Rabbit mq architecture

We look at rabbitmq concepts like binding, queue, message, producer and consumer.
Then we would learn the architectures of the rabbit MQ in spring boot.
Learn how to create multiple queues in rabbit mq architecture.

Originally the rabbit MQ implements the Advance messaging queue protocol(AMQP), it also supports other Api protocols such as STOMP, Http etc.
A message queue is made up of a producer, a broker (the message queue software), and a consumer. It serves to provide asynchronous communication between applications.

Producer => Message => Message Broker => message => consumer

PRODUCER
Producer is an application that sends messages. It does not send messages directly to the consumer but to the broker first.
CONSUMER
Consumer reads messages from the rabbit MQ broker. There are many consumers that can subscribe to the rabbitMQ broker.

QUEUE
Queue is a buffer or storage in the rabbit MQ broker to store messages. The messages are put into a queue by the producers and read from it by the consumer. Once a message is read, it is consumed and removed from the queue. A message can only be processed just once.

EXCHANGE
Exchange acts as an intermediary between the producer and a queue. Instead of sending the message to the queue directly, the producer sends them to the exchange. The exchange then sends the message to one or more queues following specified sets of rules. The producer does not need to know the queue that receives the messages.

ROUTING KEYS

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 queues.

BINDING
This is the link between a queue and an exchange. Almost like the routing key especially where we have multiple queues in the application.

Install and set up rabbit MQ

The installation and setup we are using for this is with the docker container. You can either have a docker compose file for easy spinning up of the rabbit mq or you can use the terminal which involves going to docker hub and then using the docker command of docker pull rabbitmq:specify the exact one that you wish to pull and afterward you can now add the command to set up the environment {docker run --rm -it -p 15672:15672 -p 5672:5672 rabbitmq:3.10.25-management}.

Create the application.properties in the spring boot project and have the rabbit mq configured.

spring.rabbitmq.host=localhost i.e if you are developing locally

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

After this we need to configure the rabbit mq t be able to create message queues, the exchange layer and also the binding with keys.

```
import org.springframework.amqp.core.*;
import
org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
org.springframework.amqp.support.converter.Jackson2JsonMessag
eConverter;
import
org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
```

```java
import org.springframework.context.annotation.Configuration;

@Configuration
public class RabbitMqConfig {
   @Value("${rabbitmq.queue.name}")
   private  String queue;
   @Value("${rabbitmq.queue.json.name}")
   private  String jsonQueue;
   @Value("${rabbitmq.exchange.name}")
   private String exchange;
   @Value("${rabbitmq.routing.key}")
   private String routingKey;
   @Value("${rabbitmq.routing.json.key}")
   private String jsonRoutingKey;
   //spring bean for rabbit mq queue
   @Bean
   public Queue queue(){
       return new Queue(queue);
   }

   @Bean
   public TopicExchange exchange(){
       return new TopicExchange(exchange);
   }
   @Bean
   public Queue jsonQueue(){
       return new Queue(jsonQueue);
   }

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

```
                .to(exchange())
                .with(jsonRoutingKey);
    }
    @Bean
    public MessageConverter converter(){
        return new Jackson2JsonMessageConverter();
    }

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

    //connection factory
    //RabbitTemplate
    //RabbitAdmin
}
```

The the values are injected from the application.properties

```
rabbitmq.queue.name=javaguides
rabbitmq.exchange.name=javaguides_exchange
rabbitmq.routing.key=javaguides_routing_key
rabbitmq.queue.json.name=javaguides_json
rabbitmq.routing.json.key=javaguides_routing_json_key
```

There are basically two exchanges configured the one that will be to produce and consume Objects but not in Json formats and the one that will produce JSON formats that is also why we had the Json message converter configured in our rabbitTemplate to aid in serialization and deserialization into java classes and JSON as the case may be.

Next we build our producer classes and have them use the appropriate routing keys and queues to publish the messages to have the consumers consume it

```
package com.maximillian.springbootrabbit.publisher;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import
org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;

@Service
//for the producer you will need the exchange name and the
routing key name as well
public class RabbitMQProducer {

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

   private static final Logger LOGGER =
LoggerFactory.getLogger(RabbitMQProducer.class);
   private final RabbitTemplate rabbitTemplate;
   public RabbitMQProducer(RabbitTemplate rabbitTemplate) {
       this.rabbitTemplate = rabbitTemplate;
   }

   public void sendMessage(String message){
       LOGGER.info(String.format("Message sent -> %s,",
message));
       rabbitTemplate.convertAndSend(exchange, routingKey,
message);
   }
}
```

This produces the object type that is not of Json and has the appropriate consumer to consume it.

```
package com.maximillian.springbootrabbit.consumer;

import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import
org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Service;

@Service
public class RabbitMQConsumer {
   private static final Logger LOGGER =
LoggerFactory.getLogger(RabbitMQConsumer.class);

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

Then with the rabbitTemplate configured we could now send a Json format to the queue that will be waiting to be consumed by the consumer. But first we would have to create a user class:

```
package com.maximillian.springbootrabbit.dto;

import lombok.Data;

@Data
public class User {
   private int id;
   private String firstName;
   private String lastName;
}
```

Then after having this we will now go ahead and create the user pojo which we would use as the JSON class for serializing and deserializing

```
package com.maximillian.springbootrabbit.dto;

import lombok.Data;
```

```
@Data
public class User {
   private int id;
   private String firstName;
   private String lastName;
}
```

Then we will now create a produce of the json class:

```
@Service
@RequiredArgsConstructor
public class RabbitMQJsonProducer {
   @Value("${rabbitmq.routing.json.key}")
   private String jsonRoutingKey;

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

   private final RabbitTemplate rabbitTemplate;

   private static final Logger LOGGER =
LoggerFactory.getLogger(RabbitMQJsonProducer.class);
   public void sendMessageJson(User user){
       LOGGER.info(String.format("json message sent -> %s",
user.toString()));
       rabbitTemplate.convertAndSend(exchange,
jsonRoutingKey, user);

   }
}
```

And then a consumer that would consume the message:

```
@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()));

    }
}
```

We would also create controllers for each of the produces to send the messages to the queue.

```
@RestController
@RequestMapping("/api/v1")
@RequiredArgsConstructor
public class MessageJsonController {
    private final RabbitMQJsonProducer producer;

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

}
```

This is for the JSON message publishers

Then we have the controllers for the object consumers

```

@RestController
@RequestMapping("/api/v1")
@RequiredArgsConstructor
public class MessageController {
    private final RabbitMQProducer producer;
    @GetMapping("/publish")
    public ResponseEntity<String>
sendMessage(@RequestParam("message") String message){
        producer.sendMessage(message);
```

```
        return ResponseEntity.ok("message sent to
RabbitMQ.....");
    }

}
```