

**Sending and receiving JSON messages with Spring Boot AMQP and RabbitMQ**

BY MOISÉS MACERO ON OCTOBER 23, 2016

**Exchange JSON messages with Spring Boot AMQP and RabbitMQ**

Setting up a Spring Boot application using AMQP with RabbitMQ is not really difficult, and there are many guides on the Internet showing you the basic setup. But creating an application making use of @RabbitListener annotations and producing and consuming messages in JSON format is trickier, so I would like to share with you a really simple but more serious approach that those hello-messaging apps.

This article covers:

* How to send/publish Java Objects as JSON messages using Spring Boot and RabbitMQ's RabbitTemplate.
* How to read/consume JSON messages to Java Objects using Spring Boot and RabbitMQ's @RabbitListener annotation.
* How to send and receive Java Objects through RabbitMQ using default Java serializer.

**Updated:** The code examples use Java 10 and Spring Boot 2. [More info](https://thepracticaldeveloper.com/code-examples/).

**Video**

Do you prefer the Video version? It contains not only the explanation about how to set up the configuration to send and receive JSON messages but also some extra details about how the Message Converters work. The complete video is 29min, but here you have the shortcuts to the different sections:

* 👉 Part 1 [Sending and Receiving Messages using AMQP with default Java format](https://www.youtube.com/watch?v=yuApTF_HeWM)
* 👉 Part 2 [Running the Spring Boot AMQP application](https://www.youtube.com/watch?v=yuApTF_HeWM&t=480s)
* 👉 Part 3 [Spring Boot AMQP Message Converters in detail](https://www.youtube.com/watch?v=yuApTF_HeWM&t=681s)
* 👉 Part 4 [Using JSON Converters to serialize messages](https://www.youtube.com/watch?v=yuApTF_HeWM&t=1101s)

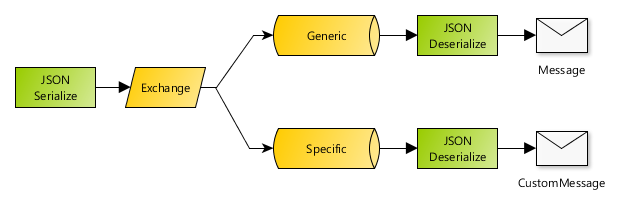
**Introduction**

As you probably know if you reached this article, [AMQP](https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol) is a specification, a protocol that defines messaging and patterns for communication between systems. [RabbitMQ](https://www.rabbitmq.com/)is a software that implements AMQP. You can find a lot of tutorials about these concepts on the Internet, apart from many useful books.

**Goal**

Within this article we’ll set up a Spring Boot application that uses RabbitMQ to send messages to a topic exchange formatted as JSON, and read them using two different queues:

* The first Queue will read the contents as a generic org.springframework.amqp.core.Message class.
* The second Queue will transform the JSON message to a specific class created by us.

[](https://thepracticaldeveloper.com/images/posts/uploads/2016/10/messaging-twoqueues.png)Queues and process view

I’ll also show you an alternative configuration to make this work without formatting the message to JSON, but nowadays this format is a good choice for intercommunication between applications and systems that may differ in programming languages, for instance.

I recommend you to check the implementation as you follow these instructions. You can do so by cloning the GitHub repository. Don’t forget to give it a star if you find it useful!

You can find all the source code on GitHub: [Spring Boot JSON AMQP messaging](https://github.com/mechero/spring-boot-amqp-messaging). If you like it, give it a star!

**Setting up the project**

**pom.xml**

Creating a simple Spring Boot application is pretty straightforward, in my case I use Spring Boot 2.0 and Maven. The only thing you need to do to make use of AMQP with RabbitMQ in Spring Boot is to include the corresponding starter dependency (AMQP) and the Jackson libraries (to work with JSON):

Maven dependencies

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-core</artifactId>

<version>2.9.6</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-annotations</artifactId>

<version>2.9.6</version>

</dependency>

<dependency>

<groupId>com.fasterxml.jackson.core</groupId>

<artifactId>jackson-databind</artifactId>

<version>2.9.6</version>

</dependency>

</dependencies>

**The message**

In this case, we’re going to send a really simple content, but enough to have an idea about how the message looks like in different formats. The class is called CustomMessage.

CustomMessage class

package com.thepracticaldeveloper.rabbitmqconfig;

import com.fasterxml.jackson.annotation.JsonProperty;

import java.io.Serializable;

public final class CustomMessage implements Serializable {

private final String text;

private final int priority;

private final boolean secret;

public CustomMessage(@JsonProperty("text") String text,

@JsonProperty("priority") int priority,

@JsonProperty("secret") boolean secret) {

this.text = text;

this.priority = priority;

this.secret = secret;

}

public String getText() {

return text;

}

public int getPriority() {

return priority;

}

public boolean isSecret() {

return secret;

}

@Override

public String toString() {

return "CustomMessage{" +

"text='" + text + ''' +

", priority=" + priority +

", secret=" + secret +

'}';

}

}

[](https://leanpub.com/practical-software-architecture)

Pretty basic POJO. There are only two parts to point out:

1. The constructor uses @JsonProperty annotations, which we need for JSON deserialization. This is an alternative to using an empty constructor and non-final fields. There is also a way to infer this from the variable names, but this is good for the example.
2. The toString() method, that will give us a human-readable representation of the objects.

**Sending messages**

As mentioned before, this example will use a topic exchange to send CustomMessage objects.

package com.thepracticaldeveloper.rabbitmqconfig;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.amqp.rabbit.core.RabbitTemplate;

import org.springframework.scheduling.annotation.Scheduled;

import org.springframework.stereotype.Service;

import java.util.Random;

@Service

public class CustomMessageSender {

private static final Logger log = LoggerFactory.getLogger(CustomMessageSender.class);

private final RabbitTemplate rabbitTemplate;

public CustomMessageSender(final RabbitTemplate rabbitTemplate) {

this.rabbitTemplate = rabbitTemplate;

}

@Scheduled(fixedDelay = 3000L)

public void sendMessage() {

final var message = new CustomMessage("Hello there!", new Random().nextInt(50), false);

log.info("Sending message...");

rabbitTemplate.convertAndSend(MessagingApplication.EXCHANGE\_NAME, MessagingApplication.ROUTING\_KEY, message);

}

}

Some comments about this service:

* @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](https://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.
* The exchange and the routing key are AMQP concepts that I recommend you to read about that if you still don't know what they do. As a summary, we are sending messages through a given channel (exchange), and the routing key allows us to filter for who are those generated. This sample application is not intended for showing advanced exchange / queue configuration so you will see that this is a basic setup.

**Receiving messages**

We need to build the other side of the communication too. For this post, I’ve created the receivers in the same application for better logistics (having a single repo), but you shouldn’t have problems to replicate this configuration with two different apps if you want. Actually, you could run this Spring Boot application twice and remove the producer in one instance, the consumers in another one, to make it more like a ‘real case’.

package com.thepracticaldeveloper.rabbitmqconfig;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.amqp.core.Message;

import org.springframework.amqp.rabbit.annotation.RabbitListener;

import org.springframework.stereotype.Service;

@Service

public class CustomMessageListener {

private static final Logger log = LoggerFactory.getLogger(CustomMessageListener.class);

@RabbitListener(queues = MessagingApplication.QUEUE\_GENERIC\_NAME)

public void receiveMessage(final Message message) {

log.info("Received message as generic: {}", message.toString());

}

@RabbitListener(queues = MessagingApplication.QUEUE\_SPECIFIC\_NAME)

public void receiveMessage(final CustomMessage customMessage) {

log.info("Received message as specific class: {}", customMessage.toString());

}

}

Here you can see our goal: the idea is to show at the same time how the same message, being sent by the CustomMessageSender class, can be received through two different queues. First queue (GENERIC) is receiving the message as a generic Message class, so there is no implicit conversion. Spoiler alert: when we log the message we should see something like a JSON string in the payload since it has not been converted.

On the other hand, the queue SPECIFIC is expecting a CustomMessage class (the method argument). That will trigger a logic inside Spring to find a converter from JSON to that specific class. **This part should be easy to configure, but sometimes is the part in which we the developers struggle most**. In this case is easy to guess that the text logged should be the CustomMessage’s implementation of toString().

**Configuration**

This is the most interesting part since it’s what is making everything work together. First, you will find some annotations in the class:

@SpringBootApplication

@EnableScheduling

public class MessagingApplication {

Apart from the one that makes Spring Boot works, @EnableScheduling needs to be added to support the @Scheduled method for sending messages that we covered before in CustomMessageSender class. You don’t need to add the @EnableRabbit annotation since it will be enabled automatically just because you added the dependency (part of the Spring Boot’s magic).

Then you’ll find some constants and, next to them, the Exchange / Queues configuration:

@Bean

public TopicExchange appExchange() {

return new TopicExchange(EXCHANGE\_NAME);

}

@Bean

public Queue appQueueGeneric() {

return new Queue(QUEUE\_GENERIC\_NAME);

}

@Bean

public Queue appQueueSpecific() {

return new Queue(QUEUE\_SPECIFIC\_NAME);

}

@Bean

public Binding declareBindingGeneric() {

return BindingBuilder.bind(appQueueGeneric()).to(appExchange()).with(ROUTING\_KEY);

}

@Bean

public Binding declareBindingSpecific() {

return BindingBuilder.bind(appQueueSpecific()).to(appExchange()).with(ROUTING\_KEY);

}

These are the topic exchange, both queues and the bindings between them (both queues are bound to the same exchange with the same routing key). Just a sample configuration.

At the bottom, in the same class, we can find the configuration related to **producing and consuming JSON messages**:

@Bean

public RabbitTemplate rabbitTemplate(final ConnectionFactory connectionFactory) {

final var rabbitTemplate = new RabbitTemplate(connectionFactory);

rabbitTemplate.setMessageConverter(producerJackson2MessageConverter());

return rabbitTemplate;

}

@Bean

public Jackson2JsonMessageConverter producerJackson2MessageConverter() {

return new Jackson2JsonMessageConverter();

}

[](https://leanpub.com/practical-software-architecture)

The only thing we’re doing here is overriding the default RabbitTemplate created by Spring Boot with another one that will use a message converter of type Jackson2JsonMessageConverter. **Remember**: this configuration only affects to messages**sent** via RabbitTemplate.

The bean Jackson2JsonMessageConverter will also take care of deserializing the JSON messages to Java classes, using a default ObjectMapper. Note that, with this configuration, you don’t need to implement the interface RabbitListenerConfigurer nor use a MappingJackson2MessageConverter as it was suggested in a previous version of this post.

**Running the application**

First, you need to make sure that the RabbitMQ server is running. In the [repository](https://github.com/mechero/spring-boot-amqp-messaging/blob/master/docker-compose.yml), there is a docker-compose.yml file that you can use if you have Docker installed. Just run:

docker-compose up -d

Also, if you want to use a configuration which is different from the default one, make sure you include those changes within the application.properties file (port number, authentication, etc.). You don’t need to do this if you use the default settings as they are set in this sample codebase (unless your docker host IP is not your own localhost).

Then you can run the application class from your IDE or executing mvn spring-boot:run from the command line. When you do that, you should see these lines in the console:

INFO 13100 --- [pool-4-thread-1] c.t.rabbitmqconfig.CustomMessageSender : Sending message...

INFO 13100 --- [cTaskExecutor-1] c.t.r.CustomMessageListener : Received message as specific class: CustomMessage{text='Hello there!', priority=40, secret=false}

INFO 13100 --- [cTaskExecutor-1] c.t.r.CustomMessageListener : Received message as generic: (Body:'{"text":"Hello there!","priority":40,"secret":false}' MessageProperties [headers={\_\_TypeId\_\_=com.thepracticaldeveloper.rabbitmqconfig.CustomMessage}, timestamp=null, messageId=null, userId=null, receivedUserId=null, appId=null, clusterId=null, type=null, correlationId=null, correlationIdString=null, replyTo=null, contentType=application/json, contentEncoding=UTF-8, contentLength=0, deliveryMode=null, receivedDeliveryMode=PERSISTENT, expiration=null, priority=0, redelivered=false, receivedExchange=appExchange, receivedRoutingKey=messages.key, receivedDelay=null, deliveryTag=3, messageCount=0, consumerTag=amq.ctag-cZstJRR8omlFYGJDJ0zvcA, consumerQueue=appGenericQueue])

Working as expected! Our listener for CustomMessage is showing us our implementation of toString() method. The last line though, the method listening for the Message class, is showing us the raw contents, which turns out to be -*as we wanted*- JSON format.

**Changing from JSON format to default**

You can comment the lines that set up the message converters (they are indicated in the source code). What you will see then is the default configuration for Spring Boot, which is using a Java serialization to encode messages.

INFO 6660 --- [pool-4-thread-1] c.t.rabbitmqconfig.CustomMessageSender : Sending message...

INFO 6660 --- [cTaskExecutor-1] c.t.r.CustomMessageListener : Received message as generic: (Body:'[B@56f321e7(byte[143])' MessageProperties [headers={}, timestamp=null, messageId=null, userId=null, receivedUserId=null, appId=null, clusterId=null, type=null, correlationId=null, correlationIdString=null, replyTo=null, contentType=application/x-java-serialized-object, contentEncoding=null, contentLength=0, deliveryMode=null, receivedDeliveryMode=PERSISTENT, expiration=null, priority=0, redelivered=false, receivedExchange=appExchange, receivedRoutingKey=messages.key, receivedDelay=null, deliveryTag=3, messageCount=0, consumerTag=amq.ctag-ovFVyrVodTF4fWW7\_PELaQ, consumerQueue=appGenericQueue])

INFO 6660 --- [cTaskExecutor-1] c.t.r.CustomMessageListener : Received message as specific class: CustomMessage{text='Hello there!', priority=45, secret=false}

As you can see in the headers of the message, now the format is contentType=application/x-java-serialized-object. This is just another approach, which could turn out to be not the best one if you are using, for instance, different kind of applications (non-Java) for receiving messages.

**More info**