RabbitMQ with Spring boot

Agenda

- What is Messaging Queue, How it works and its uses
- What is Rabbit MQ
- Different types of Exchanges in Rabbit, MQ
- What is Messaging Queue, How it works and its uses

  Different types of Exchanges in Rabbit MQ .
  - Direct Exchanges
  - Fanout Exchanges
  - Topic Exchanges
  - Header Exchanges
- Retry and Error Handling Example

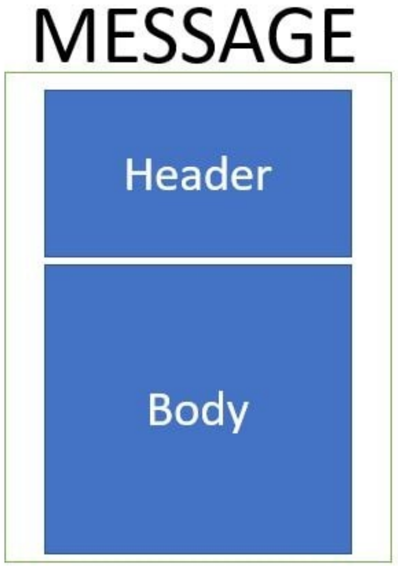
What is RabbitMQ ?

- What is RabbitMQ ?
  - RabbitMQ is a message broker that originally implements the Advance Message Queuing Protocol (AMQP)
- AMQP standardizes messaging using Producers, Broker and Consumers.
  - AMQP standards was designed with the following main characteristics Security, Reliability, Interoperability
  - Reliability confirms the message was successfully delivered to the message broker and confirms that the message was successfully processed by the consumer

What is Messaging?

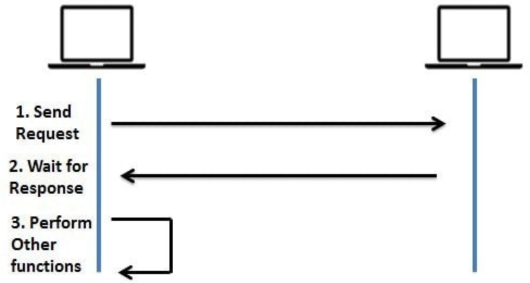
Messaging is a communication mechanism used for system interactions. In software development messaging enables distributed communication that is loosely coupled.

A messaging client can send messages to, and receive messages from, any other client. The structure of message can be defined as follows



Synchronous Messaging

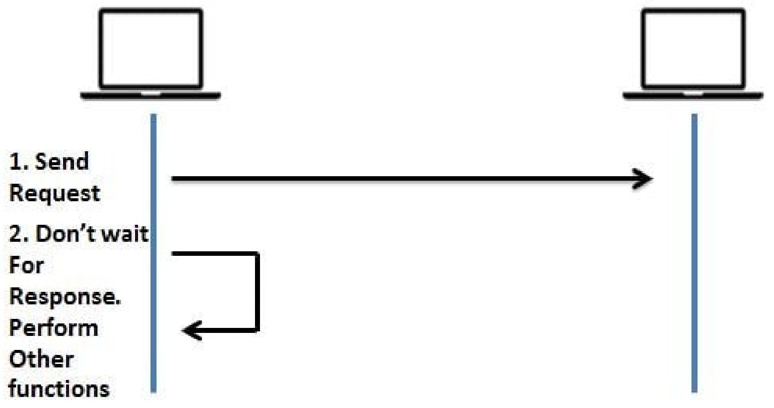
Synchronous Messaging is implemented when such that the messaging client sends a message and expects the response immediately. So the sender client waits for the response before he can execute the next task. So until and unless the message is recieved the sender is blocked.



Synchronous Communication

Asynchronous Messaging

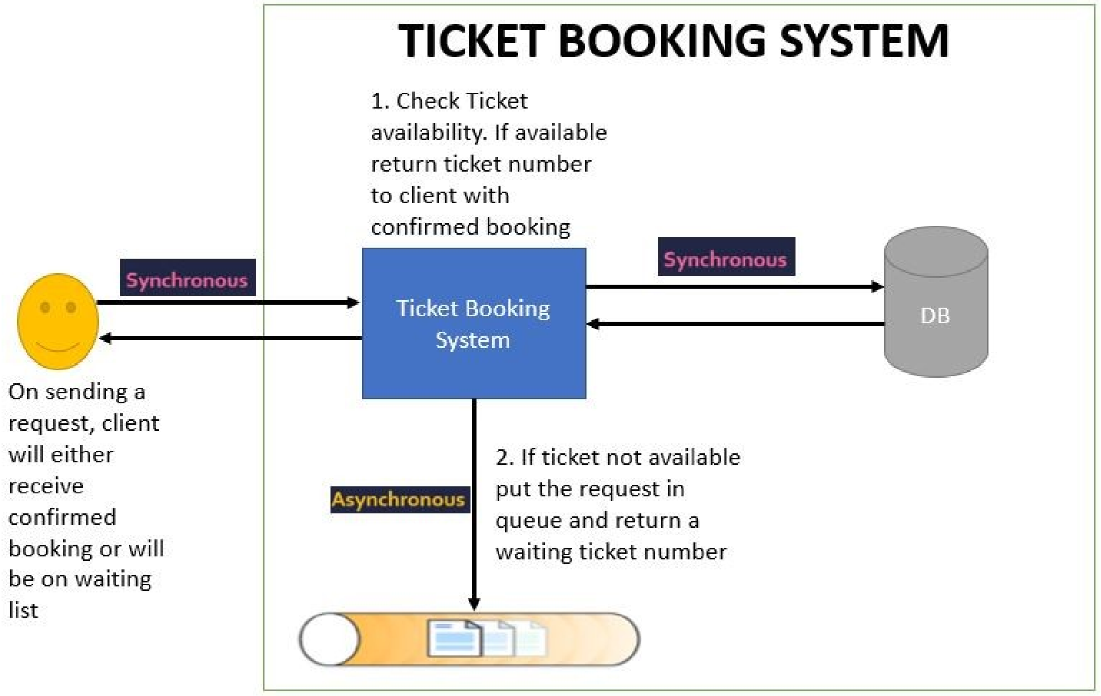
Asynchronous Messaging is implemented such that the messaging client sends a message and does not expect the response immediately. So the sender client does not for the response before he can execute the next task. So the sender is not blocked



Asynchronous Communication

Real time systems

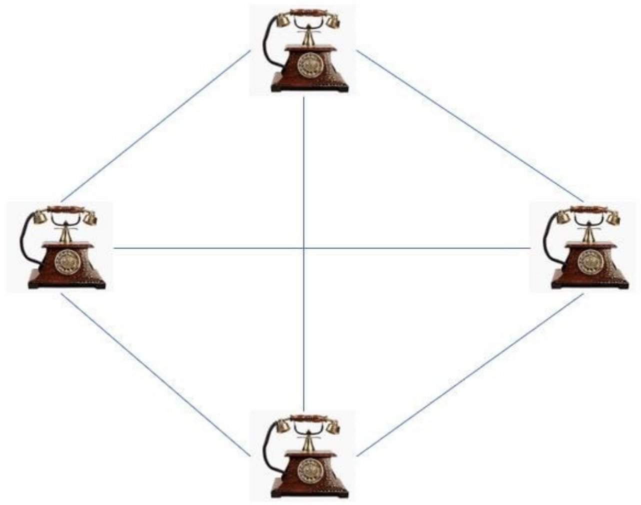
Real time systems usually have a combination of syschronous and ascynchronous communication


TICKET BOOKING SYSTEM
1. Check Ticket availability. If available return ticket number to client with confirmed booking
Synchronous
Synchronous
Ticket Booking System
DB
On sending a request, client will either receive confirmed booking or will be on waiting list
Asynchronous
2. If ticket not available put the request in queue and return a waiting ticket number

Message Broker

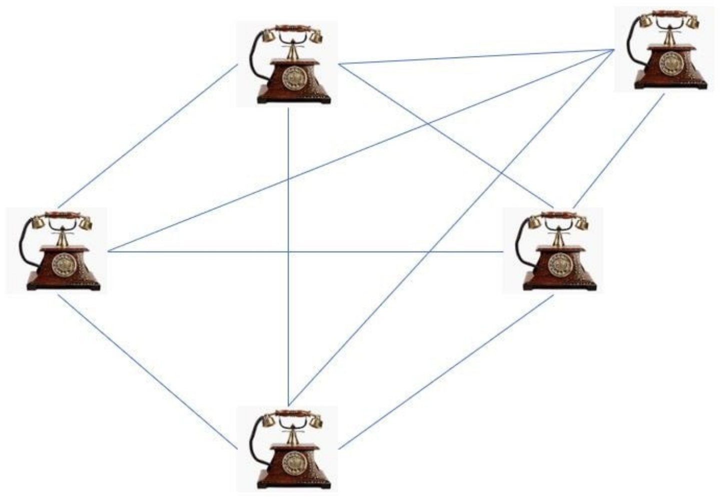
Message Broker - are responsible for establishing connections with various client systems.

Let us consider role of Message Broker in a telecom system. Suppose initially there is no message broker. Then each telephone connection will have a direct line with all other telephone connections.



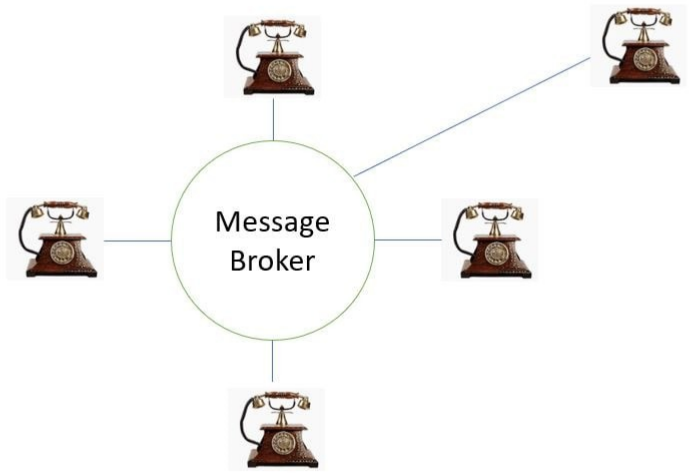
Message Broker(2)

Suppose tomorrow if another telephone connection needs to be added, then all existing telephone connections will need to get a direct line with this new telephone connection. As more connections are added this will get more complicated.



Message Broker(3)

With Message broker all connections are registered with Message Broker. So all connections only need to connect to the message broker. It will automatically route the message to the correct client based on some message configuration.



Rabbit mq installation ubuntu

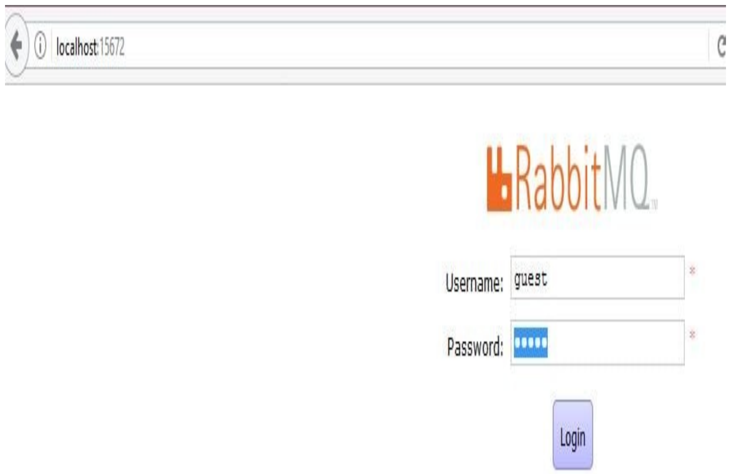
```
echo "deb http://www.rabbitmq.com/debian/ testing main" | sudo tee -a /etc/apt/sources.list

echo "deb http://packages.erlang-solutions.com/ubuntu wheezy contrib" | sudo tee -a /etc/apt/sources.list

wget http://packages.erlang-solutions.com/ubuntu/erlang_solutions.asc

sudo apt-key add erlang_solutions.asc

sudo apt-get update

sudo apt-get -y install erlang erlang-nox

sudo apt-get -y --force-yes install rabbitmq-server
```

Rabbit mq installation ubuntu(2)

```
# Enable the web interface

sudo rabbitmq-plugins enable rabbitmq_management

sudo service rabbitmq-server restart

ss -tunelp | grep 1567


sudo service rabbitmq-server start

sudo service rabbitmq-server stop
```

http://localhost:15672

https://stackoverflow.com/questions/8808909/simple-way-to-install-rabbitmq-in-ubuntu

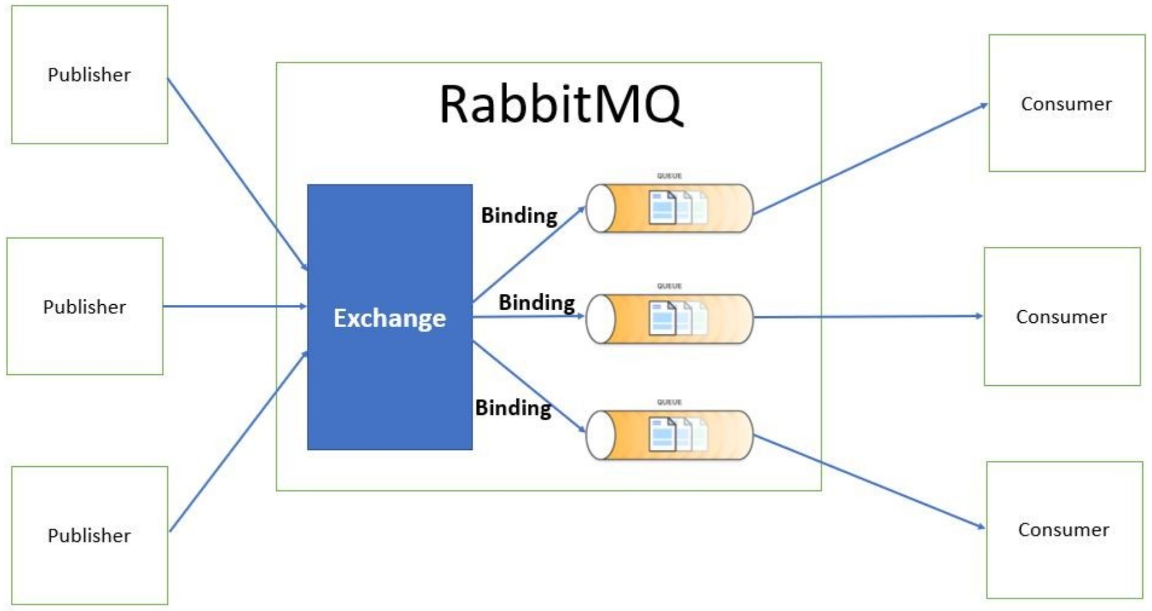

localhost:15672

Username: guest

Password: •••••

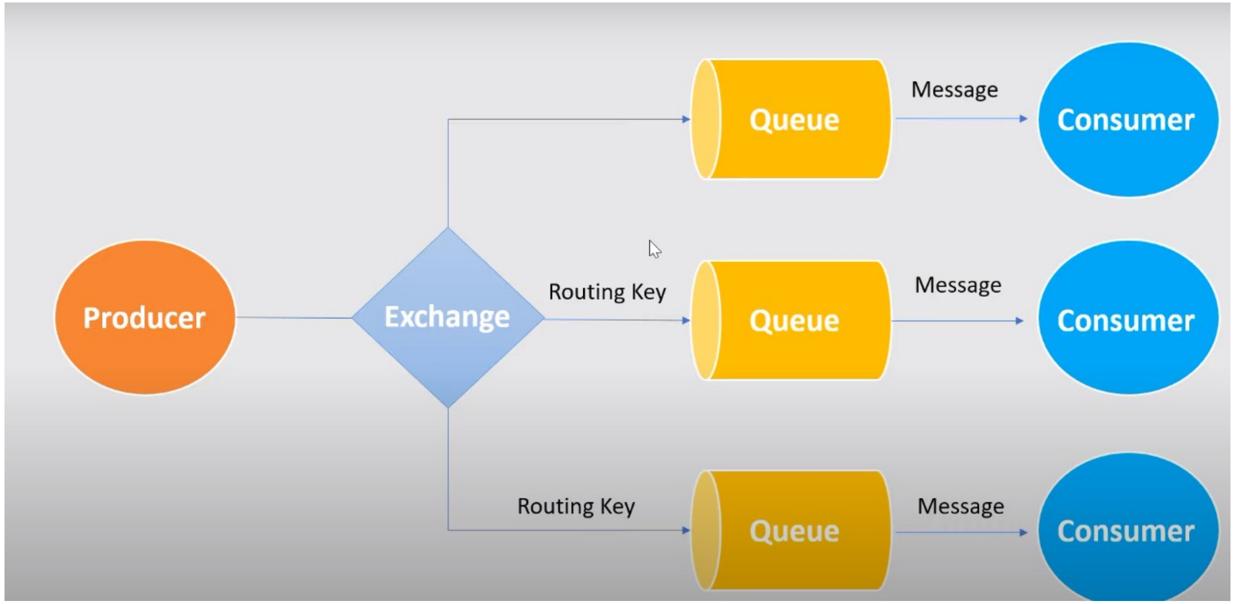
RabbitMQ Messaging Flow

When using RabbitMQ the publisher never directly sends a message to a queue. Instead, the publisher sends messages to an exchange. Exchange is responsible for sending the message to an appropriate queue based on routing keys, bindings and header attributes. Exchanges are message routing agents which we can define and bindings are what connects the exchanges to queues. So in all our examples we will be creating first a Queue and Exchange, then bind them together.

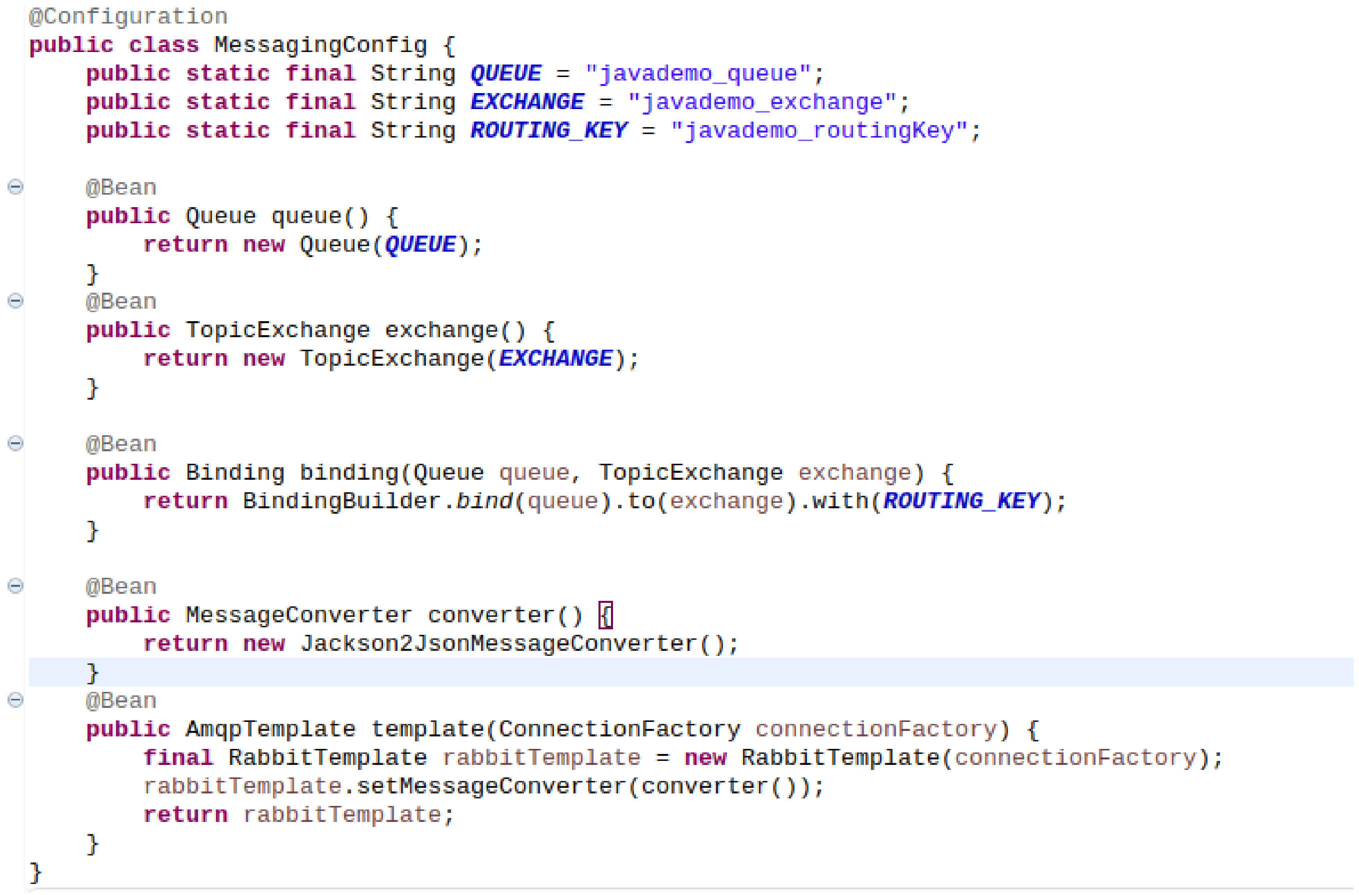


Spring boot rabbitMQ hello world

- Create spring boot project with web, ampq dependencies

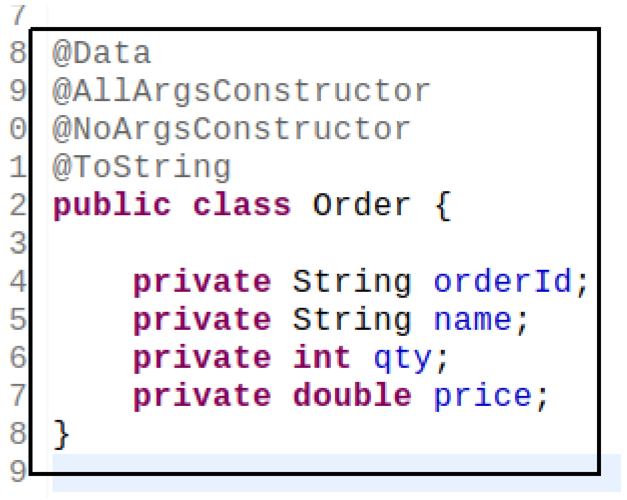


Configuration rabbitmq

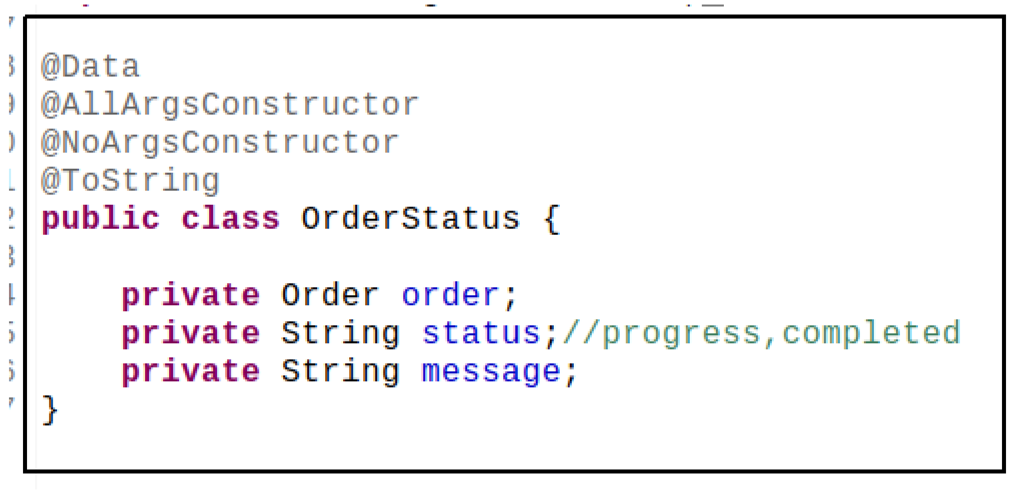


```java
@Configuration
public class MessagingConfig {
    public static final String QUEUE = "javademo_queue";
    public static final String EXCHANGE = "javademo_exchange";
    public static final String ROUTING_KEY = "javademo_routingKey";

    @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);
    }

    @Bean
    public MessageConverter converter() {
        return new Jackson2JsonMessageConverter();
    }
    @Bean
    public AmqpTemplate template(ConnectionFactory connectionFactory) {
        final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
        rabbitTemplate.setMessageConverter(converter());
        return rabbitTemplate;
    }
}
```

Request and response objects

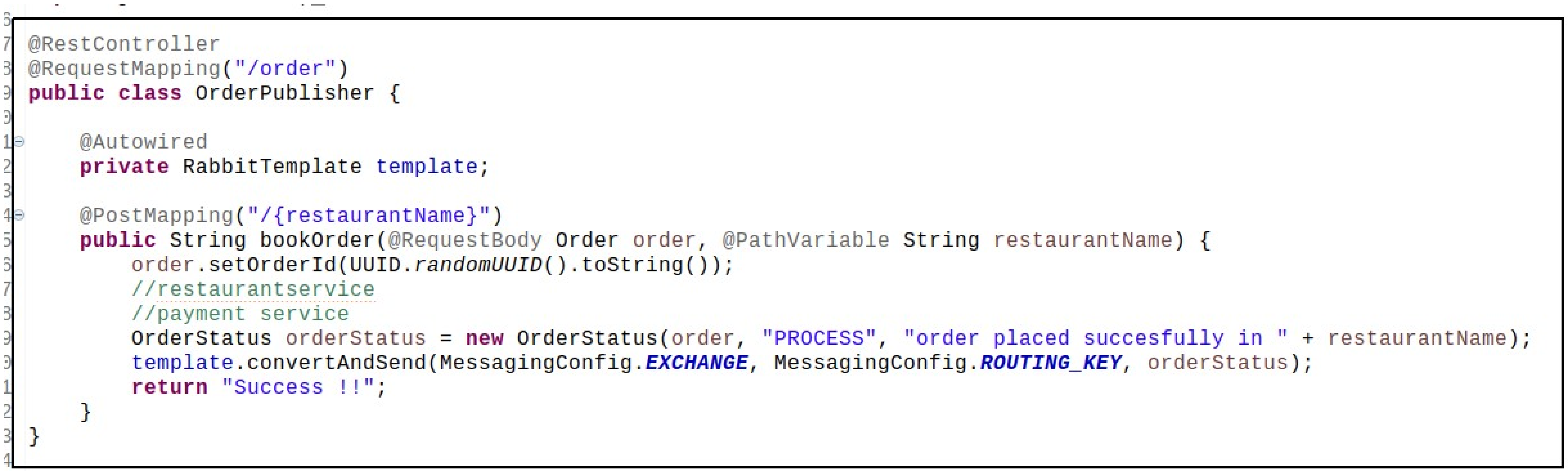


```
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class Order {

    private String orderId;
    private String name;
    private int qty;
    private double price;
}
```



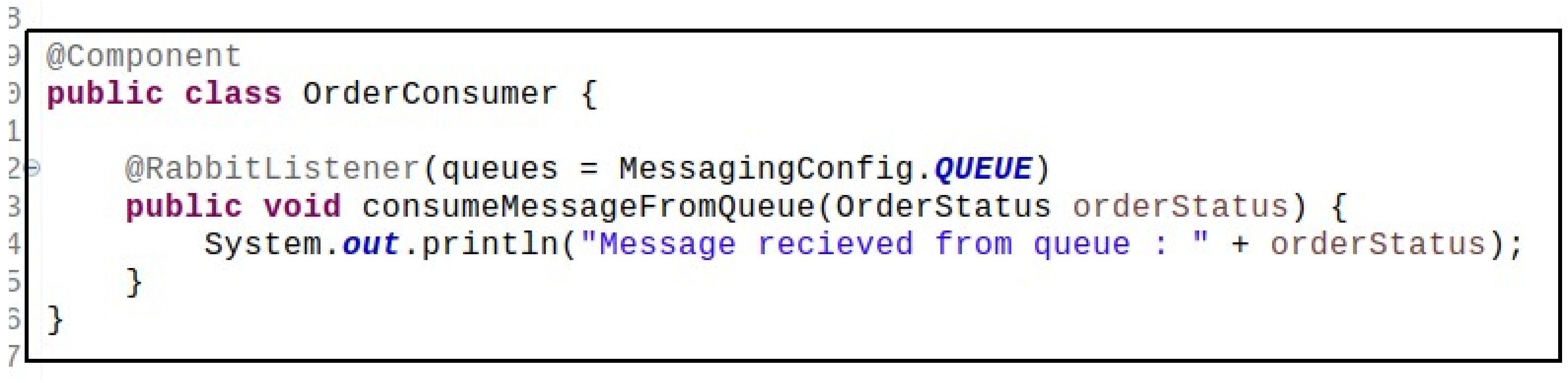
```
@Data
@AllArgsConstructor
@NoArgsConstructor
@ToString
public class OrderStatus {

    private Order order;
    private String status;//progress,completed
    private String message;
}
```

Order Producer



```java
@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 !!";
    }
}
```

Order Consumer



```
@Component
public class OrderConsumer {

    @RabbitListener(queues = MessagingConfig.QUEUE)
    public void consumeMessageFromQueue(OrderStatus orderStatus) {
        System.out.println("Message recieved from queue : " + orderStatus);
    }
}
```

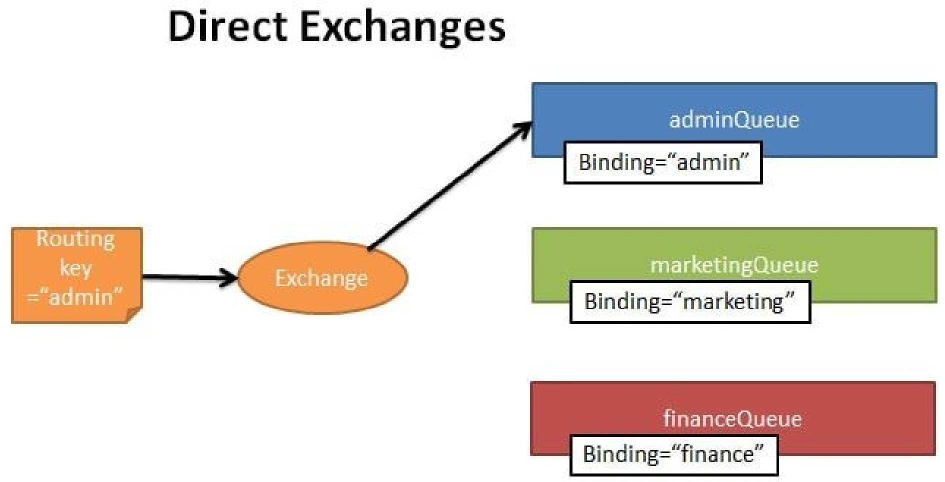
RabbitMQ types of Exchanges

With RabbitMQ we have the following types of Exchanges-

1) Direct Exchange
2) Fanout Exchange
3) Topic Exchange
4) Header Exchange

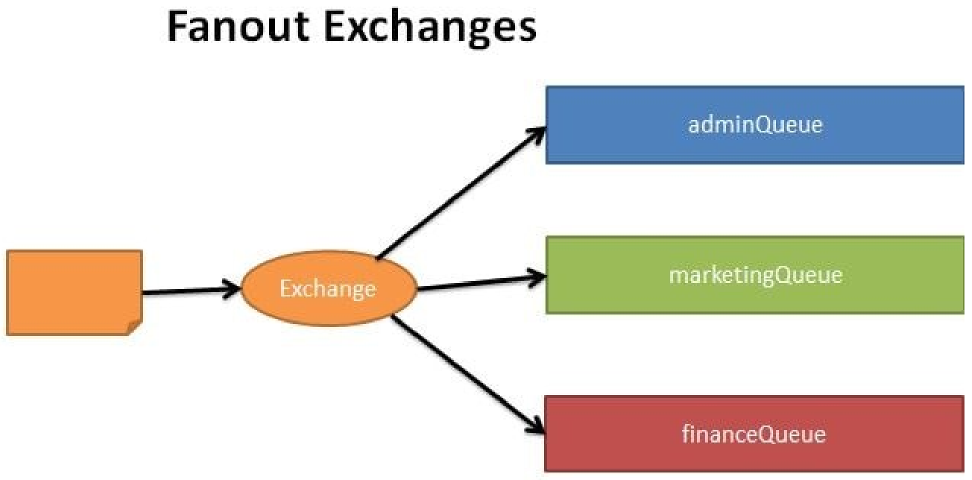
Direct Exchange

Based on the routing key a message is sent to the queue having the same routing key specified in the binding rule. The routing key of exchange and the binding queue have to be an exact match. A message is sent to exactly one queue



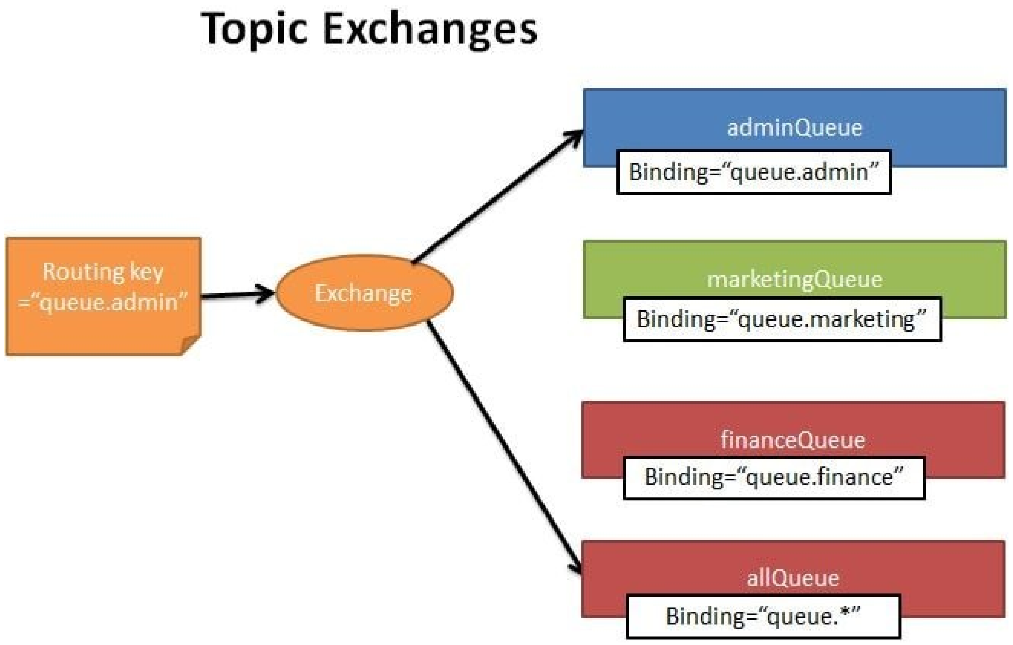
Fanout Exchange

The message is routed to all the available bounded queues. The routing key if provided is completely ignored. So this is a kind of publish-subscribe design pattern.



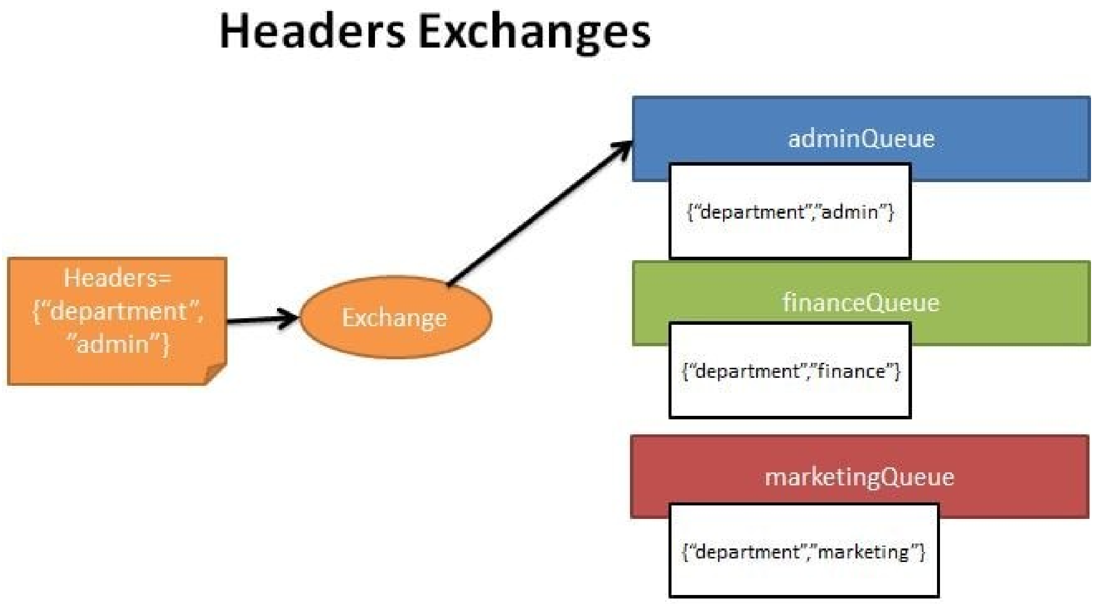
Topic Exchange

Here again the routing key is made use of. But unlike in direct exchange type, here the routing key of the exchange and the bound queues should not necessarily be an exact match. Using regular expressions like wildcard we can send the exchange to multiple bound queues.

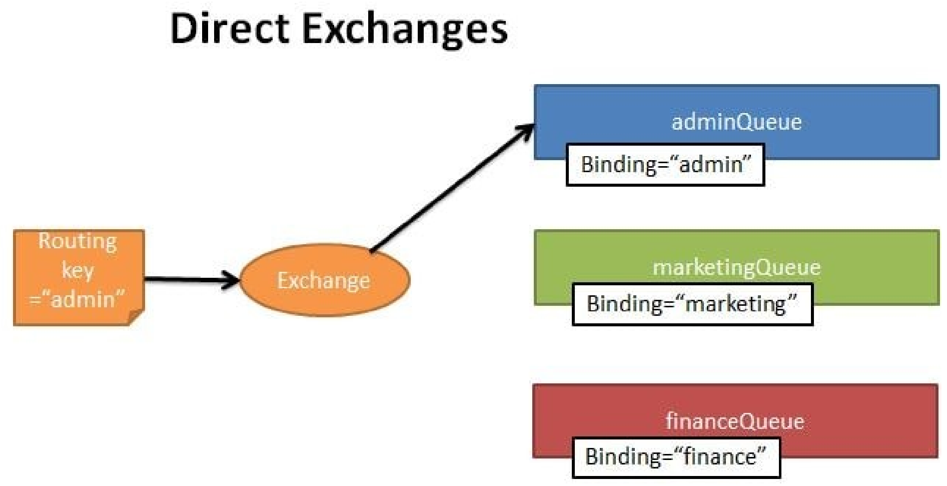


Header Exchange

In this type of exchange the routing queue is selected based on the criteria specified in the headers instead of the routing key. This is similar to topic exchange type, but here we can specify complex criteria for selecting routing queues.

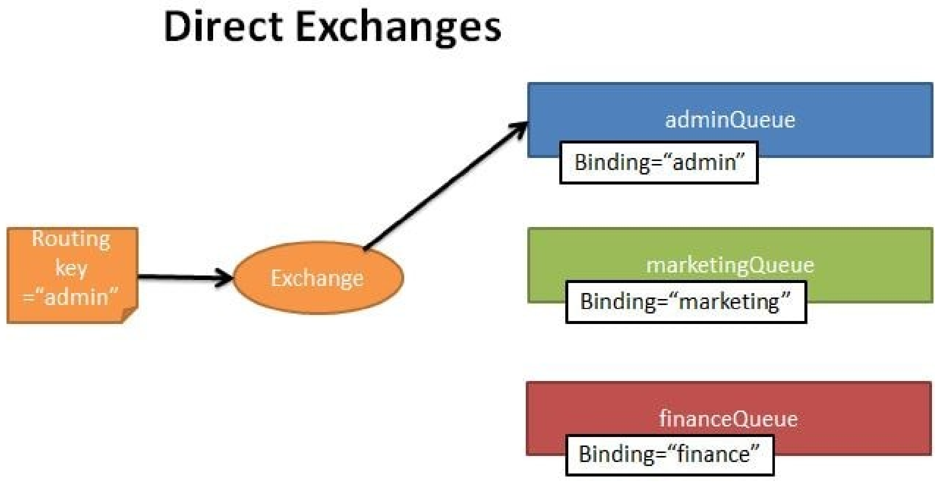


Direct exchange in details



Direct Exchange

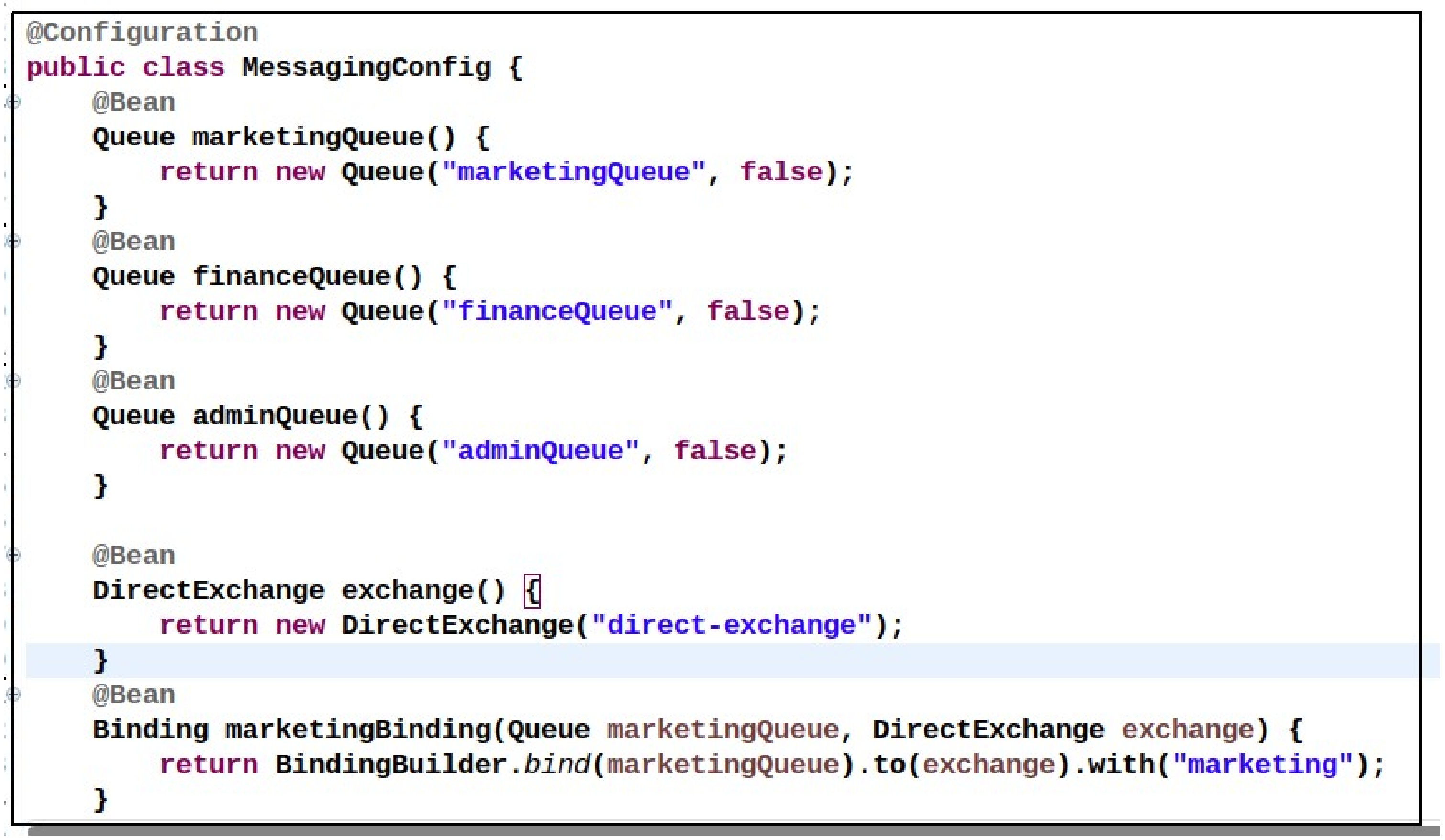
Based on the routing key a message is sent to the queue having the same routing key specified in the binding rule. The routing key of exchange and the binding queue have to be an exact match. A message is sent to exactly one queue



Direct Exchange

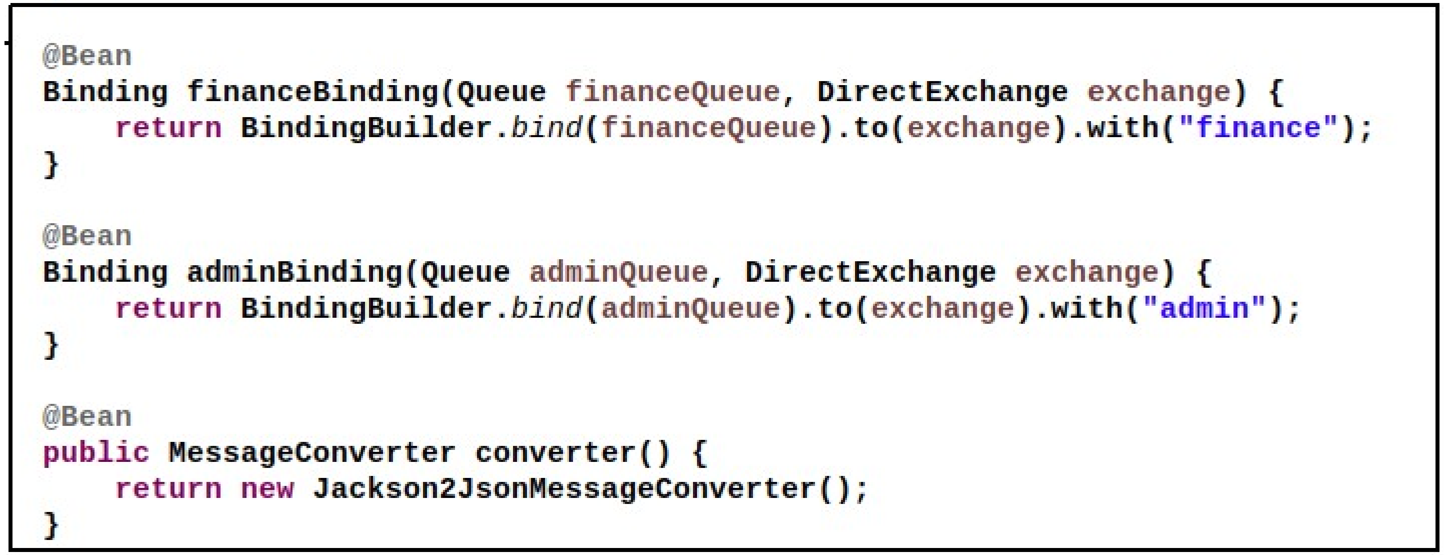
- Create the RabbitMQDirectConfig as follows-
- Create Queues named - **marketingQueue, adminQueue, financeQueue**
- Create a DirectExchange named - **direct-exchange**
- Create Bindings for each of the queue with the **DirectExchange specifying the binding key**

Direct Exchange config



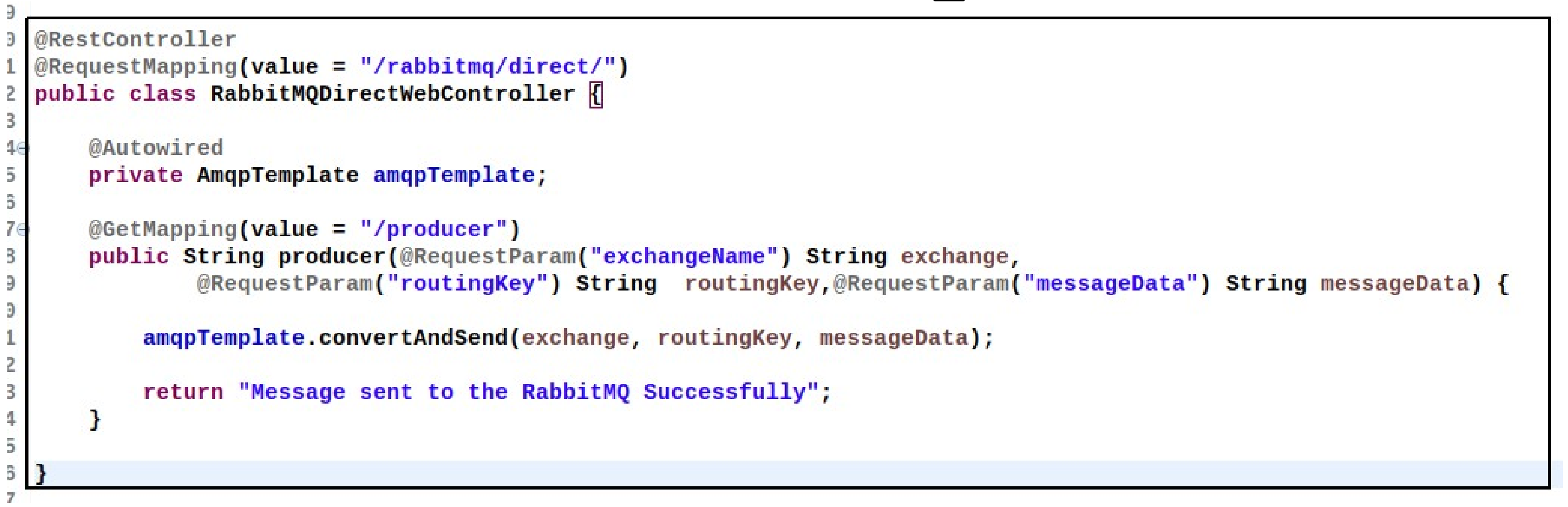
```
@Configuration
public class MessagingConfig {
    @Bean
    Queue marketingQueue() {
        return new Queue("marketingQueue", false);
    }
    @Bean
    Queue financeQueue() {
        return new Queue("financeQueue", false);
    }
    @Bean
    Queue adminQueue() {
        return new Queue("adminQueue", false);
    }

    @Bean
    DirectExchange exchange() {
        return new DirectExchange("direct-exchange");
    }
    @Bean
    Binding marketingBinding(Queue marketingQueue, DirectExchange exchange) {
        return BindingBuilder.bind(marketingQueue).to(exchange).with("marketing");
    }
```

Direct Exchange config



```
@Bean
Binding financeBinding(Queue financeQueue, DirectExchange exchange) {
    return BindingBuilder.bind(financeQueue).to(exchange).with("finance");
}

@Bean
Binding adminBinding(Queue adminQueue, DirectExchange exchange) {
    return BindingBuilder.bind(adminQueue).to(exchange).with("admin");
}

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

Direct Exchange controller

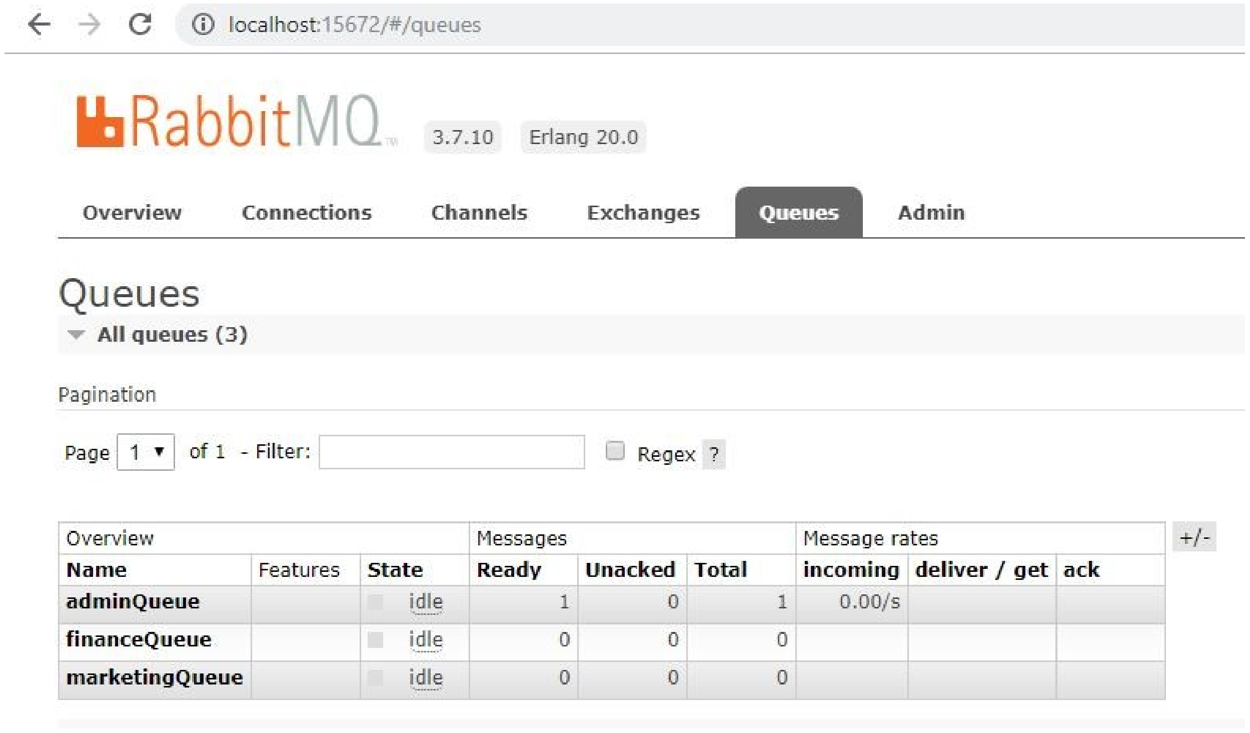


```
@RestController
@RequestMapping(value = "/rabbitmq/direct/")
public class RabbitMQDirectWebController {

    @Autowired
    private AmqpTemplate amqpTemplate;

    @GetMapping(value = "/producer")
    public String producer(@RequestParam("exchangeName") String exchange,
            @RequestParam("routingKey") String  routingKey,@RequestParam("messageData") String messageData) {

        amqpTemplate.convertAndSend(exchange, routingKey, messageData);

        return "Message sent to the RabbitMQ Successfully";
    }

}
```

- We send the message using the url
- - h**ttp://localhost:8080/rabbitmq/direct/producer?exchangeName=direct-exchange&routingKey=admin&messageData=HelloWorld**
- we will be specifying the following
- exchange name= "direct-exchange"
- routing key ="admin"
- message to sent to queue = "HelloWorld"

Direct Exchange Queues

- We can see that queues named marketingQueue, adminQueue and financeQueue are created. Also a message has been sent to the adminQueue



localhost:15672/#/queues

RabbitMQ 3.7.10 Erlang 20.0

Queues

All queues (3)

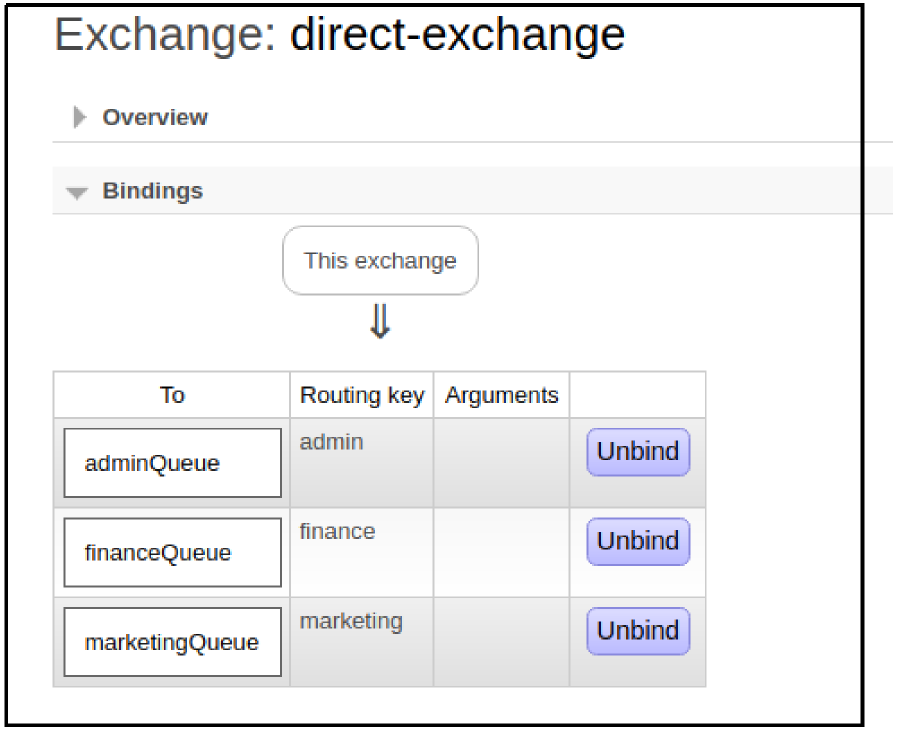
Pagination

Page 1 of 1 - Filter: Regex ?

Overview			Messages			Message rates		
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
adminQueue		idle	1	0	1	0.00/s		
financeQueue		idle	0	0	0			
marketingQueue		idle	0	0	0			

Direct Exchange Queues

- An exchange named direct-exchange is created with following bindings

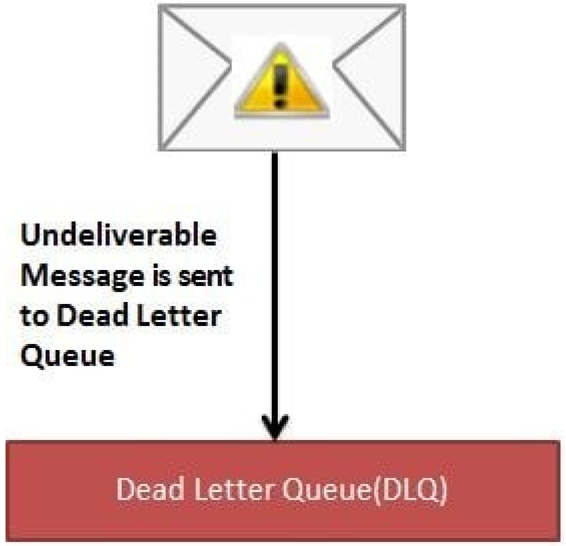


Exchange: direct-exchange

To	Routing key	Arguments	
adminQueue	admin		Unbind
financeQueue	finance		Unbind
marketingQueue	marketing		Unbind

Retry and Error Handling

- If exception still exists after maximum retries then put message in a dead letter queue where it can be analyzed and corrected later.
- What is a Dead Letter Queue?
  - In English vocabulary Dead letter mail is an undeliverable mail that cannot be delivered to the addressee.
  - A dead-letter queue (DLQ), sometimes which is also known as an undelivered-message queue, is a holding queue for messages that cannot be delivered to their destinations due to some reason or other.

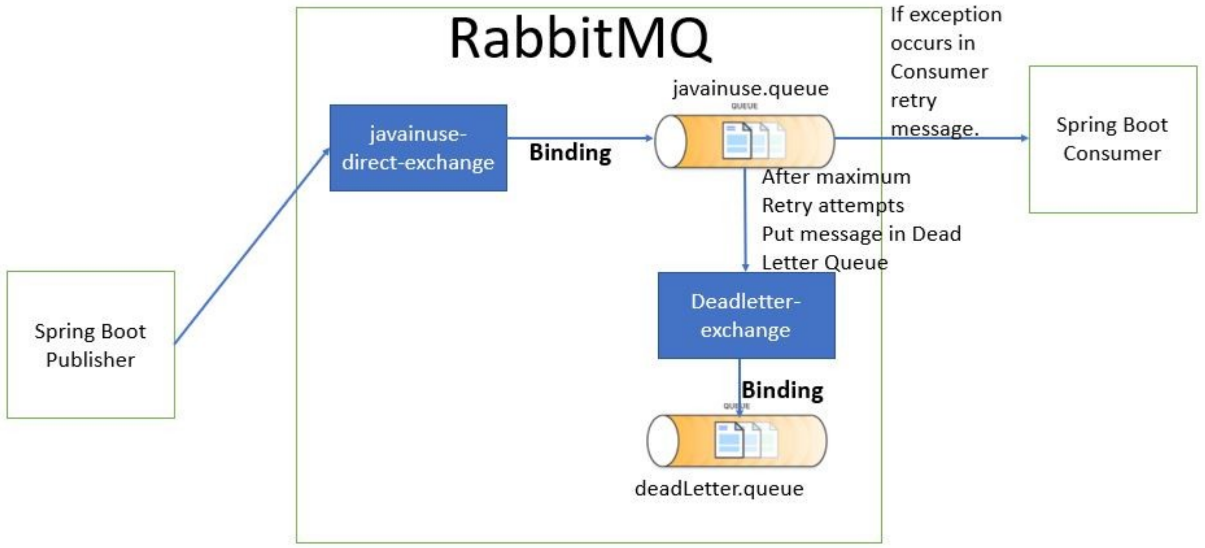


Retry and Error Handling

- In message queueing the dead letter queue is a service implementation to store messages that meet one or more of the following failure criteria:
- Message that is sent to a queue that does not exist.
- Queue length limit exceeded.
- Message length limit exceeded.
- Message is rejected by another queue exchange.
- Message reaches a threshold read counter number, because it is not consumed. Sometimes this is called a "back out queue".

Retry and Error Handling

- Spring Boot Producer Module - It will produce a message and put it in RabbitMQ queue. It will also be responsible for creating the required queues including the dead letter queue.
- Spring Boot Consumer Module - It will consume a message from RabbitMQ queue. We will be throwing an exception and then retrying the message. After maximum retries it will then be put in dead letter queue.


RabbitMQ
javainuse.queue
QUEUE
javainuse-direct-exchange
Binding
If exception occurs in Consumer retry message.
Spring Boot Consumer
After maximum Retry attempts Put message in Dead Letter Queue
Deadletter-exchange
Binding
QUEUE
deadLetter.queue
Spring Boot Publisher