Advanced Message Queuing Protocol

Ritvik Shukla

Department of Computer Science Engineering
School of Engineering and Technology
Central University Of Rajasthan

March 2, 2023

Overview

1. Distributed Systems
2. Message Queuing
3. Message Brokers
4. AMQP
5. Implementation

What are Distributed Systems?

Distributed systems are software systems that run on multiple computers or nodes and work together to achieve a common goal. In a distributed system, each node communicates with other nodes by exchanging messages over a network. The nodes can be located in the same physical location or in different locations across the globe.

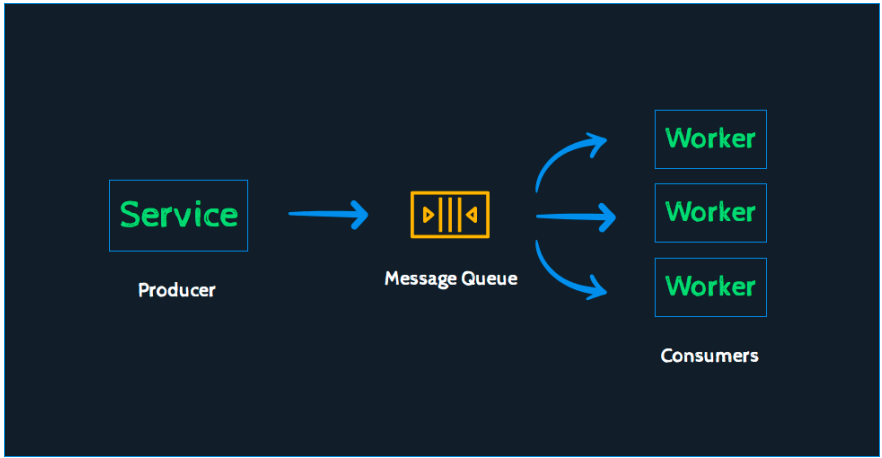
Distributed systems are designed to provide high availability, scalability, and fault tolerance. They are used to solve problems that cannot be easily solved by a single computer or centralized system. Some examples of distributed systems include web applications, cloud computing platforms, online gaming systems, and social networking platforms.

What is Message Queuing?

A message queue is a software component that allows applications to exchange messages or data with each other asynchronously. It acts as an intermediary between the sender and the receiver, ensuring reliable delivery and decoupling the sender and receiver so that they can operate independently.

Message queues are often used in distributed systems where applications may be running on different machines or in different processes. They provide a way for applications to communicate without needing to know the exact details of each other's implementation or availability. Common implementations of message queues include RabbitMQ, Apache Kafka.

Figure


Service
Producer
Message Queue
Worker
Worker
Worker
Consumers

How it work?

The working of a message queue involves several components, including the sender, the receiver, and the message queue itself.

- Sender sends a message: The sender application creates a message and sends it to the message queue.
- Message is stored in the queue: The message is stored in the message queue until a receiver retrieves it.
- Receiver retrieves the message: The receiver application polls or subscribes to the message queue and retrieves the next available message.
- Message is processed: The receiver processes the message according to its requirements.

How it work?

- Message is acknowledged: Once the receiver has processed the message, it sends an acknowledgment to the message queue to confirm that the message has been successfully received and processed.
- Message is removed from the queue: The message queue removes the message from the queue and updates its internal state accordingly.

What is Message Broker?

A message broker is an intermediary software component that facilitates communication between different applications or services by managing the exchange of messages.

In a distributed system, applications or services may need to communicate with each other asynchronously, but direct communication between them may not be possible or efficient. In such cases, a message broker acts as a middleman, receiving messages from one application or service and forwarding them to one or more other applications or services.

Message brokers provide several functions, including message routing, message filtering, message transformation, and message persistence.

Common Brokers Used

Examples

Some of the common examples of message brokers used are RabbitMQ, Apache Kafka, Apache ActiveMQ etc.

What is AMQP?

AMQP (Advanced Message Queuing Protocol) is an open standard messaging protocol that enables reliable and efficient communication between applications or services in a distributed system.

AMQP defines a messaging format and a set of rules for message routing and delivery, allowing applications to communicate asynchronously and reliably over a network. It provides features such as message queuing, message acknowledgment, and message routing to ensure that messages are delivered to their intended destination in a timely and secure manner.

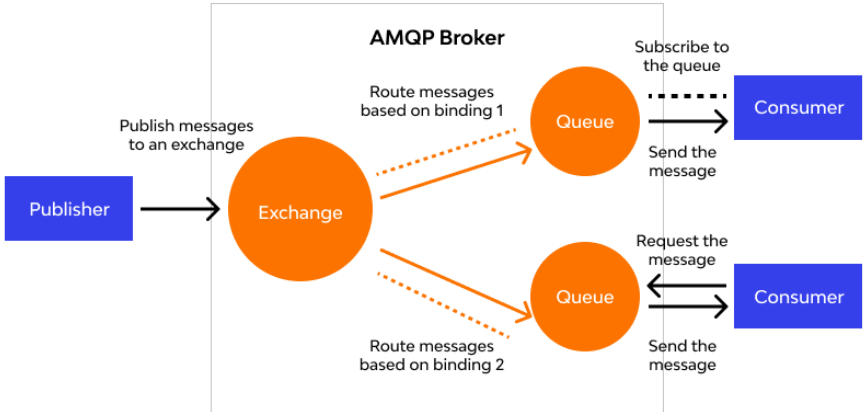
What is AMQP?

AMQP is designed to be interoperable, meaning that it can be used with different programming languages, operating systems, and messaging systems. It is widely used in various industries, including finance, healthcare, e-commerce, and telecommunications, to name a few.

Note

Popular messaging systems that implement AMQP include RabbitMQ and Apache ActiveMQ.

Working of AMQP



Working of AMQP

AMQP works on a client-server model, where the messaging system consists of one or more message brokers (servers) and multiple client applications that interact with the brokers to send and receive messages.

The basic workflow of AMQP is as follows:

- A client application sends a message to a message broker using a specified queue or exchange.
- The message broker receives the message and performs routing and delivery based on the message's destination, using AMQP routing rules.
- If the message is sent to a queue, it is stored in the queue until a consuming client retrieves it.

Working of AMQP

- If the message is sent to an exchange, the message broker routes it to the correct queue or multiple queues based on a set of predefined routing rules.
- A client application consumes the message from the queue and sends an acknowledgment to the broker indicating that the message has been received and processed.

Important Points Related to AMQP

Note

AMQP uses channels to multiplex multiple independent conversations over a single TCP connection, reducing the overhead of opening and closing multiple connections.

Note

AMQP supports different messaging patterns such as publish/subscribe, point-to-point, and request/response, allowing developers to choose the best pattern for their use case.

Messaging Patterns in AMQP

There are different messaging patterns that AMQP supports like:

- Publish/Subscribe
- Point-To-Point
- Request/Response

Publish/Subscribe Pattern

The publish-subscribe (pub/sub) messaging pattern is a communication pattern used in distributed systems where publishers send messages to a topic or channel, and subscribers receive those messages based on their interest in that topic. In this pattern, the publishers and subscribers are decoupled from each other, meaning that publishers do not need to know who the subscribers are, and vice versa.

When a publisher sends a message to a topic, the message is distributed to all subscribers that have subscribed to that topic. Subscribers can receive messages in a variety of ways, such as through a message queue or by using a WebSocket.

Point-To-Point Pattern

The point-to-point messaging pattern is a communication pattern used in distributed systems where messages are sent from one sender to one specific receiver. In this pattern, a message is sent to a specific queue, and only one consumer can receive and process that message from that queue.

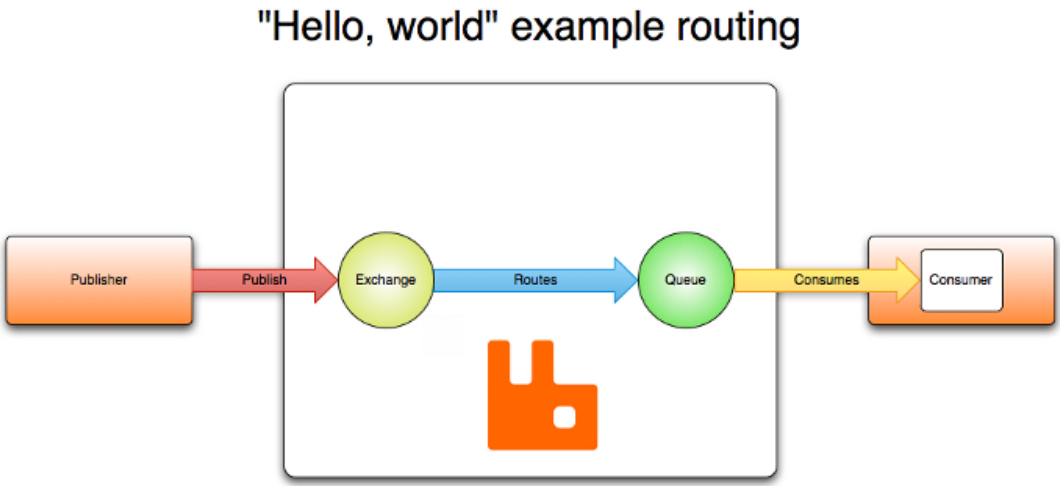
When a sender sends a message to a queue, the message is stored in the queue until a receiver processes it. When a receiver processes a message, it is removed from the queue and cannot be received by any other receiver. This ensures that each message is processed only once by one receiver.

Request/Response Pattern

The request-response messaging pattern is a communication pattern used in distributed systems where one application sends a request message to another application and expects a response message in return. In this pattern, the sender sends a request message and waits for a response from the receiver before continuing with its work.

When a requester sends a request message to a receiver, it waits for a response message before continuing with its work. The receiver processes the request message and sends a response message back to the requester. The response message contains the result of the request or an error message if the request could not be processed.

Exchanges

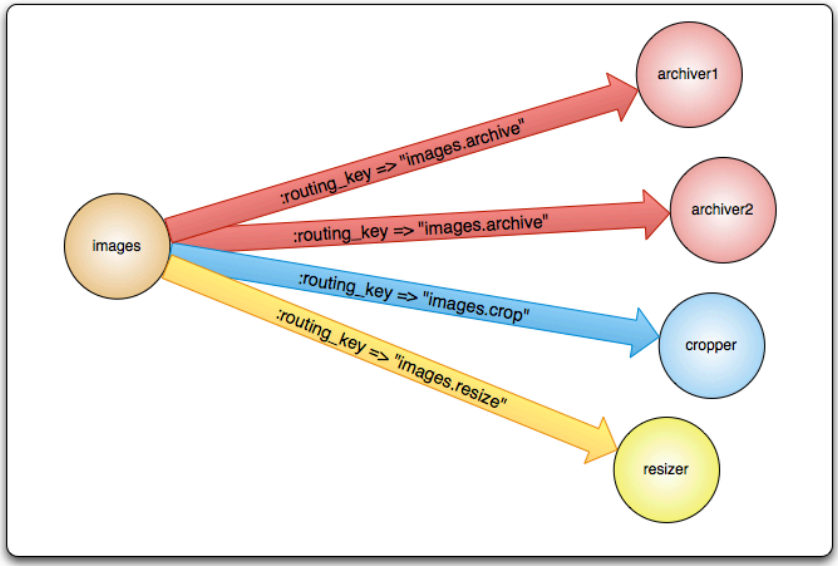

"Hello, world" example routing
Publisher
Publish
Exchange
Routes
Queue
Consumes
Consumer

Exchanges

Exchanges take a message and route it into zero or more queues. The routing algorithm used depends on the exchange type and rules called bindings. AMQP brokers provide four exchange types:

- Direct exchange
- Fanout exchange
- Topic exchange
- Headers exchange

Direct exchange routing

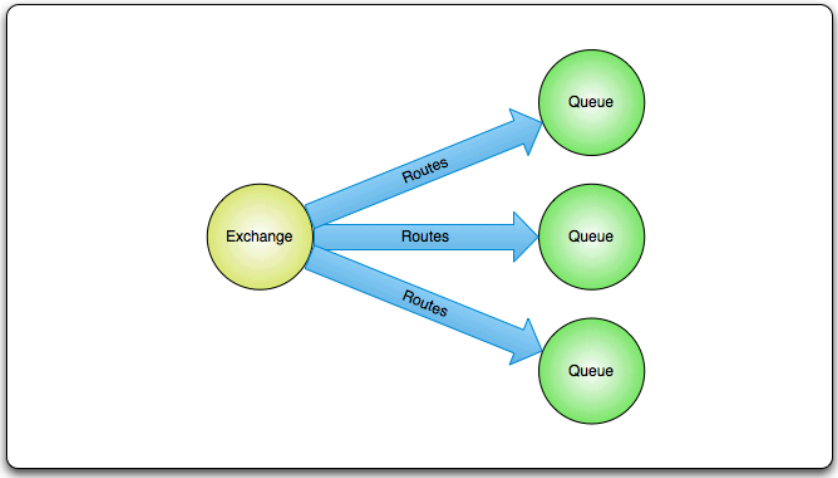


Direct exchange

A direct exchange delivers messages to queues based on the message routing key. A direct exchange is ideal for the unicast routing of messages (although they can be used for multicast routing as well).

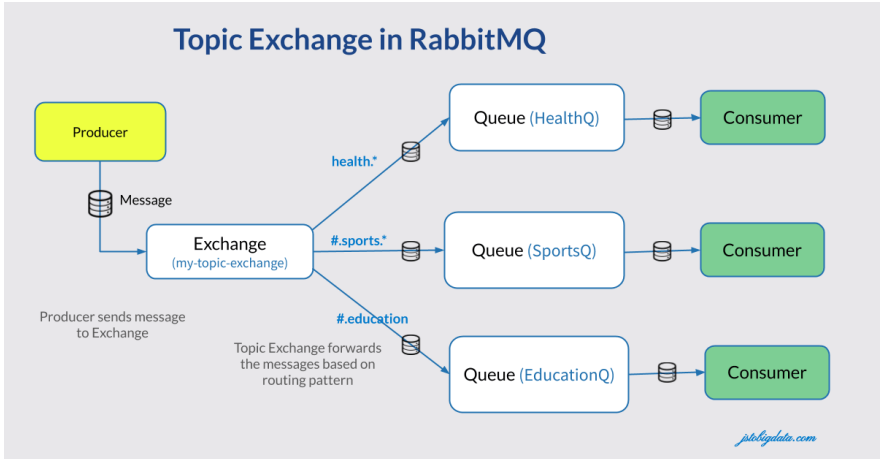
A queue binds to the exchange with a routing key K When a new message with routing key R arrives at the direct exchange, the exchange routes it to the queue if K = R

Fanout exchange routing



Fanout exchange

When a message is published to a fanout exchange, the exchange sends a copy of the message to every queue that is bound to it. This means that every message sent to a fanout exchange is delivered to all the queues that are bound to it, regardless of the routing key or any other message attribute.

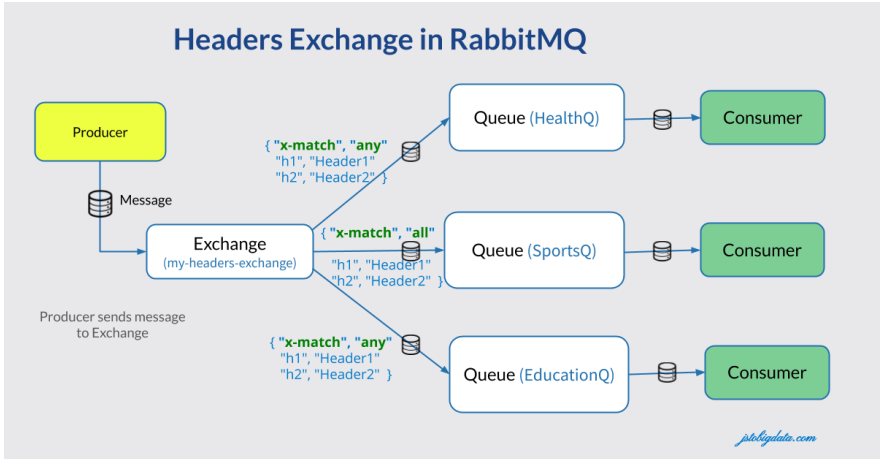

Topic Exchange in RabbitMQ
Producer
Message
Exchange
(my-topic-exchange)
Producer sends message
to Exchange
health.*
#.sports.*
#.education
Topic Exchange forwards
the messages based on
routing pattern
Queue (HealthQ)
Queue (SportsQ)
Queue (EducationQ)
Consumer
Consumer
Consumer
jabbigdata.com

Topic exchange

The topic exchange is a type of exchange where messages are routed to queues based on a pattern match between the routing key and the binding key.

When a message is published to a topic exchange, the exchange compares the routing key of the message with the routing key of each queue that it is bound to. The routing key can contain one or more words separated by dots, and can also contain wildcard characters:

- "*" (star) can substitute for exactly one word.
- "#" (hash) can substitute for zero or more words.


Headers Exchange in RabbitMQ
Producer
Message
Exchange
(my-headers-exchange)
Producer sends message
to Exchange
{ "x-match", "any"
"h1", "Header1"
"h2", "Header2" }
{ "x-match", "all"
"h1", "Header1"
"h2", "Header2" }
{ "x-match", "any"
"h1", "Header1"
"h2", "Header2" }
Queue (HealthQ)
Queue (SportsQ)
Queue (EducationQ)
Consumer
Consumer
Consumer
jobbigdata.com

Headers exchange

Header exchange is a type of exchange where messages are routed to queues based on message attributes (headers) that match the values specified in the binding.

When a message is published to a header exchange, the exchange examines the message headers and compares them to the headers specified in each binding. The exchange then delivers the message to all the queues that have bindings with header values that match the message headers.

Implementation

Ecommerce Microservice

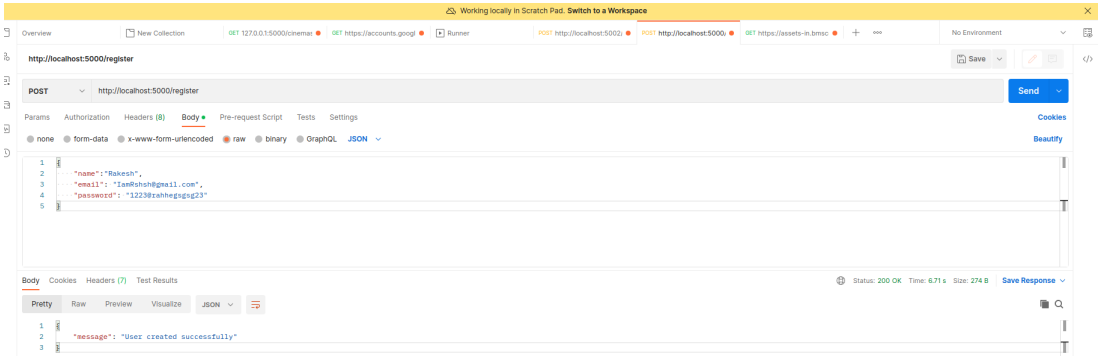
What I built?

Implementing a Microservice using RabbitMQ as a message broker.

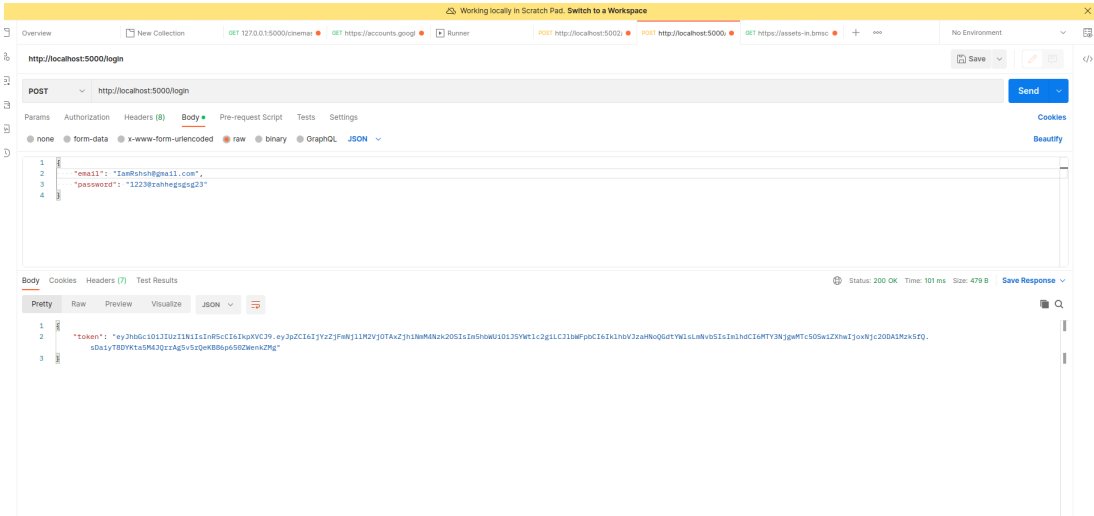
- Auth Service
- Product Service
- Order Service

Auth Service

Create User

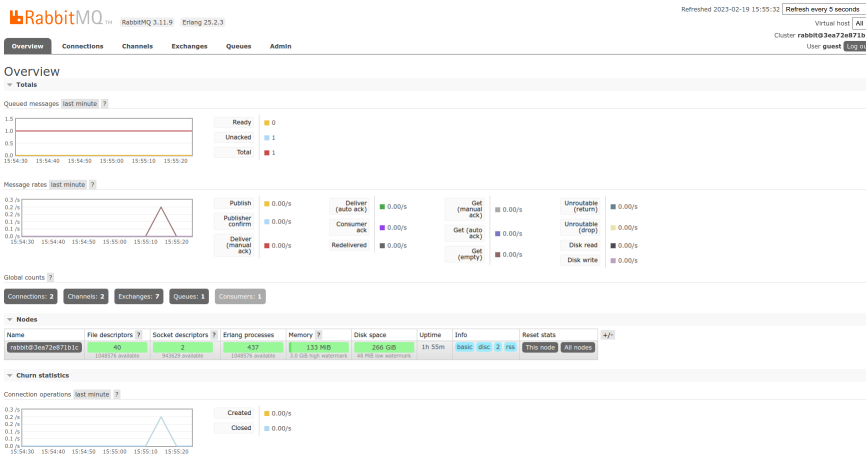

Working locally in Scratch Pad. Switch to a Workspace
Overview
New Collection
GET 127.0.0.1:5000/cinemas
GET https://accounts.google
Runner
POST http://localhost:5002/
POST http://localhost:5000/
GET https://assets-in.bmscd
No Environment
http://localhost:5000/register
Save
POST
http://localhost:5000/register
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"name":"Rakesh",
"email": "IamRahsh@gmail.com",
"password": "1223#rahhegsgsg23"
}
Body
Cookies
Headers (7)
Test Results
Status: 200 OK
Time: 6.71 s
Size: 274 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"message": "User created successfully"
}

Login User


Working locally in Scratch Pad. Switch to a Workspace
Overview
New Collection
GET 127.0.0.1:5000/cinemas
GET https://accounts.google
Runner
POST http://localhost:5002/
POST http://localhost:5000/
GET https://assets-in.bmsc
No Environment
http://localhost:5000/login
Save
POST
http://localhost:5000/login
Send
Params
Authorization
Headers (8)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"email": "IamRshsh@gmail.com",
"password": "1223@rahhegsgsg23"
}
Body
Cookies
Headers (7)
Test Results
Status: 200 OK
Time: 101 ms
Size: 479 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
{
"token": "eyJhbGciOiJIUzI1NiIsInRScCI6IkpXVCJ9.eyJpZCI6IjYzZjFmNjllM2VjOTAxZjhiNmM4N2k2OSIsImShbWUiOiJSYWtlc2giLCJlbWFpbCI6Ik1hbVJzaHNoQGdtYWlsLmNvbSIsImlhdCI6MTY3NjgwMTc5OSwiZXhwIjoxNjc2ODA1Mzk5fQ.
sDaiyTBDYKta5M4JQrzAgSv5zQeKB86p650ZWenkZMg"
}

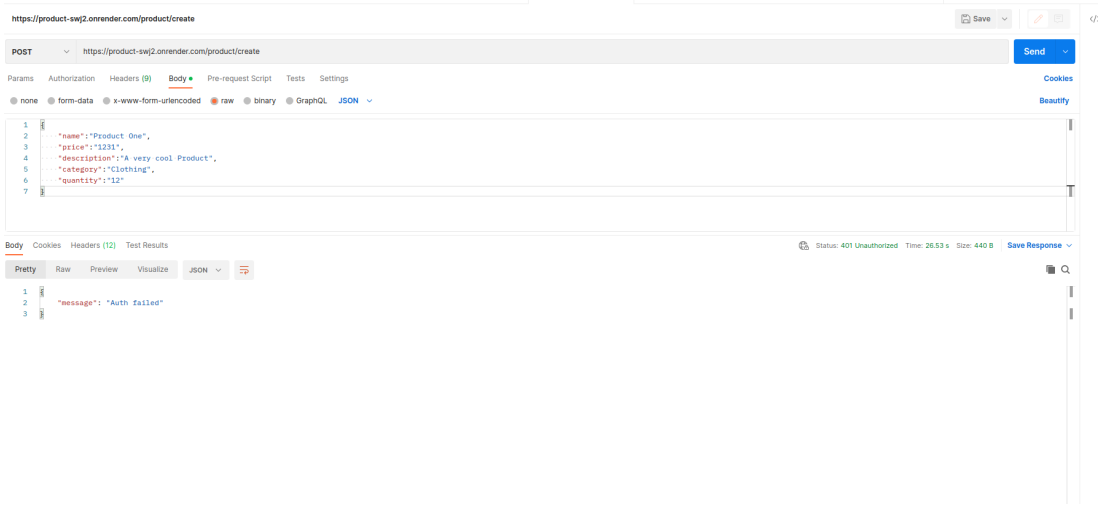
RabbitMQ on Docker

Admin UI

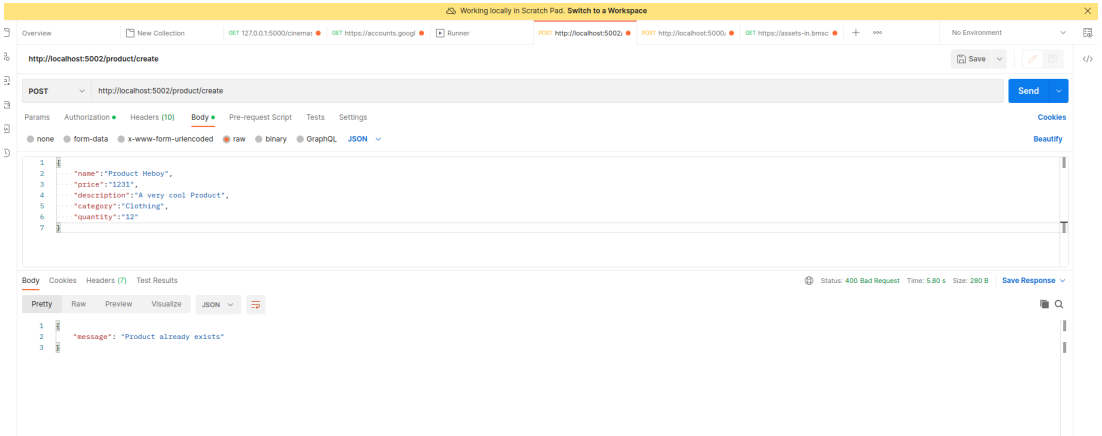

RabbitMQ
RabbitMQ 3.11.9
Erlang 25.2.3
Refreshed 2023-02-19 15:55:32
Refresh every 5 seconds
Virtual host All
Cluster rabbit@3ea72e871b
User guest
Log out
Overview
Connections
Channels
Exchanges
Queues
Admin
Overview
Totals
Queued messages last minute
Ready 0
Unacked 1
Total 1
Message rates last minute
Publish 0.00/s
Publisher confirm 0.00/s
Deliver (manual ack) 0.00/s
Deliver (auto ack) 0.00/s
Consumer ack 0.00/s
Redelivered 0.00/s
Get (manual ack) 0.00/s
Get (auto ack) 0.00/s
Get (empty) 0.00/s
Unroutable (return) 0.00/s
Unroutable (drop) 0.00/s
Disk read 0.00/s
Disk write 0.00/s
Global counts
Connections: 2
Channels: 2
Exchanges: 7
Queues: 1
Consumers: 1
Nodes
Name
File descriptors
Socket descriptors
Erlang processes
Memory
Disk space
Uptime
Info
Reset stats
rabbit@3ea72e871b1c
40
1048576 available
2
943629 available
437
1048576 available
133 MiB
3.5 GiB high watermark
266 GiB
48 MiB low watermark
1h 55m
basic disc 2 rss
This node
All nodes
Churn statistics
Connection operations last minute
Created 0.00/s
Closed 0.00/s

Product Service

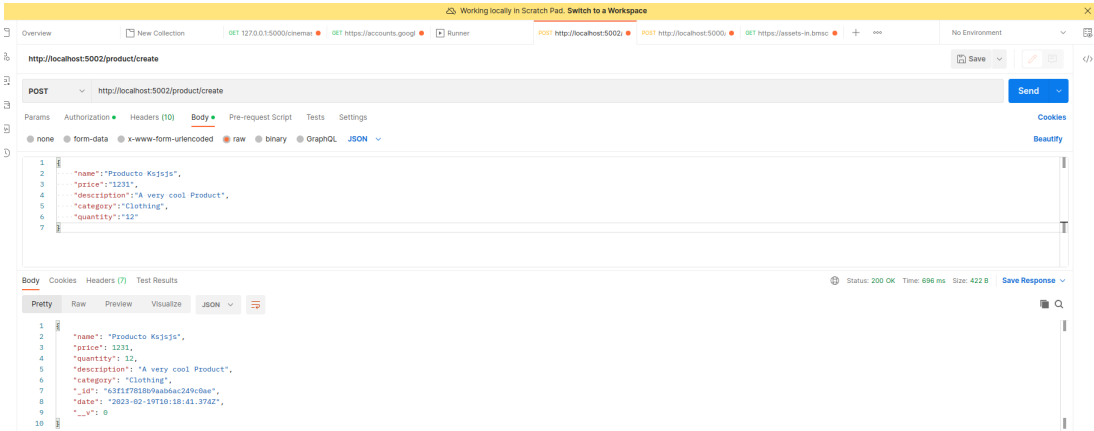
Restricted Access



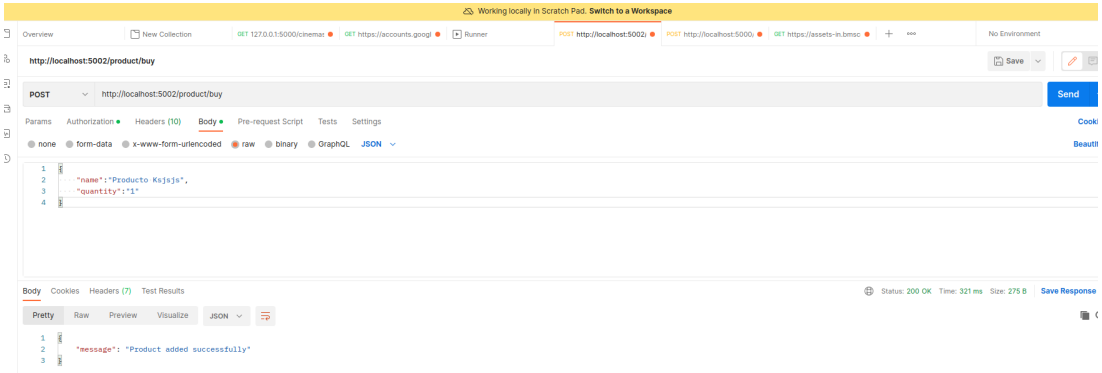
Uniqueness Check



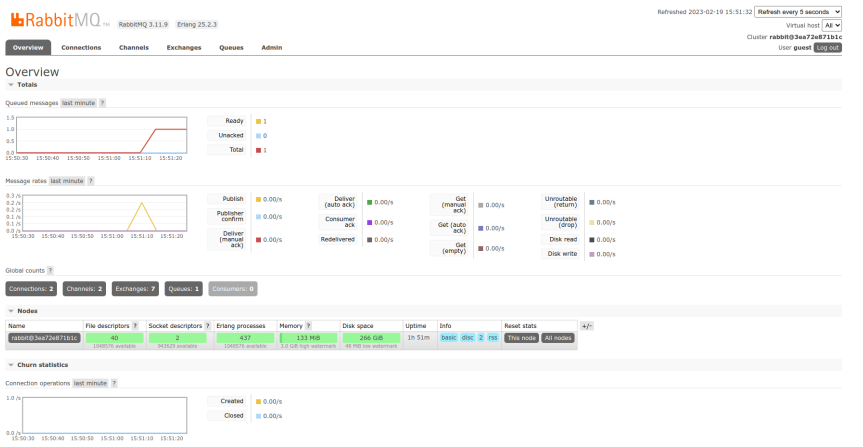
Create a Product



Buy a Product

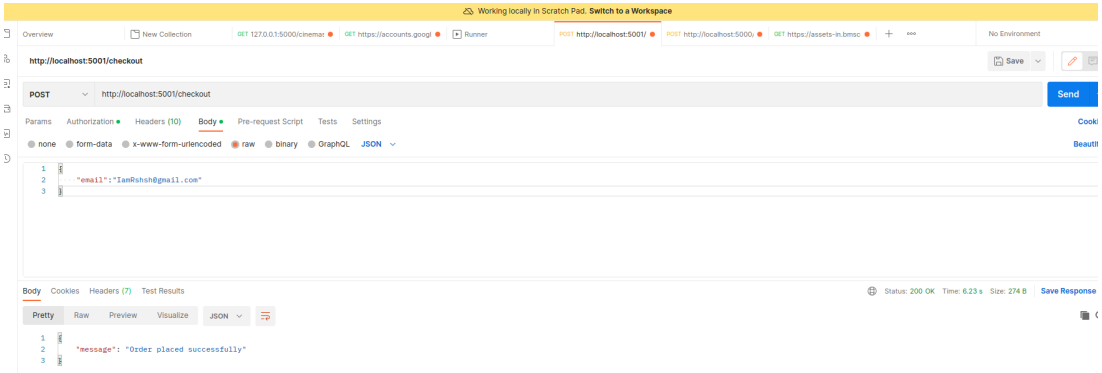


Message Queued

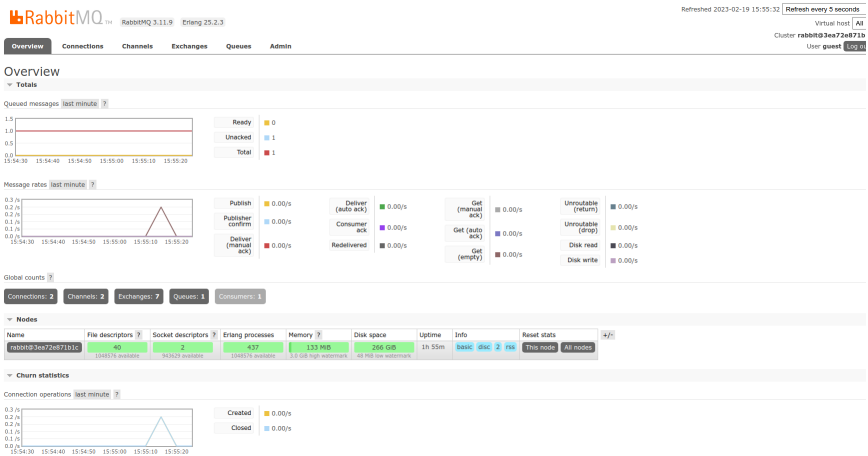


Order Service

Checkout


Working locally in Scratch Pad. Switch to a Workspace
Overview
New Collection
GET 127.0.0.1:5000/cinemas
GET https://accounts.googl
Runner
POST http://localhost:5001/
POST http://localhost:5000/
GET https://assets-in.bmsc
No Environment
http://localhost:5001/checkout
Save
POST
http://localhost:5001/checkout
Send
Params
Authorization
Headers (10)
Body
Pre-request Script
Tests
Settings
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
1 {
2 "email":"IamRshsh@gmail.com"
3 }
Body
Cookies
Headers (7)
Test Results
Status: 200 OK
Time: 6.23 s
Size: 274 B
Save Response
Pretty
Raw
Preview
Visualize
JSON
1 {
2 "message": "Order placed successfully"
3 }

Consumed Message



References

RabbitMQ Docs
https://www.rabbitmq.com/documentation.html
RabbitMQ

AMQP (0-9-1) Specification
https://www.amqp.org/specification/0-9-1/amqp-org-download
AMQP

The End