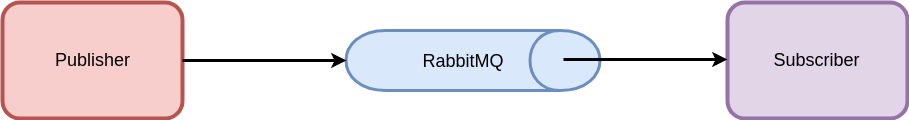
**Microservice Communication using Spring Cloud Stream and RabbitMQ**

**Communication between microservices is the backbone of distributed systems**. Usually, developers try to avoid it altogether fearing an increase in complexity. However, a combination of **Spring Cloud Stream and RabbitMQ** can make it*relatively easy to handle communication between microservices*.

**Spring Cloud Stream** is also part of the [Spring Cloud](http://progressivecoder.com/spring-cloud-microservices-a-detailed-guide/) group of projects. It provides easy integration with various message brokers with minimum configuration. Below is a **high-level view** on the overall pattern.



*Spring Cloud Stream helps in exchanging messages between two applications or microservices. It can work seamlessly with message brokers such as RabbitMQ, Kafka and so on.*

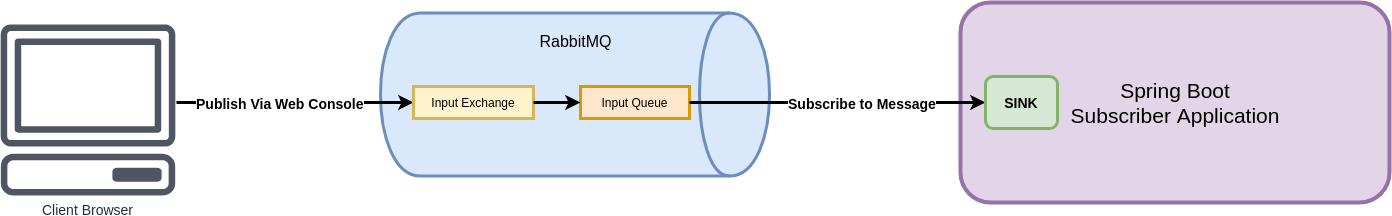
In this post, we will implement **communication between microservices** using **Spring Boot and Spring Cloud Stream**. Below is our high-level plan.

* **Step 1 –** Create a **RabbitMQ Server**using Docker
* **Step 2 –** Create a Spring Boot application using Spring Cloud Stream to **listen to messages**. We call this application as **subscriber-application**.
* **Step 3 –** Publish messages on RabbitMQ so that our **subscriber-application** can listen to those messages.
* **Step 4** – Create another Spring Boot application using Spring Cloud Stream to **publish messages** to Rabbit MQ. We call this application **publisher-application**.
* **Step 5** – Publish Messages using **publisher-application** that will be consumed by the **subscriber-application**.

**Step 2 – Create the Subscriber Application**

In this step, we can create an application using **Spring Boot and Spring Cloud Stream** to listen to messages. If you are not aware of Spring Boot, I have a detailed guide on [Spring Boot Microservices](http://progressivecoder.com/spring-boot-microservices-fastest-production-ready-microservices/).

Below is our basic plan at this point.



Basically, we will be first building our **subscriber-application** that connects to the RabbitMQ server. We will publish the messages directly through the **RabbitMQ web console**. And our Spring Boot application will have something known as **Sink** that will help us process the incoming messages. We will look at what a **Sink** is in the next section.

Important things to note here are as follows:

* We have enabled **Sink** binding by using *@EnableBinding* annotation. This step signals the underlying framework to create the necessary bindings to the messaging middleware. In other words, it will create the destination items such as queue, topic etc.
* Also, we have added a **handler** method. This method is used to receiving incoming messages of type **Message**. This is one of the most powerful features of **Spring Cloud Stream**. The framework tries to automatically convert incoming messages to the type **Message**.

**Step 3 – Publish Message through RabbitMQ**

We are ready to test our subscriber application and see it in action.

You can simply start the application using the below command:

clean package spring-boot:run

The application will automatically try to connect to a RabbitMQ server at *http://localhost:5672*. You should see something like this in the application startup logs.

2019-06-25 17:35:22.304 INFO 8115 --- [ main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#34688e58:0/SimpleConnection@1a981cf0 [delegate=amqp://guest@127.0.0.1:5672/, localPort= 52000]

2019-06-25 17:35:22.355 INFO 8115 --- [ main] o.s.i.monitor.IntegrationMBeanExporter : Registering MessageChannel input.anonymous.KWrQWmqZSYKPjOBkx4DotA.errors

2019-06-25 17:35:22.432 INFO 8115 --- [ main] o.s.c.stream.binder.BinderErrorChannel : Channel 'application.input.anonymous.KWrQWmqZSYKPjOBkx4DotA.errors' has 1 subscriber(s).

2019-06-25 17:35:22.432 INFO 8115 --- [ main] o.s.c.stream.binder.BinderErrorChannel : Channel 'application.input.anonymous.KWrQWmqZSYKPjOBkx4DotA.errors' has 2 subscriber(s).

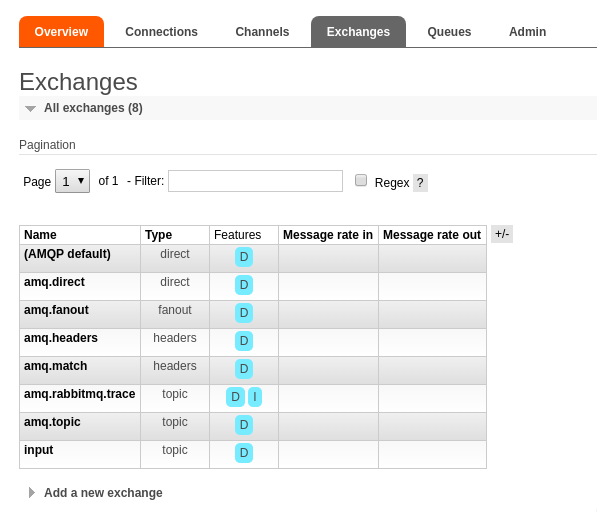
2019-06-25 17:35:22.454 INFO 8115 --- [ main] o.s.i.a.i.AmqpInboundChannelAdapter : started inbound.input.anonymous.KWrQWmqZSYKPjOBkx4DotA

2019-06-25 17:35:22.464 INFO 8115 --- [ main] c.p.d.s.SubscriberApplication : Started SubscriberApplication in 3.52 seconds (JVM running for 18.305)

To test whether our application is able to listen to messages, we can publish a message through the **RabbitMQ management console**.

To do so, you can login to the console at [*http://localhost:15672*](http://localhost:15672) using default *userid/password* as *guest/guest.*

At this point, you should see the list of exchanges as below. Note the last exchange in the list known as *input*. This is basically created automatically when we started our application.

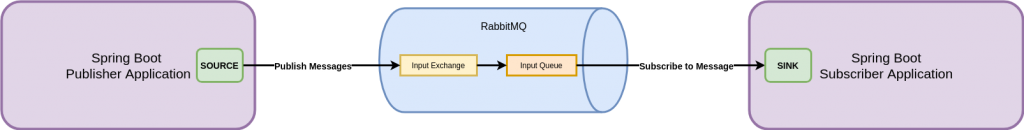


Also, a queue under the *input* exchange is created where we can publish messages.

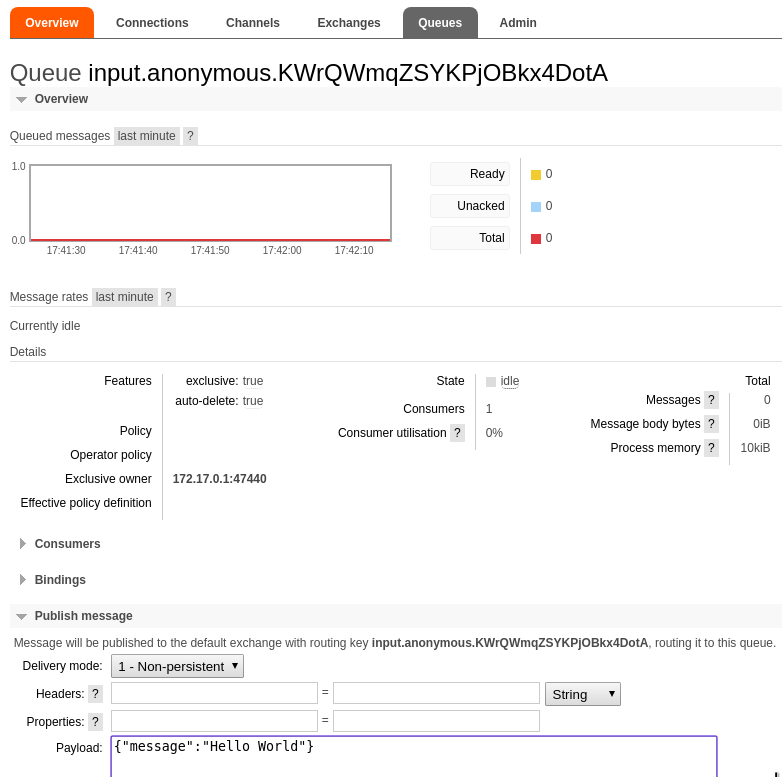
**Step 4 – Create the Publisher Application**

In this step, we will create a publisher application that has an end-point to publish messages to the **RabbitMQ** server.

Below is our high-level plan for the same.



As you can see, now we have a **publisher-application** instead of the web console. In the publisher application, we have something known as **Source**. Think of it as an opposite of the **Sink** interface we saw earlier in the **subscriber-application**. We will look at the **Source** interface in detail in this section.



Once you click **Publish**, you will see the message printed in the *subscriber-application* logs as below.

2019-06-25 17:35:22.454 INFO 8115 --- [ main] o.s.i.a.i.AmqpInboundChannelAdapter : started inbound.input.anonymous.KWrQWmqZSYKPjOBkx4DotA

2019-06-25 17:35:22.464 INFO 8115 --- [ main] c.p.d.s.SubscriberApplication : Started SubscriberApplication in 3.52 seconds (JVM running for 18.305)

Received Message is: Message{message='Hello World'}

With this, we are done with the subscriber part of our application. Now we can move onto the next step.

The important things to note here are:

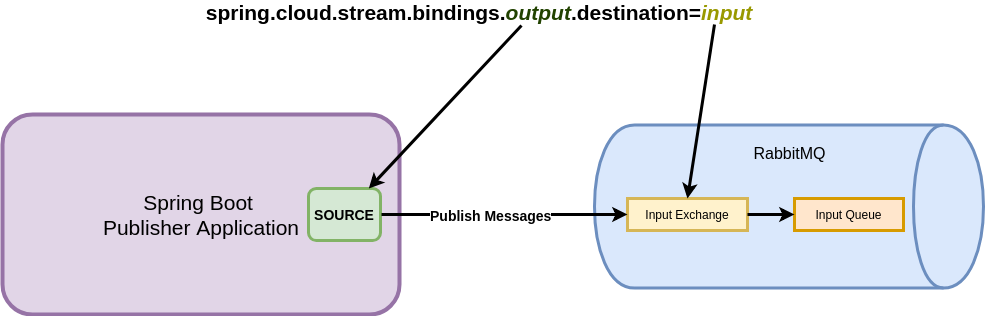
* We create a **MessagePublisher** class and annotate it as *@RestController*.
* Also, we annotate the controller class with **@EnableBinding**. However, instead of binding to **Sink** (as we did in the subscriber), we bind this class with the **Source.class**. Basically, Source and Sink are *binding interfaces* provided by Spring Cloud Stream.
* We auto-wire an instance of the **Source** class and in the */api/publish* call we use it to publish a Message object to RabbitMQ.
* We also define the **Message** class to create a new message.

There is another important setting you need to do in the *application.properties* file. We need to define the output binding as below.

spring.cloud.stream.bindings.output.destination=input

spring.cloud.stream.default.contentType=application/json

The below illustration can help understand what we mean by this property.



Basically, the **Source** interface exposes an **Output** channel that we are binding to the ***input*** exchange. This relation is required for the publisher and the subscriber to be able to exchange messages.

**Step 5 – Publish Message using publisher-application**

Now we can start-up the *publisher-application* as well. Once the application starts up, we need to trigger the endpoint [*http://localhost:8080/api/publish*](http://localhost:8080/api/publish).

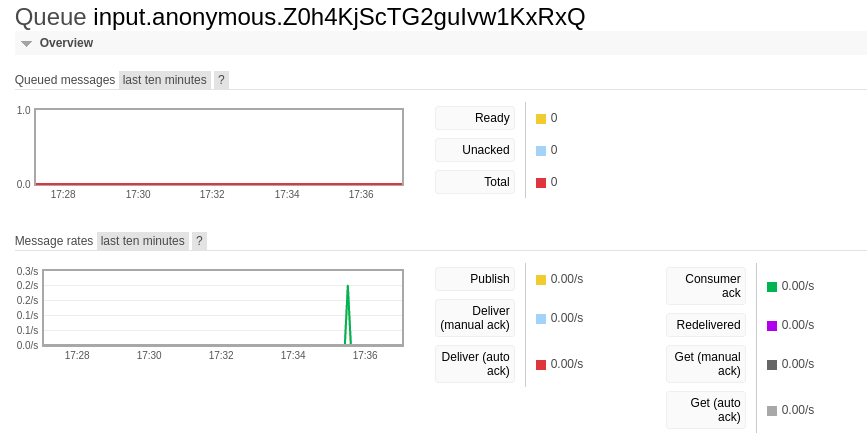
Nothing will be shown in the browser or the client as response. However, if you visit the logs of the **subscriber-application**, we will be able to see the message printed.

2019-06-26 17:18:50.366 INFO 5138 --- [ main] o.s.i.a.i.AmqpInboundChannelAdapter : started inbound.input.anonymous.Z0h4KjScTG2guIvw1KxRxQ

2019-06-26 17:18:50.374 INFO 5138 --- [ main] c.p.d.s.SubscriberApplication : Started SubscriberApplication in 2.939 seconds (JVM running for 11.335)

Received Message is: Message{message='Hello World from Publisher'}

Also, you will be able to see a green spike in the **RabbitMQ console** in the input queue. This shows the transmission of the message to the input queue to which the **subscriber-application** is listening.



**Conclusion**

With this, we have successfully developed a small application using **Spring Cloud Stream and RabbitMQ to publish and subscribe to messages**.