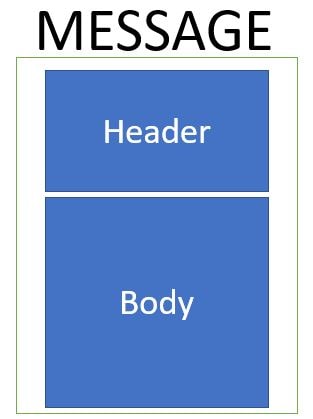
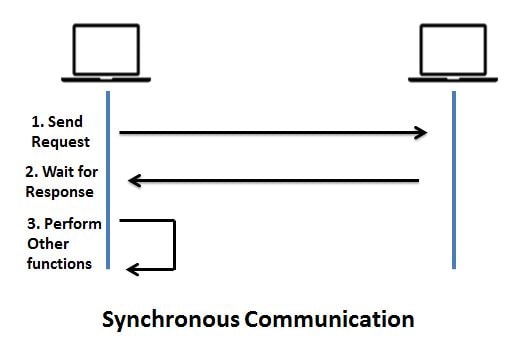
What is Messaging?

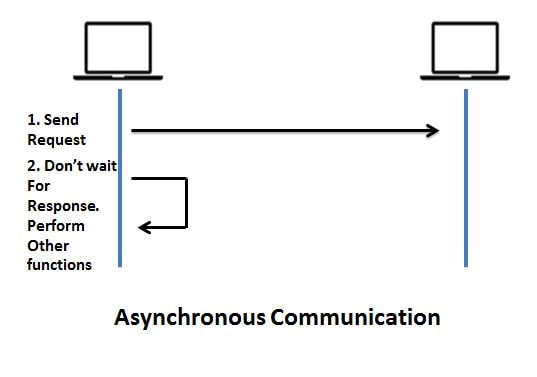
Messaging is a communication mechanism used for system interactions. In software development messaging enables distributed communication that is loosely coupled. A messaging client can send messages to, and receive messages from, any other client. The structure of message can be defined as follows-



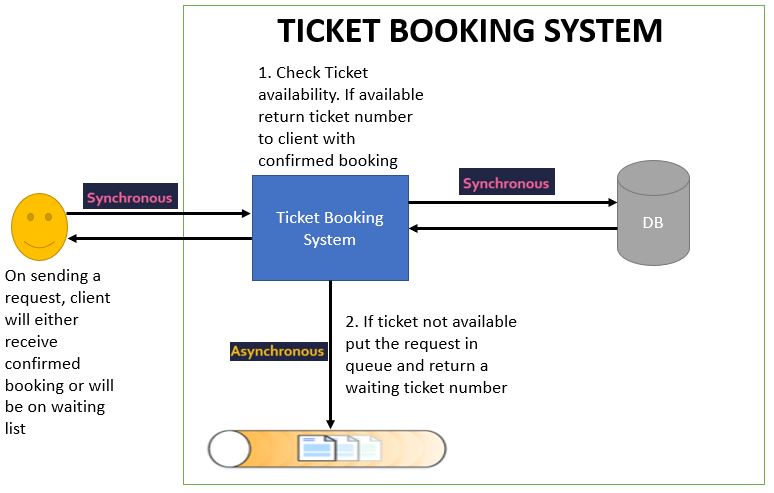
Synchronous Messaging is implemented when such that the messaging client sends a message and expects the response immediately. So the sender client waits for the response before he can execute the next task. So until and unless the message is recieved the sender is blocked.



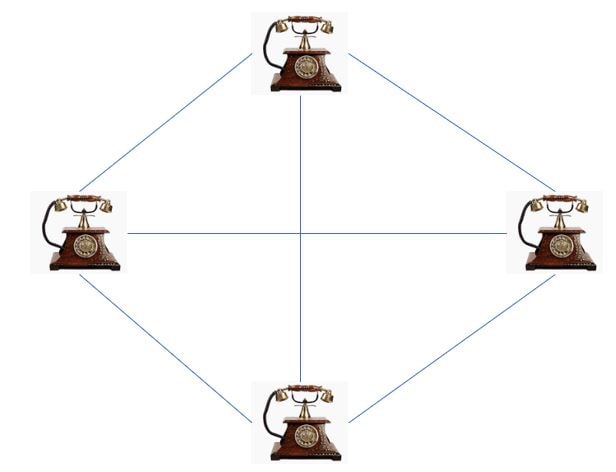
Asynchronous Messaging is implemented such that the messaging client sends a message and does not expect the response immediately. So the sender client does not for the response before he can execute the next task. So the sender is not blocked.



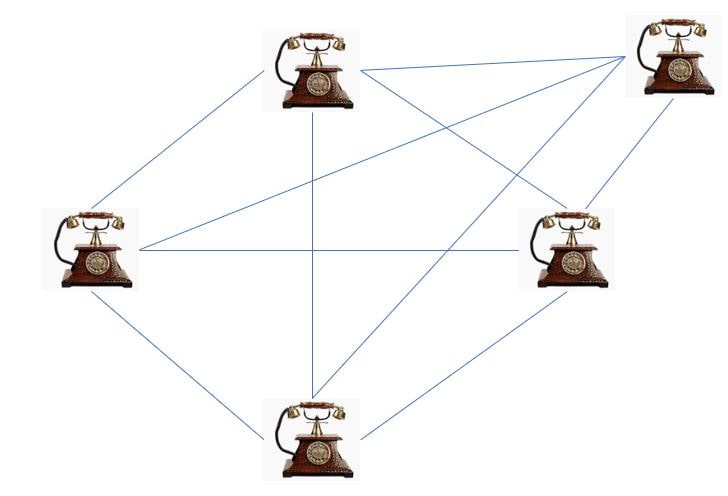
Real time systems usually have a combination of syschronous and ascynchronous communication



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.



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.



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.

Getting Starting with RabbitMQ

RabbitMQ is open source message broker software (sometimes called message-oriented middleware) that implements the Advanced Message Queuing Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language and is built on the Open Telecom Platform framework for clustering and failover.

Next we will install the RabbitMQ plugin which will give use the RabbitMQ Management Console which is accessible using the browser. For this use the command as follows-

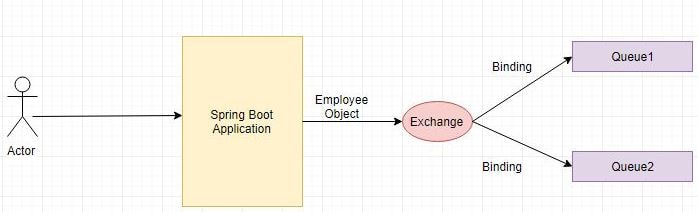
rabbitmq-plugins.bat enable rabbitmq\_management

Spring Boot + RabbitMQ Hello World Example

In this post we will integrate Spring Boot and RabbitMQ instance.

In this tutorial we will be sending a message to RabbitMQ. In next tutorial we will see how to consume a RabbitMQ message using Spring Boot.

In a previous post we had seen how to get RabbitMQ up and running. Also in this tutorial we make use of the direct exchange for publishing the message to RabbitMQ. In another tutorial we have explained the various exchange types and their implementation using Spring Boot.



pom.xml

--------

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>

<java.version>1.8</java.version>

</properties>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

</dependency>

Define the domain class Employee as follows-

@JsonIdentityInfo(generator = ObjectIdGenerators.IntSequenceGenerator.class, property = "@id", scope = Employee.class)

public class Employee

{ private String empName; private String empId; //setters and getters }

@Configuration

public class RabbitMQConfig {

@Value("${javainuse.rabbitmq.queue}")

String queueName;

@Value("${javainuse.rabbitmq.exchange}")

String exchange;

@Value("${javainuse.rabbitmq.routingkey}")

private String routingkey;

@Bean

Queue queue() { return new Queue(queueName, false); }

@Bean

DirectExchange exchange() {return new DirectExchange(exchange); }

@Bean

Binding binding(Queue queue, DirectExchange exchange) { return BindingBuilder.bind(queue).to(exchange).with(routingkey); }

@Bean

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

@Bean

public AmqpTemplate rabbitTemplate(ConnectionFactory connectionFactory) {

final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);

rabbitTemplate.setMessageConverter(jsonMessageConverter());

return rabbitTemplate; }

}

Define the Controller to expose a GET Request API as follows-Define the Controller to expose a GET Request API as follows-

@RestController

@RequestMapping(value = "/javainuse-rabbitmq/")

public class RabbitMQWebController {

@Autowired

RabbitMQSender rabbitMQSender;

@GetMapping(value = "/producer")

public String producer(@RequestParam("empName") String empName,@RequestParam("empId") String empId) {

Employee emp=new Employee();

emp.setEmpId(empId);

emp.setEmpName(empName);

rabbitMQSender.send(emp);

return "Message sent to the RabbitMQ JavaInUse Successfully"; } }

Define the RabbitMQSender class which sends the message to the RabbitMQ using AmqpTemplate. We use the exchange and the exchange key.

Exchanges are message routing agents, defined per virtual host within RabbitMQ. An exchange is responsible for the routing of the messages to the different queues. An exchange accepts messages from the producer application and routes them to message queues with help of header attributes, bindings, and routing keys.

We will use a direct exchange instead. The routing algorithm behind a direct exchange is simple - a message goes to the queues whose binding key exactly matches the routing key of the message.

@Service

public class RabbitMQSender {

@Autowired

private AmqpTemplate rabbitTemplate;

@Value("${javainuse.rabbitmq.exchange}")

private String exchange;

@Value("${javainuse.rabbitmq.routingkey}")

private String routingkey;

public void send(Employee company) {

rabbitTemplate.convertAndSend(exchange, routingkey, company);

System.out.println("Send msg = " + company); } }

Next define th following properties in application.properties-

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

javainuse.rabbitmq.exchange=javainuse.exchange

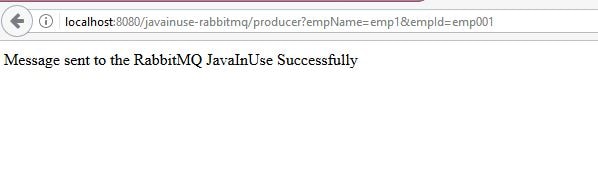
javainuse.rabbitmq.queue=javainuse.queue

javainuse.rabbitmq.routingkey=javainuse.routingkey

Finally Define the Spring Boot Class with @SpringBootApplication annotation

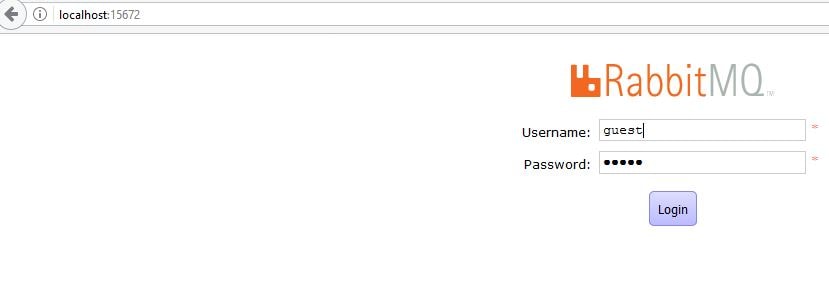
We are done with the required Java code. Now lets start RabbitMQ. As we had explained in detail in the Getting started with RabbitMQ perform the steps to start the RabbitMQ.

Next start the Spring Boot Application by running it as a Java Application. Hit the url as follows- <http://localhost:8080/javainuse-rabbitmq/producer?empName=emp1&empId=emp001>

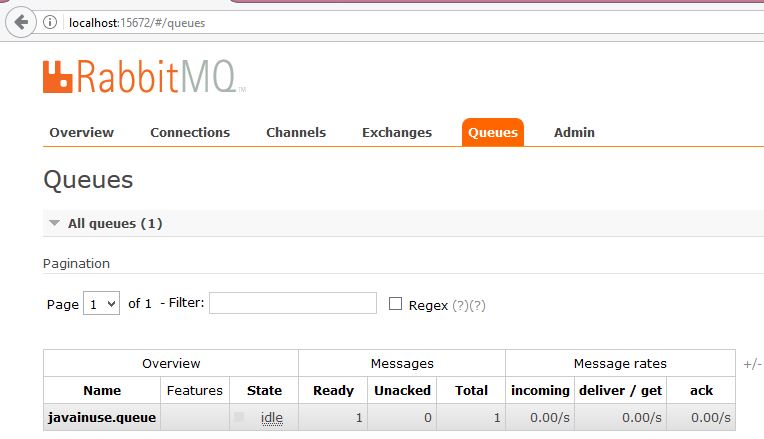


This will trigger the message to be sent to the javainuse queue.

Next go to the RabbitMQ console-**http://localhost:15672/**

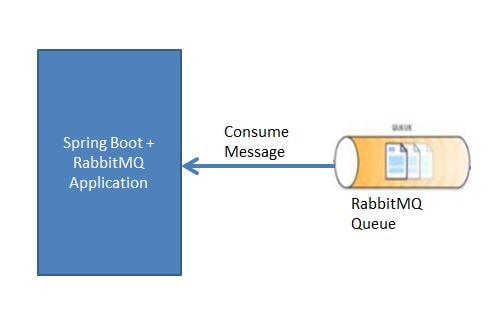


We can see in the Queues section, a queue name java gets created and it has one message.



**Spring Boot + RabbitMQ Tutorial - Configure Listeners to consume messages**

In this tutorial we will be implementing a Spring Boot + RabbitMQ example to consume message from a RabbitMQ Queue. In next tutorial we will be exploring the various RabbitMQ Exchange types and implementing them using Spring Boot.



We will first be creating a listener class which implements the AMQP MessageListener interface. This class is responsible for getting the message from the RabbitMQ queue.

@Service

public class RabbitMQListner implements MessageListener {

public void onMessage(Message message) {

S.o.p("Consuming Message - " + new String(message.getBody())); }

Next we will be creating the Spring Boot Configuration class for RabbitMQ. We will

Create Queue bean using which we create a RabbitMQ named javainuse.input-queue. This will be a non durable queue. Do not misunderstand a non durable queue to be a temporary queue. Durability property is related to how long a message will be stored in the queue. For example for in RabbitMQ restart messages in non durable queue will be lost while those in durable queue will not be lost.

Create MessageListenerContainer where we configure the RabbitMQConnections. Spring MessageListenerContainer is a replacement for a Message-Driven EJB. A connection is set up with the AMQ topic/queue, it gets messages from that topic/queue and feeds them to your MessageListener. We will be making use of the default connectionfactory. If we do not wish to use the default connectionfactory we can can create our own CachingConnectionFactory and use it.

@Configuration

public class RabbitMQConfig {

@Value("${javainuse.rabbitmq.queue}")

String queueName;

@Value("${spring.rabbitmq.username}")

String username;

@Value("${spring.rabbitmq.password}")

private String password;

@Bean

Queue queue() {

return new Queue(queueName, false);

}

//create MessageListenerContainer using default connection factory

@Bean

MessageListenerContainer messageListenerContainer(ConnectionFactory connectionFactory ) {

SimpleMessageListenerContainer simpleMessageListenerContainer = new SimpleMessageListenerContainer();

simpleMessageListenerContainer.setConnectionFactory(connectionFactory);

simpleMessageListenerContainer.setQueues(queue());

simpleMessageListenerContainer.setMessageListener(new RabbitMQListner());

return simpleMessageListenerContainer;

}

//create custom connection factory

/\*@Bean

ConnectionFactory connectionFactory() {

CachingConnectionFactory cachingConnectionFactory = new CachingConnectionFactory("localhost");

cachingConnectionFactory.setUsername(username);

cachingConnectionFactory.setUsername(password);

return cachingConnectionFactory;

}\*/

//create MessageListenerContainer using custom connection factory

/\*@Bean

MessageListenerContainer messageListenerContainer() {

SimpleMessageListenerContainer simpleMessageListenerContainer = new SimpleMessageListenerContainer();

simpleMessageListenerContainer.setConnectionFactory(connectionFactory());

simpleMessageListenerContainer.setQueues(queue());

simpleMessageListenerContainer.setMessageListener(new RabbitMQListner());

return simpleMessageListenerContainer;

}\*/

In the application.properties define the following-

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

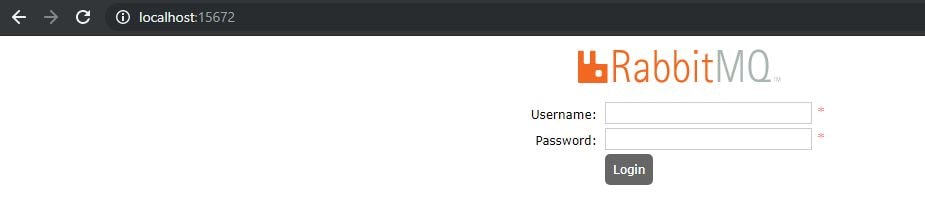
spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

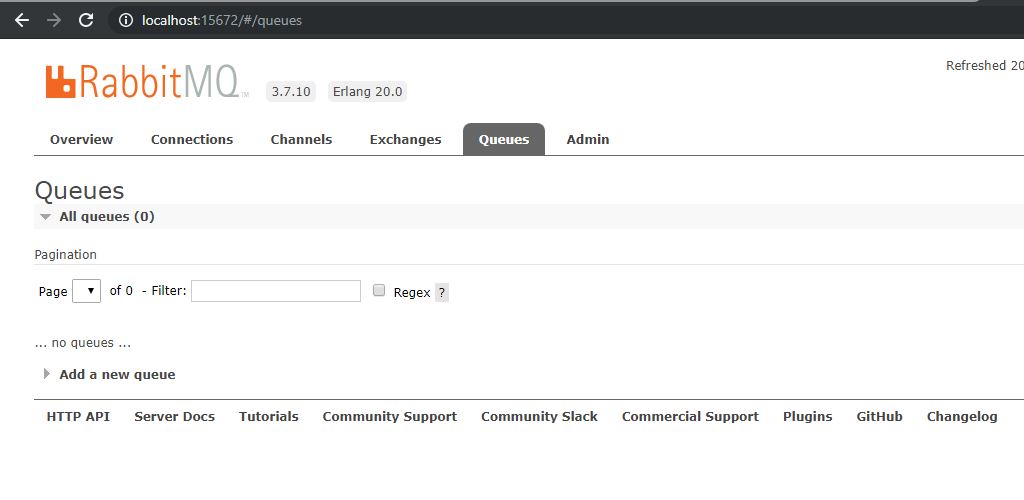
javainuse.rabbitmq.queue=javainuse.input-queue

Finally create the bootstrap class which makes use of the SpringBootApplication Annotation

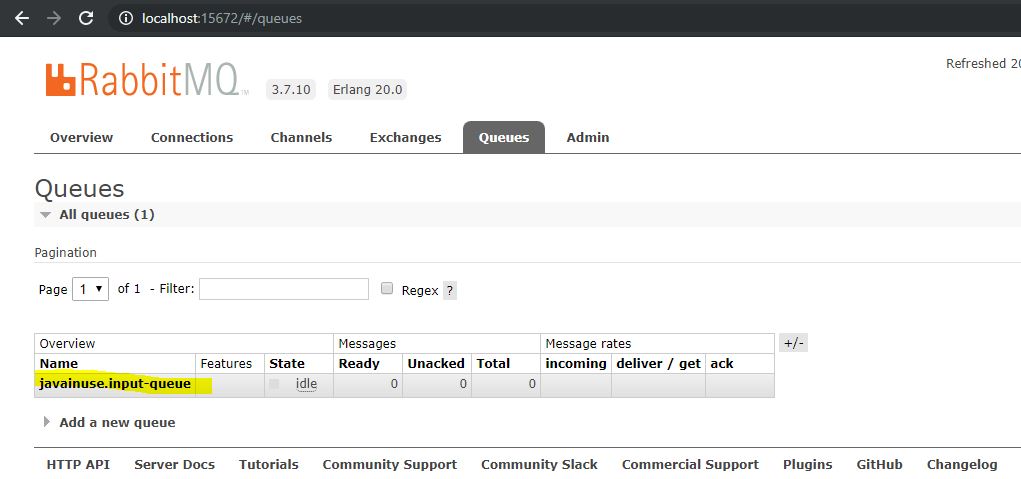
Once you have followed implemented this tutorial go to localhost:15672



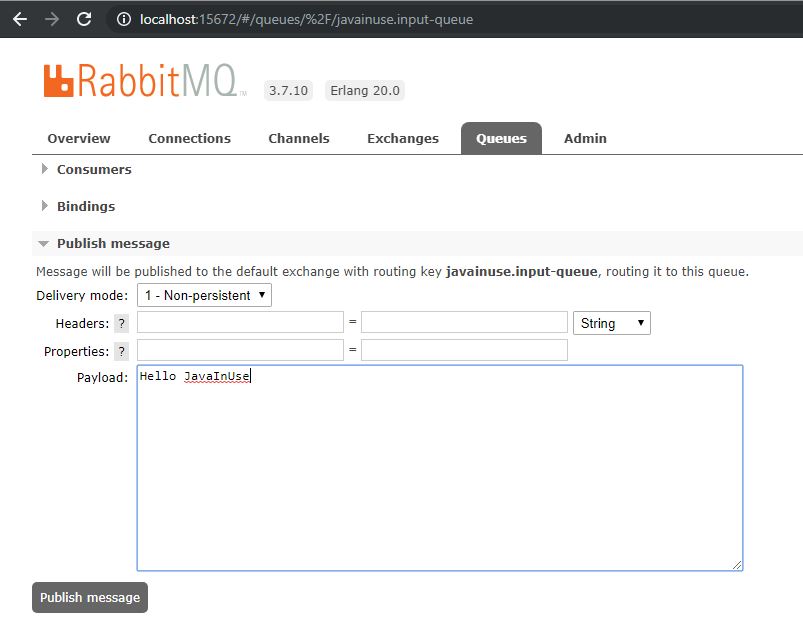
Use the username and password as guest.If we now go to the queues section, currently there are no queues



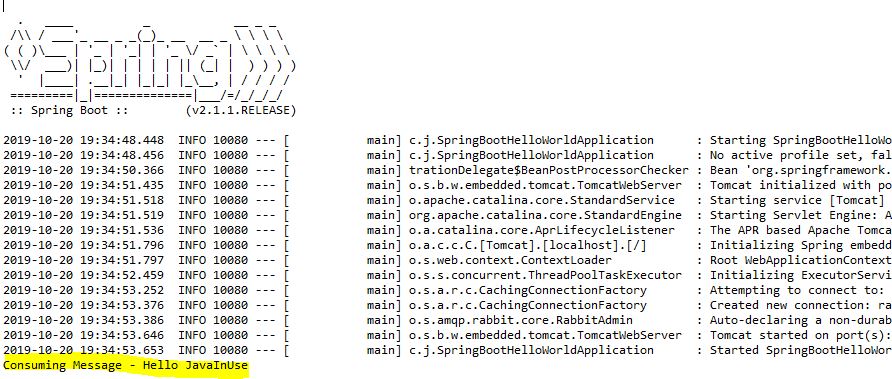
Start the Spring Boot Application. If we now again go to the RabbitMQ Management Console Queues section we can see that a queue named javainuse.input-queue has been created.



Now select the queue and publish a message.



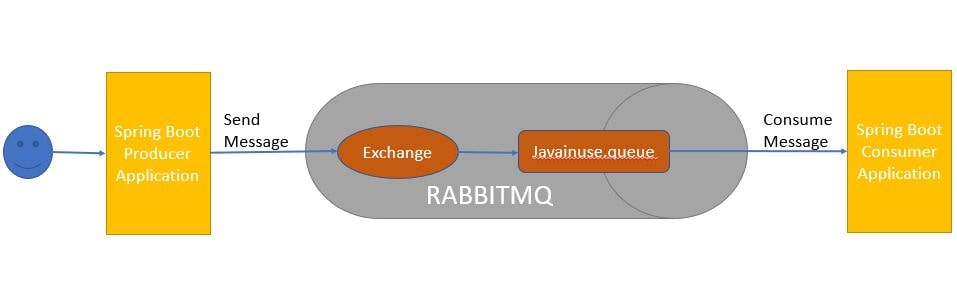
If we now go to the eclipse console we can see that our application has consumed the RabbitMQ message and printed its content.



# Spring Boot + RabbitMQ Consume Message Simple Example

In a previous post we integrated RabbitMQ and Spring Boot to send a message to RabbitMQ. In this post we consuming the message using Spring Boot and RabbitMQ.

In a previous post we had seen how to get RabbitMQ up and running.



Define the pom.xml as follows- Add the spring-boot-starter-amqp dependency.

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.json</groupId>

<artifactId>json</artifactId>

</dependency>

Define the domain class Employee as follows-

@JsonIdentityInfo(generator = ObjectIdGenerators.IntSequenceGenerator.class, property = "@id", scope = Employee.class)

public class Employee {

private String empName;

private String empId;

//setters and getters

}

Define the RabbitMQConsumer class which consumes the message from RabbitMQ using RabbitListener.The RabbitMQ Listener listens to RabbitMQ Queue for any incoming messages. For the basic configuration we specify the the Queue/Topic Name (the name of the queue/topic where the message should be consumed)

@Component

public class RabbitMQConsumer {

@RabbitListener(queues = "${javainuse.rabbitmq.queue}")

public void recievedMessage(Employee employee) {

System.out.println("Recieved Message From RabbitMQ: " + employee);

} }

Next define th following properties in application.properties-

spring.rabbitmq.host=localhost

spring.rabbitmq.port=5672

spring.rabbitmq.username=guest

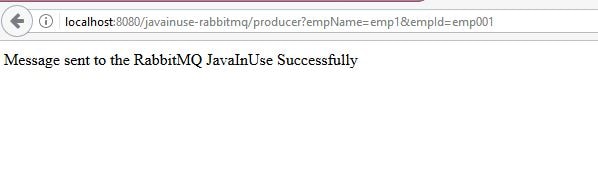
spring.rabbitmq.password=guest

javainuse.rabbitmq.queue=javainuse.queue

server.port=8081

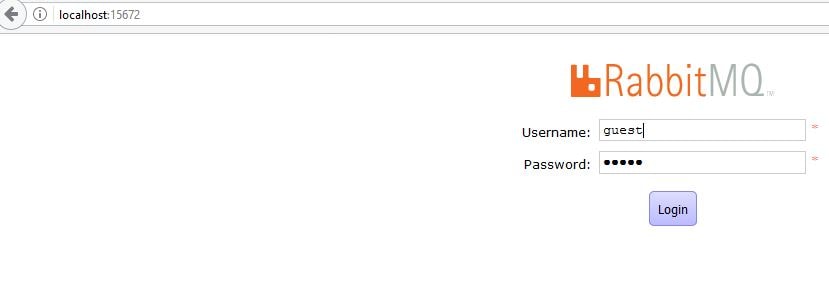
We are done with the required Java code. Now lets start RabbitMQ. As we had explained in detail in the Getting started with RabbitMQ perform the steps to start the RabbitMQ.

Next start the Spring Boot RabbitMQProducer application we developed in previous tutorial. Application by running it as a Java Application. Hit the url as follows**- http://localhost:8080/javainuse-rabbitmq/producer?empName=emp1&empId=emp001**

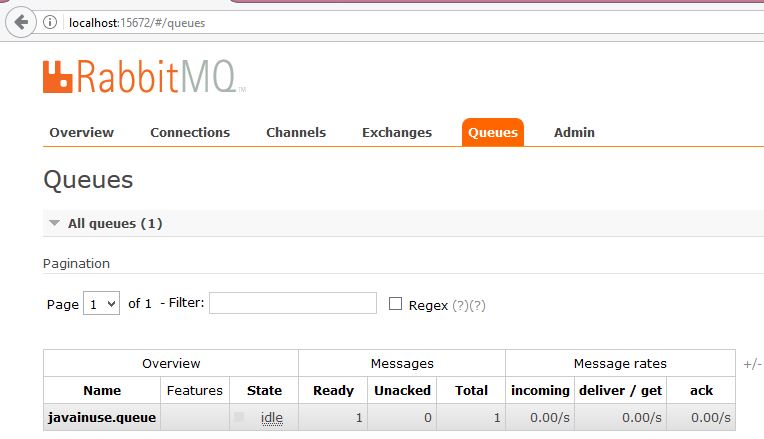


This will trigger the message to be sent to the javainuse queue.

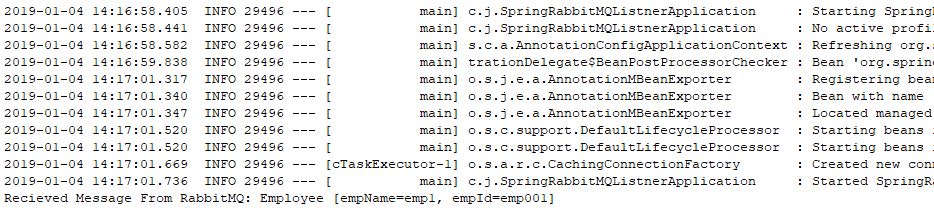
Next go to the RabbitMQ console-http://localhost:15672/



We can see in the Queues section, a queue name java gets created and it has one message.



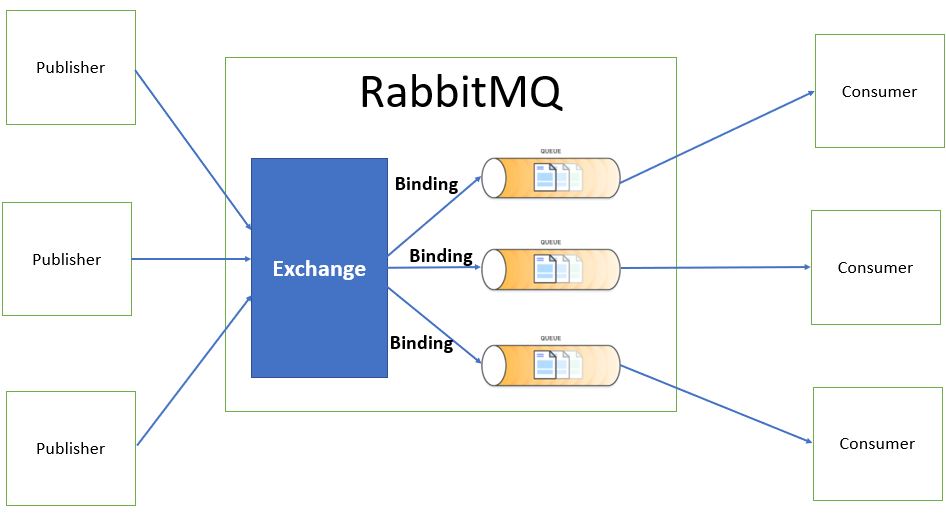
Next start the Spring Boot RabbitMQ Consumer application we just developed. It will consume the message from the queue named javainuse.queue.



# Spring Boot + RabbitMQ Tutorial - Implement Exchange Types

In this tutorial we will be implementing and understanding the various exchange types and their usage.

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.



With RabbitMQ we have the following types of Exchanges-

Direct Exchange

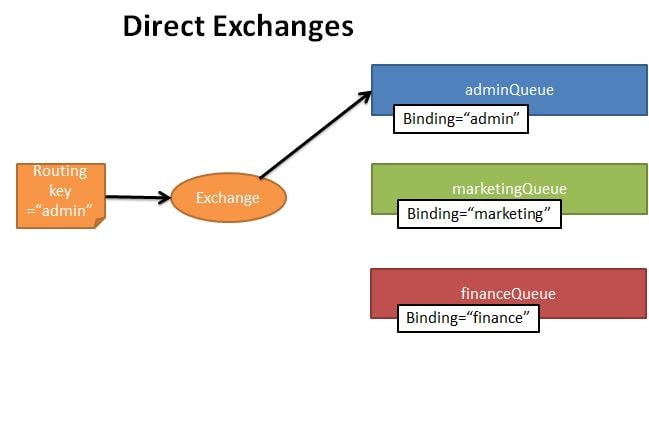
Fanout Exchange

Topic Exchange

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.



pom.xml

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

@Configuration

public class RabbitMQDirectConfig {

@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");

}

@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");

} }

Create the RabbitMQDirectWebController class where we expose API to send message to RabbitMQ Exchange.

@RestController

@RequestMapping(value = "/javainuse-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) {

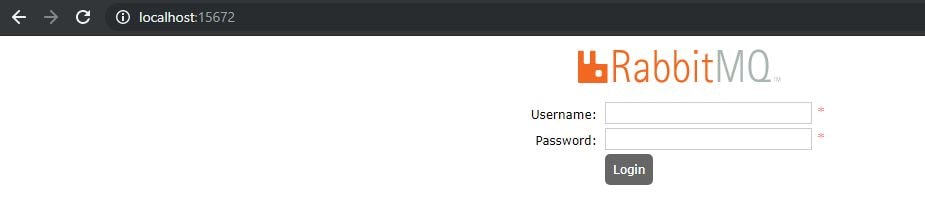
amqpTemplate.convertAndSend(exchange, routingKey, messageData);

return "Message sent to the RabbitMQ Successfully";

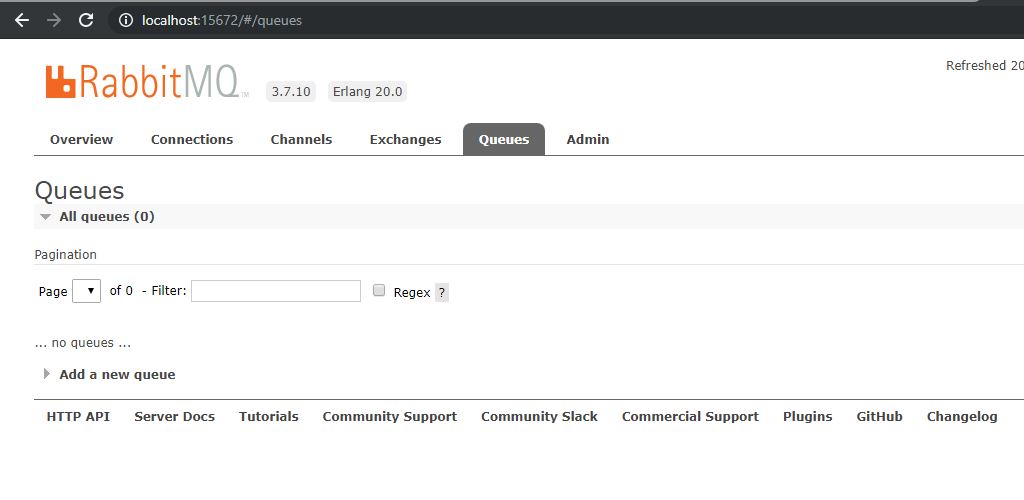
}

Create the bootstrap class with SpringBoot Annotation

Once you have followed implemented this tutorial go to localhost:15672



Use the username and password as guest.If we now go to the queues section, currently there are no queues



Run the Spring Boot Application

We send the message using the url - http://localhost:8080/javainuse-rabbitmq/direct/producer?exchangeName=direct-exchange&routingKey=admin&messageData=HelloWorldJavaInUse

we will be specifying the following

exchange name= "direct-exchange"

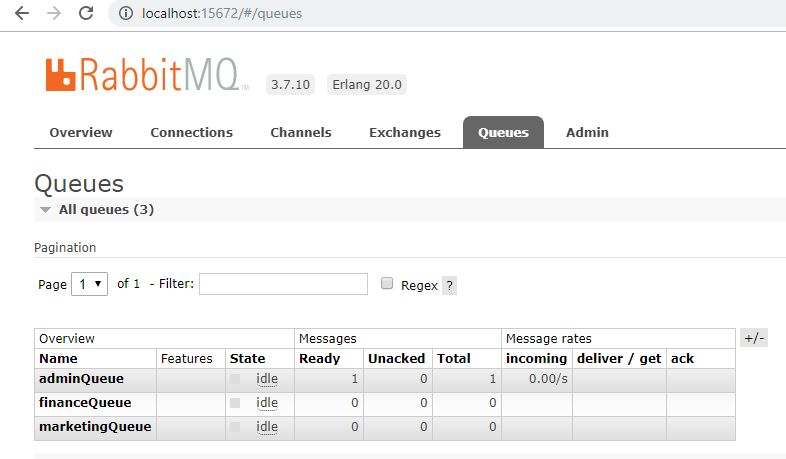
routing key ="admin"

message to sent to queue = "HelloWorldJavaInUse"

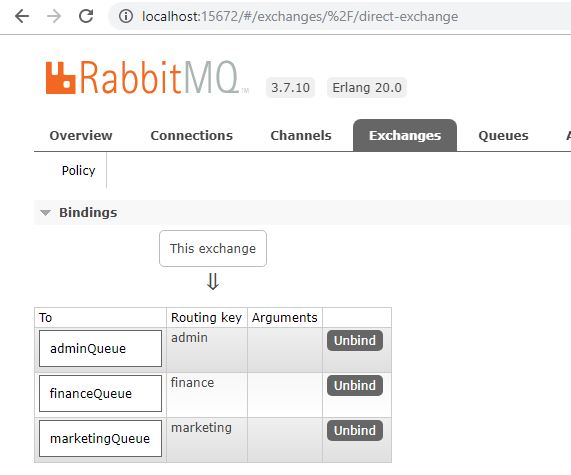
The message is sent to the admin queue.We get the web output as-



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

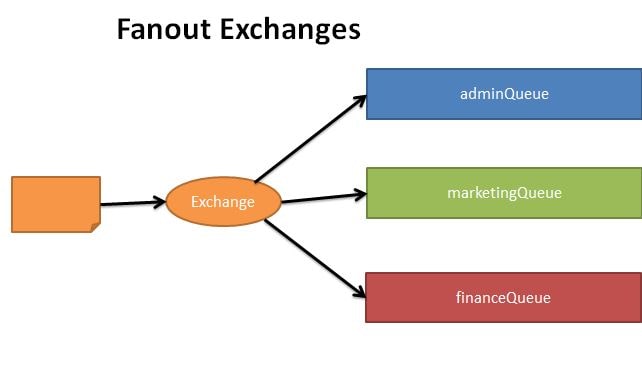


An exchange named direct-exchange is created with following bindings.



**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.



Modify the RabbitMQFanoutConfig as follows-

* Create Queues named - marketingQueue, adminQueue, financeQueue
* Create a FanoutExchange named - fanout-exchange
* Create Bindings for each of the queue with the FanoutExchange. Also as this is a fanout exchange we do not need to specify a binding key.

@Configuration

public class RabbitMQFanoutConfig {

@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

FanoutExchange exchange() { return new FanoutExchange("fanout-exchange"); }

@Bean

Binding marketingBinding(Queue marketingQueue, FanoutExchange exchange) {

return BindingBuilder.bind(marketingQueue).to(exchange); }

@Bean

Binding financeBinding(Queue financeQueue, FanoutExchange exchange) {

return BindingBuilder.bind(financeQueue).to(exchange); }

@Bean

Binding adminBinding(Queue adminQueue, FanoutExchange exchange) {

return BindingBuilder.bind(adminQueue).to(exchange); }

Create the RabbitMQFanoutWebController class where we expose API to send message to RabbitMQ Exchange.

@RestController

@RequestMapping(value = "/javainuse-rabbitmq/fanout/")

public class RabbitMQFanoutWebController {

@Autowired

private AmqpTemplate amqpTemplate;

@GetMapping(value = "/producer")

public String producer(@RequestParam("exchangeName") String exchange,

@RequestParam("messageData") String messageData) {

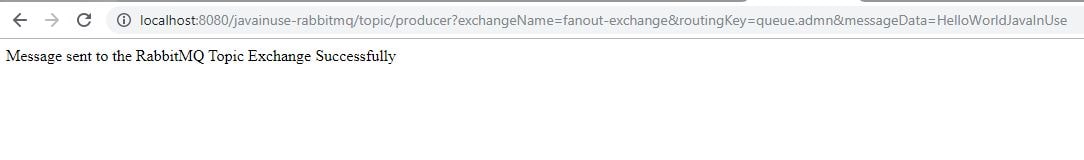
amqpTemplate.convertAndSend(exchange, "", messageData);

return "Message sent to the RabbitMQ Fanout Exchange Successfully"; }

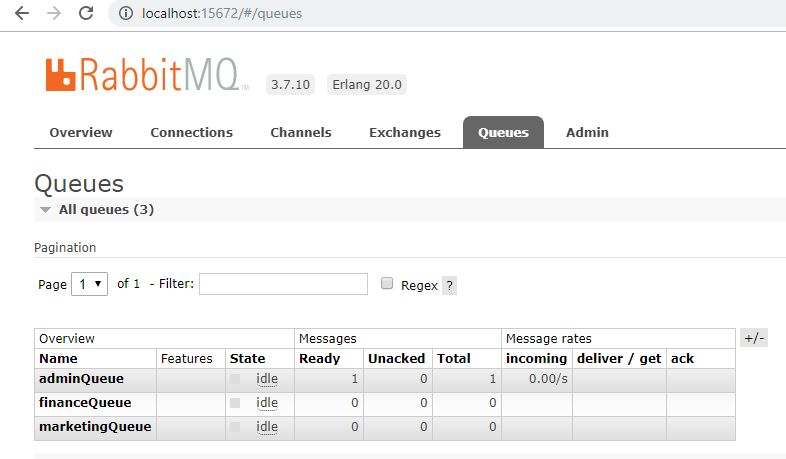
Run the Spring Boot Application

We send the message using the url - http://localhost:8080/javainuse-rabbitmq/fanout/producer?exchangeName=fanout-exchange&messageData=HelloWorldJavaInUse

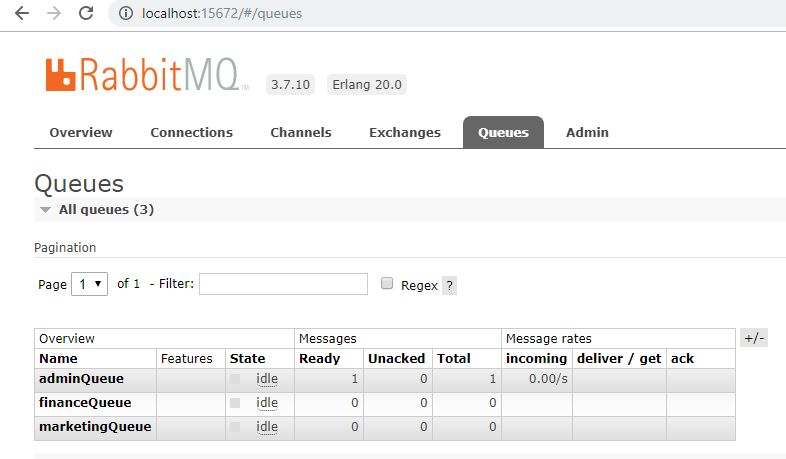
* exchange name= "fanout-exchange"
* message to sent to queue = "HelloWorldJavaInUse"



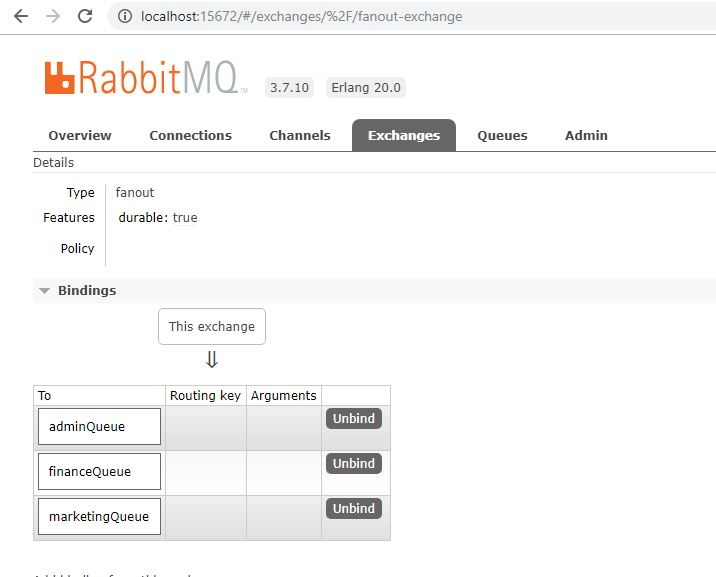
We do not need to specify the routing key here as message is published to all the queues. The message is sent to the admin queue.We get the web output as-



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

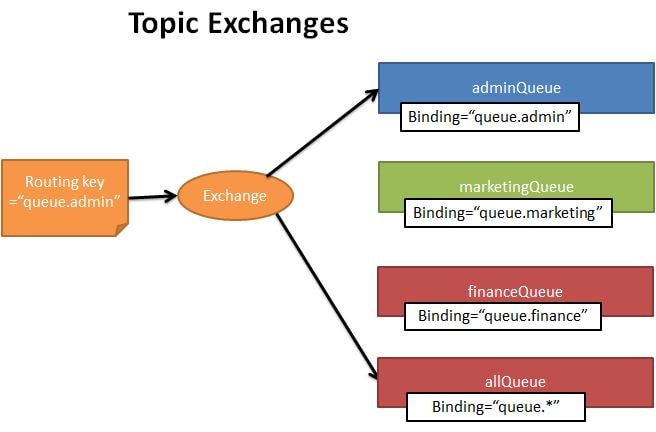


An exchange named fanout-exchange is created with following bindings



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.



Modify the RabbitMQTopicConfig as follows-

Create Queues named - marketingQueue, adminQueue, financeQueue and allQueue

Create a TopicExchange named - topic-exchange

Create Bindings for each of the queue with the TopicExchange. We specify routing key for each binding. Also for allQueue binding we specify the binding key with wildcard.

@Configuration

public class RabbitMQTopicConfig {

@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

Queue allQueue() {

return new Queue("allQueue", false);

}

@Bean

TopicExchange topicExchange() {

return new TopicExchange("topic-exchange");

}

@Bean

Binding marketingBinding(Queue marketingQueue, TopicExchange topicExchange) {

return BindingBuilder.bind(marketingQueue).to(topicExchange).with("queue.marketing");

}

@Bean

Binding financeBinding(Queue financeQueue, TopicExchange topicExchange) {

return BindingBuilder.bind(financeQueue).to(topicExchange).with("queue.finance");

}

@Bean

Binding adminBinding(Queue adminQueue, TopicExchange topicExchange) {

return BindingBuilder.bind(adminQueue).to(topicExchange).with("queue.admin");

}

@Bean

Binding allBinding(Queue allQueue, TopicExchange topicExchange) {

return BindingBuilder.bind(allQueue).to(topicExchange).with("queue.\*");

}

Create the RabbitMQTopicWebController class where we expose API to send message to RabbitMQ Exchange.

@RestController

@RequestMapping(value = "/javainuse-rabbitmq/topic/")

public class RabbitMQTopicWebController {

@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 Topic Exchange Successfully";

}

Run the Spring Boot Application

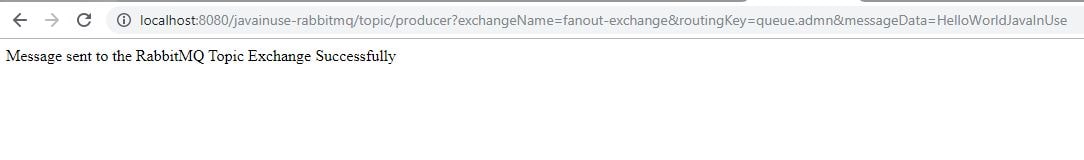
We send the message using the url - http://localhost:8080//javainuse-rabbitmq/topic/producer?exchangeName=topic-exchange&routingKey=queue.admin&messageData=HelloWorldJavaInUse

exchange name= "topic-exchange"

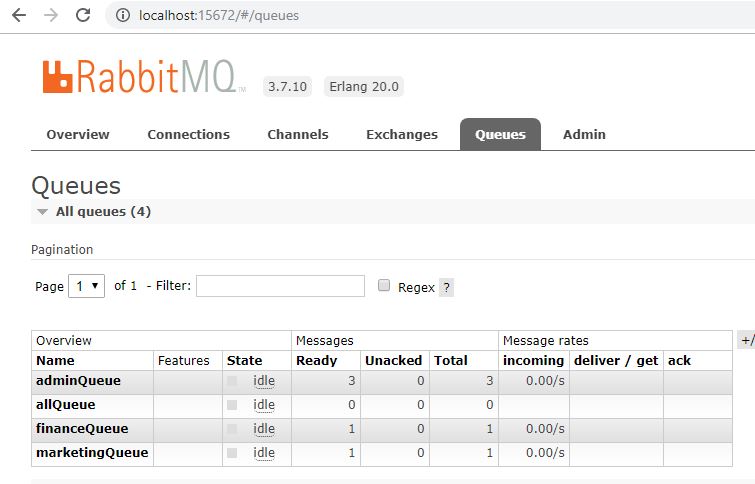
routing key ="queue.admin"

message to be sent to queue = "HelloWorldJavaInUse"

The message is sent to the admin queue.We get the web output as-

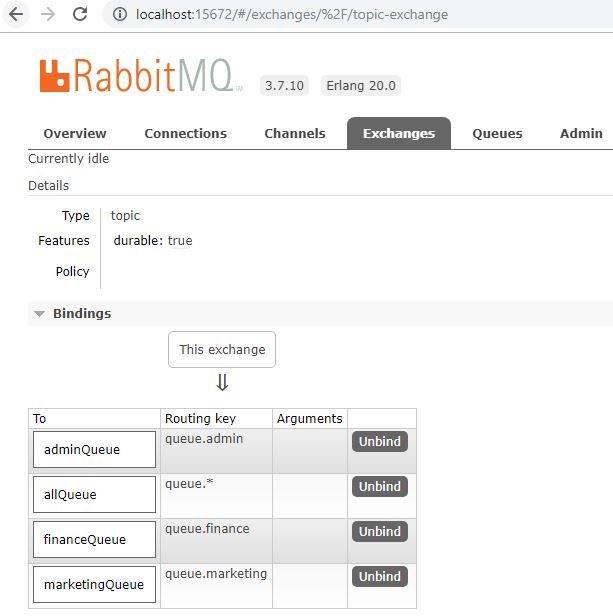


We can see that queues named marketingQueue, adminQueue and financeQueue and allQueue are created.



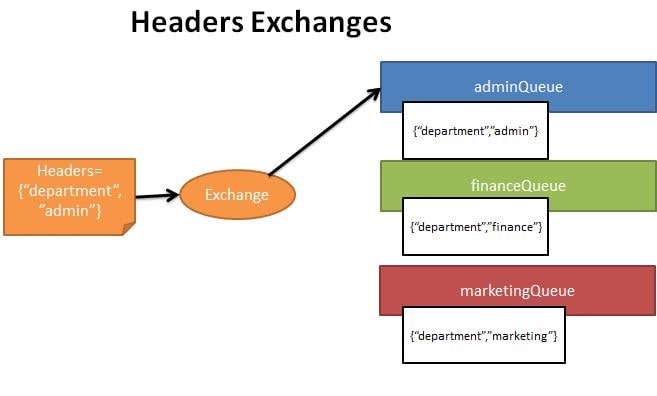
RabbitMQ Topic Exchange Queues

An exchange named topic-exchange is created with following bindings.



**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.



Modify the RabbitMQHeaderConfig as follows-

* Create Queues named - marketingQueue, adminQueue, financeQueue
* Create a HeaderExchange named - header-exchange
* Create Bindings for each of the queue with the HeaderExchange. As this is a Header Exchange, instead of binding key we specify the criteria rules which should be present in the message header.

@Configuration

public class RabbitMQHeaderConfig {

@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

HeadersExchange headerExchange() {return new HeadersExchange("header-exchange"); }

@Bean

Binding marketingBinding(Queue marketingQueue, HeadersExchange headerExchange) {

return BindingBuilder.bind(marketingQueue).to(headerExchange).where("department").

matches("marketing"); }

@Bean

Binding financeBinding(Queue financeQueue, HeadersExchange headerExchange) {

return BindingBuilder.bind(financeQueue).to(headerExchange).where("department"). matches("finance"); }

@Bean

Binding adminBinding(Queue adminQueue, HeadersExchange headerExchange) {

return BindingBuilder.bind(adminQueue).to(headerExchange).where("department").matches("admin");}

Create the RabbitMQHeaderWebController class where we expose API to send message to RabbitMQ Exchange.

@RestController

@RequestMapping(value = "/javainuse-rabbitmq/header/")

public class **RabbitMQHeaderWebController** {

@Autowired

private AmqpTemplate amqpTemplate;

@GetMapping(value = "/producer")

public String producer(@RequestParam("exchangeName") String exchange, @RequestParam("department") String department,

@RequestParam("messageData") String messageData) {

MessageProperties messageProperties = new MessageProperties();

messageProperties.setHeader("department", department);

MessageConverter messageConverter = new SimpleMessageConverter();

Message message = messageConverter.toMessage(messageData, messageProperties);

amqpTemplate.send(exchange, "", message);

return "Message sent to the RabbitMQ Header Exchange Successfully";

}

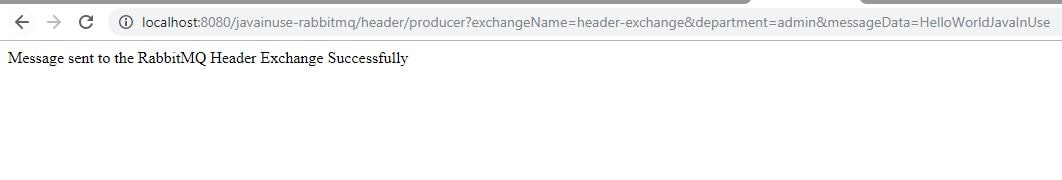
Run the Spring Boot Application

We send the message using the url - **http://localhost:8080/javainuse-rabbitmq/header/producer?exchangeName=header-exchange&department=admin&messageData=HelloWorldJavaInUse**

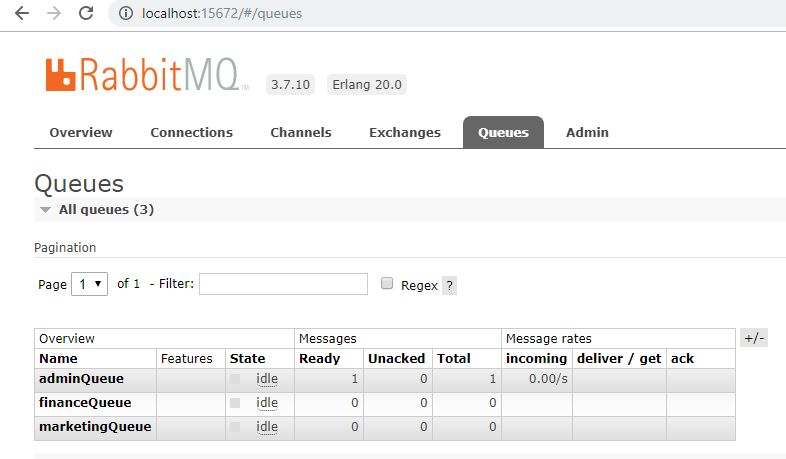
* exchange name= "header-exchange"
* header key ="admin"
* message to be sent to queue = "HelloWorldJavaInUse"

The message is sent to the admin queue.We get the web output as-

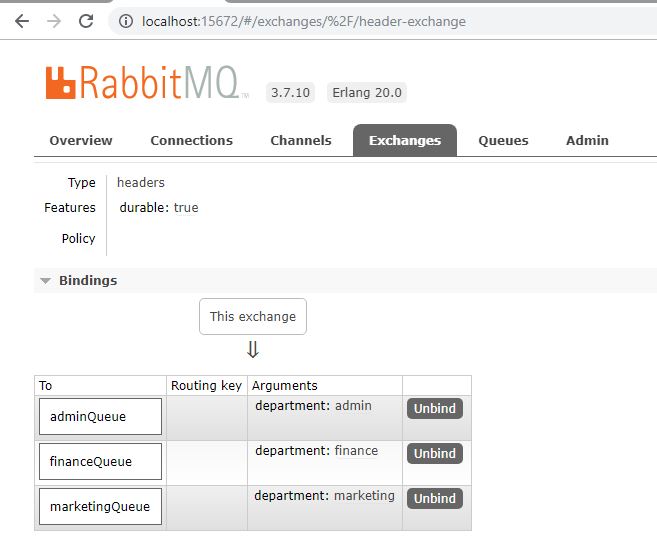
RabbitMQ Heaader Exchange Output



We can see that queues named marketingQueue, adminQueue and financeQueue are created. Also adminQueue has a message.



An exchange named header-exchange is created with following bindings.



# Spring Boot + RabbitMQ Tutorial - Retry and Error Handling Example | JavaInUse

In this tutorial we will be implementing a Spring Boot + RabbitMQ example to retry messages on exception and 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.

According to Wikipedia page - 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.

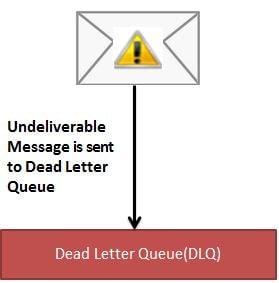
Queue length limit exceeded.

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".

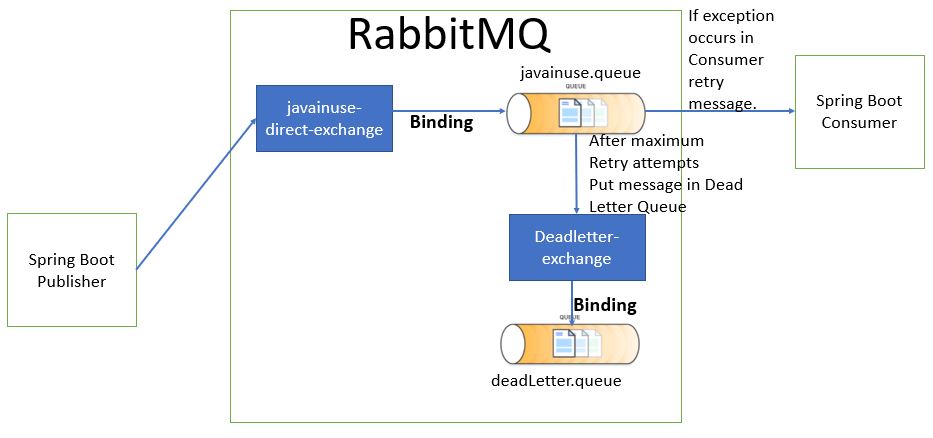
Later on we can analyze the messages in the DLQ to know the reason why the messages are failing.



We will be implementing two modules -

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.



Spring Boot + RabbitMQ Producer Module

POM.xml

Define the domain class Employee as follows-

@JsonIdentityInfo(generator = ObjectIdGenerators.IntSequenceGenerator.class, property = "@id", scope = Employee.class)

public class Employee {

private String empName;

private String empId;

private int salary;

@JsonIdentityInfo(generator = ObjectIdGenerators.IntSequenceGenerator.class, property = "@id", scope = Employee.class)

public class Employee {

private String empName;

private String empId;

private int salary;

}

Next define the configuration class where we-

Create Direct Exchanges named - deadLetterExchange and javainuseExchange.

Create Queue named javainuse and dlq. For the javainuse queue specify the x-dead-letter-exchange argument as the deadLetterExchange. This means that any message in javainuse queue that cannot be delivered will be sent to the deadLetterExchange.

Bind the javainuse queue with javainuseExchange and the dlq queue with deadLetterExchange.

@Configuration

public class RabbitMQConfig {

@Bean

DirectExchange deadLetterExchange() {

return new DirectExchange("deadLetterExchange");

}

@Bean

DirectExchange exchange() {

return new DirectExchange("javainuseExchange");

}

@Bean

Queue dlq() {

return QueueBuilder.durable("deadLetter.queue").build();

}

@Bean

Queue queue() {

return QueueBuilder.durable("javainuse.queue").withArgument("x-dead-letter-exchange", "deadLetterExchange")

.withArgument("x-dead-letter-routing-key", "deadLetter").build();

}

@Bean

Binding DLQbinding() {

return BindingBuilder.bind(dlq()).to(deadLetterExchange()).with("deadLetter");

}

@Bean

Binding binding() {

return BindingBuilder.bind(queue()).to(exchange()).with("javainuse");

}

@Bean

public MessageConverter jsonMessageConverter() {

return new Jackson2JsonMessageConverter();

}

public AmqpTemplate rabbitTemplate(ConnectionFactory connectionFactory) {

final RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);

rabbitTemplate.setMessageConverter(jsonMessageConverter());

return rabbitTemplate;

}

Create the RabbitMQWebController class where we expose API to send message to RabbitMQ Exchange.

@RestController

@RequestMapping(value = "/javainuse-rabbitmq/")

public class RabbitMQWebController {

@Autowired

private AmqpTemplate amqpTemplate;

@GetMapping(value = "/producer")

public String producer(@RequestParam("empName") String empName,@RequestParam("empId") String empId,@RequestParam("salary") int salary) {

Employee emp=new Employee();

emp.setEmpId(empId);

emp.setEmpName(empName);

emp.setSalary(salary);

amqpTemplate.convertAndSend("javainuseExchange", "javainuse", emp);

return "Message sent to the RabbitMQ Successfully";

}

Create the Spring Boot Bootstrap class with SpringBootApplication annotation.

Spring Boot Consumer Module

Define the domain class Employee as follows-

@JsonIdentityInfo(generator = ObjectIdGenerators.IntSequenceGenerator.class, property = "@id", scope = Employee.class)

public class Employee {

private String empName;

private String empId;

private int salary;

}

Define a custom checked exception named InvalidSalaryException as follows-

public class InvalidSalaryException extends Exception {

private static final long serialVersionUID = -3154618962130084535L; }

Define the RabbitMQConsumer class which consumes the message from RabbitMQ using RabbitListener.The RabbitMQ Listener listens to RabbitMQ Queue for any incoming messages. For the basic configuration we specify the the Queue/Topic Name (the name of the queue/topic where the message should be consumed). Also here we will be checking the incoming message for salary field. If this field is negative we will be throwing an InvalidSalaryException.

@Component

public class RabbitMQConsumer {

private static final Logger logger = LoggerFactory.getLogger(RabbitMQConsumer.class);

@RabbitListener(queues = "javainuse.queue")

public void recievedMessage(Employee employee) throws InvalidSalaryException {

logger.info("Recieved Message From RabbitMQ: " + employee);

if (employee.getSalary() < 0) {

throw new InvalidSalaryException();

}

}

Next define th following properties in application.yml. Here we enable the Spring Boot RabbitMQ retry mechanism and specify some more additional parameters-

initial interval- The message should be retried after an interval of 3s.

max-attempts- The message should be retried maximum of 6 times. After which it will be sent to dead letter Queue.

max-interval- The maximum time interval between two retries should never exceed 10s.

multiplier- The interval between second retry gets multiplied by 2. But this interval can never exceed the max-interval. So the retry interval values will be 3s, 6s, 10s, 10s, 10s. As 10 sec is the max interval specified.

spring.rabbitmq.listener.simple.retry.enabled=true

spring.rabbitmq.listener.simple.retry.initial-interval=3s

spring.rabbitmq.listener.simple.retry.max-attempts=6

spring.rabbitmq.listener.simple.retry.max-interval=10s

spring.rabbitmq.listener.simple.retry.multiplier=2

server.port=8081

Finally Define the Spring Boot Class with @SpringBootApplication annotation

@Bean

public Jackson2JsonMessageConverter converter() {

return new Jackson2JsonMessageConverter();

}

In a previous tutorial we have shown how to install RabbitMQ and get started.

Start the Producer and Consumer applications. And go to http://localhost:8080/javainuse-rabbitmq/producer?empName=emp1&empId=emp001&salary=-50 The message will be sent to rabbitMQ queue named javainuse.queue and consumed by the consumer application. As the salary is negative , InvalidSalaryException will be thrown. This message will be retried 6 times and then will be put in dead letter queue.

