# Lab 11

## Part A:

For this exercise we will use ActiveMQ as JMS middleware so we first have to start ActiveMQ.

The starter files is in the directory f **C:\ CS544\Tools\activemq\bin\startactivemq.** You can find a shortcut of this file in the Desktop inside of the folder called **"Tool Shortcuts".**

Open the given **Lab11Sender** project and the **Lab11Receiver** project.

First run the SpringJmsReceiverApplication.java in the **Lab11Receiver** project.
Then run the SpringJmsPersonSenderApplication in the **Lab11Sender** project.

You should now see the following in the sender console:

```
Sending a JMS message.
```

And you should see the following in the receiver console:

```
Receiver has started ...
Received message:Frank Brown
```

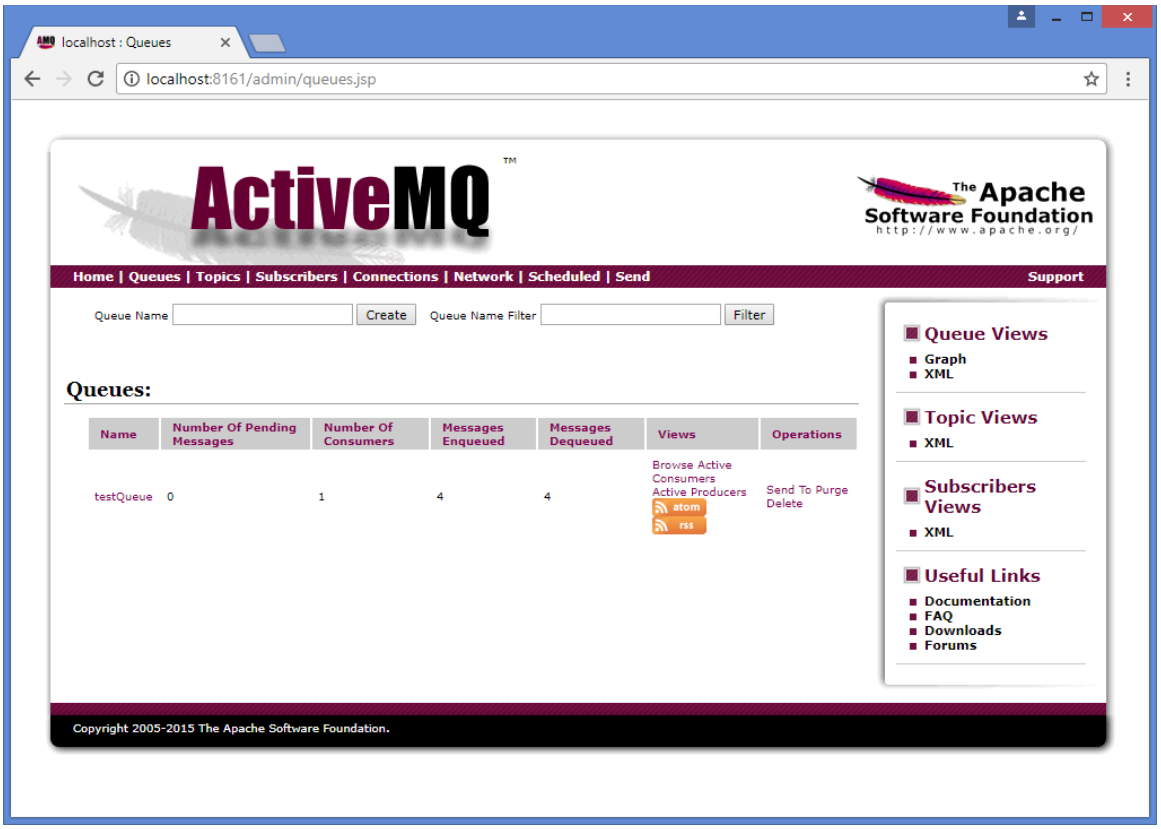
Now run the sender a few times.

```
Receiver has started ...
Received message:Frank Brown
Received message:Frank Brown
Received message:Frank Brown
```

Then open the ActiveMQ console at http://localhost:8161/admin.

You can login with username **admin** and password **admin**

Select the Queues page from the menu:




**You see that the queue with name** testqueue **has one consumer, and 4 message have been received and consumed.**

Now write a JMS calculator application where the JMS receiver implements a calculator that receives commands by means of JMS messages. So the sender sends an object containing an operator (+,-,*) and an value (integer). The Receiver receives the message and does the requested calculation. Example: The calculator starts with value =0. The sender sends + and 7. The receiver prints out that the result is 7. The sender sends + and 8. The receiver prints out that the result is 15.

Have the sender output the commands it is about to send, and have the receiver output the resulting calculations.

## Part B. –JMS for the Bank Application

The goal in this exercise is to implement the actual sending of JMS messages in the bank application where we previously only had JMSSender that did a System.out.println().

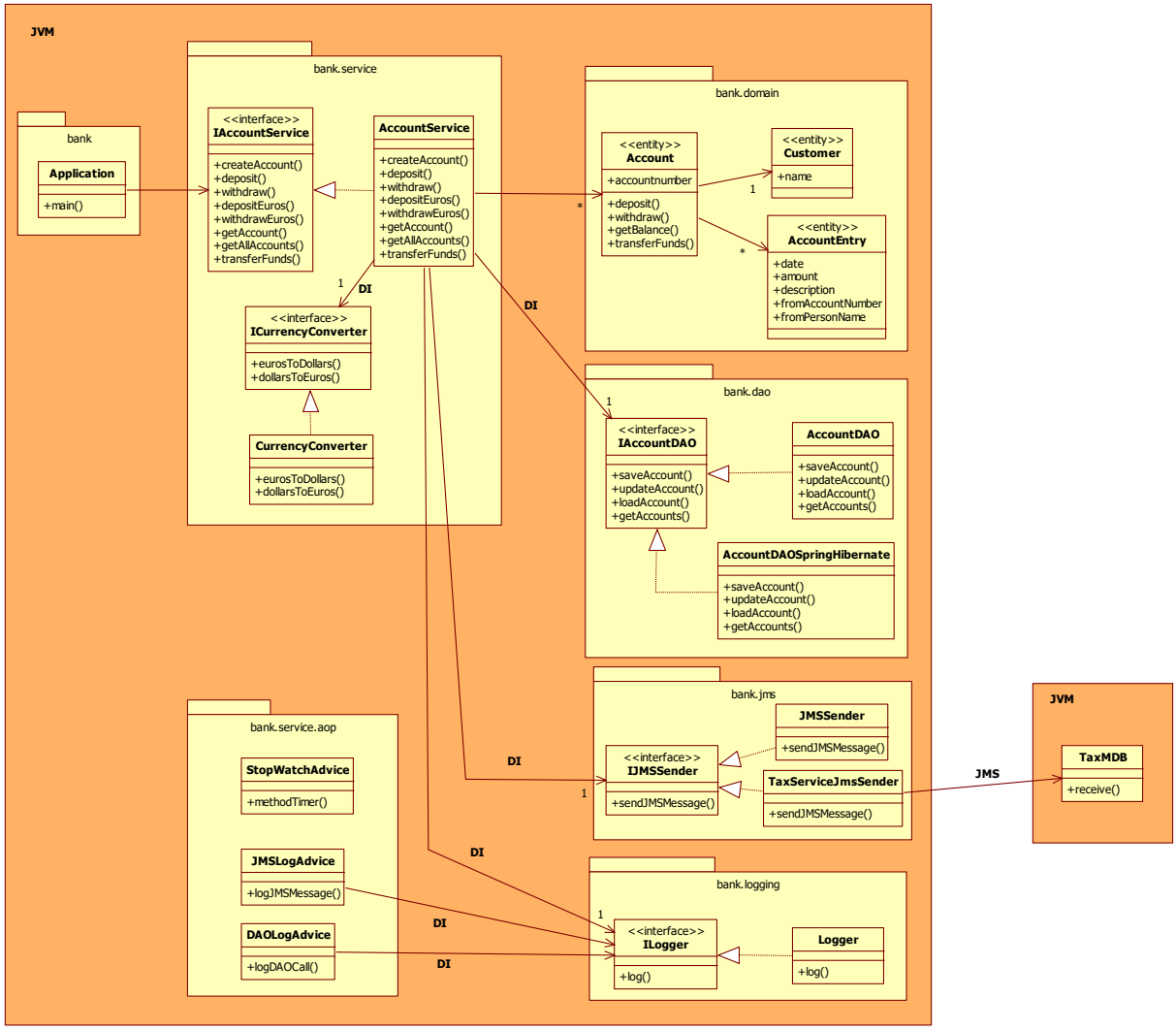
Create a new Spring Boot application with the name TaxService. This TaxService can receive JMS messages and outputs the contents of the message to the console.

Then change the bank application in such a way that whenever a deposit of 10,000 Euros or greater is made, an actual JMS message is sent to the to the TaxService application.

In order for both the sender as the receiver application to work, add the following dependencies to the POM file:

```
<dependency>
	<groupId>org.apache.activemq</groupId>
	<artifactId>activemq-broker</artifactId>
	<version>5.18.1</version>
	<exclusions>
		<exclusion>
			<groupId>org.apache.activemq</groupId>
			<artifactId>activemq-client</artifactId>
		</exclusion>
	</exclusions>
</dependency>
<dependency>
	<groupId>org.apache.activemq</groupId>
	<artifactId>activemq-client-jakarta</artifactId>
	<version>5.18.1</version>
</dependency>
<dependency>
	<groupId>com.fasterxml.jackson.core</groupId>
	<artifactId>jackson-databind</artifactId>
</dependency>
```

For both the Tax Application and Bank Application you need to add the JmsConfig class to have a proper configuration for JMS. Sender and Receiver JMS have different JmsConfig classes so be carefull with this. The Sender Service has a **@Bean** for **“JmsTemplate”**.


JVM
bank.service
bank
Application
+main()
<<interface>>
IAccountService
+createAccount()
+deposit()
+withdraw()
+depositEuros()
+withdrawEuros()
+getAccount()
+getAllAccounts()
+transferFunds()
AccountService
+createAccount()
+deposit()
+withdraw()
+depositEuros()
+withdrawEuros()
+getAccount()
+getAllAccounts()
+transferFunds()
1
DI
<<interface>>
ICurrencyConverter
+eurosToDollars()
+dollarsToEuros()
CurrencyConverter
+eurosToDollars()
+dollarsToEuros()
bank.domain
<<entity>>
Account
+accountnumber
+deposit()
+withdraw()
+getBalance()
+transferFunds()
*
1
<<entity>>
Customer
+name
*
<<entity>>
AccountEntry
+date
+amount
+description
+fromAccountNumber
+fromPersonName
DI
bank.dao
1
<<interface>>
IAccountDAO
+saveAccount()
+updateAccount()
+loadAccount()
+getAccounts()
AccountDAO
+saveAccount()
+updateAccount()
+loadAccount()
+getAccounts()
AccountDAOSpringHibernate
+saveAccount()
+updateAccount()
+loadAccount()
+getAccounts()
bank.jms
JMSSender
+sendJMSMessage()
<<interface>>
IJMSSender
+sendJMSMessage()
TaxServiceJmsSender
+sendJMSMessage()
1
DI
JMS
JVM
TaxMDB
+receive()
bank.service.aop
StopWatchAdvice
+methodTimer()
JMSLogAdvice
+logJMSMessage()
DAOLogAdvice
+logDAOCall()
DI
DI
DI
bank.logging
1
<<interface>>
ILogger
+log()
Logger
+log()

## Part C. Kafka

### First start Zookeeper:

First start Zookeeper by double clicking the file **C:\ CS544\Tools\kafka\startzookeeper**

### Then start Kafka:

Wait till you see logging data in the prompt window of zookeeper and then start Kafka by double clicking the file **C:\ CS544\Tools\kafka\\startkafka**

You can find shortcuts of these executable files in the Desktop inside of the folder called **"Tool Shortcuts".**

In IntelliJ open the projects **KafkaReceiver and KafkaSender**

Run KafkaReceiver

```
Receiver is running and waiting for messages
```

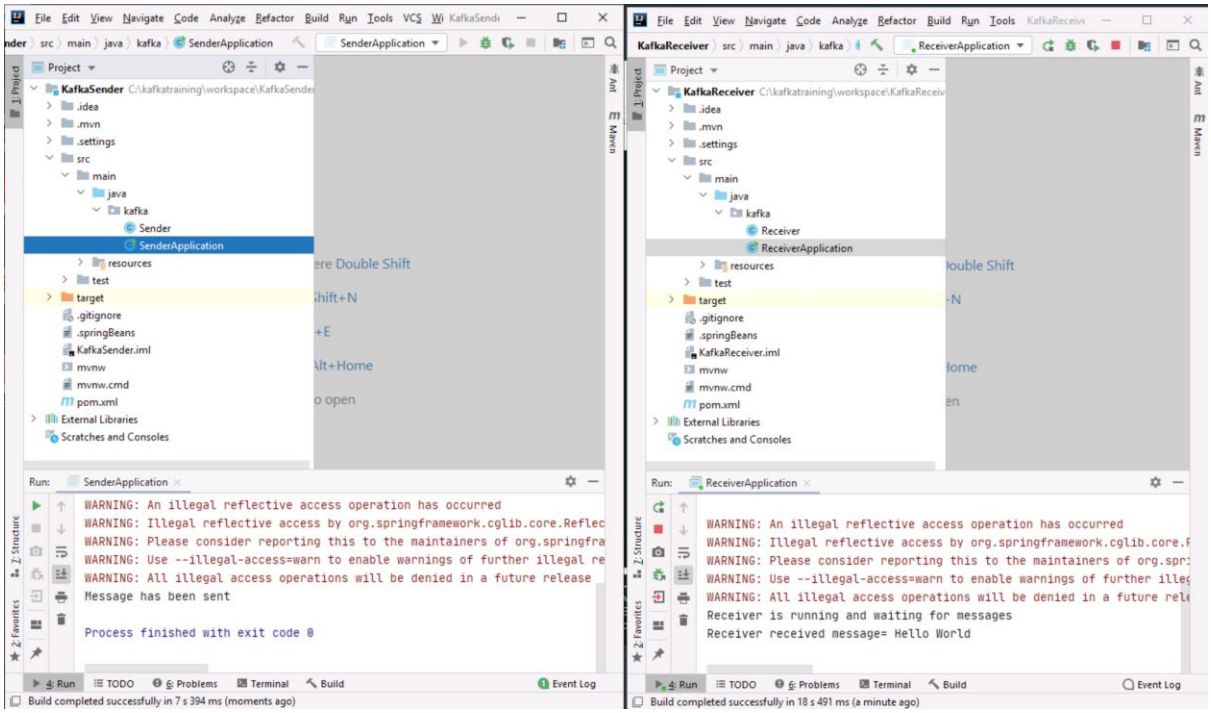
Run KafkaSender

In the console you will see the message:

```
Message has been sent
```

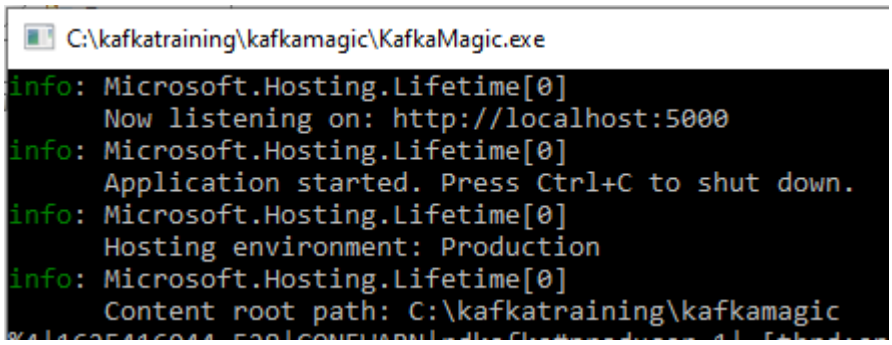
In the receiver you see the following output:

```
Receiver is running and waiting for messages
Receiver received message= Hello World
```




Run the sender a few times more.

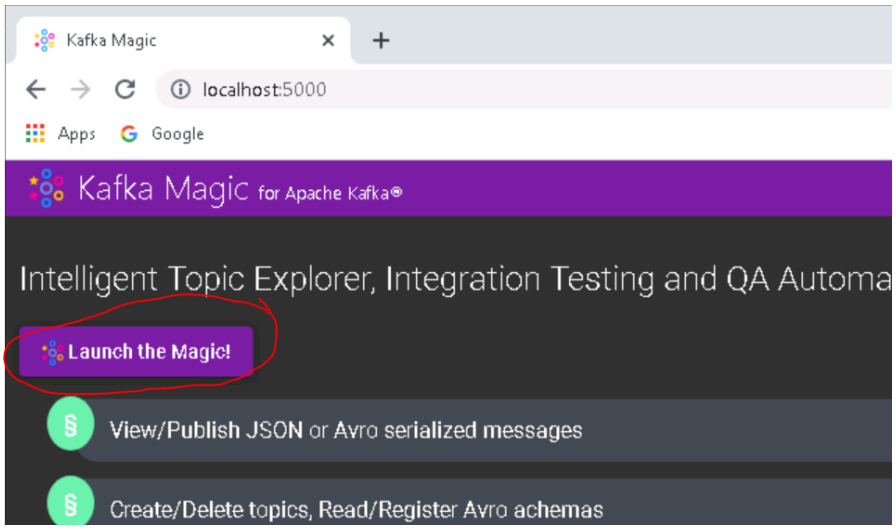
Now start KafkaMagic by double-clicking the file **C:\CS544\Tools kafkamagic\KafkaMagic.exe**



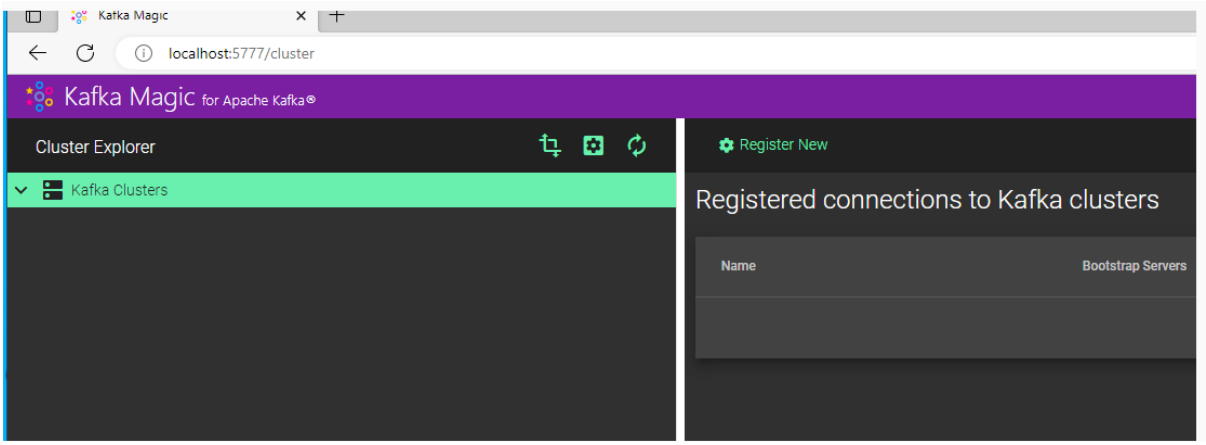

Open in a browser the following URL: **http://localhost:5000/.**

Before you Laungh the Magic tool you need to register your service

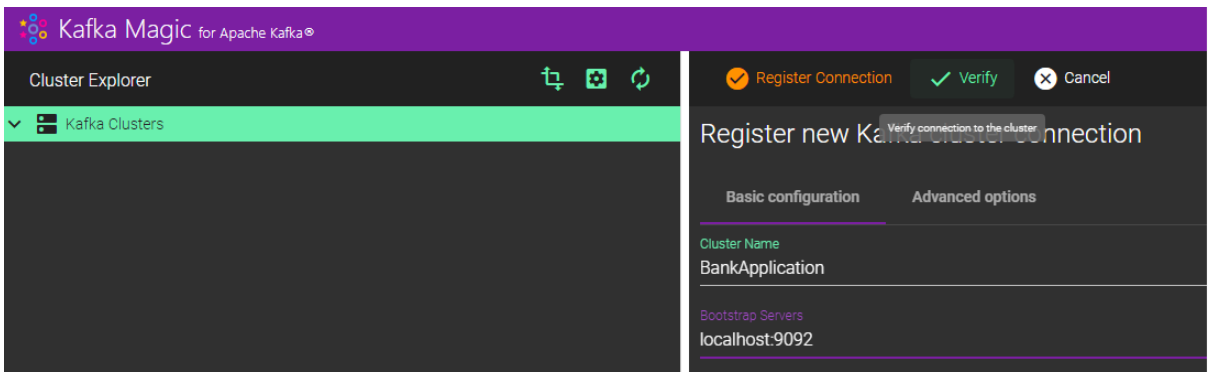
Click the **Launch the Magic!** Button



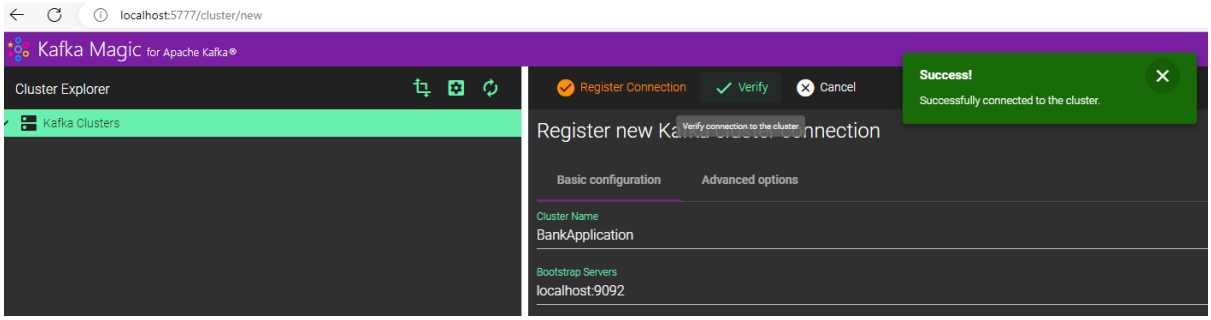

Click on “Register New”



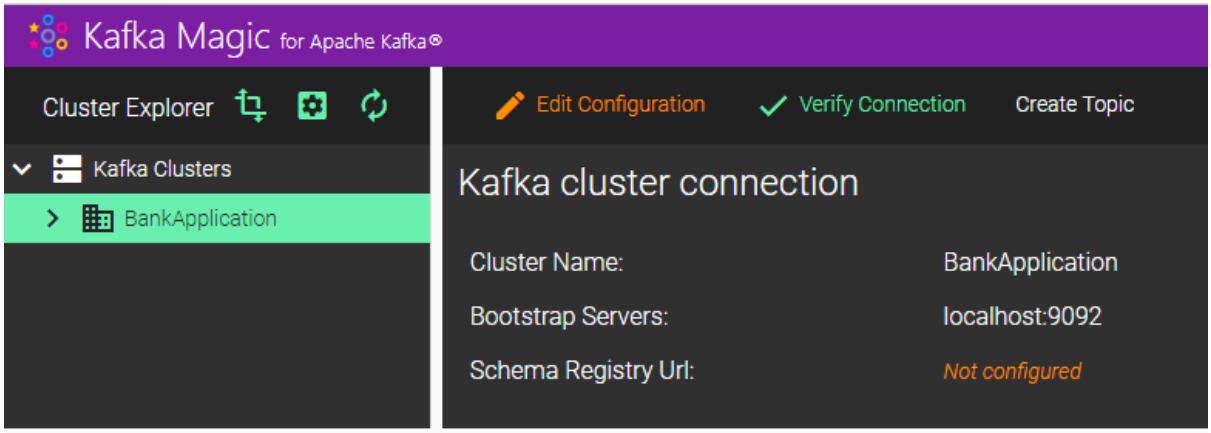

Then fill in the **“Cluster Name”** with the name you want to with to your cluster. Example: **“BankApplication”.** Then on Bootstrap Servers add the address **“localhost:9092”** as you can see on the snapshot:



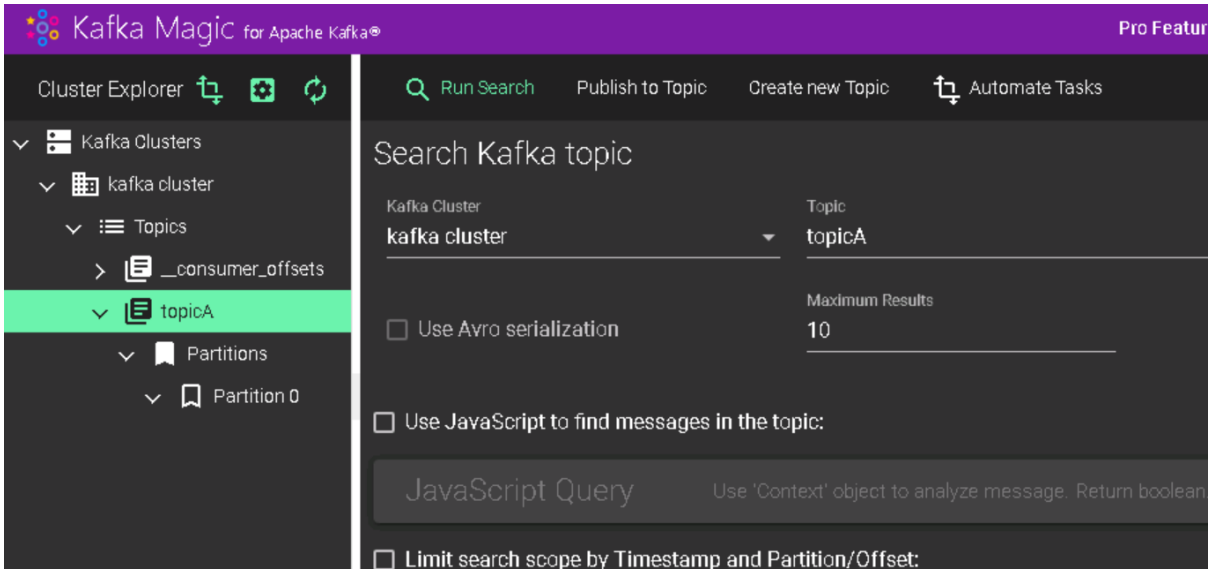
Click on **“Verify”** to check the connection:




Finally click on **“Register Connection”** to register your cluster:




Now you can see your cluster:




Select **TopicA** and click **Run Search**

Scroll to the bottom and you can now inspect the messages in the topicA
Run the sender again and see the message appear in the topic and in the receiver application.

Play a little bit with the applications. Let the sender send multiple messages when you run it.

In the ReceiverApplication write a new Receiver class:

```
@Service
public class Receiver2 {

    @KafkaListener(topics = {"topicA"})
    public void receive(@Payload String message) { System.out.println("Receiver2 received message= "+ message); }

}
```

Restart the Receiver and see that only one receiver received the message send by the sender.

Modify Receiver2 as follows

```
@Service
public class Receiver2 {

    @KafkaListener(topics = {"topicA"}, groupId = "gid2")
    public void receive(@Payload String message) { System.out.println("Receiver2

}
```

Restart the Receiver and notice that both receivers received the message send by the sender.

In the KafkaSender project, create a new Sender that sends a message to **TopicA2**. In the KafkaReceiver project add a new Receiver that listens to messages in **TopicA2**

## Part D. –Kafka for the Bank Application

Write a new Client application that can call the following functionality of the bank application by sending a message to the bank application using Kafka:

- Create an account
- Deposit money
- Withdraw money

The bank application does not need to send a message to this Client application.

## What to hand in:

1. A separate zip file with the solution of part A
2. A separate zip file with the solution of part B
3. A separate zip file with the solution of part C
4. A separate zip file with the solution of part D