***Message Brokers for Microservices-RabbitMQ tutorial***

RabbitMQ is a message broker that implements Advanced Message Queuing Protocol ([AMQP](https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol)). It provides client libraries for major programming languages. In this tutorial, I will present how RabbitMQ can be utilized to communicate between 2 Java microservices. The first microservice is Book and the second one is Review. These microservices are connected to an API Gateway that communicates with a frontend application in Angular. Each time the API that gets book information based on a title is called, the Review microservice gets the information about the book, which will be queued by the Book microservice, and shows it.

The system works like a producer-consumer model. In this case, the Book microservice will act as the producer, who will send the messages to the broker, and the Review microservice will be the consumer, which will receive the messages from the broker. RabbitMQ will act as a communication middleware between the microservices.

First, RabbitMQ needs to be installed. To do this, first we must install Erlang from <https://www.erlang.org/downloads> and RabbitMQ from <https://www.rabbitmq.com/download.html>. After that, we can open the following

A purple rectangle with white text

Description automatically generated

and run the command rabbitmq-plugins enable rabbitmq\_management, in order to enable a management plugin. After that, if we go in the browser and type <http://localhost:15672> (which are the default host and port for rabbitmq, but can be changed), we can see information about the queues and messages. The username and password are guest and guest.

After that, the following dependency needs to be added in both microservices, in pom.xml:

<dependency>  
 <groupId>com.rabbitmq</groupId>  
 <artifactId>amqp-client</artifactId>  
 <version>4.0.0</version>  
</dependency>

Then, the producer will have to start publishing messages to the queue. The code will look similar to this:

A screen shot of a computer code

Description automatically generated

First, we need to connect to the RabbitMQ broker using java client. This is achieved with ConnectionFactory. We can set the host, set to localhost here, and also the port, which is 15672 by default. Also, we need to create a channel.

Then, we will have to declare the queue. In this case, we named the queue “book\_queue”. We can also see it in the management plugin.

A white background with black text

Description automatically generated

Then, each time the API is called and a book is returned, the information about the book will be published to the queue. If the other microservice which is the consumer is not running, then we can see in the management plugin that the messaged is queued.

A screen shot of a computer

Description automatically generated

The following will be printed in the console by the producer:

In the end, the connection and the channel need to be closed.

Next, the consumer will feature code similar to this:

A screen shot of a computer screen

Description automatically generated

A new connection also has to be established, similar to how it was established in the producer. Then, the consumer is declared in order for it receive the message from the queue, which has to have the same name as the one defined in the producer. As soon the microservice is running, in the management plugin the message will no longer be ready.

A graph with a red line

Description automatically generated

And also, it will be delivered(green, yellow represented the message when it was only published and ready to be delivered, waiting for the consumer.

A screenshot of a computer

Description automatically generated

In the consumer, the following message will be printed, which is the same as the one the producer published.



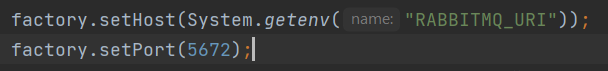
If we want to use RabbitMQ with Docker, a few changes should be made. First, in the docker-compose.yml file, the following should be added:

A screenshot of a computer code

Description automatically generated

A white background with black text

Description automatically generated And, for the services that will communicate between each other, book and review, the following:

 Then, in both the producer and consumer, the host and the port should be set to work with docker.

The name of the environment from docker-compose.yml should correspond with the name passed to the host in the Java code. Now, the communication between the producer and consumer should be visible from the terminal, and from the management plugin like before.

Dumistracel Sabina-258/1