**RabbitMQ**

**What is RabbitMQ?**

**RabbitMQ** is an **AMQP** messaging broker and it is the most popular open source and cross-platform message broker.

RabbitMQ is also a way to exchange the data between different platform applications such as a message sent from .Netapplication can be read by a Node.js application or Java application.

 The RabbitMQ is built on Erlang general-purpose programming language and it is also used by WhatsApp for messaging.

**RabbitMQ we send messages to Exchange and depending on Routing Key that message will be forwarded to Queue(s)**

**Why and when to Use RabbitMQ?**

Now a days most people will perform a multiple tasks in single application like sending emails or SMS, reports and it will create a heavy load on application so if you separate these tasks, then we will get more space (memory) to serve more requests.

By using RabbitMQ, we can remove some heavy work from our web applications such as sending a reports in Excel or Pdf format’s or sending an email, SMS or another task such as trigger some other applications to start processing.

RabbitMQ is an open source and cross-platform message broker so it’s easy to use with many languages such as .Net, Java, Python, Ruby, Node.Js.

**RabbitMQ Supported Client Libraries**

RabbitMQ will support multiple operating systems and programming languages. RabbitMQ has provided a various client libraries for following programming languages.

* .Net
* Java
* Spring Framework
* Ruby
* Python
* PHP
* Objective-C and Swift
* JavaScript
* GO
* Perl

### **AMQP:**

AMQP stands for Advanced Message Queuing Protocol. AMQP is an open standard protocol for implementing MOMs (Message Oriented Middleware).

The features of AMQP are message orientation, queuing, routing (including point-to-point and publish-and-subscribe), reliability and security.

It was developed by JPMorgan and iMatix Corporation. AMQP was designed with the following main characteristics as goals:

* Security
* Reliability
* Interoperability
* Standard
* Open

RabbitMQ is lightweight and easy to deploy on available premises and it supports multiple messaging protocols. RabbitMQ can be deployed in distributed and federated configurations to meet high-scale, high-availability requirements.

The main drawback or limitation of JMS API is interoperability that means we can develop Messaging systems that will work only in Java-based applications. It does not support other languages.

### AMQP solves the JMS API problem. The major advantage of AMQP is that it supports interoperability between heterogeneous platforms and messaging brokers. We can develop our Messaging systems in any language (Java, C++, C#, Ruby etc.) and in any operating system; still, they can communicate with each other by using AMQP based message broker

**RabbitMQ Exchanges:**

In rabbitmq, when producer creates a message that will not directly sent to a queue, instead first the message will be send to exchanges, then after that a routing agent reads and sends it to the appropriate queue with help of header attributes, bindings, and routing keys.

## **RabbitMQ Exchange Types**

In rabbitmq, we have a four type of Exchanges are available to route the message in different ways.

Following are the different type of exchanges available in rabbitmq.

* Default
* Direct
* Fanout
* Topic
* Headers

### **Bindings**

A *binding* is an association between a queue and an exchange. A queue must be bound to at least one exchange in order to receive messages from publishers

**Exchange attributes**

Exchanges have several attributes associated with them:

* Name
* Type (direct, fanout, topic, headers or some custom type)
* Durability
* Whether the exchange is auto-deleted when no longer used
* Other metadata (sometimes known as *X-arguments*)

## **Message attributes**

Every AMQP message has a number of *attributes*. Some attributes are important and used very often, others are rarely used. Message attributes are metadata and are similar in purpose to HTTP request and response headers.

Every message has an attribute called *routing key*. The routing key is an "address" that the exchange may use to decide how to route the message . This is similar to, but more generic than, a URL in HTTP. Most exchange types use the routing key to implement routing logic, but some ignore it and use other criteria (e.g. message content).

## **Direct Exchange**

A direct exchange delivers messages to queues based on a *message routing key*, an attribute that every AMQP v0.9.1 message contains.

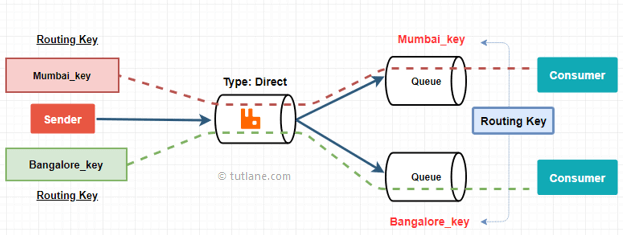
In rabbitmq, **direct** exchange will deliver a messages to the queues based on the message routing key. In direct exchange, the message is routed to the queues whose binding key exactly matches with the routing key of the message.

0000000000000000

**Direct exchanges can be used in a wide variety of cases:**

* Direct (near real-time) messages to individual players in an MMO game
* Delivering notifications to specific geographic locations (for example, points of sale)
* Distributing tasks between multiple instances of the same application all having the same function, for example, image processors
* Passing data between workflow steps, each having an identifier (also consider using headers exchange)
* Delivering notifications to individual software services in the network

Following is the pictorial representation of message flow in rabbit direct exchange.



**Default exchange:**

The default exchange is implicitly bound to every queue, with a routing key equal to the queue name. It is not possible to explicitly bind to, or unbind from the default exchange. It also cannot be deleted.

*Every queue is automatically bound to it with a routing key which is the same as the queue name*.

For example, when you declare a queue with the name of "search.indexing.online", RabbitMQ will bind it to the default exchange using "search.indexing.online" as the routing key. Therefore a message published to the default exchange with routing key = "search.indexing.online" will be routed to the queue "search.indexing.online"

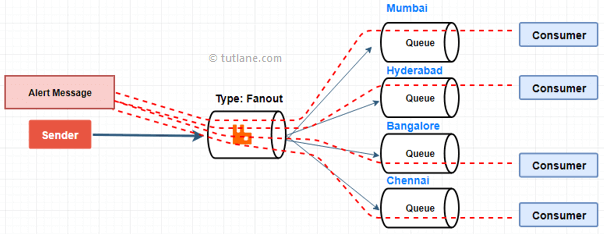
The default exchange is a pre-declared direct exchange with no name, usually referred by the empty string "". When you use the default exchange, your message is delivered to the queue with a name equal to the routing key of the message. Every queue is automatically bound to the default exchange with a routing key which is the same as the queue name.

**Purpose:**This is a Default exchange type.It is used when a message needs to send to a queue

## **Fanout Exchange:**

In rabbitmq, fanout exchange will route messages to all of the queues that are bound to it,

Following is the pictorial representation of message flow in rabbitmq fanout exchange.



**Purpose:**Useful for broadcast feature using publish subscribe pattern

**Use cases:**

* Because a fanout exchange delivers a copy of a message to every queue bound to it, its use cases are quite similar:
* Massively multiplayer online (MMO) games can use it for leaderboard updates or other global events
* Sport news sites can use fanout exchanges for distributing score updates to mobile clients in near real-time
* Distributed systems can broadcast various state and configuration updates
* Group chats can distribute messages between participants using a fanout exchange

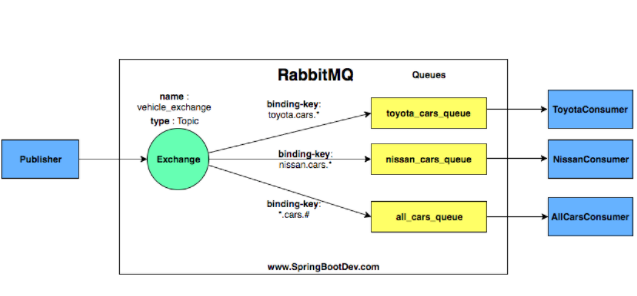
**Topic Exchange:**

Topic exchanges are similar to direct, but they use patterns instead of exact matching. Pattern is a list of words separated by dots – aaa.bbb.ccc. Similar to direct exchanges, pattern is not specified on exchange level, but for each of binding separately.

Same as Direct, but wildcards are allowed in the binding key. ‘#‘ matches zero or more dot-delimited words and ‘\*‘ matches exactly one such word.

* (star) can substitute for exactly one word.
* # (hash) can substitute for zero or more words.

Topic exchanges route messages to one or many queues based on matching between a message routing key and the pattern that was used to bind a queue to an exchange. The topic exchange type is often used to implement various publish/subscribe pattern variations. Topic exchanges are commonly used for the multicast routing of messages.



manufacturer\_name.car.type

As you can see, we have created a Topic exchange called vehicle\_exchange. In addition, we have created three queues and bound them to the Topic exchange (vehicle\_exchange) with following binding keys.

toyota\_cars\_queue  ( binding key – toyota.cars.\* )

toyota\_cars\_queue is bound to the exchange with wildcard biding key toyota.cars.\*  This means that the any message starting with routing key toyota.cars and ending with exactly any text should be directed to this queue (toyota\_cars\_queue).

nissan\_cars\_queue  ( binding key – nissan.cars.\* )

In the same way, any message starting with routing key nissan.cars and ending with exactly any text should be directed to the nissan\_cars\_queue.

all\_cars\_queue  ( binding key – \*.cars.#)

According to the binding key, the routing key should start with any text (exactly one) and followed by cars keyword. it can be ended with zero or more texts.

e.g:-  The following set of routing keys can be identified as valid routing key for this binding pattern.

nissan.cars ,  toyota,cars  , any.cars  , anything.cars.everything , anything.cars.anything.and.everything

The following table will show you a list of routing keys and their destination queue(s)

|  |  |
| --- | --- |
| routing key | delivered to queue(s) |
| nissan.cars.japan | nissan\_cars\_queue and all\_cars\_queue |
| nissan.cars | all\_cars\_queue |
| toyota.cars.japan.manufactured | all\_cars\_queue |
| japan.toyota.cars | No matching queue. message will be discard |
| import.nissan.cars.from.japan | No matching queue. message will be discard |
| toyota.cars.manufatured | toyota\_cars\_queue and all\_cars\_queue |
| no.latest.cars.toyota | No matching queue. message will be discard |

#### Topic exchange : More to consider

Topic exchange is powerful and can behave like other exchanges.

When a queue is bound with “#” (hash) binding key – it will receive all the messages, regardless of the routing key – like in fanout exchange.

When special characters “\*” (star) and “#” (hash) aren’t used in bindings, the topic exchange will behave just like a direct one.

Reference:

<https://springbootdev.com/2017/11/12/spring-amqp-rabbitmq-topic-exchange-example-part-1-producer-application/>

<https://grokonez.com/spring-framework/spring-amqp/springboot-rabbitmq-headers-exchange>

**Headers Exchange**:

**Headers Exchange** will use the message header attributes to send a messages to queue instead of routing key.

The headers exchange is useful when we want to route a messages based on header values instead of routing keys.

Reference: <https://www.tutlane.com/tutorial/rabbitmq/csharp-rabbitmq-headers-exchange>

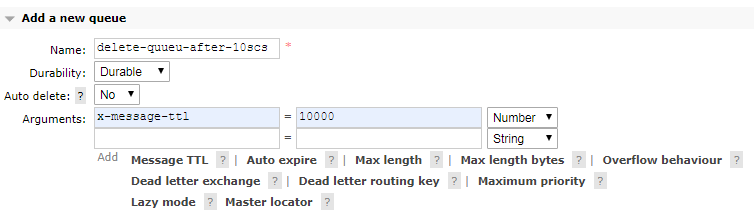
**Delete Message after 10 seconds:**

In this queue once publish the message to queue there is no consumer consume the message,

That message will be automatically deleted 10 seconds from the queue.

Attribute for queue:

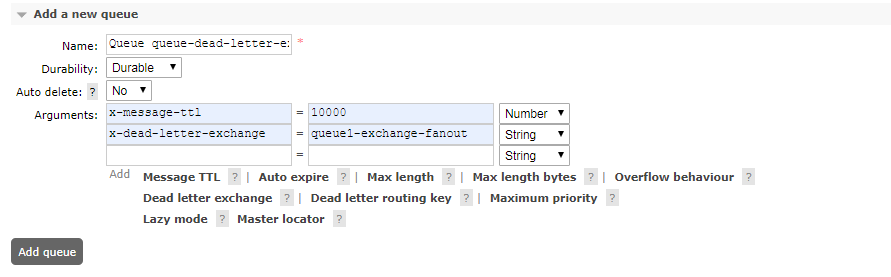
x-message-ttl: 10000



**Dead letter exchange(dlx):**

In this queue once publish the message to queue there is no consumer consume the message,

That message will be automatically deleted 10 seconds from the queue and send that message to other exchange (queues).



Attribute for queue:

x-message-ttl: 10000

x-dead-letter-exchange: queue1-exchange-fanout

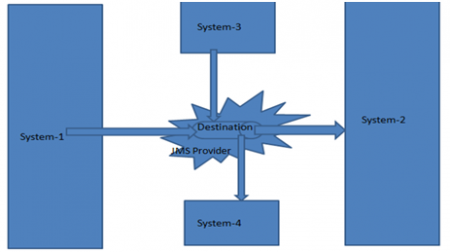
### **Difference between JMS and AMQP**

In this section, we will discuss about what are the differences between JMS and AMQP protocols.

1. **Interoperability**

JMS Application works in any OS environment, but it supports only Java platform. If we want to develop the below system architecture by using JMS API then all those systems should be developed by using Java language only.

But if we use AMQP standard to develop the same system, then we can develop those systems in any language that means System-1(JAVA), System-2(C#), System-3(Ruby) and System-4(C++).



JMS API is specific for Java Platform only, but AMQP supports many technologies.

**Definition :**

**JMS :** Java Message Service is an API that is part of Java EE for sending messages between two or more clients.  There are many JMS providers such as OpenMQ (glassfish’s default), HornetQ(Jboss), and ActiveMQ.

**RabbitMQ:** is an open source message broker software which uses the AMQP standard and is written by Erlang.

**Messaging Model:**

JMS supports two models: one to one and publish/subscriber. RabbitMQ supports the AMQP model which has 4 models : direct, fanout, topic, headers.

**Data types:**

JMS supports 5 different data types but RabbitMQ supports only the binary data type.

**Workflow strategy:**

In AMQP, producers send to the exchange then the queue, but in JMS, producers send to the queue or topic directly.

**Technology compatibility:**

JMS is specific for java users only, but RabbitMQ supports many technologies.

**Jackson JSON**

Jackson JSON Parser API provides easy way to convert JSON to java, java to Json and supports easy conversion to Map from JSON data.

how to serialize Java objects into JSON and deserialize JSON string into Java objects

## **Create Message Listeners**

Implementing message listener is tricky as it requires handling some of the scenarios like:

* How to auto deserialize the message to a POJO
* What if listener is making a REST call to some API which is unreachable, or what if an error occurred on the API side while processing the request?
* How to make multiple listeners to concurrently pop the message from queue and process
* When and how to re-queue the message in the message queue in failure scenarios

### **Deserializing Message to POJO**

Spring provides an annotation @RabbitListener , which can be used to receive messages from the queue. It has a great feature of deserializing the message to a POJO while receiving. The below example illustrates that.

### **Error Handling and Message Re-Queuing Feature in Listener**

* Listener is trying to call an API to process the request that is unreachable
* Listener has called the API but error occurred in API while processing the request

In this situation, depending on your business requirement, either you should not re-queue the message, or you should re-queue with max number of trial option to re-try to process it up to a limit.

To not requeue the message in queue, you can throw exception AmqpRejectAndDontRequeueException . For max number of trial handling, you can add an additional parameter in the message to set max number of trial and use it while receiving the message by incrementing it’s value and checking whether total number of trial has not exceeded max limit.

There is an alternative of above approach to add this properties in application.properties and specify max number of attempts:-

spring.rabbitmq.listener.simple.retry.max-attempts=3

## **Concurrency Capability**

Concurrency feature can be implemented in 2 ways:

* Create a thread pool with a specified number of max threads and using ThreadExecutorcall the methods/APIs to process the request.
* By using Inbuild concurrency feature. I believe this is the simplest approach to implement concurrency. This requires just making use of 2 properties in application.properties file.

**Note:** You can set the values of these properties as per your application scalability.

spring.rabbitmq.listener.simple.concurrency=4

spring.rabbitmq.listener.simple.max-concurrency=8

**Example:**

|  |
| --- |
| #AMQP RabbitMQ configuration  spring.rabbitmq.host=localhost  spring.rabbitmq.port=5672  spring.rabbitmq.username=guest  spring.rabbitmq.password=guest  # Message Queue specific configs for app  rabbitmq.exchange=demo.exchange  rabbitmq.queue=demo.queue  rabbitmq.routingkey=demo.routingkey  # Additional RabbitMQ properties  spring.rabbitmq.listener.simple.concurrency=4  spring.rabbitmq.listener.simple.max-concurrency=8  spring.rabbitmq.listener.simple.retry.initial-interval=5000  spring.rabbitmq.listener.acknowledge-mode=manual  Configration files:  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.core.RabbitTemplate;  import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;  import org.springframework.amqp.support.converter.MessageConverter;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.context.annotation.Bean;  import org.springframework.context.annotation.Configuration;  import org.springframework.amqp.rabbit.connection.ConnectionFactory;  @Configuration  public class RabbitMqConfig {  @Value("${rabbitmq.queue}")  private String queueName;  @Value("${rabbitmq.exchange}")  private String exchange;  @Value("${rabbitmq.routingkey}")  private String routingKey;  @Bean  public Queue queue() {  return new Queue(queueName, true);  }  @Bean  public DirectExchangeexchange() {  return new DirectExchange(exchange);  }  @Bean  public Binding binding(Queue queue, DirectExchange exchange) {  return BindingBuilder.bind(queue).to(exchange).with(routingKey);  }  @Bean  public MessageConverterjsonMessageConverter() {  return new Jackson2JsonMessageConverter();  }    public AmqpTemplaterabbitTemplate(ConnectionFactoryconnectionFactory) {  final RabbitTemplaterabbitTemplate = new RabbitTemplate(connectionFactory);  rabbitTemplate.setMessageConverter(jsonMessageConverter());  return rabbitTemplate;  }  }  **Publisher:**  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.beans.factory.annotation.Value;  import org.springframework.stereotype.Component;  import com.mng.spring.model.Employee;  @Component  public class RabbitMqPublisher {    @Autowired  public RabbitTemplaterabbitTemplate;    @Value("${rabbitmq.exchange}")  private String exchange;  @Value("${rabbitmq.routingkey}")  private String routingKey;    @Scheduled(fixedDelay = 3000L)  public void send(Employee employee) {  rabbitTemplate.convertAndSend(exchange, routingKey, employee);  System.out.println("Send msg = " + employee);  }  }   * @Scheduled  annotation is used to schedule a method to be executed at a given time, or with a given frequency. In this case, this method will be executed every 3 seconds (until the end of the times or you stop the application, whatever happens earlier). You can check more info [here](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/scheduling.html). * RabbitTemplate is used to convert and send a message using RabbitMQ. It is a helper class, as many other Template classes existing in Spring (such as JdbcTemplate , RestTemplate , etc.). Spring Boot creates a default version for you, but in this case, we will need to tune it a little bit to make it use JSON converter when producing messages.   **Subscriber:**  package com.mng.spring.rabbitmq;  import org.slf4j.Logger;  import org.slf4j.LoggerFactory;  import org.springframework.amqp.AmqpRejectAndDontRequeueException;  import org.springframework.amqp.rabbit.annotation.RabbitListener;  import org.springframework.http.HttpStatus;  import org.springframework.stereotype.Component;  import org.springframework.web.client.HttpClientErrorException;  import org.springframework.web.client.RestTemplate;  import com.mng.spring.model.Employee;  import com.rabbitmq.client.Channel;  import com.rabbitmq.client.Envelope;  @Component  public class RabbitMqSubcriber {    private static final Logger log = LoggerFactory.getLogger(RabbitMqSubcriber.class);    @RabbitListener(queues="${rabbitmq.queue}")  public void receiveObject(Employee employee, Envelope env, Channel channel) {  try {  log.info("Received Message: " + employee);  log.info("Making REST call to the API");    RestTemplaterestTemplate = new RestTemplate();  restTemplate.postForObject("http://localhost:8585/msg", employee, Employee.class);  //restTemplate.postForObject("http://localhost:8585/msg",Employee.class, employee);    //TODO: Code to make REST call  log.info("<< Exiting receiveMessageCrawlCI() after API call.");  } catch(HttpClientErrorException ex) {  if(ex.getStatusCode() == HttpStatus.NOT\_FOUND) {  log.info("Delay...");  try {  Thread.sleep(5000);  } catch (InterruptedException e) { }    log.info("Throwing exception so that message will be requed in the queue.");  // Note: Typically Application specific exception can be thrown below  throw new RuntimeException();  } else {  throw new AmqpRejectAndDontRequeueException(ex);  }    } catch(Exception e) {  log.error("Internal server error occurred in python server. Bypassing message requeue {}", e);  throw new AmqpRejectAndDontRequeueException(e);  }  }  }  **Controller:**  package com.mng.spring.controller;  import org.springframework.beans.factory.annotation.Autowired;  import org.springframework.http.MediaType;  import org.springframework.web.bind.annotation.RequestBody;  import org.springframework.web.bind.annotation.RequestMapping;  import org.springframework.web.bind.annotation.RequestMethod;  import org.springframework.web.bind.annotation.RestController;  import com.mng.spring.model.Employee;  import com.mng.spring.rabbitmq.RabbitMqPublisher;  @RestController  public class RabbitMqEmployeeController {    @Autowired  public RabbitMqPublisherrabbitMqPublisher;    @RequestMapping("/hello")  public String hello() {  return "welcome rabbit mq programing";  }    @RequestMapping("/emp")  public Employee getEmployee() {  Employee obj = new Employee();  obj.setEmpId("1235");  obj.setEmpName("Nagendar");  return obj;  }    @RequestMapping(path = "/save", method = RequestMethod.POST, consumes = {  MediaType.APPLICATION\_JSON\_VALUE })  public void save(@RequestBody Employee employee) {  rabbitMqPublisher.send(employee);  System.out.println("employee object processed");  }  }  **Model class:**  @JsonIdentityInfo(generator= ObjectIdGenerators.PropertyGenerator.**class**,  property="empId")  **publicclass** Employee {  **private** String empName;  **private** String empId;  **public** String getEmpName() {  **return**empName;  }  **publicvoid**setEmpName(String empName) {  **this**.empName = empName;  }  **public** String getEmpId() {  **return**empId;  }  **publicvoid**setEmpId(String empId) {  **this**.empId = empId;  }  @Override  **public** String toString() {  **return**"Employee [empName=" + empName + ", empId=" + empId + ", getEmpName()=" + getEmpName() + ", getEmpId()="  + getEmpId() + ", getClass()=" + getClass() + ", hashCode()=" + hashCode() + ", toString()="  + **super**.toString() + "]";  }  } |

Jackson is a suite of data-processing tools for Java comprising of three components:

* Streaming (jackson-core) defines low-level streaming APIs and includes JSON-specific implementations.
* Annotations (jackson-annotations) contains standard Jackson annotations.
* Databind (jackson-databind) implements data-binding (and object serialization) support on the streaming package. This package depends on both the streaming and annotations packages.

Annotations:

* @JsonIgnore
* @JsonIgnoreProperties
* @JsonIgnoreType
* @JsonAutoDetect

### @JsonIgnore

The @JsonIgnore annotation marks a field of a POJO to be ignored by Jackson during serialization and deserialization. Jackson ignores the field both JSON serialization and deserialization

### @JsonIgnoreProperties

The @JsonIgnoreProperties annotation is used at the class level to ignore fields during serialization and deserialization

### @JsonIgnoreType

The @JsonIgnoreType annotation is used to mark a class to be ignored during serialization and deserialization

### @JsonPropertyOrder

The @JsonPropertyOrder annotation tells Jackson to serialize the Java object to JSON in the order specified as the arguments of the annotation.

### @JsonRootName

The @JsonRootName annotation can be used to tell Jackson to wrap the object to be serialized with a top-level element. You can pass the name as a parameter to the @JsonRootName annotation.

### @JsonGetter

The @JsonGetter annotation is used to customize the generated JSON keys.

|  |
| --- |
| publicclassIgnoreDemoBean {  @JsonIgnore  publiclongpersonId = 0;  publicString name = "James Clark";  ---  }  @JsonIgnoreProperties({"userId", "gender"})  publicclassIgnorePropertiesDemoBean {  publiclonguserId=0;  publicStringname="James Clark";  publicStringgender=null;  ----  }  @JsonPropertyOrder({"name", "personId"})  publicclassPropertyOrderDemoBean {  publiclongpersonId=123L;  publicStringname="James Clark";  ----  }  @JsonRootName(value="user")  publicclassRootNameDemoBean {  publiclongpersonId=0;  publicStringname="James Clark";  ----  }  publicclassGetterDemoBean {  publiclongpersonId=123L;  publicStringpersonName="James Clark";  @JsonGetter(value="person-id")  publiclonggetPersonId() {  returnpersonId;  }  @JsonGetter(value="person-name")  publicStringgetPersonName() {  returnpersonName;  }  } |

Good spring boot with rabbit mq:

<https://grokonez.com/spring-framework/spring-amqp/springboot-rabbitmq-topic-exchange>

<https://springbootdev.com/category/docker/>

<https://www.tutlane.com/tutorial/rabbitmq/rabbitmq-exchanges><http://rubymarchhare.info/articles/exchanges.html>

<https://dzone.com/articles/all-you-need-to-know-about-asyncronous-messaging-u>

<https://dzone.com/articles/rabbitmq-and-spring-boot-integration-with-fault-to>

<https://dzone.com/articles/getting-started-with-nodejs-and-apache-ignite>

<https://thepracticaldeveloper.com/category/docker/>

<https://thepracticaldeveloper.com/2018/11/24/spring-boot-kafka-config/>

docker

<https://thepracticaldeveloper.com/2018/07/31/java-and-json-jackson-serialization-with-objectmapper/>

exception and dead later queue

<https://fatihdirlikli.wordpress.com/2016/06/22/spring-rabbit-dead-letter-queue-configuration/>

<https://zoltanaltfatter.com/2016/09/06/dead-letter-queue-configuration-rabbitmq/>

<https://fatihdirlikli.wordpress.com/2015/10/02/spring-boot-spring-cloud-aws-and-elasticbeanstalk-part-2-deployment/>