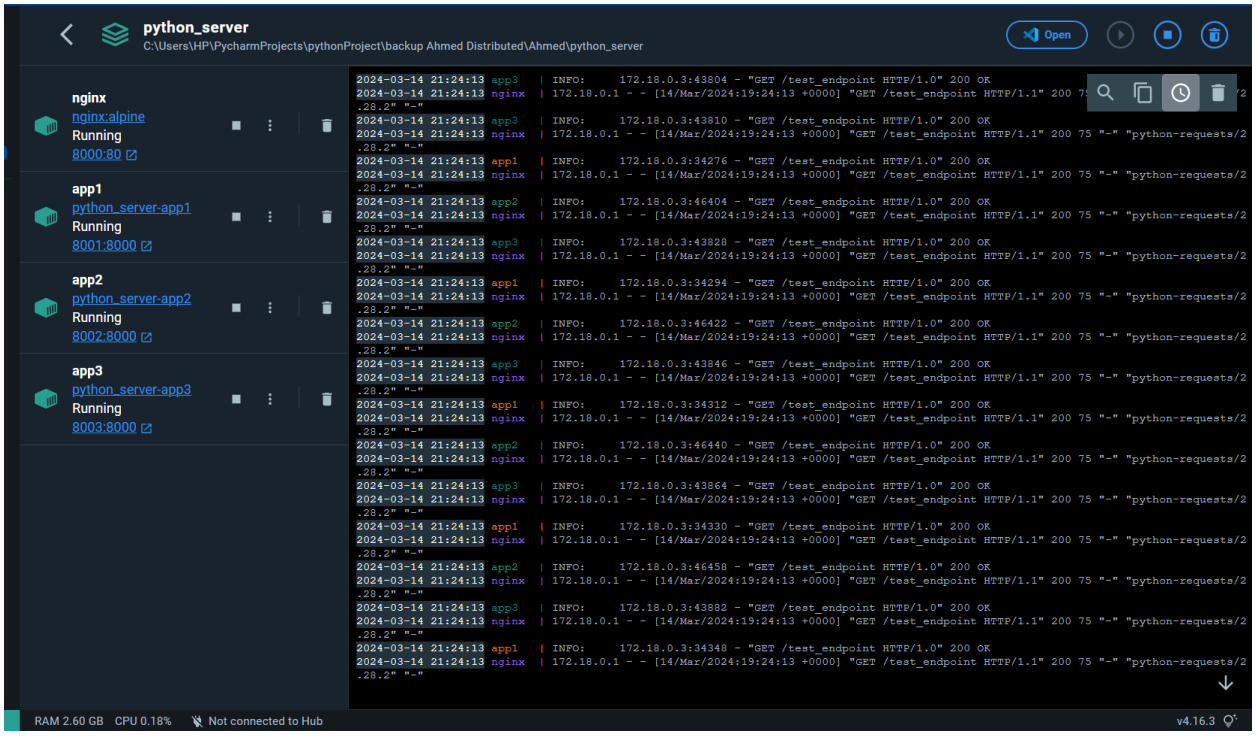
```
C:\Users\HP\PycharmProjects\pythonProject\venv\Scripts\python.exe "C:\Users\HP\Pycharm
{'37b2949fd702': 78, '761ceaa20d5b': 11, '05dc1cacd4cb': 11}

Process finished with exit code 0
```



Least Connection Algorithm:

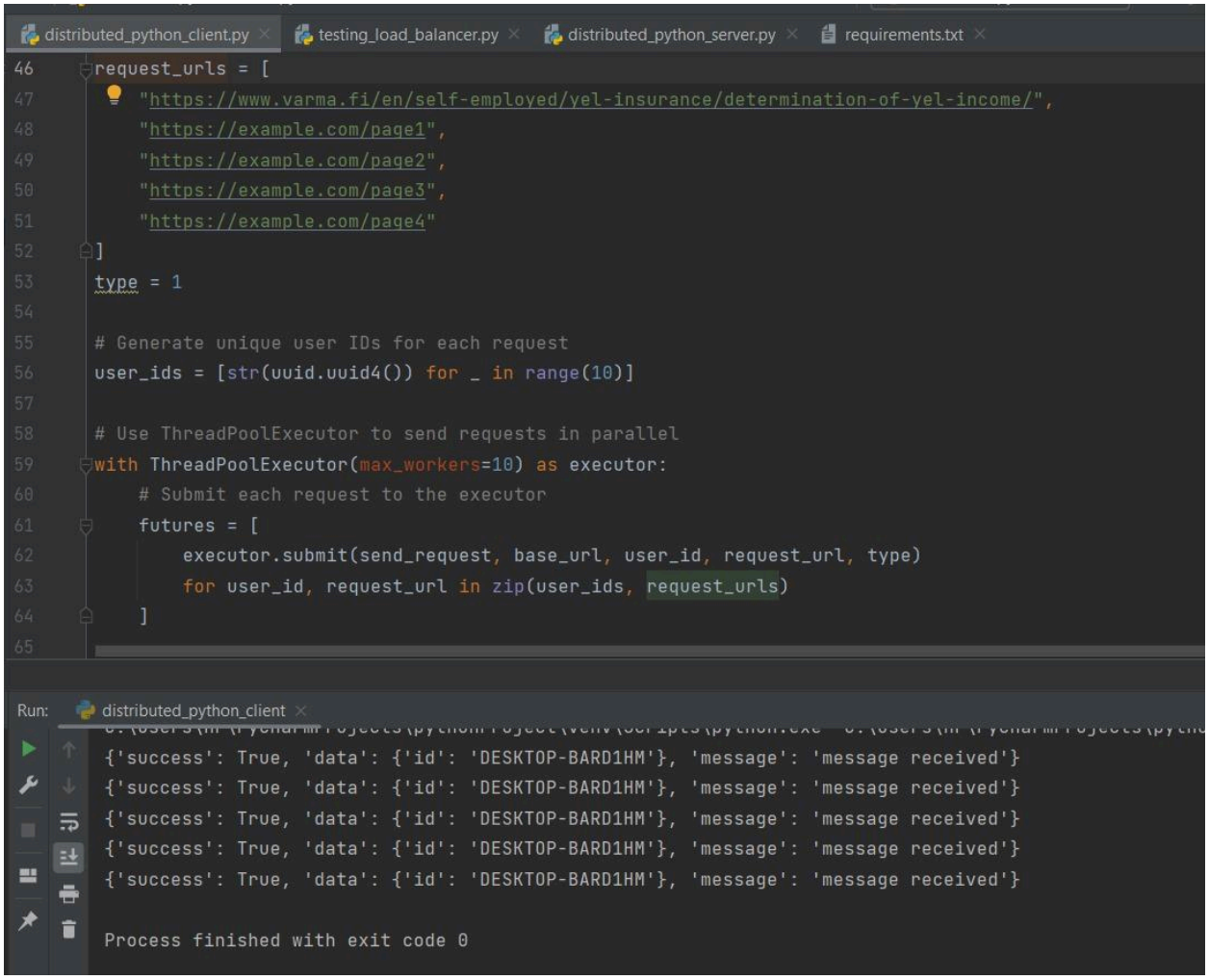


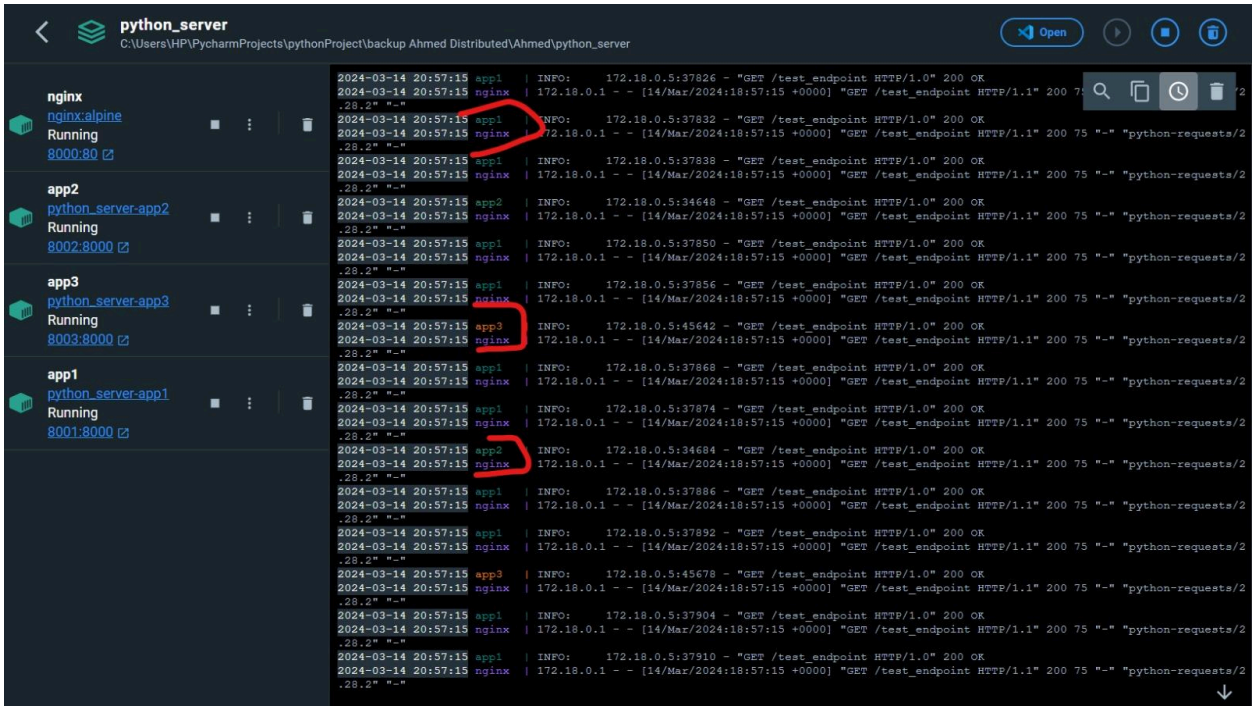

distributed_python_client.py
Dockerfile
docker-compose.yaml
nginx.conf
testing_load_balancer.py
import requests
MAX_REQUESTS = 100
data = {}
for i in range(100):
res = requests.get("http://localhost:8000/test_endpoint")
# print(res)
id = res.json()['data']["id"]
if data.get(id):
data[id] = data[id] + 1
else:
data[id] = 1
print(data)
for i in range(100)
Run:
testing_load_balancer
C:\Users\HP\PycharmProjects\pythonProject\venv\Scripts\python.exe "C:\Users\HP\PycharmProjects
{'c612ea6e67a8': 34, '78d1d5cbf819': 32, '01510bca8c23': 34}
Process finished with exit code 0


python_server
C:\Users\HP\PycharmProjects\pythonProject\backup Ahmed Distributed\Ahmed\python_server
Open
nginx
nginx:alpine
Running
8000:80
app1
python_server-app1
Running
8001:8000
app2
python_server-app2
Running
8002:8000
app3
python_server-app3
Running
8003:8000
RAM 2.60 GB CPU 0.18%
Not connected to Hub
v4.16.3

Concurrent Calls

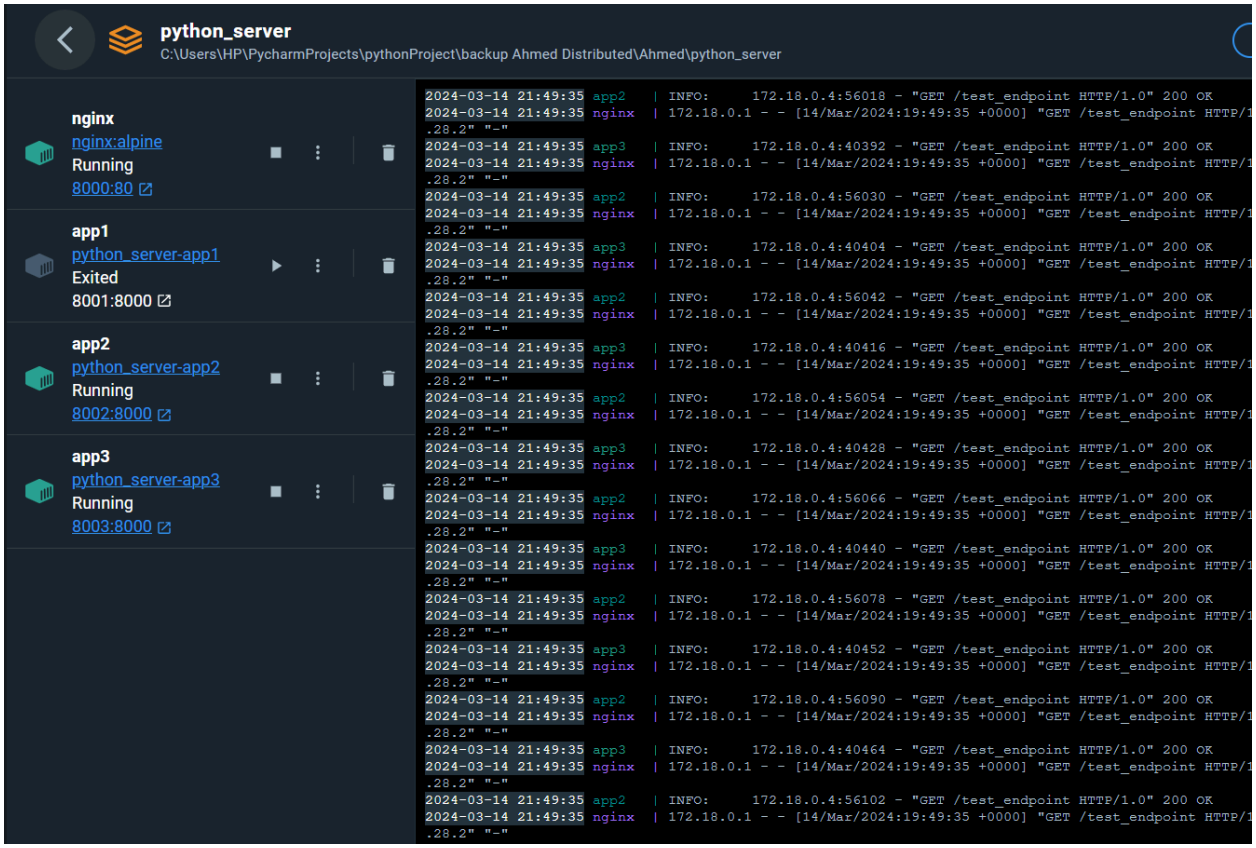
Here we are making concurrent calls like 5 parallel calls to our application and due to our distributed system we can make thousands of calls concurrent because we are using queue system in the middle.

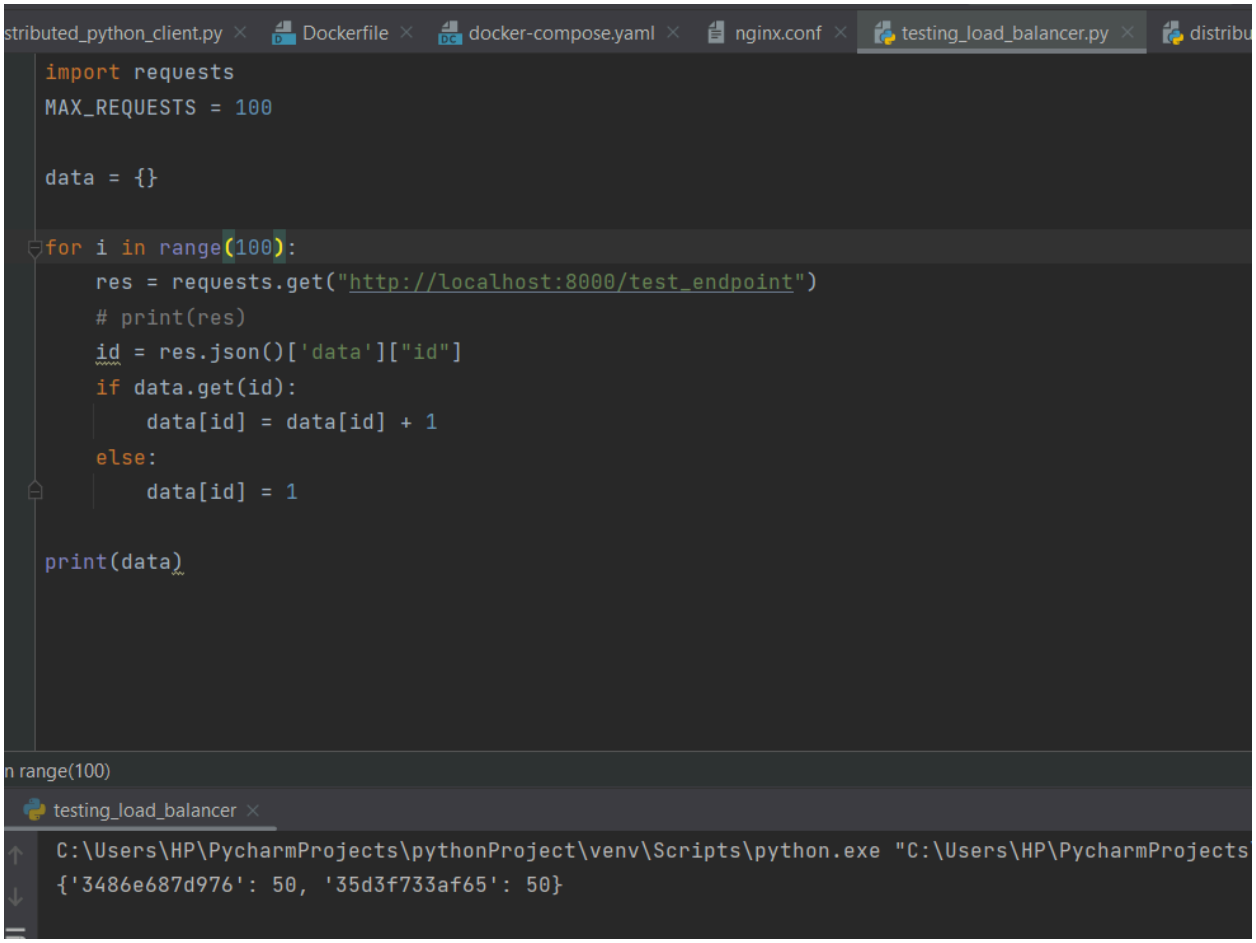




Container is down:

When we make one our container stopped to check how the traffic is routes 50 % routes from each app.


python_server
C:\Users\HP\PycharmProjects\pythonProject\backup Ahmed Distributed\Ahmed\python_server
nginx
nginx:alpine
Running
8000:80
app1
python_server-app1
Exited
8001:8000
app2
python_server-app2
Running
8002:8000
app3
python_server-app3
Running
8003:8000
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56018 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app3 | INFO: 172.18.0.4:40392 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56030 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app3 | INFO: 172.18.0.4:40404 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56042 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app3 | INFO: 172.18.0.4:40416 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56054 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app3 | INFO: 172.18.0.4:40428 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56066 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app3 | INFO: 172.18.0.4:40440 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56078 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app3 | INFO: 172.18.0.4:40452 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56090 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app3 | INFO: 172.18.0.4:40464 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"
2024-03-14 21:49:35 app2 | INFO: 172.18.0.4:56102 - "GET /test_endpoint HTTP/1.0" 200 OK
2024-03-14 21:49:35 nginx | 172.18.0.1 - - [14/Mar/2024:19:49:35 +0000] "GET /test_endpoint HTTP/1
.28.2" "-"



Workload distribution

Plan of sharing the workload (wl)and estimated hours. Please fill in the real calculated workload only on the final submission of the report.

Student name	Tasks	Estimated wl. (h)	Real wl. (h)
Muhammad Ahmed	Docker, Ngnix, python scripts, Python Client, Server	120 h	105 h
Muhammad Talha Arshad	RabbitMQ, C# scripts, MSSQL Database	120h	105 h
Md Mobusshar Islam	System Architecture, Research, Documentation	120h	105 h

Getting Started

To Setup the system locally, please follow the following steps

1- Installation

1- Docker Container https://www.docker.com/
2- python version 3.10 https://www.python.org/
3- .NET Core 8.0, C#10 https://dotnet.microsoft.com/en-us/download
4- RabbitMQ https://www.rabbitmq.com/
5- MSSQL Server https://www.microsoft.com/en-us/sql-server/sql-server-downloads
6- NGINX https://www.nginx.com/

Note: please enable system to use .net framework in your OS system if not enabled by default

2- Clone the github repository

https://github.com/Mobusshar/distributed-systems-project-msc-oulu-2024.git

3- Python Scripts

From the folder in the repository "Python Scripts", run the python server.py file with the following command

```
docker compose up --build
```

Some additional files are also attached in the same folder to mock the data of the clients, please run those files as needed.

4- MSSQL

After installing the MSSQL, change the server name and create a database table with the name "Messages" with the following script

```
USE [Messages]
GO

/****** Object:  Table [dbo].[Messages]    Script Date: 3/14/2024 10:16:15 PM ******/
SET ANSI_NULLS ON
GO

SET QUOTED_IDENTIFIER ON
GO

CREATE TABLE [dbo].[Messages](
	[Id] [int] IDENTITY(1,1) NOT NULL,
	[user_Id] [nvarchar](100) NULL,
	[request_url] [nvarchar](100) NULL,
```

```
	[type] [int] NULL,
	[Is_processed] [bit] NULL,
	[callback_url] [nvarchar](100) NULL,
	[date_added] [datetime] NULL,
	[date_process_complete] [datetime] NULL,
	[Is_processing] [bit] NULL,
PRIMARY KEY CLUSTERED
(
	[Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY
= OFF, ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON,
OPTIMIZE_FOR_SEQUENTIAL_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]
GO

ALTER TABLE [dbo].[Messages] ADD  CONSTRAINT [DF_Messages_IsProcessed]
DEFAULT ((0)) FOR [Is_processed]
GO

ALTER TABLE [dbo].[Messages] ADD  CONSTRAINT [DF_Messages_date_added]
DEFAULT (getdate()) FOR [date_added]
GO

ALTER TABLE [dbo].[Messages] ADD  CONSTRAINT [DF_Messages_Is_processing]
DEFAULT ((0)) FOR [Is_processing]
GO
```

4- CSharp Scripts

MessageBroker_1_RequestRecording file contains the solution to run that will save the requests from the servers and store them in the database created above.

PublisherFromDB solution in the same directory will be publishing the requests or messages to be consumed by RabbitMQ where actual processing will be happen by consumers

RabbitMQConsumer has two solutions that will act as subscribers and process the request along with sending callback after request completion.