RabbitMQ:

- 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 asynchronous communication between applications.
- A message broker acts as a central hub (like a post office) for the exchange of messages between different applications and services.
- RabbitMQ is an open-source message-broker(message queue software) software that act as an intermediary platform where different applications can send and receive messages.
- RabbitMQ originally implemented the Advanced Message Queuing Protocol (AMQP) but has since been extended to support other messaging protocols such as MQTT, STOMP, and HTTP.
- It is written in Erlang and provides a messaging queue that allows applications to communicate with each other asynchronously.
- RabbitMQ is widely used for its reliability, flexibility, and scalability in distributed systems and microservices architectures.
- RabbitMQ is freely available and known for its robustness, making it popular in production environments.
- RabbitMQ is a powerful messaging system that enables developers to build distributed and scalable applications by providing reliable message queuing and delivery mechanisms.

Key Concepts:

- ✓ **Message:** Information that is sent from the producer to a consumer through RabbitMQ.
- ✓ **Producers:** Applications that send messages to the broker.
- ✓ **Consumers:** Applications that receive messages from the broker.
- ✓ **Queues:**
    - Queue is a buffer or storage in a RabbitMQ broker to store the messages.
    - Messages are placed into queues by producers and then consumed by consumers.
    - Once a message Is read, it is consumed and removed from the queue.
    - A message can thus only be processed exactly once.
    - Queues are the basic mechanism that RabbitMQ provides for message storage and delivery.

- ✓ **Exchanges:**
  - Entry points where producers send messages. Producers send messages to exchanges, which then route them to one or more queues based on defined rules (bindings).
  - 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 queue based on specified set of rules.
- ✓ **Bindings:** Bindings define the relationship between exchanges and queues, determining how messages are routed.
- ✓ **Routing Keys:** Additional attributes on messages used by exchanges to make more refined routing decisions. Decides how to route the message to queues.
- ✓ **Durability:** Messages and queues can be made durable, ensuring that they survive broker restarts.

Features:

- ✓ **Message Broker:** RabbitMQ acts as a middleman between producers (applications that send messages) and consumers (applications that receive messages).
- ✓ **Messaging Protocols:** It supports multiple messaging protocols including AMQP, MQTT, STOMP, and HTTP.
- ✓ **Management Interface:** RabbitMQ comes with a web-based management interface that allows administrators to monitor and manage the broker.

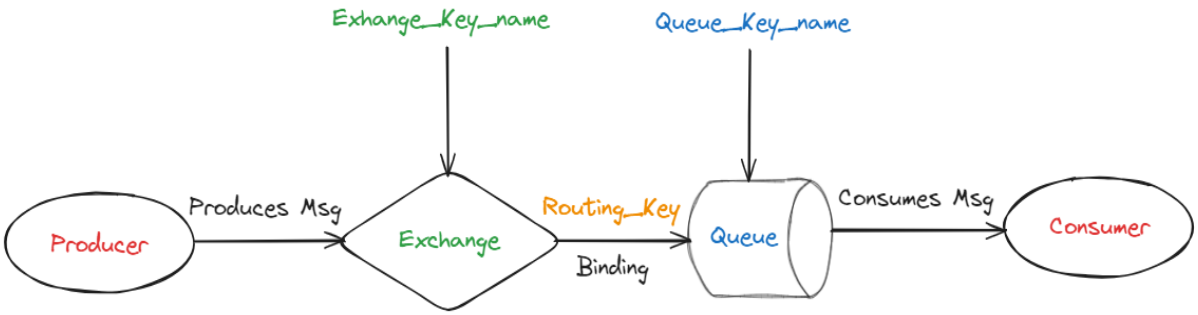

Fig. RabbitMQ Architecture

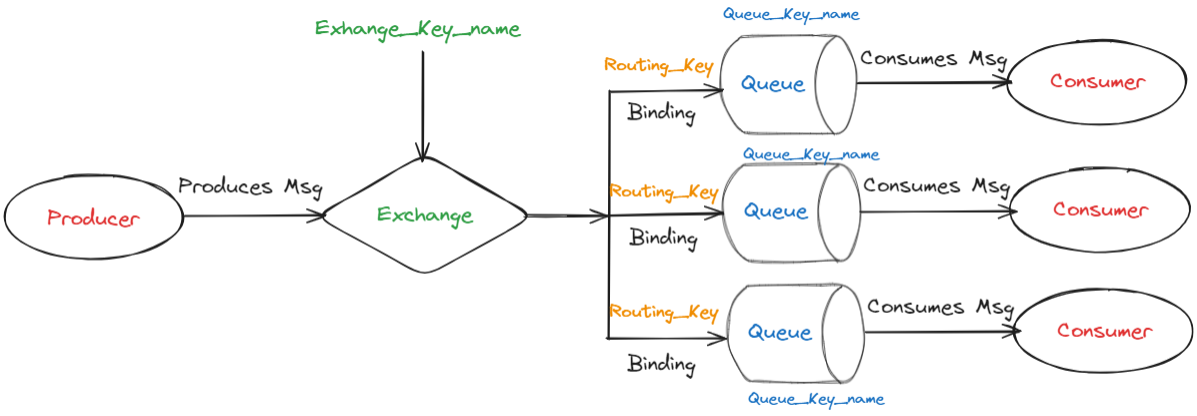

Fig. RabbitMQ Architecture With Multiple Queues

Installation and Setup:

RabbitMQ will listen to port on 5672 by default

RabbitMQ management will listen on port 15672

Installation and running in Local: (Always choose management version)

1. Install Erlang - https://www.erlang.org/downloads
2. Install RabbitMQ - https://www.rabbitmq.com/docs/install-windows
3. Set Environment variable-(if not set automatically during installation)

   RABBITMQ_BASE c:\RabbitMQ Server (Not necessary)

   ERLANG_HOME C:\Program Files\erl10.7
4. Open CMD as administrator and change directory to RabbitMQ sbin directory (C:\Program Files\RabbitMQ Server\rabbitmq_server-3.13.0\sbin)
5. Enable the plugins.

   $ rabbitmq-plugins enable rabbitmq_management
6. Start the service ,

   $ rabbitmq-service start
7. Go to http://localhost:15672/

Run using Docker in Local: (Always choose management version)

1. Install Docker Desktop software locally.
2. Open CMD or PowerShell and execute command,

   $ docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.13.0-management
3. Go to http://localhost:15672/

Run on AWS cloud:

1. Login into AWS and Launch EC2 instance (Amazon Linux Machine)
2. Connect with Linux VM using MobaXterm software or Putty.
3. Install docker in Linux VM,

   $ sudo yum update -y

   $ sudo yum install docker -y

4. Start docker service,

   $ sudo service docker start

5. Add docker ec2-user to user group,

   $ sudo usermod -aG docker ec2-user

6. Exit and restart the session by pressing R in MobaXterm

   $ exit

   and Press R

7. Verify docker installed or not by using,

   $ docker -v

8. Execute the below command,

   $ docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.13.0-management

9. Enable 15672, 5672 port on Aws Security group>Edit Inbound rules.
10. Go to http://aws-public-ip:15672/

If RabbitMQ not working properly(RabbitMQ fails to start), Follow the below steps:

1. Run the command prompt as Administrator and change directory to RabbitMQ sbin directory.
2. Navigate to the sbin directory and uninstall the service.

   $ rabbitmq-service remove

3. Reinstall the service

   $ rabbitmq-service install

4. Enable the plugins.

   $ rabbitmq-plugins enable rabbitmq_management

5. Start the service

   $ rabbitmq-service start

6. Go to http://localhost:15672/

Spring Boot with RabbitMQ:

Publisher and Subcriber Model:

Publisher:

1. Create Spring Boot App and Add required maven Dependency,

```
<dependency>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

2. Add RabbitMQ configuration in appication.propertiesor yml file,
   - If RabbitMQ running on local then no need of configuration, by default spring boot will auto configure RabbitMQ connection.
   - If ruuning on different machine or host(example: Running On AWS), then use,

```
spring.rabbitmq.host=3.108.66.77
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
```

3. Create Constant class to define Names to Queue, Exchange, Routing Key or fetch from properties file or yml using @Value.

```
package in.nk.tech.rabbitmq.constants;
public class AppConstants {
	public static final String QUEUE = "nk_pubsub_queue";
	public static final String EXCHANGE = "nk_pubsub_exchange";
	public static final String ROUTING_KEY = "nk_pubsub_routing_key";
}
```

4. Create RabbitMQConfig class, to configure Queue, Exchange, Binding, MessageConverter(In case of JSON), and RabbitTemplate(only if we use MessageConverter, or else by default auto configured)

```
package in.nk.tech.rabbitmq.config;

import org.springframework.amqp.core.AmqpTemplate;
import org.springframework.amqp.core.Binding;
import org.springframework.amqp.core.BindingBuilder;
import org.springframework.amqp.core.DirectExchange;
import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import in.nk.tech.rabbitmq.constants.AppConstants;

@Configuration
public class RabbitMQConfig {

	@Bean
	public Queue queue() {
		return new Queue(AppConstants.QUEUE);
	}

	@Bean
	public DirectExchange directExchange() {
		return new DirectExchange(AppConstants.EXCHANGE);
	}

	@Bean
	public Binding binding() {
		return
BindingBuilder.bind(queue()).to(directExchange()).with(AppConstants.ROUTING_KEY);
	}

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

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

}
```

5. Create DTO class as per requirement.

6. Create Publisher class to publish message to Queue,

```
package in.nk.tech.rabbitmq.publisher;

import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Service;

import in.nk.tech.rabbitmq.constants.AppConstants;
import in.nk.tech.rabbitmq.dto.OrderStatus;
import lombok.extern.slf4j.Slf4j;
```

```
@Service
@Slf4j
public class Publisher {

	private RabbitTemplate rabbitTemplate;

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

	public String sendUserDetails(OrderStatus orderStatus) {
		rabbitTemplate.convertAndSend(AppConstants.EXCHANGE,
AppConstants.ROUTING_KEY, orderStatus);
		log.info("Message Sent {}", orderStatus);
		return "Message Sent - " + orderStatus;
	}

}
```

7. Create Restcontroller class as per requirement and publish messages.
8. Go to RabbitMQ Management http://localhost:15672/

Subscriber:

1. Follow the publisher steps from 1-5 (everything should be same)
2. Create a Subscriber class to listen to the published messages as per your requirement,(Must listen to same Queue).

```
package in.nk.tech.rabbitmq.consumer;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import in.nk.tech.rabbitmq.constants.AppConstants;
import in.nk.tech.rabbitmq.dto.OrderStatus;
import in.nk.tech.rabbitmq.utils.EmailUtils;
import lombok.extern.slf4j.Slf4j;

@Service
@Slf4j
public class Consumer {

	@Autowired
	private EmailUtils emailUtils;

	@RabbitListener(queues = AppConstants.QUEUE)
	public void consumeUserData(OrderStatus orderStatus) {
		String to = orderStatus.getOrder().getUser().getEmail();
		System.out.println("to: " + to);
```

```
                String name = orderStatus.getOrder().getUser().getName();
                String subject = null;
                if ("Processing".equalsIgnoreCase(orderStatus.getStatus())) {
                        subject = "Order Placed";
                        String body = "<h3>Hi " + name + ", </h3> " + "<p>Your order
placed successfully.</p>" + "<p>Order Id: "
                                        + "<b>" + orderStatus.getOrder().getOrderId() +
"</b></p>" + "<p>Order Name: " + "<b>"
                                        + orderStatus.getOrder().getProduct() +
"</b></p>" + "<p>Quantity: " + "<b>"
                                        + orderStatus.getOrder().getQuantitiy() +
"</b></p>" + "<p>Price: " + "<b>"
                                        + orderStatus.getOrder().getPrice() + "</b></p>";

                        emailUtils.sendEmail(to, subject, body);
                        log.info("Data Consumed, {}", orderStatus);
                } else {
                        subject = "Order Delivered";
                        String body = "<h3>Hi " + name + ", </h3> " + "<p>Your order
delivered successfully.</p>" + "<p>Order Id: "
                                        + "<b>" + orderStatus.getOrder().getOrderId() +
"</b></p>" + "<p>Order Name: " + "<b>"
                                        + orderStatus.getOrder().getProduct() +
"</b></p>" + "<p>Quantity: " + "<b>"
                                        + orderStatus.getOrder().getQuantitiy() +
"</b></p>" + "<p>Price: " + "<b>"
                                        + orderStatus.getOrder().getPrice() + "</b></p>" +
"<p>Thank You</p>";

                        emailUtils.sendEmail(to, subject, body);
                        log.info("Data Consumed, {}", orderStatus);
                }
        }

}
```

3. Check in console for messages.