RabbitMQ + Springboot tutorial

RabbitMQ is a lightweight, open-source, and easy-to-deploy message broker that enables services and applications to communicate with each other. It supports multiple messaging protocols like AMQP (Advanced Message Queuing Protocol), and MQTT (MQ Telemetry Transport). STOMP (Simple Text Orientated Messaging Protocol) and a few more. In this tutorial, we will explore how to set it up locally and how to use it in order to communicate between 2 SpringBoot 3.2 applications.

RabbitMQ features:

Unlike the standard JMS (Java Message Service), RabbitMQ does not simply provide a messaging queue. Instead, a producer will put each of their messages into a specific exchange and, based on the internal configuration, the exchange will route each message to 1 or more queues (in which case it will create duplicates). A queue is a simple FIFO container, to which 1 or more consumers can connect. Each message inside a queue will only be consumed once, while a message inside an exchange can be sent to multiple queues, in parallel.
In RabbitMQ each message, besides its actual content, contains its key, a string value based on which the exchange can perform routing operations. There are several types of exchanges, such as:

- Direct exchange: it uses the message routing key to transport messages to queues. For each incoming message, the direct exchange will route it to the queue which exactly matches the key.
- Topic exchange: unlike the direct exchange, each incoming message will be routed by the topic exchange to all the queues which match its key. In this case, the routing key of each queue can contain wildcard characters, such as * (a word) or # (zero or more words). As such, if the topic exchange receives a message with the key “soa.matei.tutorial”, it will route it to both a queue with the pattern “soa.#” and one with the pattern “soa.*.tutorial”, but not to one with the pattern “tutorial”.
- Fanout exchange: it duplicates and routes a received message to any associated queues, regardless of routing keys or pattern matching. Here, the provided keys will be entirely ignored.
- Headers exchange: it uses arguments with headers and optional values to route messages. Header exchanges are identical to topic exchanges, except that instead of using routing keys, messages are routed based on header values. If the value of the header equals the value of supply during binding, the message matches.
- Default exchange: it is an unnamed pre-declared direct exchange. An empty string is frequently used to indicate it. The messages will be delivered to a queue with the same name as the routing key.
- Dead letter exchange: all messages that can not be send to any queue end up in here, for later reprocessing.

Local installation:

The easiest way to run RabbitMQ locally is through Docker. The docker compose yaml definition is the following:

```yaml
    version: '3.8'
    services:
```

```
    rabbitmq:
      image: rabbitmq:management
      ports:
        - "5672:5672"
        - "15672:15672"
```

Example with SpringBoot 3.2

To illustrate its usage, a demo project consisting of 2 SpringBoot applications will be used. Java 21 is used, together with Maven. The common maven configuration is extracted into a parent pom file, while the individual configurations are specified for each of the 2 projects (the org.springframework.boot:spring-boot-starter-amqp dependency contains all the required dependencies in order to produce messages and listen to a RabbitMQ server). The general flow is this: the producer receives a message through a rest call, and sends it to a RabbitMQ topic exchange. The other application listens to a queue, and will print any incoming messages. All components are defined using Spring Beans, so they could be used in any context. Since the RabbitMQ application is configured to run on the default ports on localhost, no additional connectivity configurations are required. If the broker was running in a non-default configuration, the connection can be configured using spring properties such as spring.rabbitmq.host, spring.rabbitmq.password, spring.rabbitmq.username, spring.rabbitmq.port.

In addition to that, Lombok is used in the example applications in order to log operations and to perform autowiring from the constructor of each component.

Producer application:

This application does not require the definition of any additional beans. The Spring Context provides a RabbitTemplate bean, which can be autowired into any other component. The rabbitTemplate has a convertAndSend method, which can be used to send a message to any kind of exchange (identified by its name), together with a message routing key and the message contents. The message content is a primitive array of bytes, using the Message class from org.springframework.amqp.core as a wrapper. However, the convertAndSend method has an overload which accepts any object and will perform the conversion to a AMQP Message as long as that object is either an array of bytes, a string, or implements the Serializable interface. However, while more efficient, Java native serialization and deserialization poses security risks. As such, it is easier to send and receive simple strings, and to do any kind of conversions to Java objects using Jackson’s ObjectMapper or similar libraries.

Consumer application:

The consumer application will both define methods for receiving RabbitMQ messages as well as beans for the internal exchange routing configuration.

Configuration:
For the consumer, several additional beans need to be declared, which also have the purpose of performing the internal routing configuration for the used RabbitMQ exchanges. For the purpose of this tutorial a topic exchange will be used. First of all, the queue and the

exchange need to be defined as beans (besides the TopicExchange class, DirectExchange, FanoutExchange, etc. are provided.). Using the defined beans, a Binding bean is created, in which the internal routing rules for rabbitmq will be defined.

Consumption:
In order to get the message into the application, a simple spring component is defined, called Receiver. It has a single method, with a single String parameter. Similar to the producer, instead of String the input parameter type could be a Message, an array of bytes or any POJO which implements the serializable interface. In order to integrate our Receiver with RabbitMQ, a MessageListenerAdapter bean is defined. It takes in the Receiver component and creates a new MessageListenerAdapter object, using the reference to the receiver bean and the name of the method through which we expect to receive messages, which will be invoked whenever a message is received using reflection. Finally, the container bean is defined, which makes the connection between the queue (identified by its name) and the previously defined MessageListenerAdapter. This approach uses generic components, and when using it, RabbitMQ could be replaced with any other message service provider which implements the AMQP interface simply from the configurations.
Alternatively, for a more simple approach, the @RabbitListener annotation can be used for any method defined in a spring bean, with a String or Serializable parameter. The actual implementation is similar to the previous configuration, just without the need to declare the beans in code. With this annotation, we can specify any number of queues which will be handled by the method.
Within a single SpringBoot application, multiple exchanges, queues, bindings, and receivers can be defined, in order to listen to a multitude of RabbitMQ messages.