

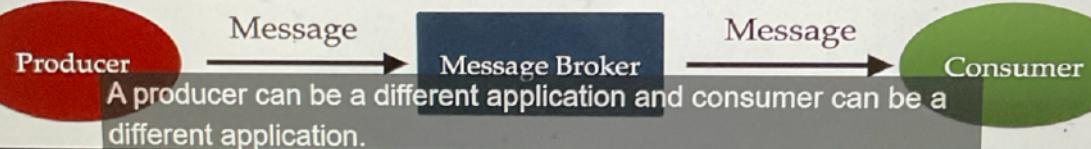
RabbitMQ

What is a Message Queue?

Message queuing allows applications to communicate by sending messages to each other. The message queue provides temporary message storage when the destination program is busy or not connected.

A message queue is made up of a producer, a broker (the message queue software), and a consumer.

A message queue provides an asynchronous communication between applications.



RabbitMQ

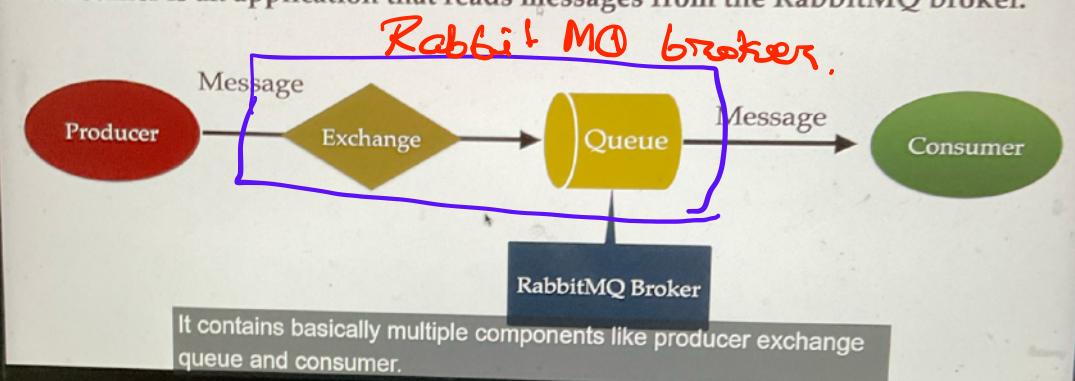
RabbitMQ is a message queue software (message broker/queue manager) that acts as an intermediary platform where different applications can send and receive messages.

RabbitMQ originally implements the Advanced Message Queuing Protocol (AMQP). But now RabbitMQ also supports several other API protocols such as STOMP, MQTT and HTTP.

Well, rabbit is nothing but a message queue that acts as an intermediary platform where different applications

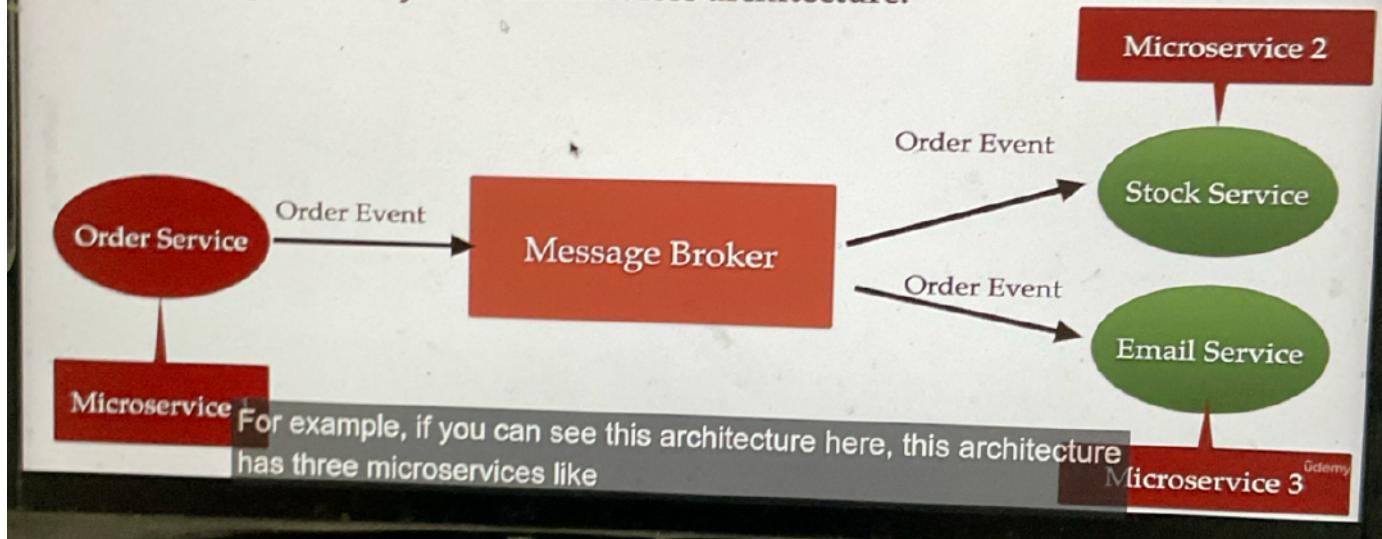
RabbitMQ

Producer is an application that sends messages to the RabbitMQ broker and Consumer is an application that reads messages from the RabbitMQ broker.



Use of RabbitMQ in Microservices

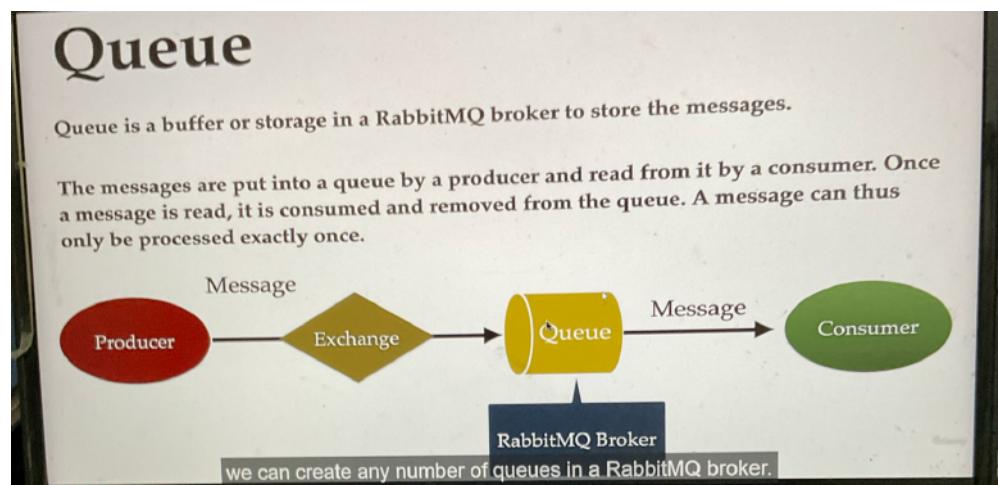
RabbitMQ is one of the simplest freely available options for implementing messaging queues in your microservices architecture.



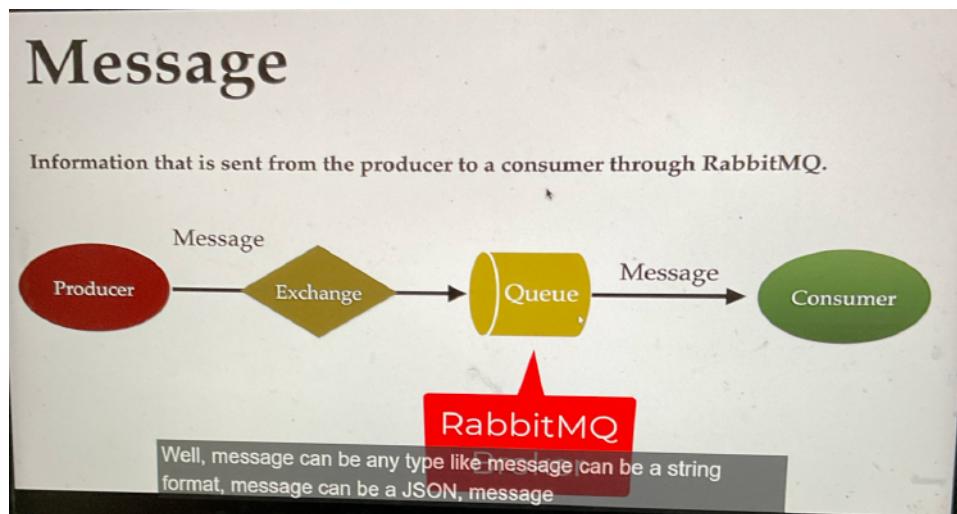
Producers : Producer sends message to broker.
↓
[Exchange + Queue]

Consumer : Consumer reads message from the RabbitMQ broker.

Queue :



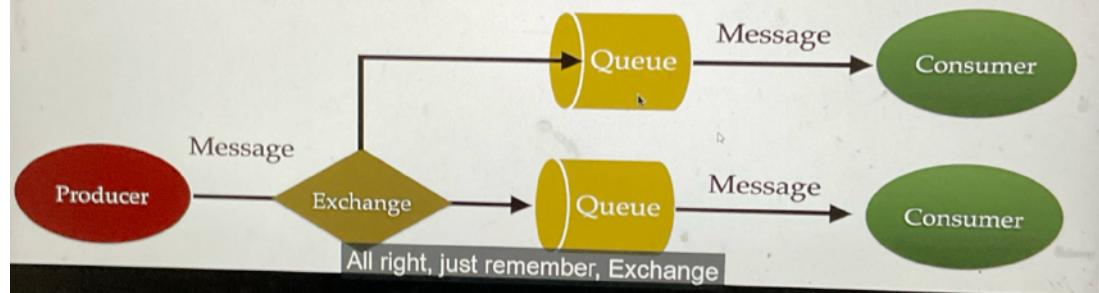
Message :



Exchange :

Exchange

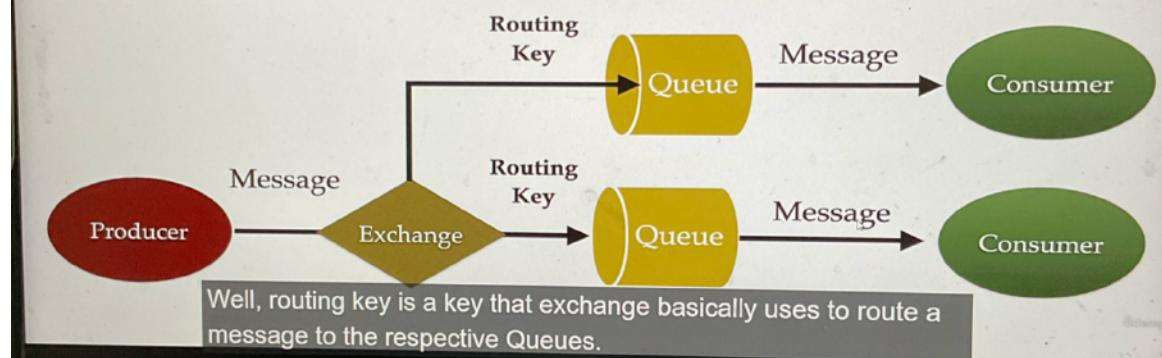
Basically, it acts as an intermediary between the producer and a queue. Instead of sending messages directly to a queue, a producer can send them to an exchange instead. The exchange then sends those messages to one or more queues following a specified set of rules. Thus, the producer does not need to know the queues that eventually receive those messages.



Routing Key :

Routing Key

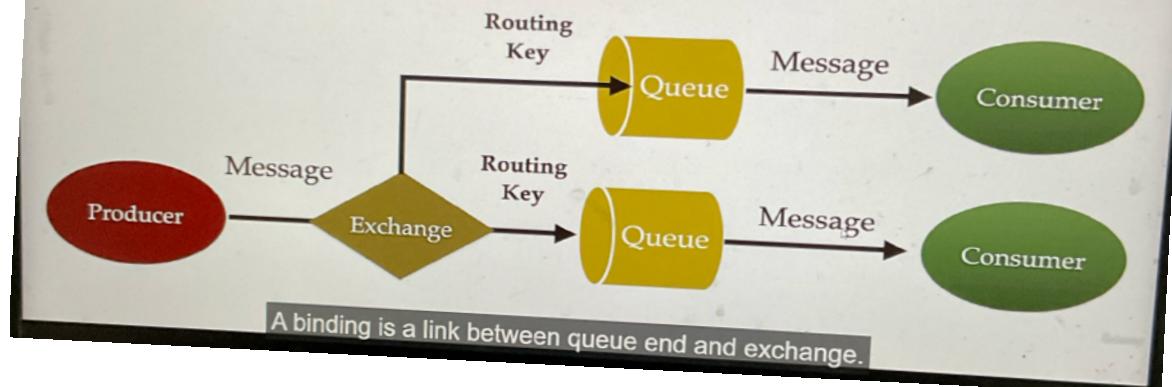
The routing key is a key that the exchange looks at to decide how to route the message to queues. The routing key is like an address for the message.



Binding

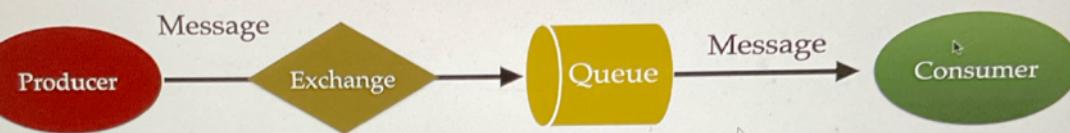
Binding

A binding is a link between a queue and an exchange.



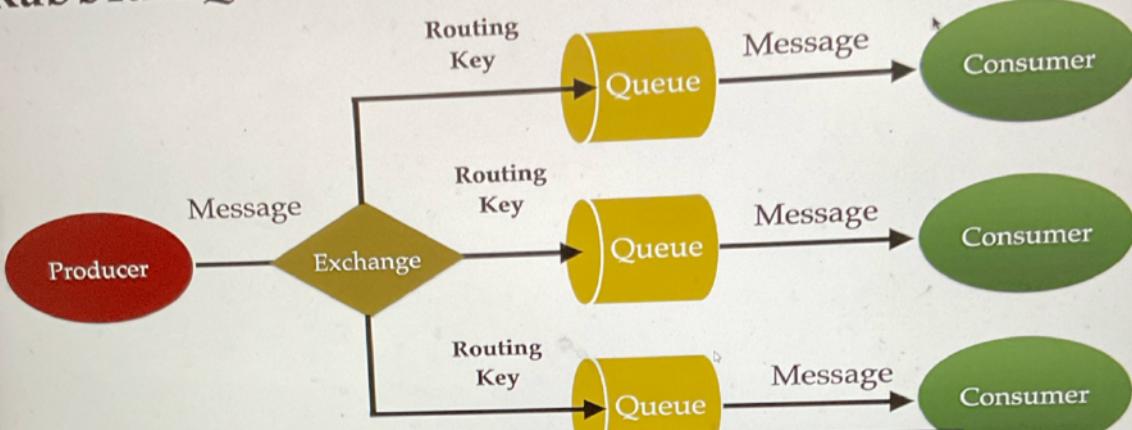
RabbitMQ Architecture

Simple RabbitMQ Architecture



message to the Queue and then consumer will basically consume or read that message from the queue.

RabbitMQ Architecture with Multiple Queues



And then consumer will basically consume or read the message from the respective queues.

* Producer send a message along with Routing Key . to the exchange .

Key . to the exchange .

* Exchange use that Routing key send that message into respective Queue

RabbitMQ :

Run :

```
Last login: Mon Jun 20 19:10:17 on ttys000
ramesh@adarsh-MacBook-Air ~ % docker pull rabbitmq:3.10.5-management
3.10.5-management: Pulling from library/rabbitmq
11e23ac719b3: Pull complete
431d6e9b403d: Pull complete
39626a8e8f80: Pull complete
25839a1a1cbb: Pull complete
d01e97e9a4c9: Pull complete
cbfbef8c41c9: Pull complete
d018da6f6dd: Pull complete
857399c34ab7: Pull complete
ec1ad240cb9c: Pull complete
dcc5dd42a10: Pull complete
Digest: sha256:244b4f37f778dbf10e2a6c0713101945100f5b2dbff62ec57f812fd6b452c6e
Status: Downloaded newer image for rabbitmq:3.10.5-management
docker.io/library/rabbitmq:3.10.5-management
ramesh@adarsh-MacBook-Air ~ % docker run --rm -it -p 15672:5672 rabbitmq:3.10.5-management
```

Port 15672 is for the RabbitMQ management website

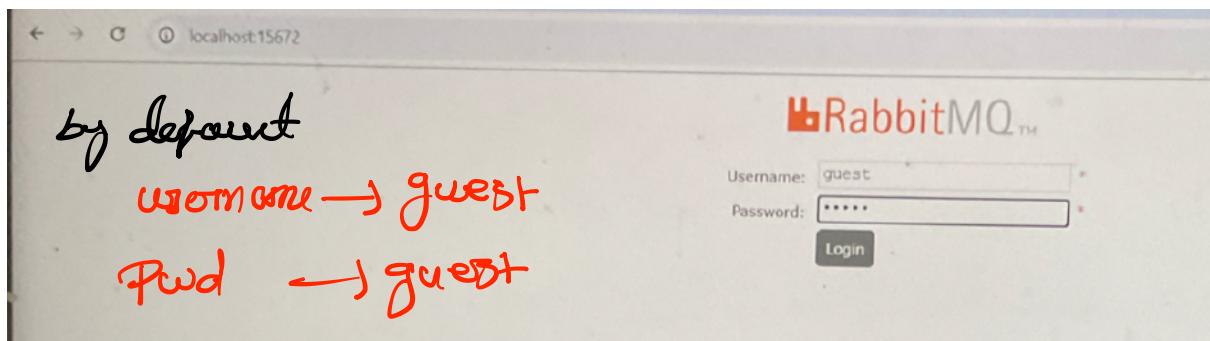
So this port is for the rabbit management website and this port is to connect to a RabbitMQ client.

```
Last login: Mon Jun 20 19:10:17 on ttys000
ramesh@adarsh-MacBook-Air ~ % docker pull rabbitmq:3.10.5-management
3.10.5-management: Pulling from library/rabbitmq
11e23ac719b3: Pull complete
431d6e9b403d: Pull complete
39626a8e8f80: Pull complete
25839a1a1cbb: Pull complete
d01e97e9a4c9: Pull complete
cbfbef8c41c9: Pull complete
d018da6f6dd: Pull complete
857399c34ab7: Pull complete
ec1ad240cb9c: Pull complete
dcc5dd42a10: Pull complete
Digest: sha256:244b4f37f778dbf10e2a6c0713101945100f5b2dbff62ec57f812fd6b452c6e
Status: Downloaded newer image for rabbitmq:3.10.5-management
docker.io/library/rabbitmq:3.10.5-management
ramesh@adarsh-MacBook-Air ~ % docker run --rm -it -p 15672:5672 rabbitmq:3.10.5-management
```

Port 5672 is used for the RabbitMQ client connections

So this port is for the rabbit management website and this port is to connect to a RabbitMQ client.

RabbitMQ using RabbitMQ Management UI :



A screenshot of the RabbitMQ Management UI Connections page at localhost:15672/#/connections. The page title is "Connections". It shows a table with one row: "All connections (0)". Below the table is a "Pagination" section with a dropdown for "Page" (set to 1), a "Filter" input field, and a "Displaying 0 item, page size up to: 100" message. At the bottom of the page, there is a footer with links: HTTP API, Server Docs, Tutorials, Community Support, Community Slack, Commercial Support, Plugins, GitHub, and Changelog. A red callout arrow points from the handwritten note "Well, whenever we connect our application with the RabbitMQ server that entry you can able to see" to the "All connections (0)" row in the table.

→ Producer and Consumer will basically exchange a message over a channel.

Exchange :

Create Exchange :

RabbitMQ Management

Refreshed 2022-06-20 19:26:03

Displaying 8 items, page size

Cluster rabbit User

Exchanges

Name	Type	Features	Message rate in	Message rate out
(AMQP default)	direct	D		
amq.direct	direct	D		
amq.fanout	fanout	D		
amq.headers	headers	D		
amq.match	headers	D		
amq.rabbitmq.trace	topic	D I		
amq.topic	topic	D		
exchange_demo	direct	D		

Add a new exchange

Name:

Type: direct

Durability: Durable

Auto delete: No

Internal: No

Arguments:

Add Alternative exchange

Add exchange

And as soon as you click on Add Exchange, a new entry will be added.

Creating Queues :

RabbitMQ Management

Refreshed 2022-06-20 19:26:03

Displaying 0 items, page size

Queues

All queues (0)

Add a new queue

Type: Classic

Name: queue_demo

Durability: Durable

Auto delete: No

Arguments:

Add Auto expire | Message TTL | Overflow behaviour
 Single active consumer | Dead letter exchange | Dead letter routing key
 Max length | Max length bytes
 Maximum priority | Lazy mode | Version | Master locator

Add queue

and let's keep all these values as a default and let's click on add queue here.

HTTP API Server Docs Community Support Commercial Support Plugins GitHub Changelog

Binding queue and Routing key in Exchange:

RabbitMQ Management - localhost:15672/exchanges/exchange_demo

Overview Connections Channels Exchanges **Queues** Admin

Virtual host Cluster rabbit@e679cf User guest

Refreshed 2022-06-20 19:29:51 Refresh every 5 seconds

Bindings

This exchange

↓

... no bindings ...

Add binding from this exchange

To queue: queue_demo *

Routing key: routing_key_demo

Arguments: String

Bind → Then click

Publish message

Routing key: ? Headers: ? Properties: ?

Payload: So just remember, we are binding this exchange named exchange_demo with a queue name queue

Sending Msg Exchange → Queue

RabbitMQ Management - localhost:15672/exchanges/exchange_demo

Overview Connections Channels Exchanges Queues Admin

Virtual host Cluster rabbit@e679cf User guest

Refreshed 2022-06-20 19:33:50 Refresh every 5 seconds

Bindings

Routing key: Arguments: String

Publish message

Routing key: routing_key_demo Headers: ? Properties: ?

Payload: hello world

Payload encoding: String (default)

Publish message

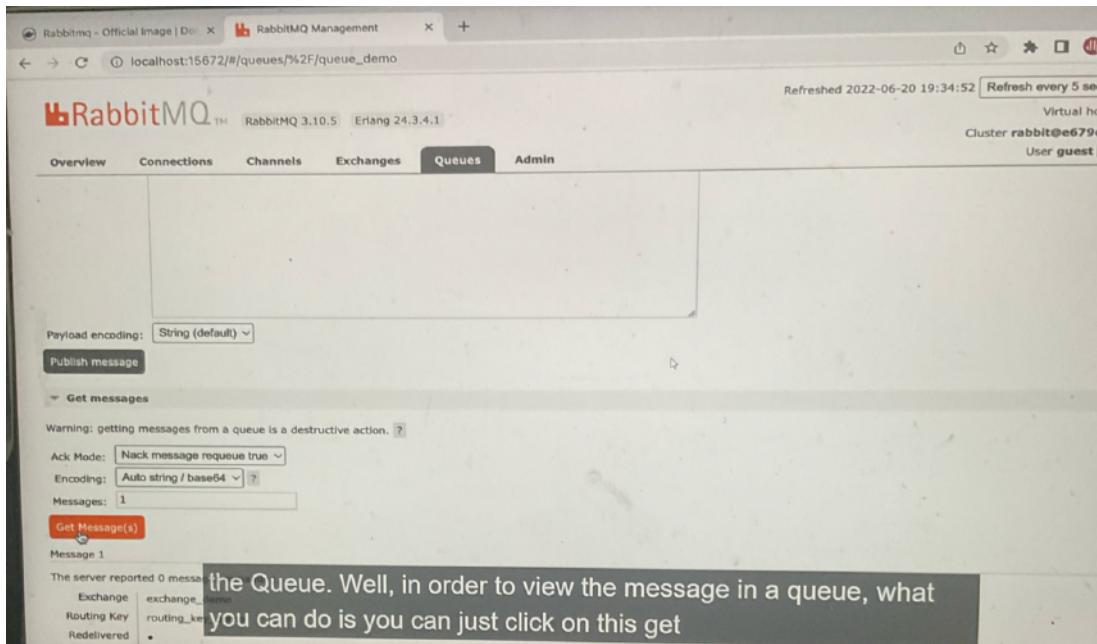
Delete this exchange

all right. Now, let's go ahead and click on this publish message

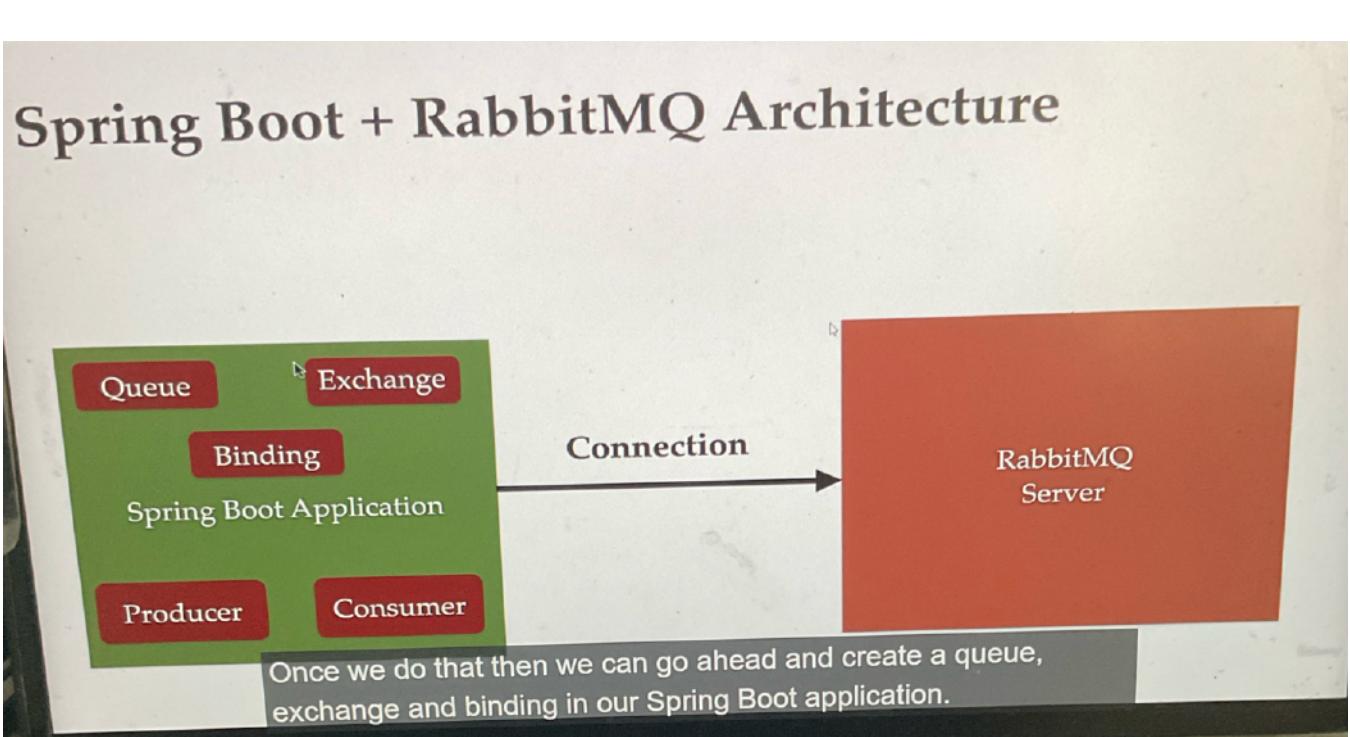
HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub ChangeLog

Read from Queue :

Click "get message to read from Queue".



Spring Amqp - advanced message Querying Protocol



Spring Boot Autoconfiguration for Spring AMQP (RabbitMQ)

We get a connection to our RabbitMQ broker on port 5672 using the default username and password of "guest".

Define these properties in a application.properties:

```
spring.rabbitmq.host=localhost  
spring.rabbitmq.port=5672  
spring.rabbitmq.username=guest  
spring.rabbitmq.password=guest
```

the RabbitMQ Broker on Port 5672 and uses the username password as a guest.

The screenshot shows an IDE interface with the following details:

- Project Structure:** A tree view showing a project named "springboot-rabbitmq" with packages like `main.java` and `resources`.
- Code Editor:** An open file `RabbitMQConfig.java` containing Java code for RabbitMQ configuration. It includes annotations like `@Value`, `@Bean`, and `@Builder`.
- Diagram:** A diagram titled "RabbitMQ Architecture with Multiple Queues" illustrating the flow from a Producer to an Exchange, which then routes messages to multiple Queues, each connected to a Consumer.
- IDE Status:** Shows various toolbars and status bars typical of an IDE environment.

and few more Configuration needed

// ConnectionFactory } RabbitMQ Auto Config
// RabbitTemplate
// RabbitAdmin } Take care of everything.

The screenshot shows the IntelliJ IDEA interface with the following details:

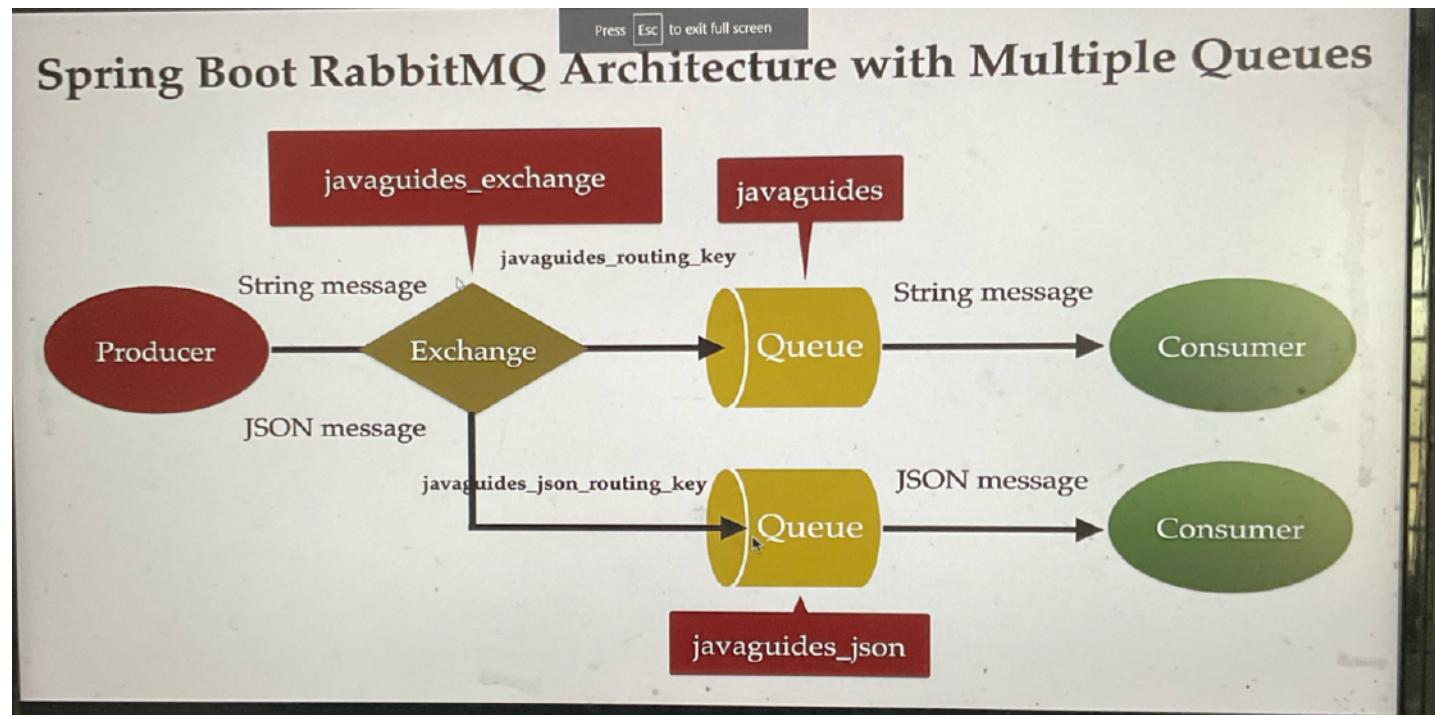
- Project Structure:** The project is named "SpringbootRabbitmqTutorialApplication". It contains a "src" directory with a "main" package. Inside "main", there are "java", "net", "javaguides", "springboot", and "publisher" packages. The "publisher" package is currently selected.
- Code Editor:** The file "RabbitMQProducer.java" is open. The code defines a class "RabbitMQProducer" with fields "exchange" and "routingKey" annotated with @Value("\${...}"). It also includes a logger field and a constructor that takes a "RabbitTemplate" parameter. A method "sendMessage" is defined which logs the message and uses the template to convert and send it.
- Toolbars and Status Bar:** The bottom toolbar includes buttons for Version Control, Run, TODO, Problems, Terminal, Build, and Dependencies. The status bar at the bottom right shows the time as 22:11 and the date as Tue Aug 14.

Sending Message:

< Home [NEW] Building Microservices with Spring Boot & Spring Cloud

```
 3 @RestController
 4 @RequestMapping("/api/v1")
 5 public class MessageController {
 6
 7     private RabbitMQProducer producer;
 8
 9     public MessageController(RabbitMQProducer producer) {
10         this.producer = producer;
11     }
12
13     // http://localhost:8080/api/v1/publish?message=hello
14     @GetMapping("/publish")
15     public ResponseEntity<String> sendMessage(@RequestParam("message") String message){
16         producer.sendMessage(message);
17         return ResponseEntity.ok("Message sent to RabbitMQ ...");
18     }
19 }
```

JSON Communication

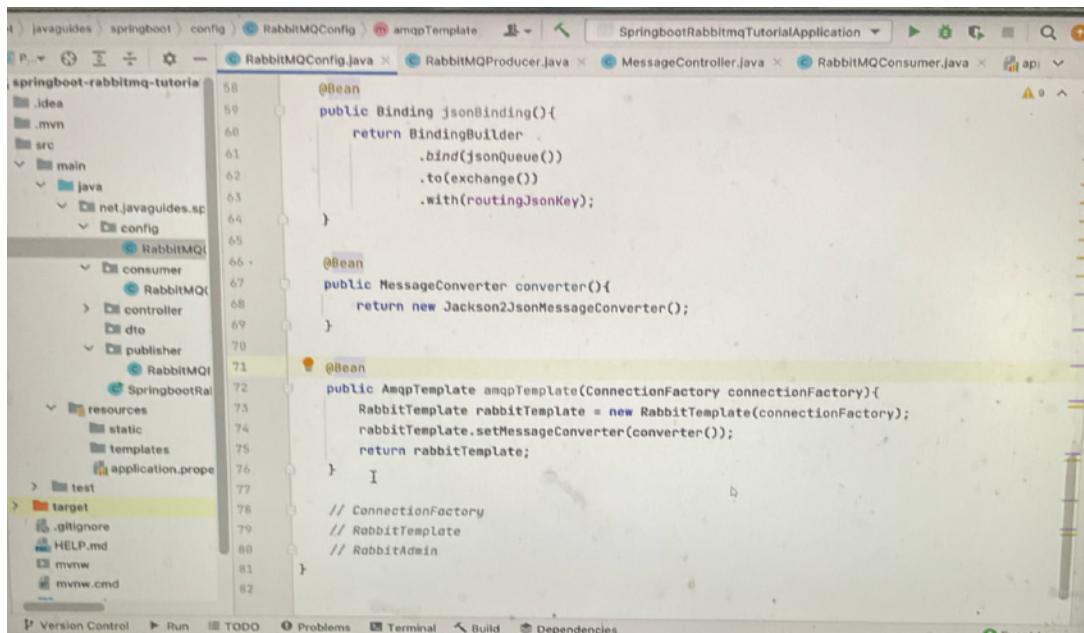


```
    // Spring boot JPA tutorial example
    @Bean
    public TopicExchange exchange(){
        return new TopicExchange(exchange);
    }

    // binding between queue and exchange using routing key
    @Bean
    public Binding binding(){
        return BindingBuilder
            .bind(queue())
            .to(exchange())
            .with(routingKey);
    }

    // binding between json queue and exchange using routing key
    @Bean
    public Binding jsonBinding(){
        return BindingBuilder
            .bind(jsonQueue())
            .to(exchange())
            .with(routingJsonKey);
    }

    // ConnectionFactory
    // RabbitTemplate
```

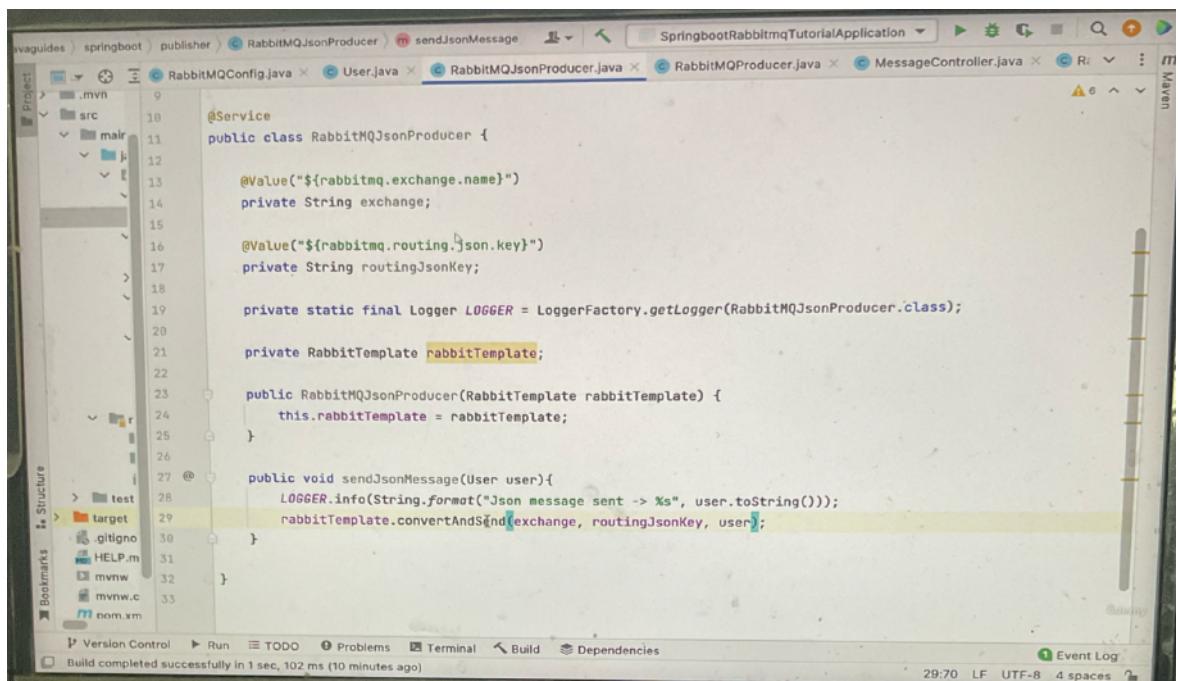


```
public Binding jsonBinding(){
    return BindingBuilder
        .bind(jsonQueue())
        .to(exchange())
        .with(routingJsonKey);
}

public MessageConverter converter(){
    return new Jackson2JsonMessageConverter();
}

public AmqpTemplate amqpTemplate(ConnectionFactory connectionFactory){
    RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
    rabbitTemplate.setMessageConverter(converter());
    return rabbitTemplate;
}
```

Producers:



```
@Service
public class RabbitMQJsonProducer {

    @Value("${rabbitmq.exchange.name}")
    private String exchange;

    @Value("${rabbitmq.routing.json.key}")
    private String routingJsonKey;

    private static final Logger LOGGER = LoggerFactory.getLogger(RabbitMQJsonProducer.class);

    private RabbitTemplate rabbitTemplate;

    public RabbitMQJsonProducer(RabbitTemplate rabbitTemplate) {
        this.rabbitTemplate = rabbitTemplate;
    }

    public void sendJsonMessage(User user){
        LOGGER.info(String.format("Json message sent -> %s", user.toString()));
        rabbitTemplate.convertAndSend(exchange, routingJsonKey, user);
    }
}
```

Controller Json :

A screenshot of an IDE (Spring Boot IDE) showing the `MessageJsonController.java` file. The code defines a controller for sending JSON messages to RabbitMQ. It includes imports for Spring Framework annotations and a producer interface. The controller has a constructor dependency injection and a POST mapping method that sends a JSON message to the queue.

```
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/v1")
public class MessageJsonController {

    private RabbitMQJsonProducer jsonProducer;

    public MessageJsonController(RabbitMQJsonProducer jsonProducer) {
        this.jsonProducer = jsonProducer;
    }

    @PostMapping("/publish")
    public ResponseEntity<String> sendJsonMessage(@RequestBody User user){
        jsonProducer.sendJsonMessage(user);
        return ResponseEntity.ok("Json message sent to RabbitMQ ...");
    }
}
```

Consumer Json :

A screenshot of an IDE showing the `RabbitMQJsonConsumer.java` file. The code defines a consumer service that listens to a queue and logs received JSON messages. It uses Spring's `@RabbitListener` annotation to handle incoming messages.

```
package net.javaguides.springboot.consumer;

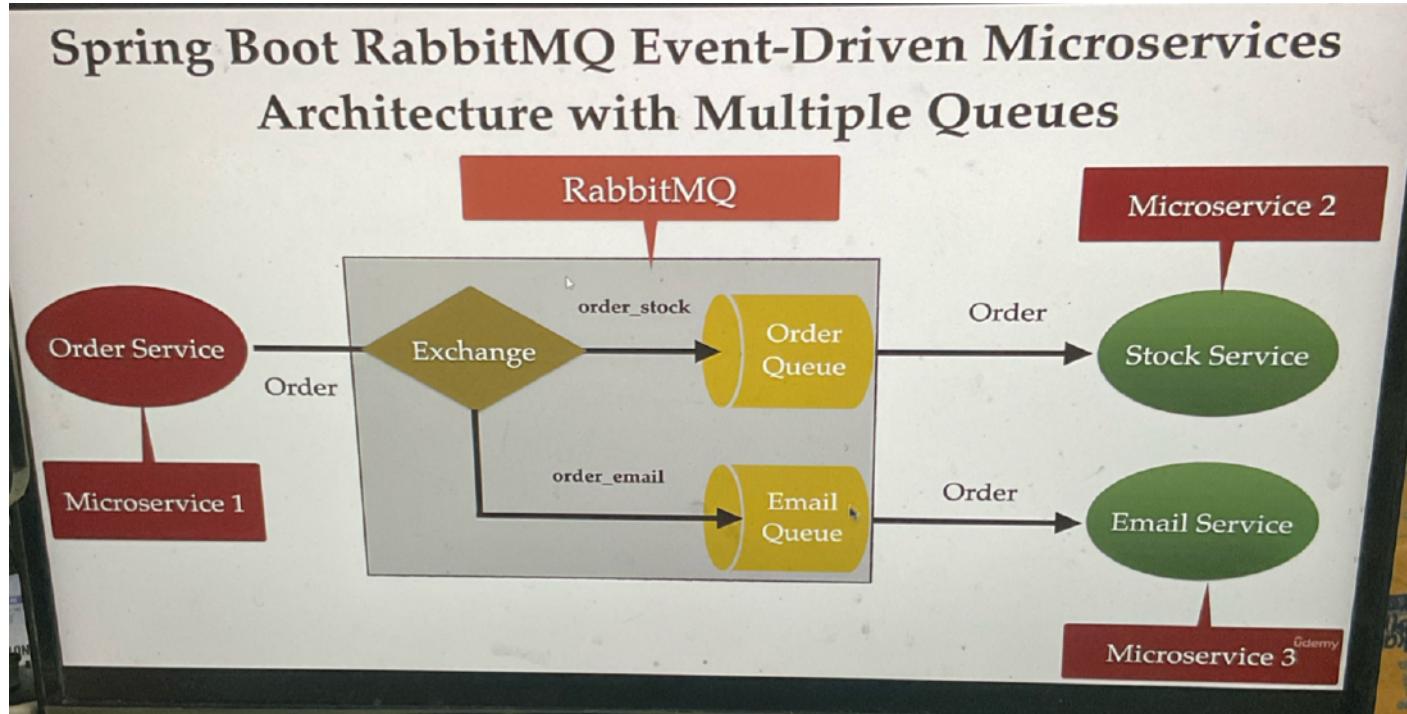
import net.javaguides.springboot.dto.User;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class RabbitMQJsonConsumer {

    private static final Logger LOGGER = LoggerFactory.getLogger(RabbitMQJsonConsumer.class);

    @RabbitListener(queues = {"${rabbitmq.queue.json.name}"})
    public void consumeJsonMessage(User user){
        LOGGER.info(String.format("Received JSON message -> %s", user.toString()));
    }
}
```

Event Driven Architecture



Report github : Dinesh333s



Practice RabbitMQ

Activity Public repository