**RabbitMQ**

* RabbitMQ is message broker that originally implements the AMQP (Advanced Message Queueing Protocol
* AMQP standardizes messaging using producers, Broker, and Consumers

**Key Features:**

* Security -support authentication,authorization,LDAP,and TLS via RabbitMQ plugins
* Reliability – confirms the message was successfully delivered to the message broker and confirms that the message was successfully processed by consumer.
* Interoperability – message is transfer as stream of bytes so any clients can operate on it irrespective of any language.

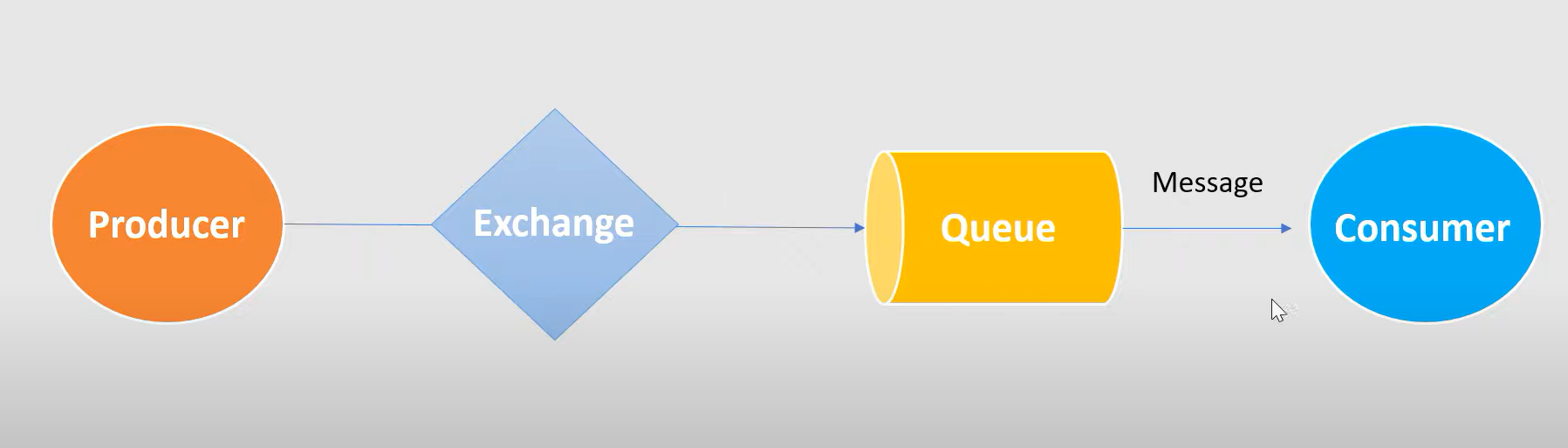
**Working of RabbitMQ Architecture**

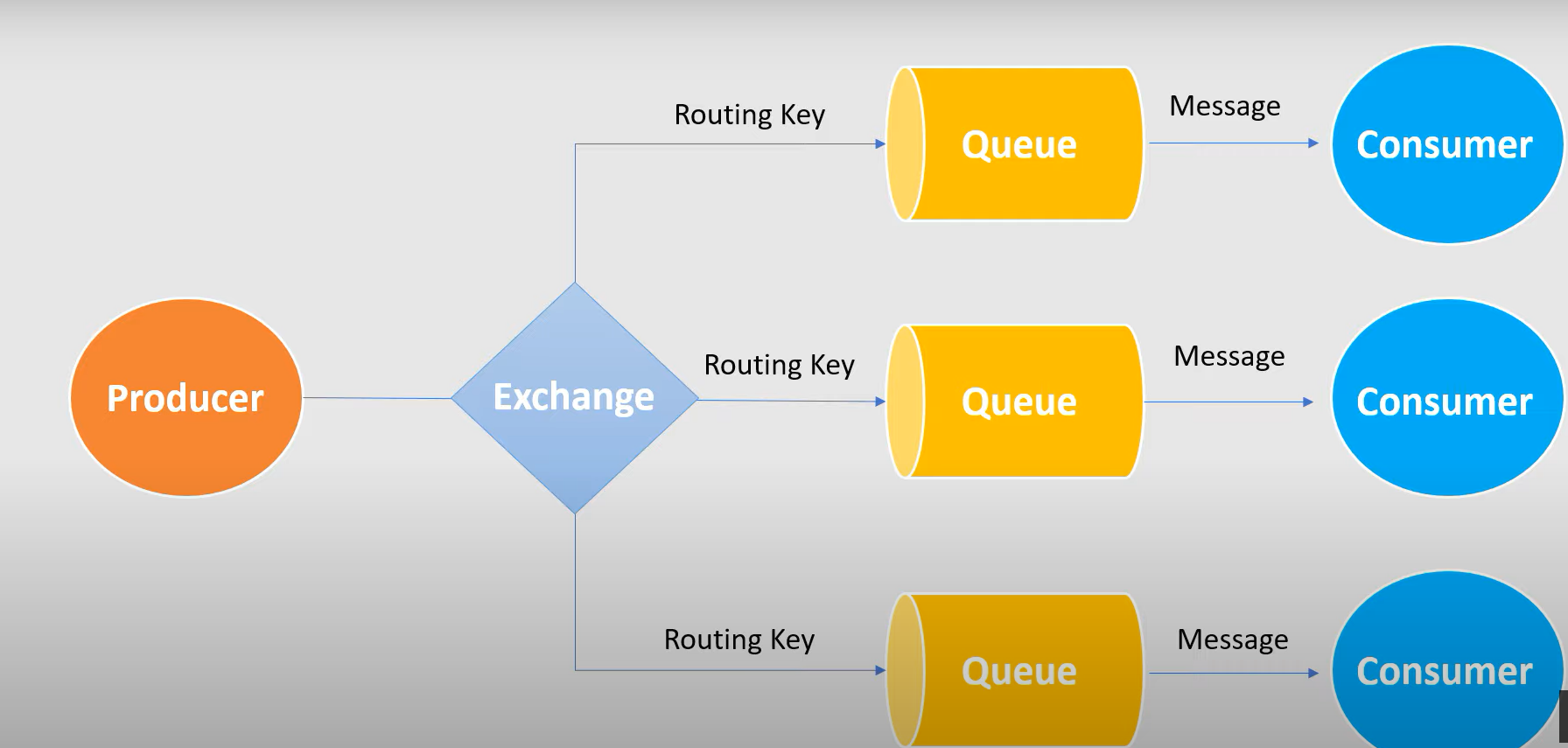
**Producer**: The producer is responsible for creating messages and sending them to RabbitMQ. Messages can be any form of data, such as text, JSON, XML, etc.

**Exchange**: The exchange receives messages from producers and routes them to the appropriate queues. Exchanges use routing rules, called bindings, to determine how messages should be distributed.

**Queue**: Queues are the ones who store the messages of the Producers and send to Consumers if they are ready.

**Bindings**: Bindings define the relationship between exchanges and queues. They specify which queues should receive messages from a particular exchange and may include routing keys or headers that further refine the routing behavior.



**RoutingKey**: The message from producer is sent to the queue with the help of Routing Key

In Summary, the producer will produce the message and Queue will store the message and use it for the transmission. The exchange is used to route to queue with the help of the Routing key. If the consumer is not ready, then the messages will be in the queue itself. If the consumer is ready the messages will be consumed.

**Demo Project was done to illustrate the RabbitMQ.**

**Dependencies used for the project.**

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-amqp</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<optional>true</optional>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

<dependency>

<groupId>org.springframework.amqp</groupId>

<artifactId>spring-rabbit-test</artifactId>

<scope>test</scope>

</dependency>

**MessagingConfig Class:**

The QUEUE, EXCHANGE,ROUTING\_KEY are the variables which are used for the Transmission. The converter () is used to convert the message to Json and the template method is used to connect to RabbitMQ

@Configuration

public class MessagingConfig {

public static final String QUEUE = "neokred\_queue";

public static final String EXCHANGE = "neokred\_exchange";

public static final String ROUTING\_KEY = "neokred\_routingKey";

@Bean

public Queue queue() {

return new Queue(QUEUE);

}

@Bean

public TopicExchange exchange() {

return new TopicExchange(EXCHANGE);

}

@Bean

public Binding binding(Queue queue,TopicExchange exchange)

{

return BindingBuilder.bind(queue).to(exchange).with(ROUTING\_KEY);

}

//To covert the maeesage to JSON

@Bean

public MessageConverter converter() {

return new Jackson2JsonMessageConverter();

}

@Bean

public AmqpTemplate template(ConnectionFactory connectionFactory) {

final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);

rabbitTemplate.setMessageConverter(converter());

return rabbitTemplate;

}

}

**Order Entity Class:**

@Data

@AllArgsConstructor

@NoArgsConstructor

@ToString

public class Order {

private String orderId;

private String name;

private int qty;

private double price;

}

**OrderStatus Class:**

@Data

@AllArgsConstructor

@NoArgsConstructor

@ToString

public class OrderStatus {

private Order order;

private String status;//progress,completed

private String message;

}

**OrderPublisher Class:**

This is the controller class in the Project. Here we are taking the object of Order as RequestBody and restaurant name as PathVariable. The message will be stored in Order status and will be sent by the template.convertAndSend().The message will ne queued in the Queue itself and will not be delivered to Consumer until it is ready.

@RestController

@RequestMapping("/order")

public class OrderPublisher {

@Autowired

private RabbitTemplate template;

@PostMapping("/{restaurantName}")

public String bookOrder(@RequestBody Order order, @PathVariable String restaurantName) {

order.setOrderId(UUID.randomUUID().toString());

//restaurantservice

//payment service

OrderStatus orderStatus = new OrderStatus(order, "PROCESS", "order placed succesfully in " + restaurantName);

template.convertAndSend(MessagingConfig.EXCHANGE, MessagingConfig.ROUTING\_KEY, orderStatus);

return "Success !!";

}

**Steps to view the queued message in the RabbitMQ UI**

* Hit the API in the postman.
* Go to RabbitMQ source folder in C Drive until u get the sbin folder and copy the path
* Go to cmd and Run as Administrator
* Execute the below commands.

cd C:\Program Files\RabbitMQ Server\rabbitmq\_server-3.8.8\sbin

rabbitmq-plugins enable rabbitmq\_management

rabbitmq-service.bat start

If it throws the error for rabbitmq-service.bat, then execute the following command

rabbitmq-service remove

rabbitmq-service install

rabbitmq-service.bat start

* Hit the <http://localhost:15672/> endpoint.
* The default login and password will be guest and guest.
* Go to Queues
* Select the queue which u have created.

A screenshot of a computer

Description automatically generated

* We will get the UI
* The messages will be in the ready state to reach the end user or Consumer
* Enter the number of messages and click on getMessages, we will get all the messages

A screenshot of a computer

Description automatically generated

**User Class:**

The consumeMesssageFromQueue is used to consume the message produced from the Producer and when we hit the API, there will be zero messages in the RabbitMQ UI. The message consumed by the consumer will be displayed in the console.

@Component

public class User {

@RabbitListener(queues = MessagingConfig.QUEUE)

public void consumeMessageFromQueue(OrderStatus orderStatus) {

System.out.println("Message recieved from queue : " + orderStatus);

}

}

A screenshot of a computer

Description automatically generated