Inhoud

[Getting started with Kafka 3](#_Toc474337318)

[Install the VM 3](#_Toc474337319)

[Getting to know the VM 4](#_Toc474337320)

[Optional: Set Networking in VM 7](#_Toc474337321)

[Create a topic 9](#_Toc474337322)

[Produce a message 10](#_Toc474337323)

[Consume a message 13](#_Toc474337324)

[Availability 14](#_Toc474337325)

[Questions 15](#_Toc474337326)

[Download Resources for the workshop 16](#_Toc474337327)

[Java producer and client 17](#_Toc474337328)

[Produce a message 17](#_Toc474337329)

[Consume a message 18](#_Toc474337330)

[Questions 20](#_Toc474337331)

[Run Node.js Kafka Client 21](#_Toc474337332)

[Consume Message from Kafka in Node.JS 21](#_Toc474337333)

[Produce Message to Kafka from Node.JS 22](#_Toc474337334)

[Run Node.js Kafka Client on your host laptop 24](#_Toc474337335)

[First Steps 24](#_Toc474337336)

[Consume Message from Kafka in Node.JS 24](#_Toc474337337)

[Produce messages to Kafka from Node.JS 25](#_Toc474337338)

[Under the Kafka hood 28](#_Toc474337339)

[Partitions 28](#_Toc474337340)

[Brokers 31](#_Toc474337341)

[Kafka REST 38](#_Toc474337342)

[Produce a message 38](#_Toc474337343)

[Create a consumer 38](#_Toc474337344)

[Bonus 40](#_Toc474337345)

[Kafka Streams 41](#_Toc474337346)

[Preparation – publish Country Messages from node client 41](#_Toc474337347)

[Preparation 2 – Install maven on VM 41](#_Toc474337348)

[DIY Streams in node.js 42](#_Toc474337349)

[Run Kafka Streams Application 44](#_Toc474337350)

[Streams Resources 49](#_Toc474337351)

[References 50](#_Toc474337352)

# Getting started with Kafka

## Install the VM

Running the VM requires 2Gb of RAM and approximately 7Gb of free HD space

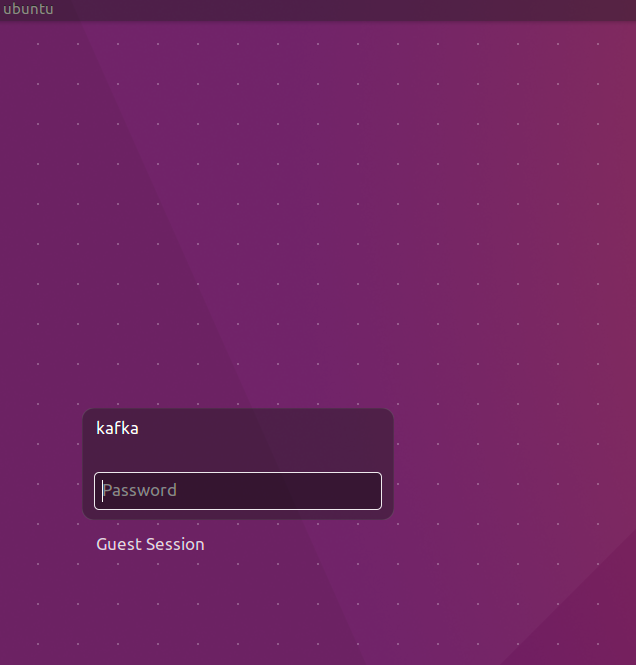
Download VirtualBox from <https://www.virtualbox.org>.

Start VirtualBox

Import the VM image by going to File, Import Virtual Appliance and browse to the image file. The VM requires 2Gb of RAM to run.

Start the VM

The VM will boot to a graphical login screen



The VM has two users: root and kafka. Both have password Welcome01.

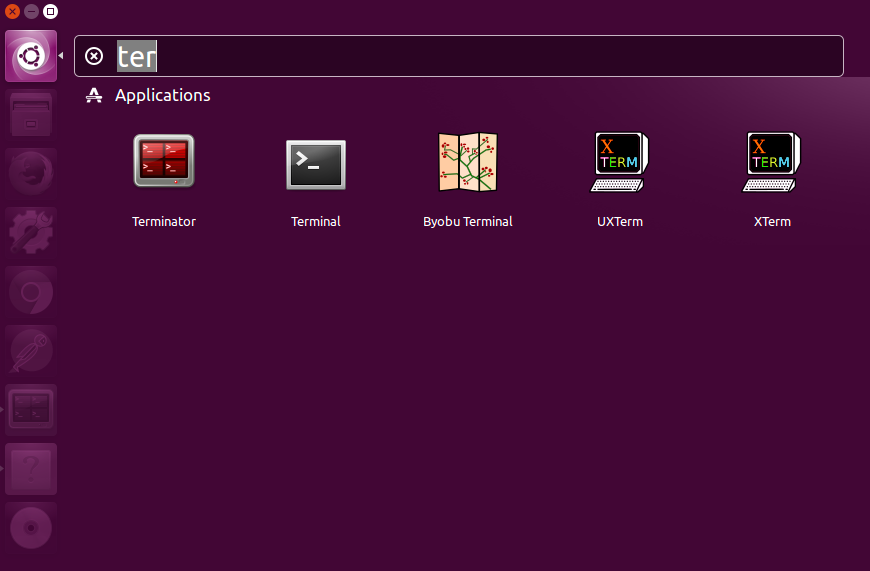
The VM contains:

* Ubuntu 16.04 LTS
* Oracle JDK 8
* Eclipse Neon.2
* Node 7.4
* Zookeeper
* Firefox
* Chrome
* Postman
* confluent-platform-oss-2.11
* kafka-manager 1.3.2.1. When started runs on <http://localhost:9000>
* kafkatool 1.0.1

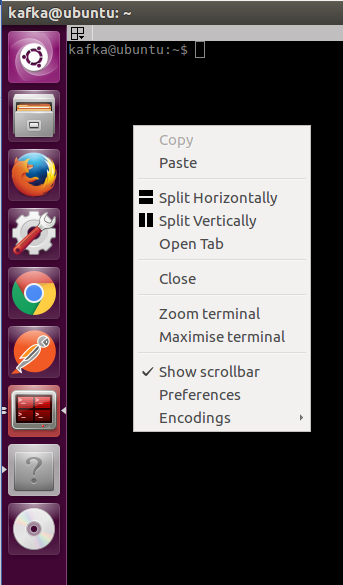
## Getting to know the VM

Login to the VM: kafka/Welcome01

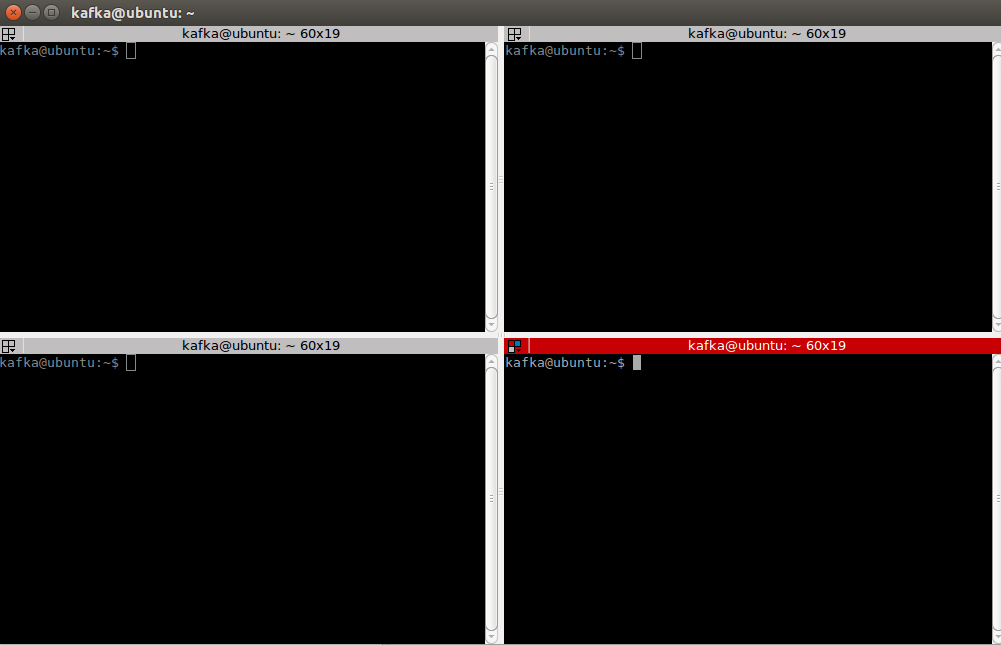
Start Terminator from the Unity menu



Split the screen horizontally – as shown in the next figure - and both screens vertically by right clicking in the window.



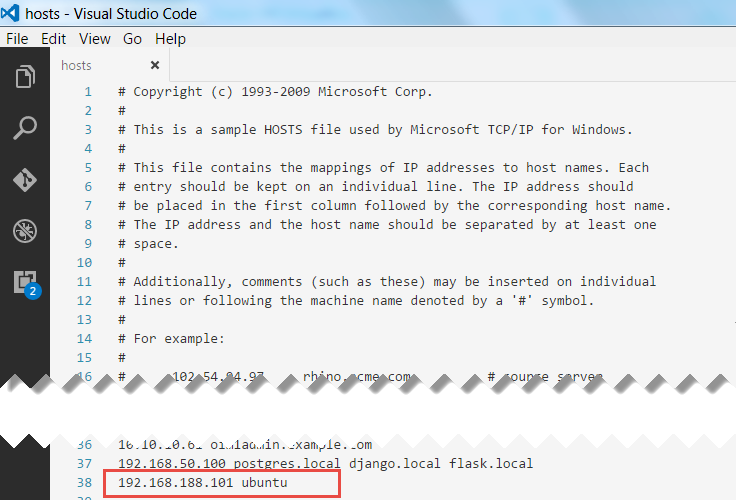
You will end up with four terminal windows.



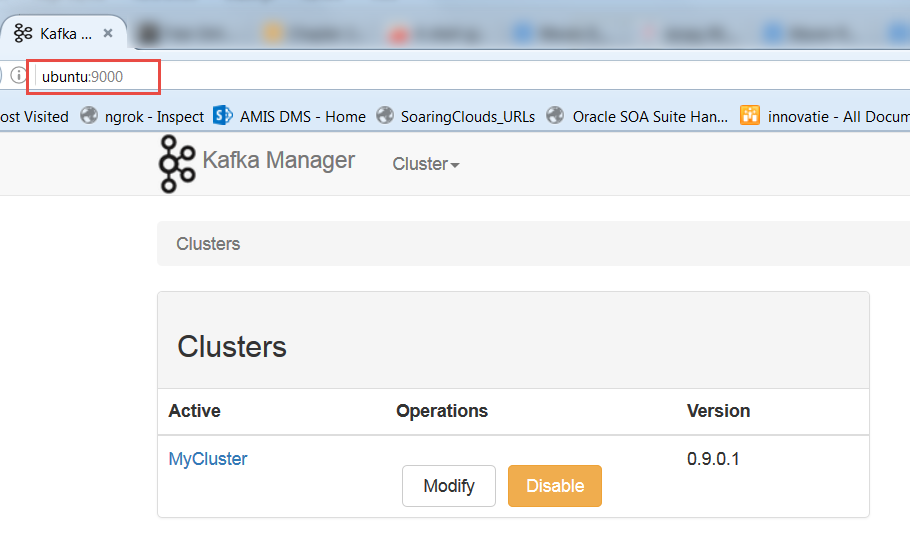
# Optional: Set Networking in VM

If and when you want to access the Kafka instance running inside the VM from your laptop, you need to ensure that you enable network access to the VM. The instructions in this article (AMIS Technology Blog: <https://technology.amis.nl/2017/01/29/network-access-to-ubuntu-virtual-box-vm-from-host-laptop/> ) show what you have to configure in order to be able to access the VM on its own IP address. Note that for this first series of labs, you do not need this. So you can skip this step for now, and optionally return to it later.

After arranging access to the VM, you may want to define a logical host name for the Virtual Machine – by adding an entry to the hosts file: C:\Windows\System32\drivers\etc\hosts, as shows below:



Note: ubuntu is the host name of the VM; to play nice with Zookeeper it seems convenient to use that same name as the host name in our hosts file.

Getting started with Kafka

In the first terminal window start a Kafka broker:

kafka-server-start /etc/kafka/server.properties

Start in the second terminal window start the kafka manager:

kafka-manager

Start kafkatool in the third terminal window:

kafkatool

Minimize the tool. You will use it at a later time.

Continue in the fourth terminal window

### Create a topic

Create a topic:

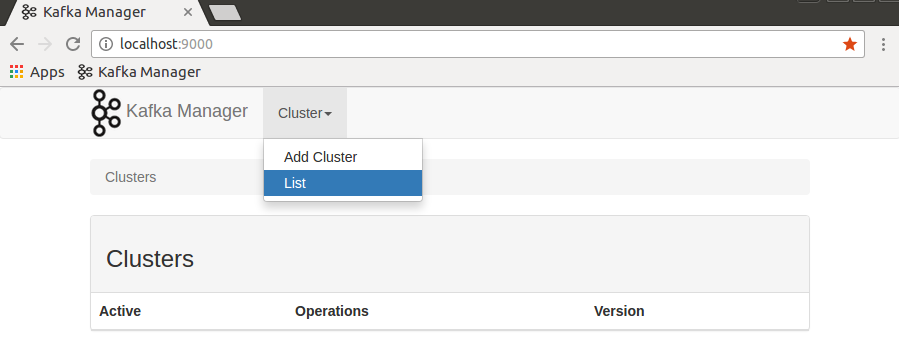
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test

Confirm the topic is created:

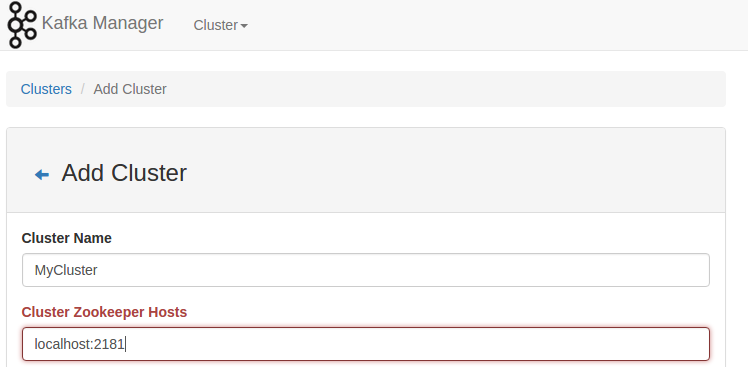
kafka-topics --list --zookeeper localhost:2181

Open a browser and go to <http://localhost:9000> and confirm the topic is created using a webinterface.

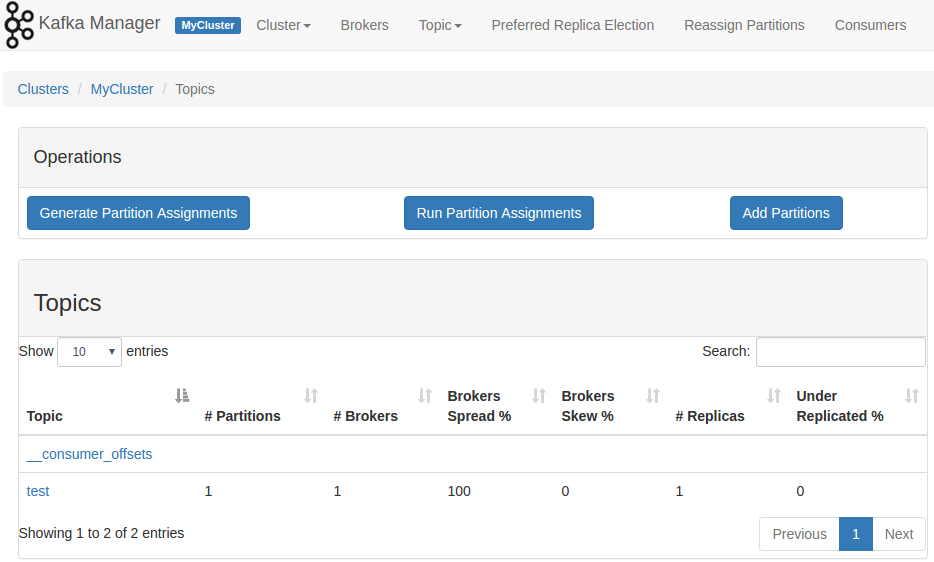
First create a cluster



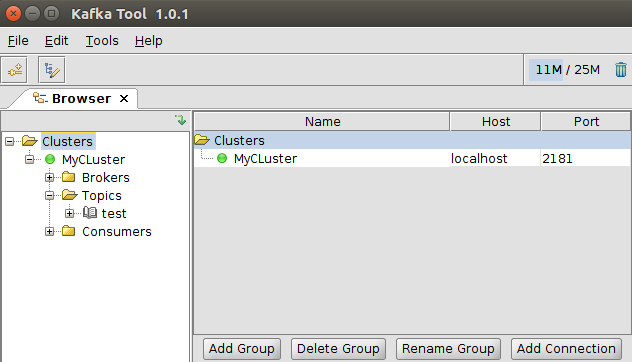
Name the cluster MyCluster and enter localhost:2181 as Zookeeper cluster. Scroll down and click save



Go to the cluster view and determine the topic has been created



Open kafkatool and confirm the topic is created



### Produce a message

Focus on the fourth terminal window again and type:

kafka-console-producer --broker-list localhost:9092 --topic test

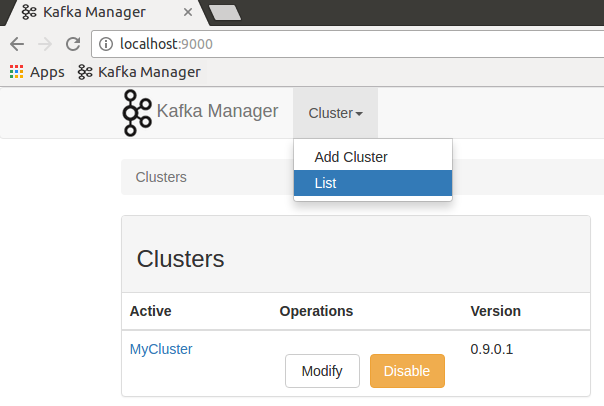
Press enter. No feedback appears; the cursor sits blinking and waiting for your input.

Type a message and press enter:

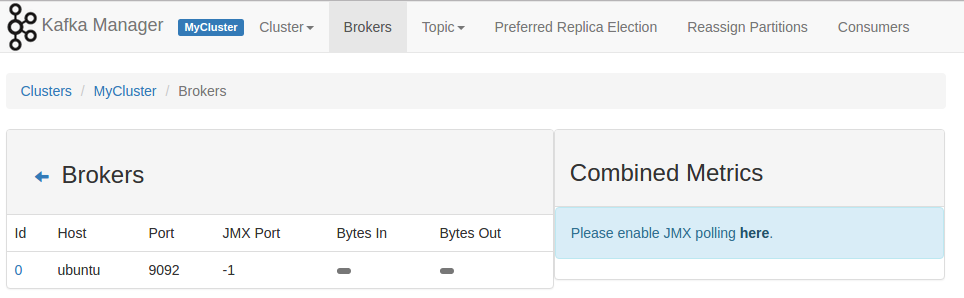
Hello World!

Check the message is created using kafka-manager and kafkatool.

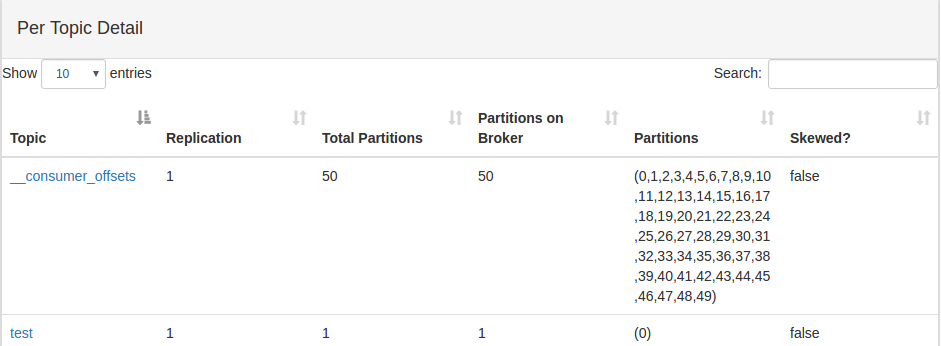
In the kafka-manager, go to Cluster, List and click the cluster you previously created



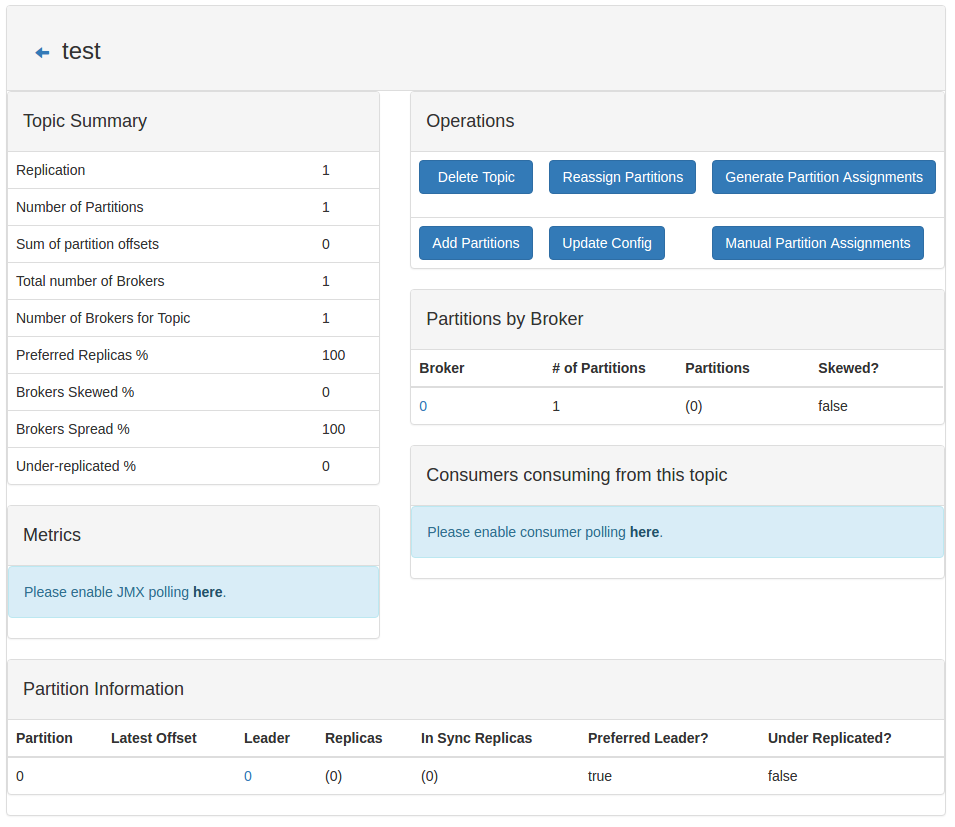
Click brokers, click 0



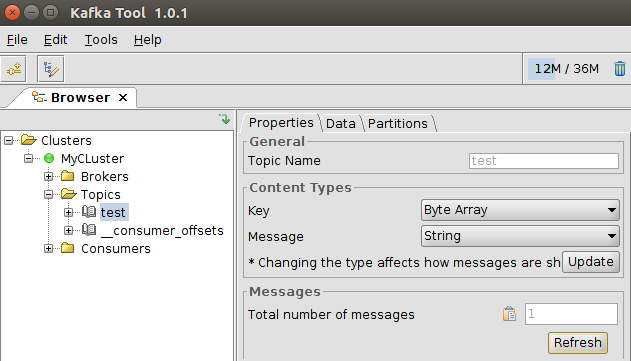
Scroll down and click the test topic



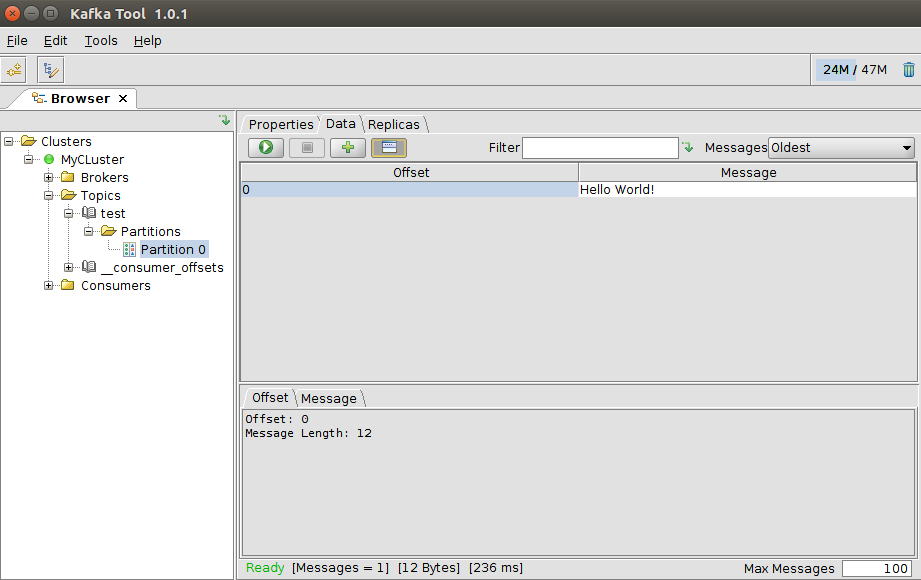
Scroll down again and confirm the latest offset is empty. No consumer has picked up messages yet.



Open kafkatool. Indicate the messages on the topic are string. Click the update button. Click the refresh button to view how many messages are present on the topic. Confirm the number is 1.



Open the partition under the test topic. Click the green play button. Confirm the Hello World! message has arrived.



### Consume a message

Consume a message

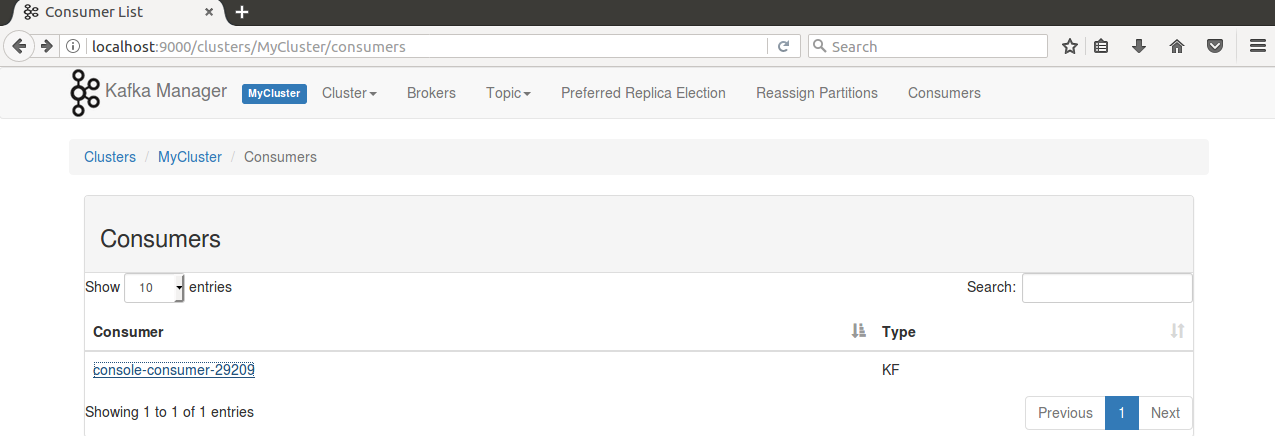
kafka-console-consumer --bootstrap-server localhost:9092 --topic test --from-beginning



Confirm that the previously posted message has been consumed.

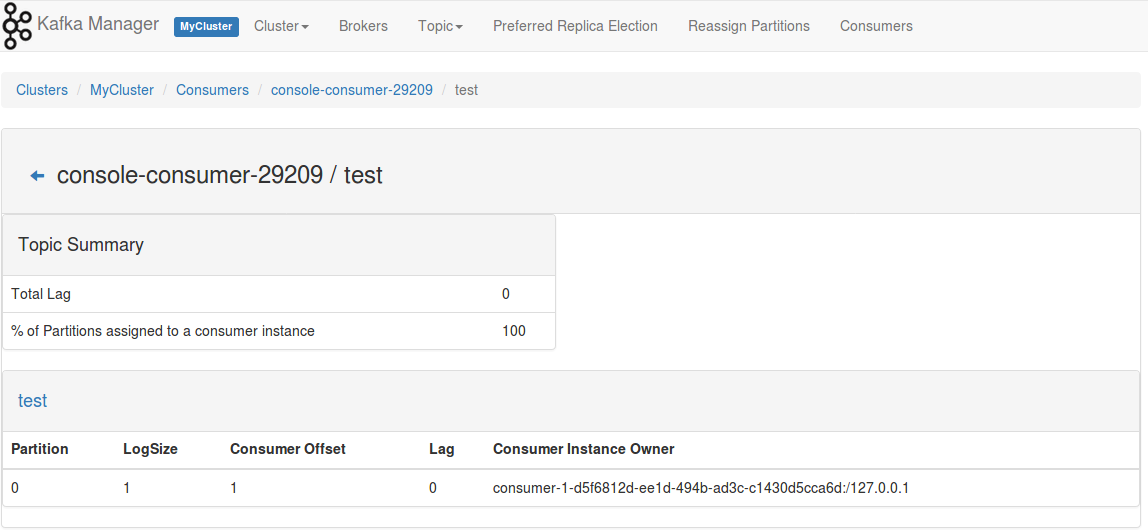
Look at kafka-manager: <http://localhost:9000>

Click on MyCluster, Consumers



Confirm the console consumer is visible

Click the consumer name. Next click the topic test.



Confirm that this consumer has an offset of 1.

End the console consumer by pressing ctrl-C in the terminal window where the consumer is running. End the console producer by pressing ctrl-C in the terminal window where the producer is running.

### Availability

Start a consumer

kafka-console-consumer --consumer-property group.id=TestConsumer --bootstrap-server localhost:9092 --topic test

Start a producer

kafka-console-producer --broker-list localhost:9092 --topic test

Send a message from the producer

Hello World!

Confirm the consumer receives the message.

End the consumer by pressing ctrl-C in the consumer console window

Produce a new message

Hello World!

Start the consumer

Confirm the consumer receives the message it had missed while being down.

Remove the consumer offset registrations.

kafka-consumer-groups --delete --zookeeper localhost:2181 --topic test

Remove the topic

kafka-topics --delete --zookeeper localhost:2181 --topic test

### Questions

* Why does the consumer not pickup missed messages when the group.id is not specified?
* Where is the offset for the consumers stored? Zookeeper, the consumer, the producer or the broker?

# Download Resources for the workshop

The resources can be downloaded from: <https://github.com/MaartenSmeets/kafka-workshop>. You can use the *download zip* function to download the Git repo in a single zip file.

The easiest way to get all sources into your VM is using the git command line tool.

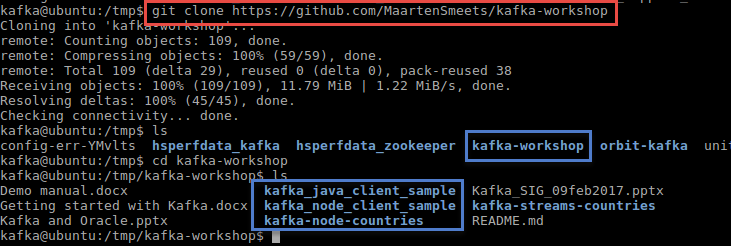
Open a terminal window in the VM. Navigate to a suitable destination directory, for example:

cd /tmp

then execute this command:

git clone https://github.com/MaartenSmeets/kafka-workshop

The GIT repository is cloned from GitHub and downloaded to your VM. The directory kafka-workshop is created and contains all sources used for the hands-on.



# Java producer and client

Start the Kafka broker if it is not already running

kafka-server-start /etc/kafka/server.properties

In a second terminal window start eclipse

cd ~/eclipse

./eclipse

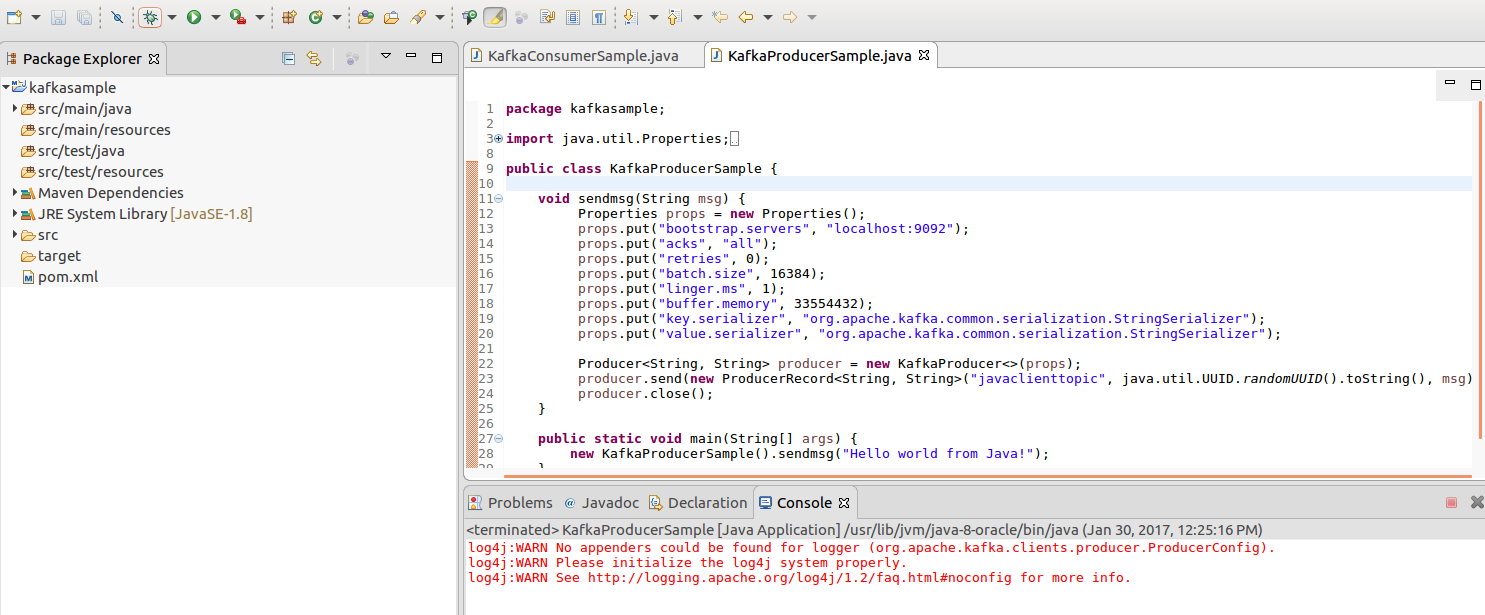
In a third terminal start kafkatool

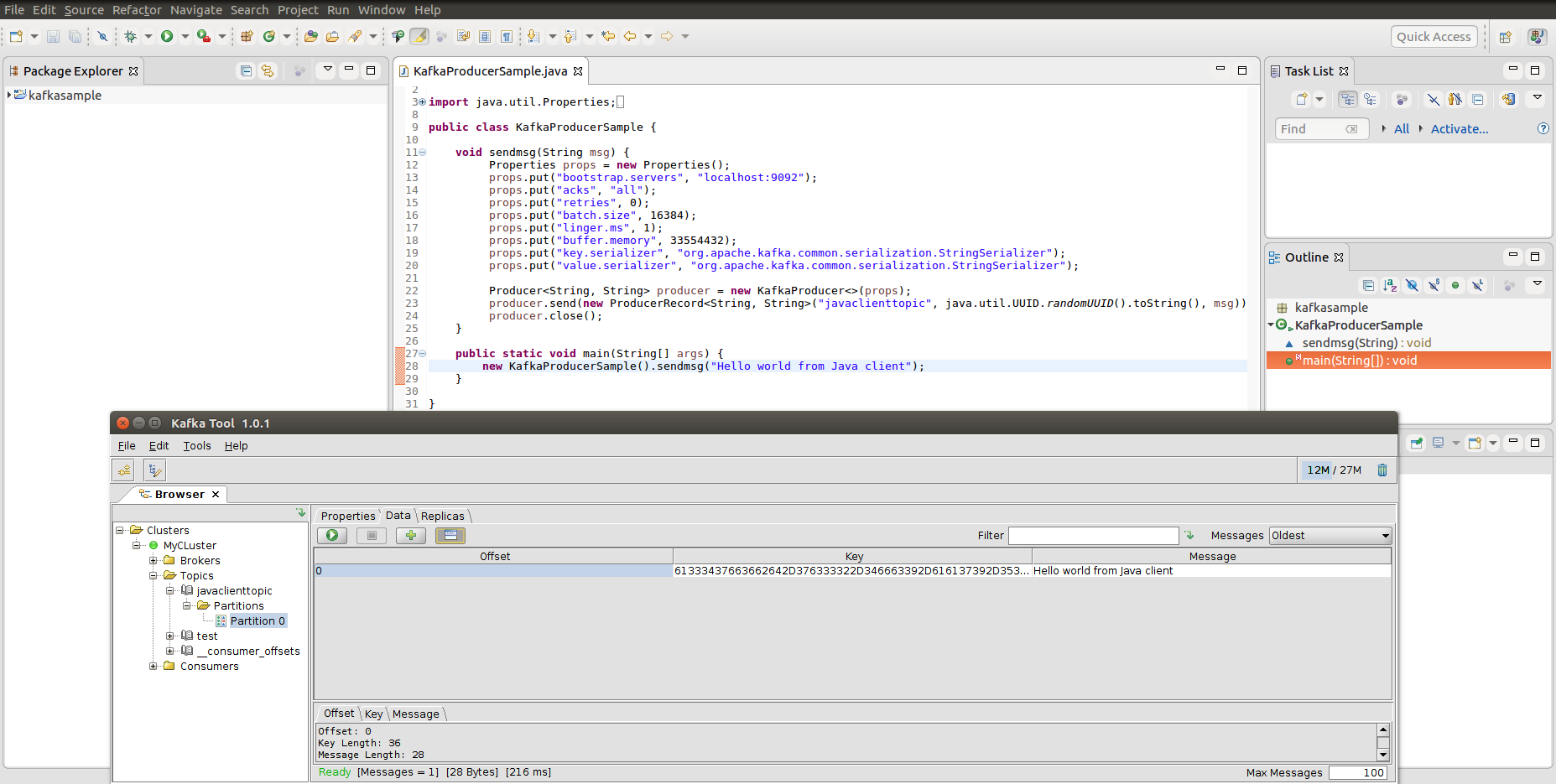
kafkatool

You can run the below code in the VM or outside within your own IDE.

## Produce a message

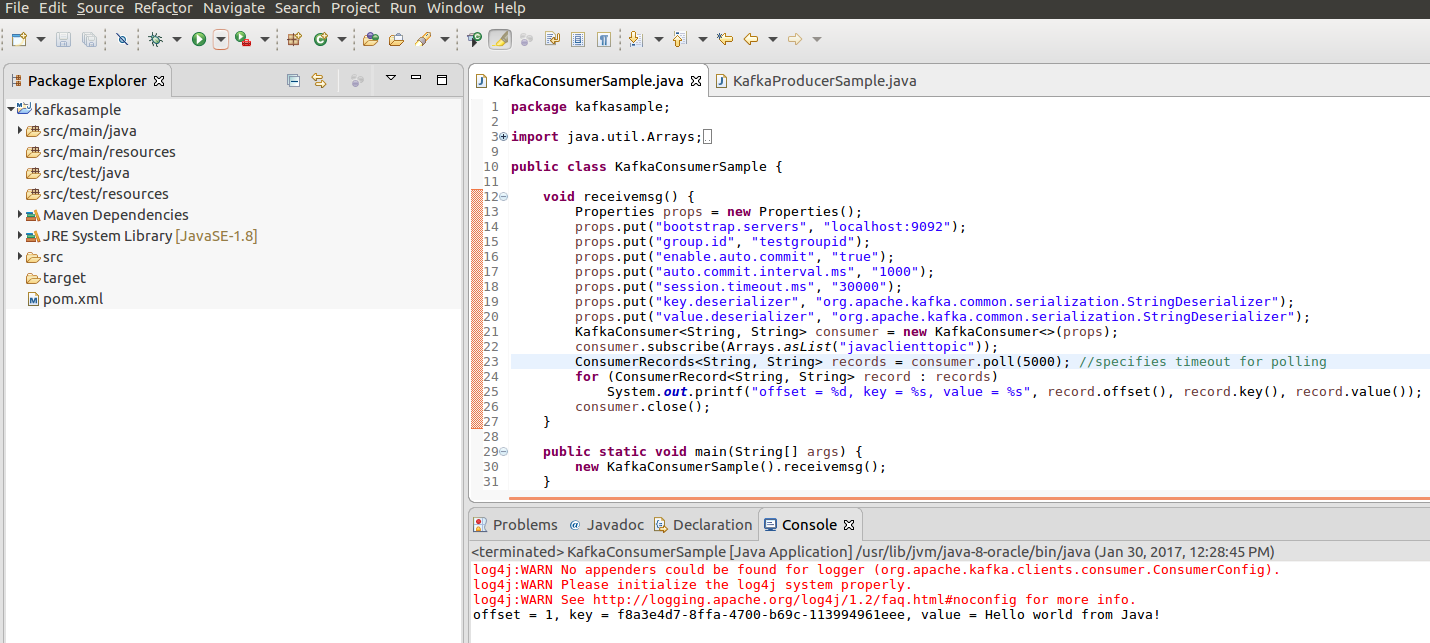
Run the Java producer (green play icon, run as Java application) and confirm message arrives on the topic specified with kafkatool. When running the producer, no output is produced.



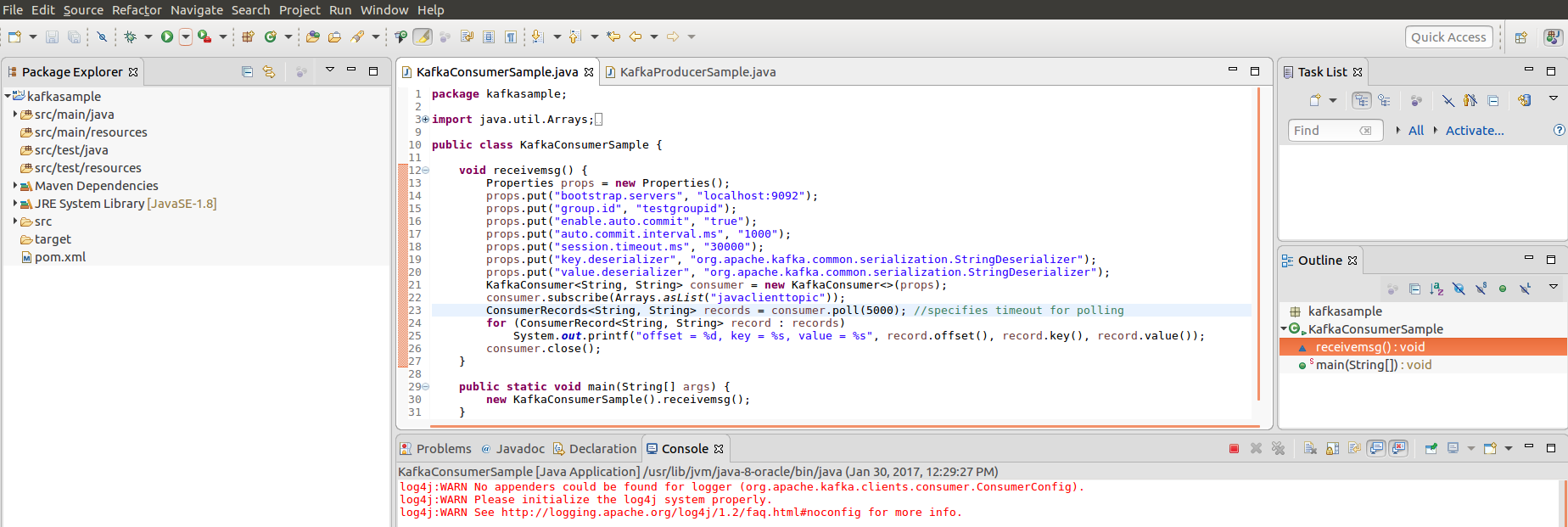


## Consume a message

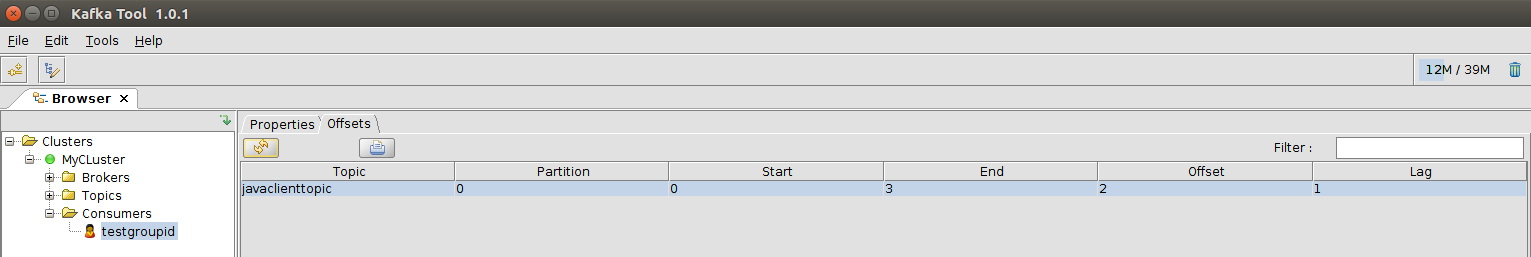
Run the Java consumer. Confirm in the console the message is consumed.



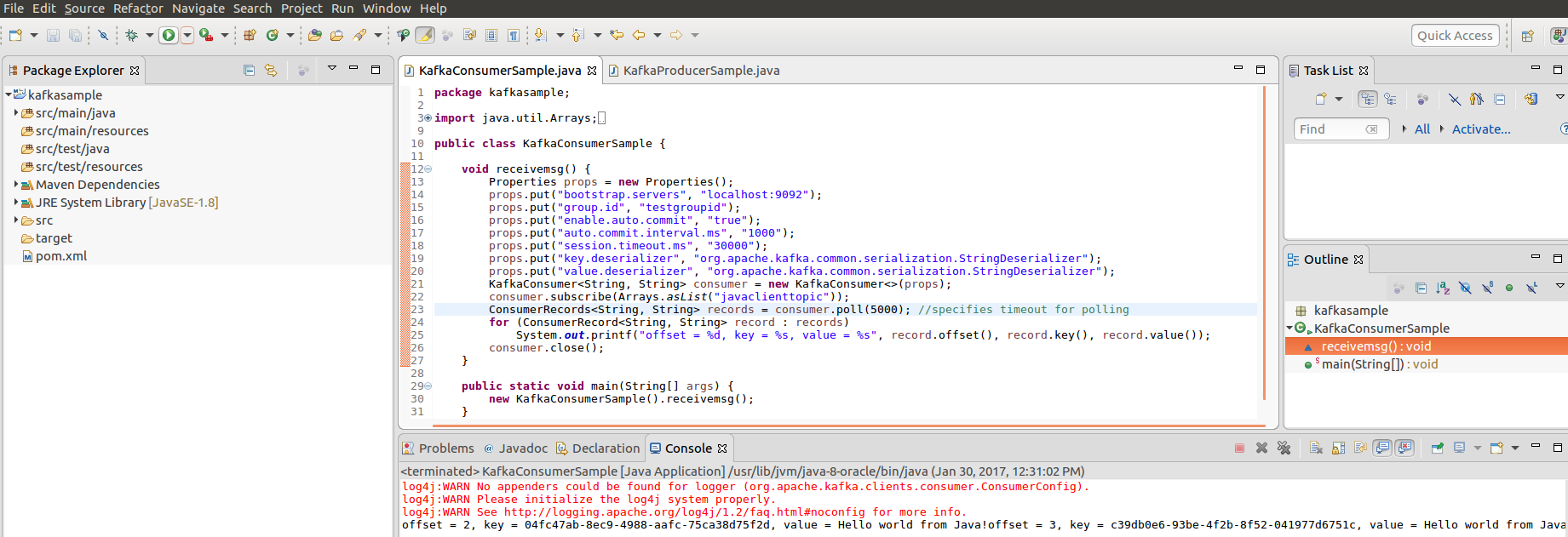
Run the consumer again. This time no messages are consumed. Why? How does the consumer know which offset to use?



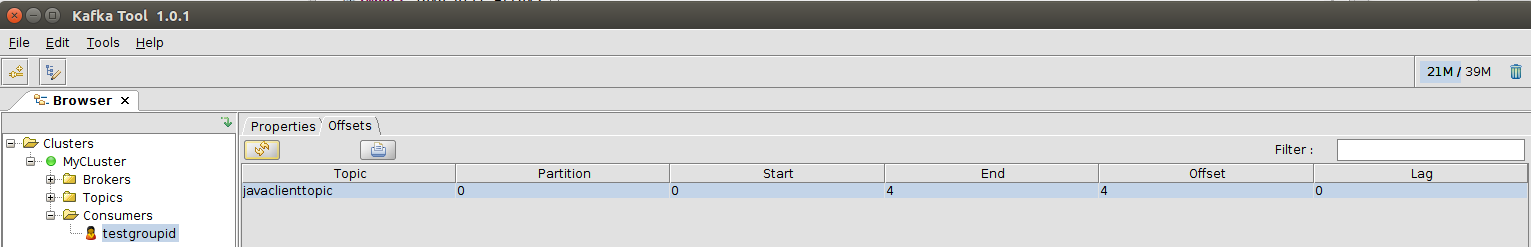
Produce a new message and check the offset and lag for the consumer in kafkatool



Produce a new message and run the consumer once. Confirm two messages are picked up



Confirm the lag has disappeared (is 0) for this consumer in kafkatool



Confirm the offset with a command-line

kafka-run-class kafka.tools.GetOffsetShell --broker-list localhost:9092 --topic javaclienttopic --time -1

## Questions

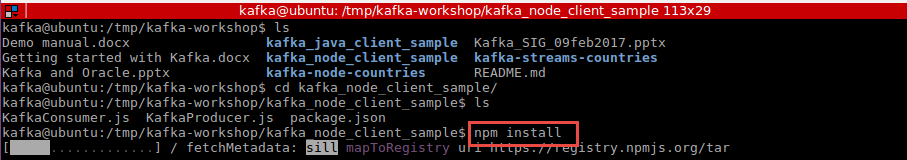
What do the different timeout values in the consumer mean? Play with them and check the behavior. For example make the poll timeout in the consumer really small (1). What happens? Why?

# Run Node.js Kafka Client

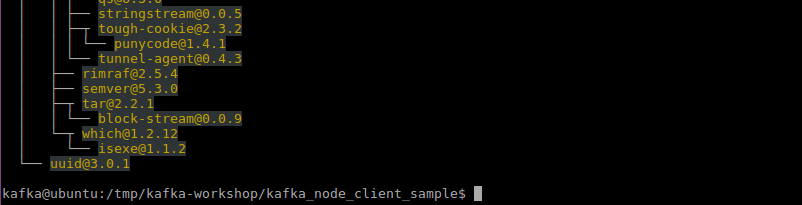
This section discusses running a Kafka client in Node.js. This will show how a Node.js application can easily produce messages to Kafka and consume messages from Kafka.

We assume that you have downloaded the workshop resources into the VM. Navigate to the directory kafka-workshop/kafka\_node\_client\_sample.

Then type npm install. This will install all node modules required for this lab – those specified in package.json.



It may take up to several minutes to install all modules. When done, you can proceed.



## Consume Message from Kafka in Node.JS

Inspect the contents of file KafkaConsumer.js:

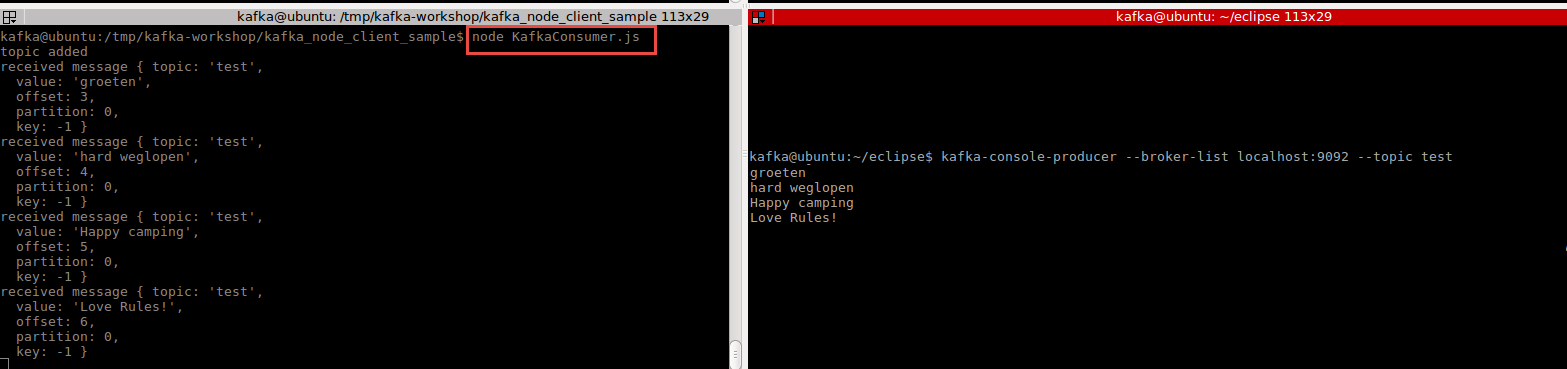
vi KafkaConsumer.js.



This code registers a new Consumer with the ZooKeeper running on ubuntu (local host) port 2181. It then expresses interest in topic *test* . Any message on that topic is pulled and processed in the on message event handler.

Run this simple Node application; from the command line:

node KafkaConsumer.js



The messages you have send to the topic earlier on in this tutorial are shown – all of them. If you end the program and then start it again – none of the messages are shown. This consumer is now up to date. If you produce new messages to topic *test*, they will be consumed and displayed.

## Produce Message to Kafka from Node.JS

Check out the contents of file KafkaProducer.js. This Node application produces messages to Kafka – through the local ZooKeeper at the familiar port.

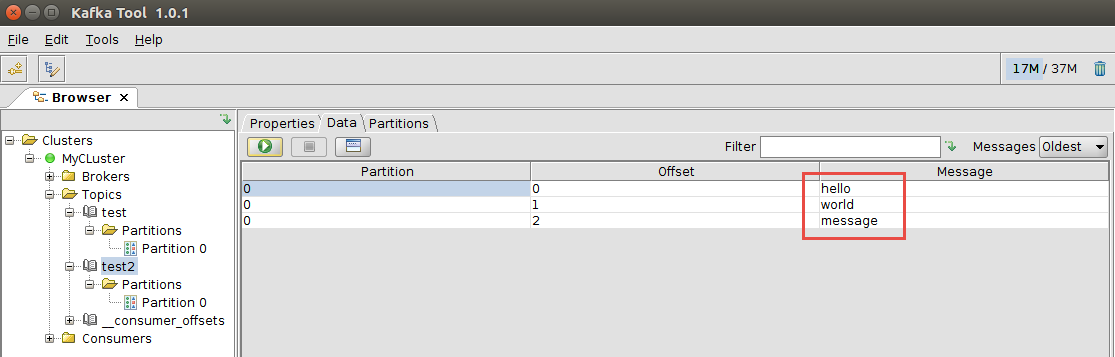
Now run this application:

node KafkaProducer.js

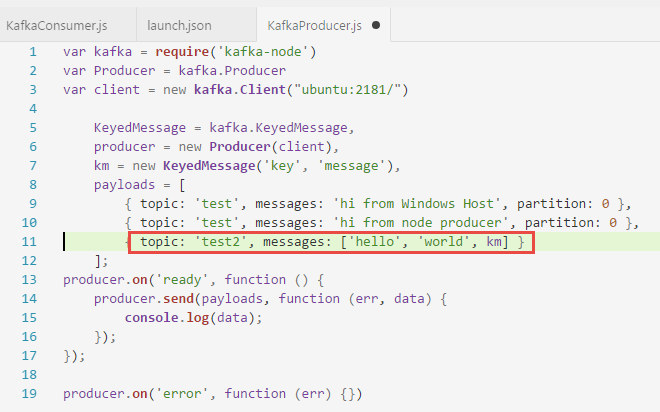


Several messages are published to the two topics test and test2, as can be seen for example in Kafka Tool.

This screenshot shows the three messages on topic test2:



Corresponding with this line in the KafkaProducer.js program:



# Run Node.js Kafka Client on your host laptop

As an example of how easy communicating with Kafka can be – even from other hosts than those running a node in the Kafka cluster – this section discusses running a Kafka client in Node.js. This will show how a Node.js application can easily produce messages to Kafka and consume messages from Kafka.

Prerequisites:

you need to have Node.JS and npm installed, either on your laptop host or inside the VM (where it is already installed) – and at least Node.JS release 4.4.7 at the time of writing. To install or upgrade Node.JS, go to <https://nodejs.org/en/download/> . The latest stable release at the time of writing is v6.9.4

## First Steps

Create a directory where you want to create new the Node.JS application to interact with Kafka. Optionally, create a new Node application with:

npm init

Open the command line window (a terminal window) for this directory. Install kafka-node, the Node.JS library for Kafka:

npm install kafka-node --save



## Consume Message from Kafka in Node.JS

Next, create a new file: KafkaConsumer.js. Add the following code to the file:

var kafka = require('kafka-node')

var Consumer = kafka.Consumer

var client = new kafka.Client("ubuntu:2181/")

var consumer = new Consumer(

client,

[],

{fromOffset: true}

);

consumer.on('message', function (message) {

console.log("received message", message);

});

consumer.addTopics([

{ topic: 'test', partition: 0, offset: 0}

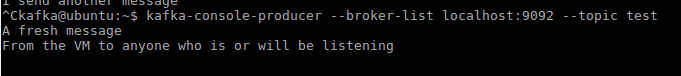
], () => console.log("topic added"));

Check if the topic name is correct and if the host (*ubuntu* – the logical name for the VM running the ZooKeeper for the Kafka cluster) and the port are correct. They should be if you have followed the steps in this tutorial.

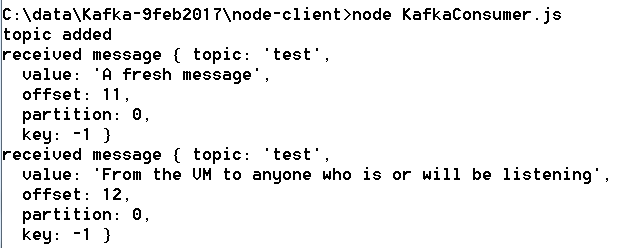
Run this simple Node application; from the command line:

node KafkaConsumer.js

The messages you have send to the topic earlier on in this tutorial are shown – all of them. If you end the program and then start it again – none of the messages are shown. This consumer is now up to date. If you were to add a new message to the topic, for example from inside the VM,



this new message will be shown:



## Produce messages to Kafka from Node.JS

Create a new file in the same directory, called KafkaProducer.js. Add the following content:

var kafka = require('kafka-node')

var Producer = kafka.Producer

var client = new kafka.Client("ubuntu:2181/")

KeyedMessage = kafka.KeyedMessage,

producer = new Producer(client),

km = new KeyedMessage('key', 'message'),

payloads = [

{ topic: 'test', messages: 'hi from Windows Host', partition: 0 },

{ topic: 'test', messages: 'hi from node producer', partition: 0 },

{ topic: 'test2', messages: ['hello', 'world', km] }

];

producer.on('ready', function () {

producer.send(payloads, function (err, data) {

console.log(data);

});

});

producer.on('error', function (err) {})

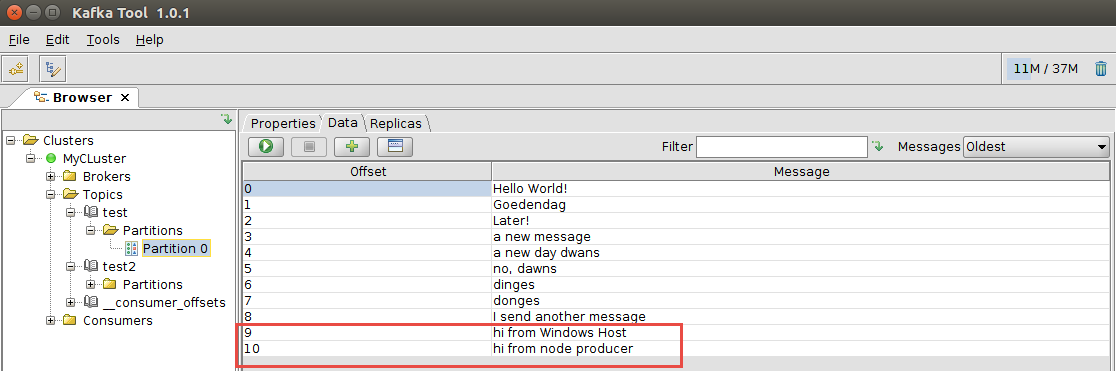
Again, verify the names of the topics (test and test1) and the name of the host running ZooKeeper and the port it is running on.

Now run the program:

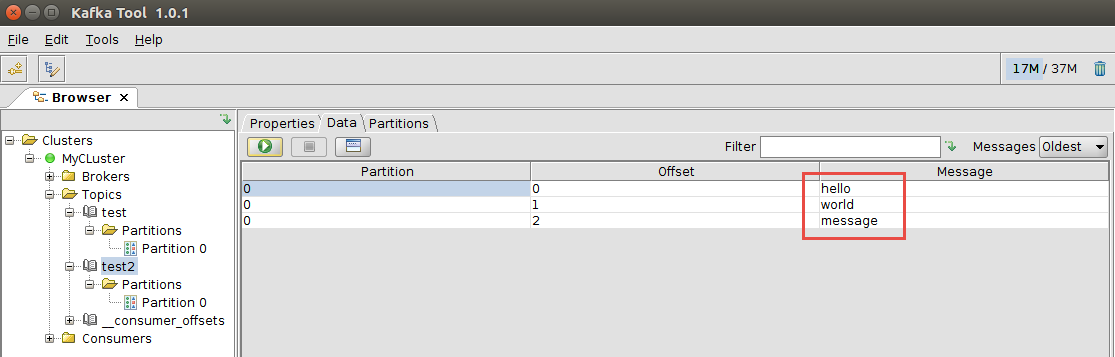
node KafkaProducer.js

Several messages are published to the two topics test and test2, as can be seen for example in Kafka Tool.

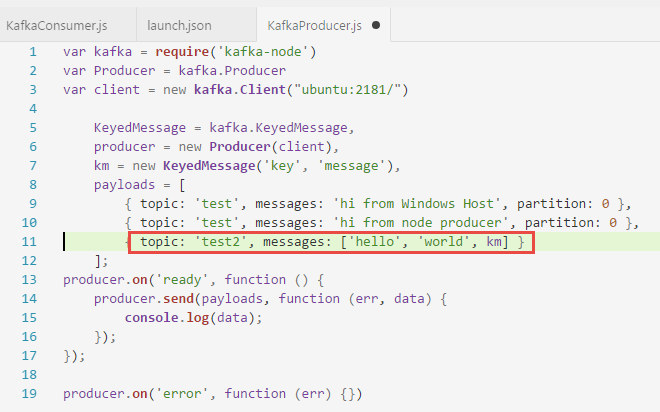
The first screenshot shows the messages in partition 0 of topic test:



and this figure the three messages on topic test2:



Corresponding with this line in the KafkaProducer.js program:



For more details on kafka-node: <https://www.npmjs.com/package/kafka-node> .

# Under the Kafka hood

In this section, we will take a closer look at some of the internals of Kafka.

First we look at partitions – and in doing so we look at some configuration details and the file structure under the message logs. Next we focus on brokers – the individual nodes in a Kafka cluster. Partitions and brokers provide the foundation for scalability and availability.

## Partitions

Partitions – used to increase scalability by decreasing serialization – are an important concept in Kafka. It is at the level of partitions that the logfiles are created on disk. It is at the level of partitions that consumer offsets are specified, replication is performed and many administrative tasks are done.

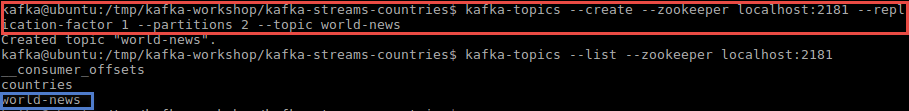
Each topic is segmented into one or more partitions. There is a logfile per partition (and per broker – more on that later). The partition logfile is a serialization point – and therefore a potential bottleneck. Only one process can write to a partition (and its logfile) at any one moment, even though multiple processes can consume from a partition at the same time. Using multiple partitions is one way to decrease the contention for a resources.

Create a topic in one of the VM terminal windows:

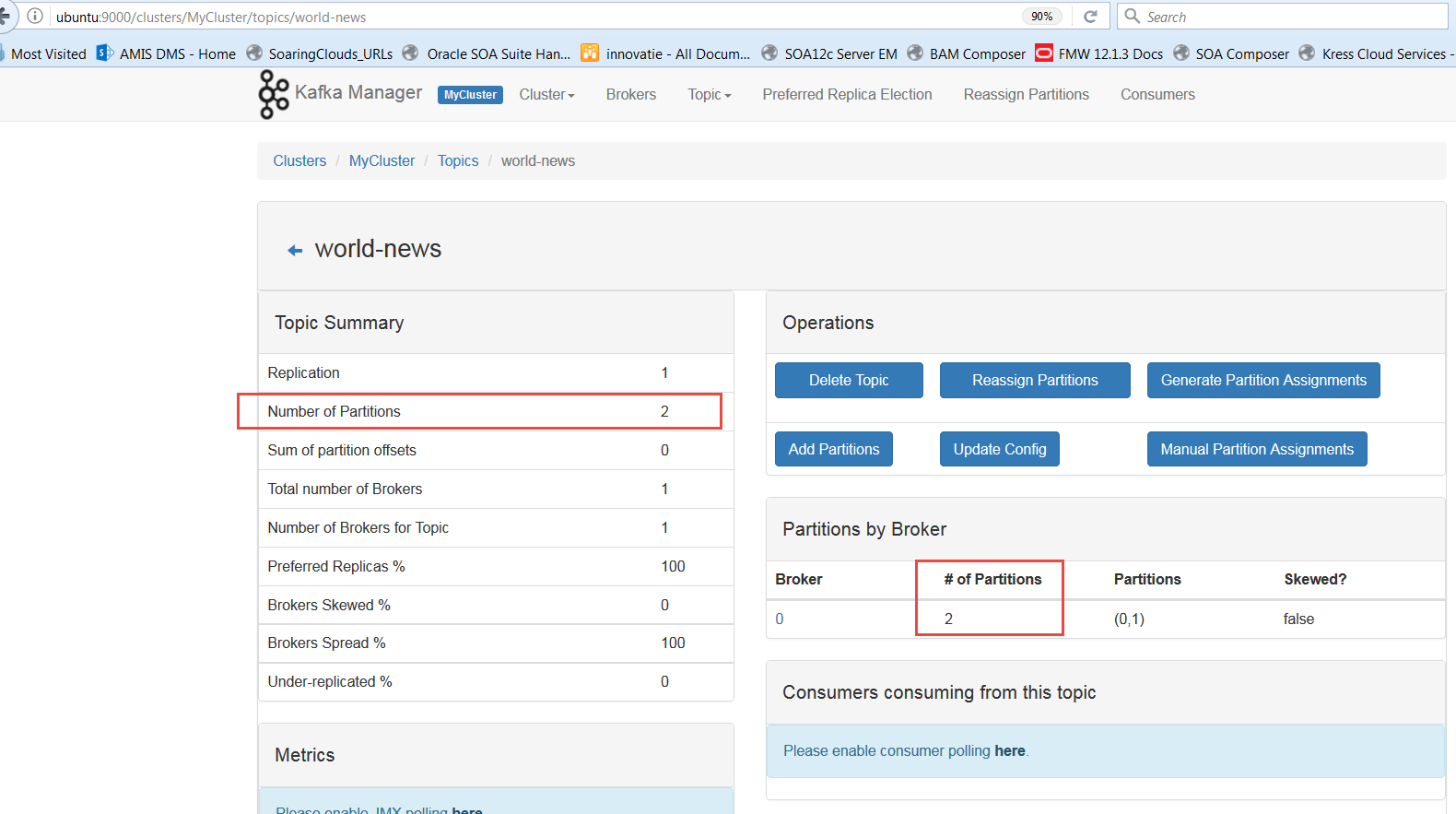
kafka-topics --create --zookeeper localhost:2181 --replication-factor 1 --partitions 2 --topic world-news

Confirm the topic is created:

kafka-topics --list --zookeeper localhost:2181

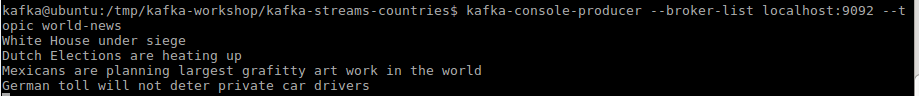


Open a browser – either inside the VM and go to <http://localhost:9000> or on the host and goto <http://ubuntu:9000> - and inspect the topic details – specially the number of partitions.

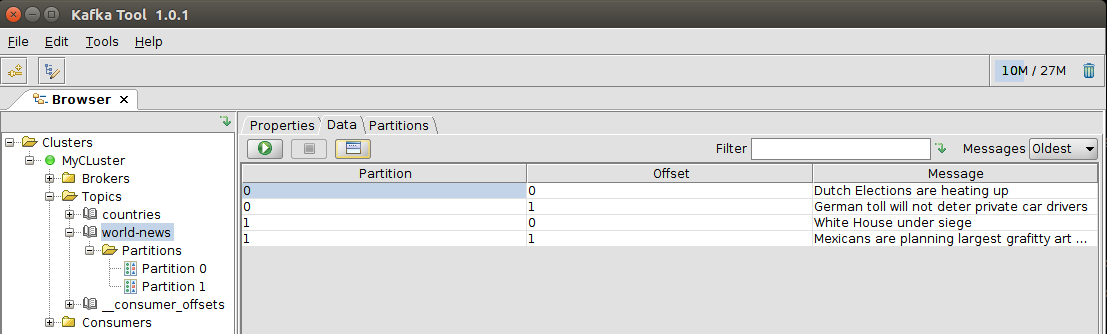


Publish a few messages:

kafka-console-producer --broker-list localhost:9092 --topic world-news

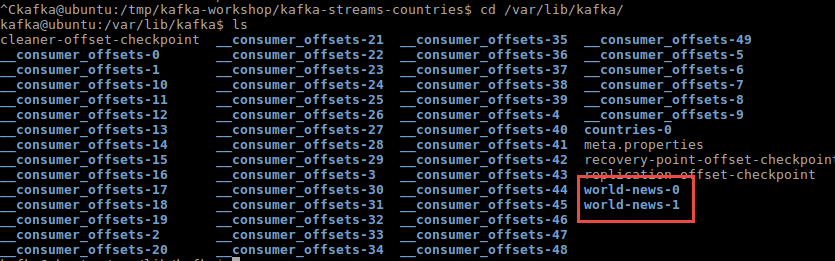


Check in KafkaTool what happened with these messages:

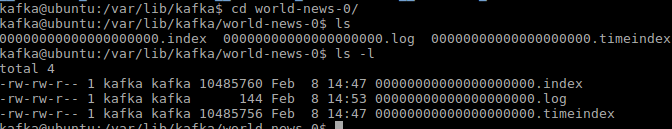


Two partitions were created for this topic. With no special partitioning logic defined, the messages are partitioned based on a hash of their contents, nicely balanced across partitions. If a logical partitioning key exists, it can be defined.

For each partition, a separate log file has been created. For the broker that came with your VM, the logfile directory is /var/lib/kafka. Inspect the contents of this directory:



You can check out the contents of the directory world-news-0, created for the first partition under topic world-news:



The files are not legible as they are.



(see <https://kafka.apache.org/documentation/#log> for documentation on the log files)

A tool to inspect the contents of log files: <https://cwiki.apache.org/confluence/display/KAFKA/System+Tools#SystemTools-DumpLogSegment>

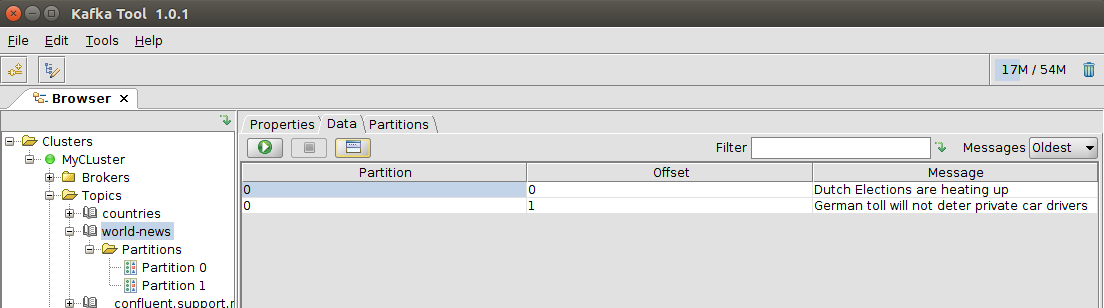
Let’s see what happens if the directory for the log files for one partition is lost. Delete the directory /var/lib/kafka/world-news-1 with its contents, using:

rm -rf /var/lib/kafka/world-news-1

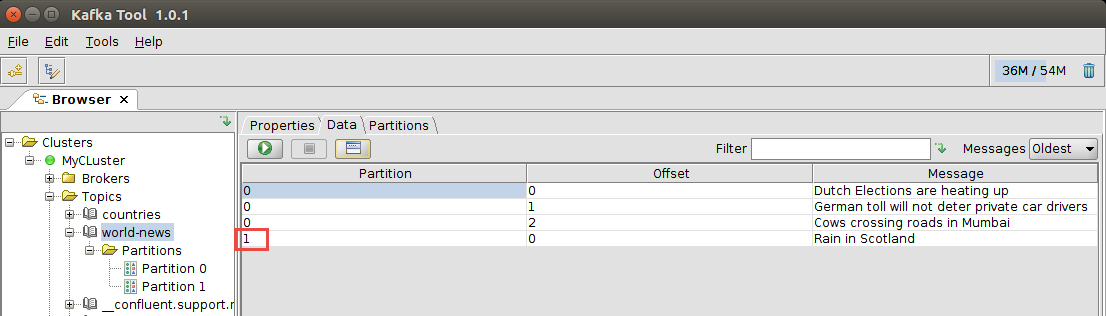


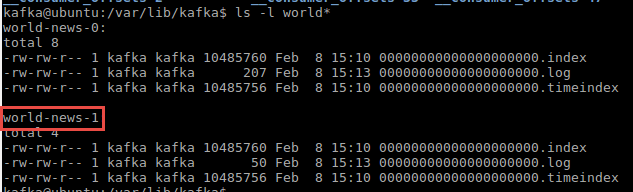
At this point, half of the messages are lost . Gone, not to be retrieved. However, because of memory caching, the Kafka tooling may not be aware of this fact yet. To establish without doubt that these messages are indeed gone, please stop and restart the broker. Also stop and restart the Kafka Tool.

You can check in Kafka Tool – both partitions are still here. But the messages stored in partition 1 are now vanished:



As soon as new messages are stored in partition 1, the directory we have just removed is recreated.



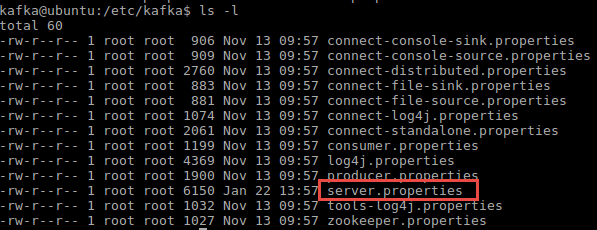


## Brokers

Brokers are independent processes that can run on distributed hosts. Brokers provide the resilience to our Kafka topics, as well as a level of scalability. In this section, we will create a second broker in our VM and create a topic with a replication factor of two. This will have the partition in the topics replicated across brokers. That in turn results in topics being available for production when at least one broker is still running, and messages being available if at least one broker is running (even if the message was produced when that broker was down).

Configure a second broker.

In directory /etc/kafka are the configuration files for many aspects of our Kafka environment. The broker we have worked with until now is started using the configuration defined in the file server.properties.



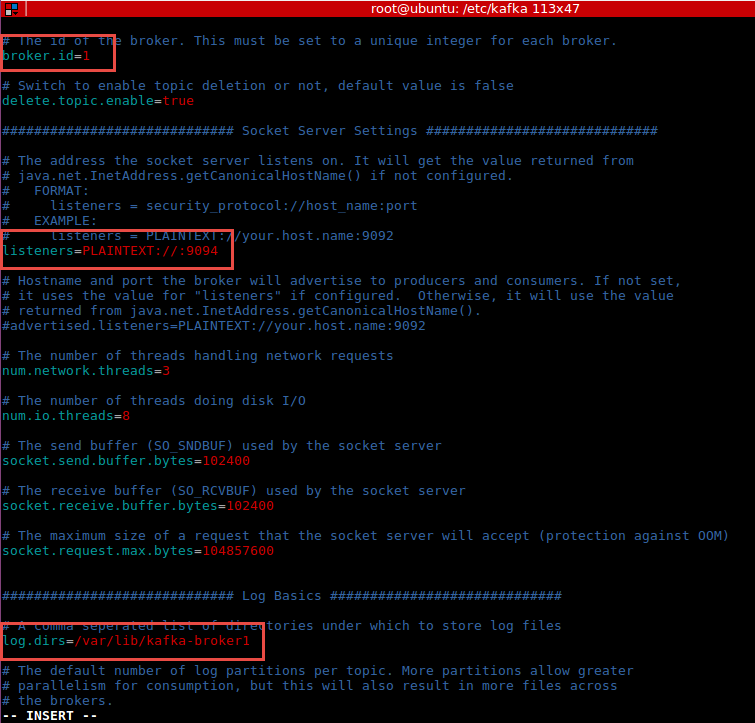
Copy the file server.properties, as sudo:

sudo su <Welcome01>

cp server.properties server2.properties

Edit the new file server2.properties. We will use this file to start a second broker. This second broker needs to have its own identifier (brokerId), log file directory (log.dirs) and its own port to listen on (listeners).

vi server2.properties

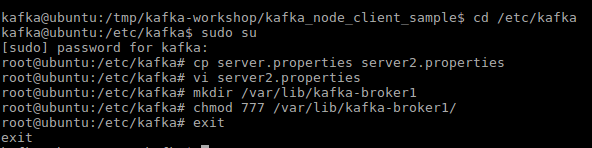


create directory

mkdir /var/lib/kafka-broker1

make directory accessible (the easy way)

chmod /var/lib/kafka-broker1

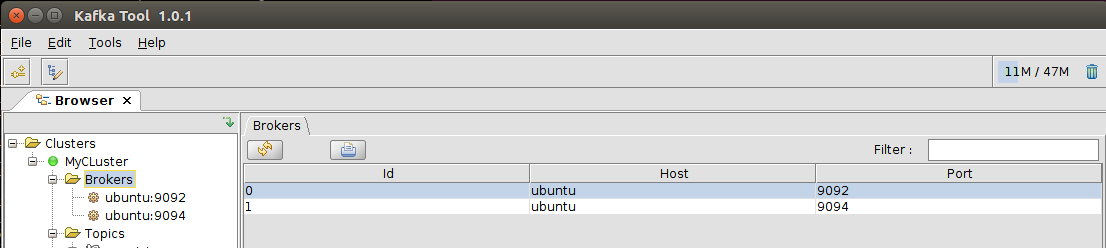


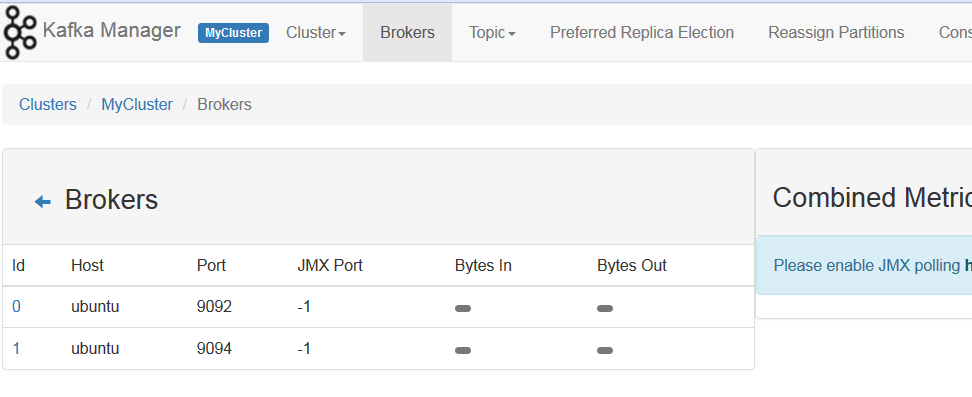
start both brokers:

kafka-server-start /etc/kafka/server.properties

kafka-server-start /etc/kafka/server2.properties

Verify in kafkatool and/or kafka-manager:





Create a topic with two partitions and replication across the two brokers:

kafka-topics --create --zookeeper localhost:2181 --replication-factor 2 --partitions 1 --topic super-safe

Prepare to produce some messages to the new topic:

kafka-console-producer --broker-list localhost:9092, localhost:9094 --topic super-safe

and enter on separate lines (and therefore in separate messages):

1

2

In a new terminal window, start consumption from topic super-safe:

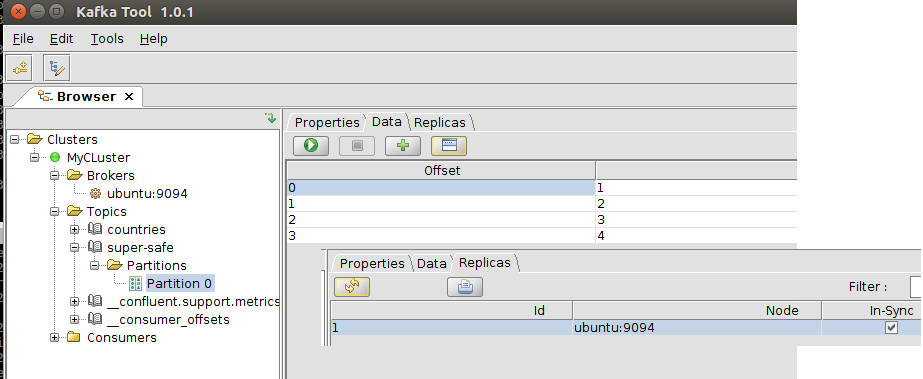
kafka-console-consumer --bootstrap-server localhost:9092 --from-beginning --topic super-safe

Kill the first broker started from server.properties.

Enter two more messages into the producer window:

3

4



Start the first broker again:

kafka-server-start /etc/kafka/server.properties

Produce messages

5

6

Now kill the second broker. Note: it can be hard to kill the broker. You may have to use brute force:

sudo su <Welcome01>

ps –ef | grep server2.properties

kill -9 <process id>

Enter two more messages:

7

8

And start the second broker.

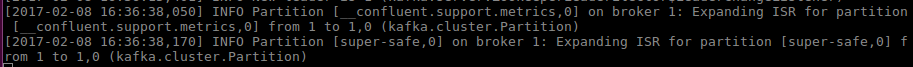
kafka-server-start /etc/kafka/server2.properties

Produce two messages while it is starting

9

10

When it is running (after the message on ISR is reported for the second broker)



kill the first broker once more.

Produce messages

11

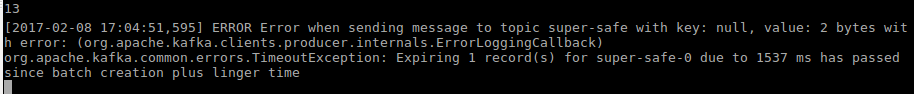
12

Now kill the second broker. Both brokers are down at this point.

Try to produce another message:

13

This will fail – as no broker remains to take your call.



Did you manage to lose a message during all the time that at least one broker was running? When you start a new consumer – it will be able to consume all messages, from the beginning, regardless of which broker(s) were running when that message was produced and regardless of which broker is currently running.

Start the first broker:

kafka-server-start /etc/kafka/server.properties

Messages 3,4, 11 and 12 were all produced while the first broker was not running. At this moment, it is the only broker running.

Now in a fresh terminal, consume all messages on topic super-safe, from the beginning:

kafka-console-consumer --bootstrap-server localhost:9092 --from-beginning --topic super-safe

If this statement leads to consumption of the messages 3, 4, 11 and 12, we have proven our point of fail over and high availability.

# Kafka REST

Start Kafka Server and Kafka REST

kafka-server-start /etc/kafka/server.properties

kafka-rest-start

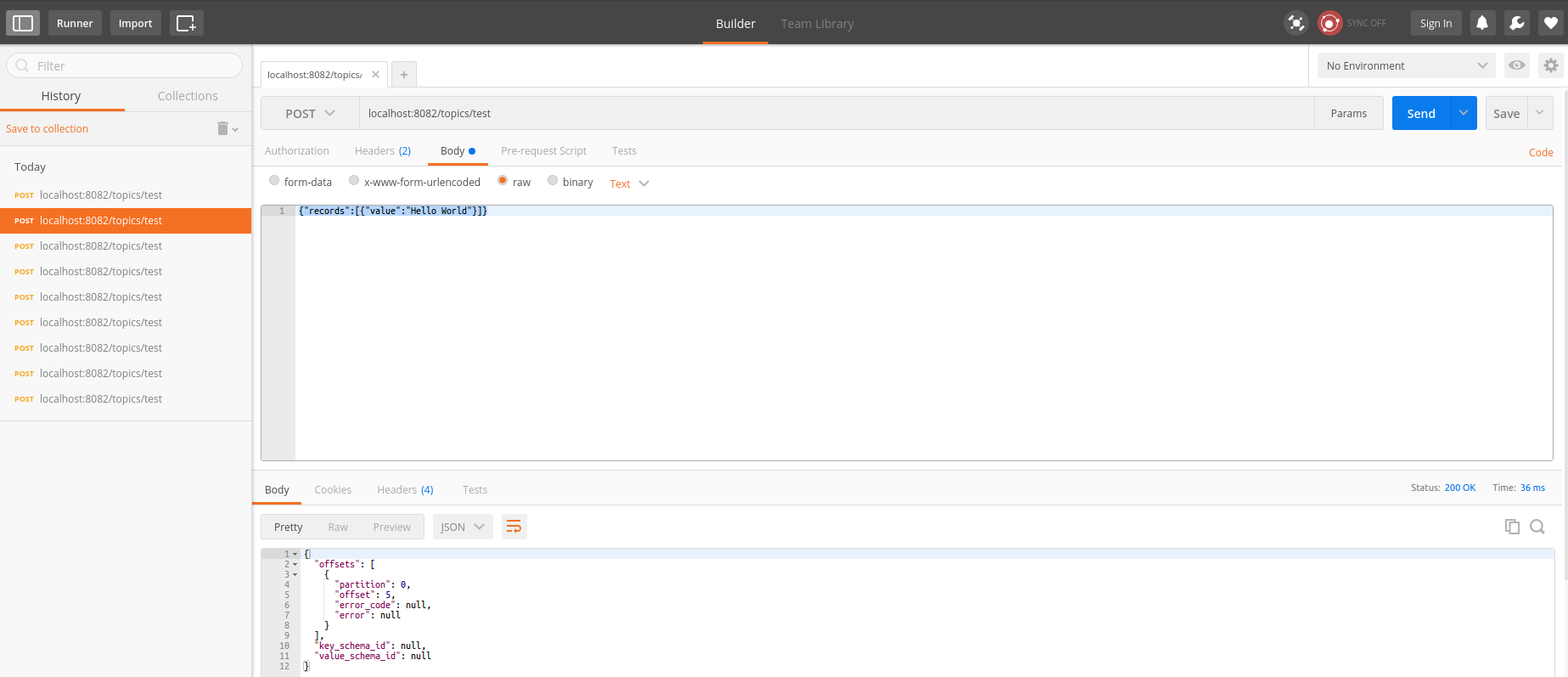
## Produce a message

Start Postman. Select POST and input localhost:8082/topics/test as URL

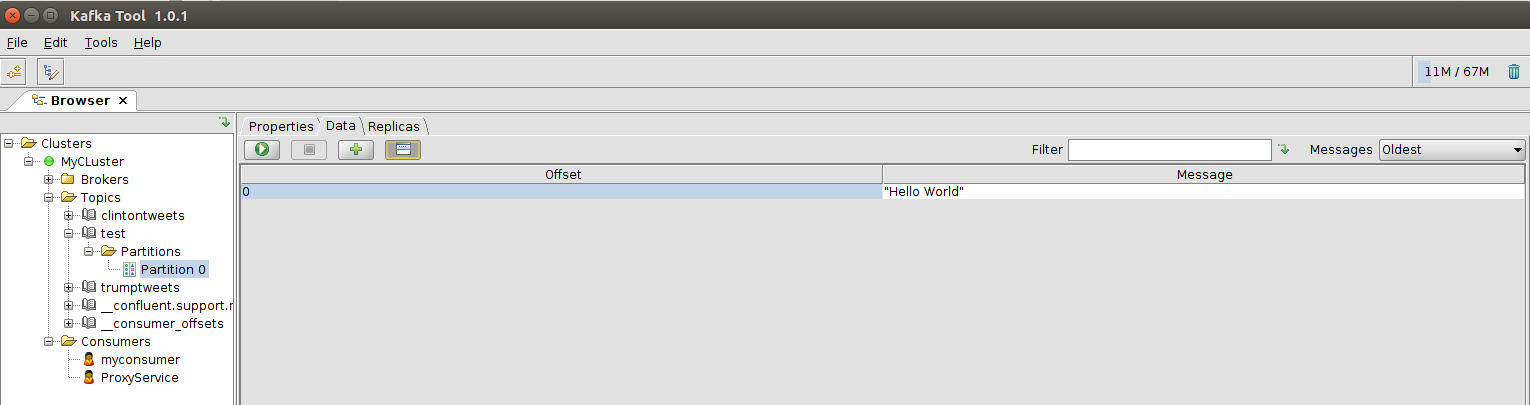
Add a Header: Content-Type application/vnd.kafka.json.v1+json

Use the body: {"records":[{"value":"Hello World"}]}

Click send



Start Kafkatool and confirm the message has arrived.



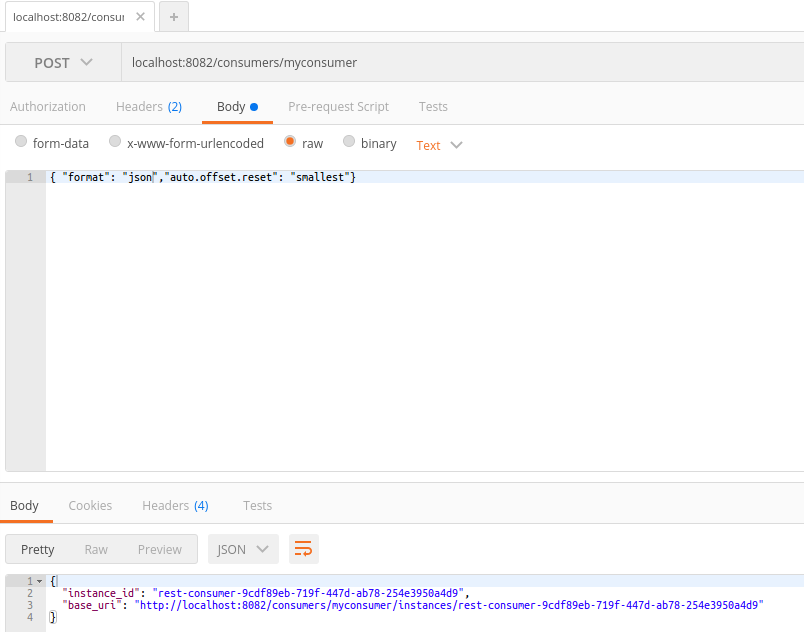
## Create a consumer

Go back to Postman

Do a POST request to localhost:8082/consumers/myconsumer

Use the following body

{ "format": "json","auto.offset.reset": "smallest"}



The response will be something like:

{

"instance\_id": "rest-consumer-a3f46a50-6fbf-4c5d-b48a-c576af53ecc6",

"base\_uri": "http://localhost:8082/consumers/myconsumer/instances/rest-consumer-a3f46a50-6fbf-4c5d-b48a-c576af53ecc6"

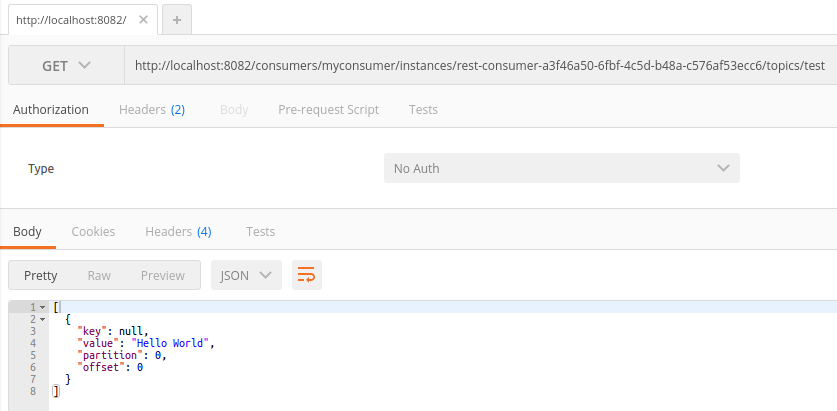
}

Do a GET request on (rest-consumer part is copied from the response below):

http://localhost:8082/consumers/myconsumer/instances/rest-consumer-a3f46a50-6fbf-4c5d-b48a-c576af53ecc6/topics/test

Specify Accept header application/vnd.kafka.json.v1+json

Confirm you can receive the response:



Repeat the request

Confirm you are not receiving new messages



## Bonus

Repeat this exercise with binary messages. See <https://github.com/confluentinc/kafka-rest> for inspiration.

Repeat the exercise using Node.js and kafka-rest-node. See <https://github.com/confluentinc/kafka-rest-node>.

# Kafka Streams

A brief introduction to Kafka Streams

## Preparation – publish Country Messages from node client

In a terminal window in the VM, go to directory kafka-workshop/kafka-node-countries.

Type

npm install

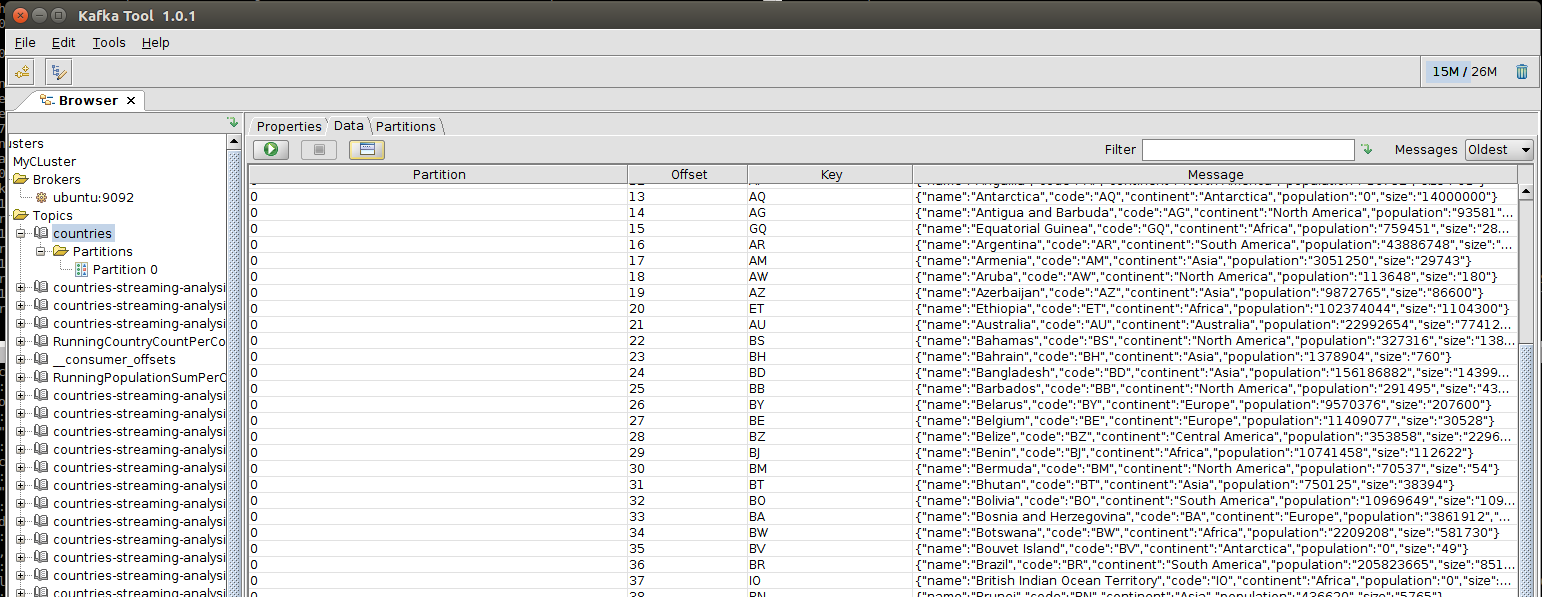
to have the node modules installed that are required by the Node producer to Kafka.

Next, run the producer:

node KafkaCountryProducer.js

The Node application starts publishing country messages to topic *countries*. The countries are read from the file countries2.csv.

You can check the country messages in Kafka Tool:



Note: when you inspect the JavaScript code, you will see how you can change the speed with which the messages are produced.

## Preparation 2 – Install maven on VM

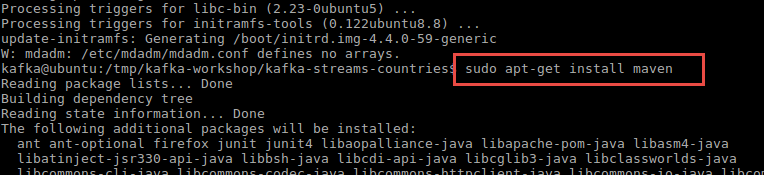
Install Maven on the VM. (See for example instructions here: <http://howtoprogram.xyz/2016/09/08/install-maven-ubuntu-16-04-lts-xenial-xerus/> ).

Type

sudo apt-get update

and provide the root password (Welcome01).

After some time, the process completes.



Next, type:

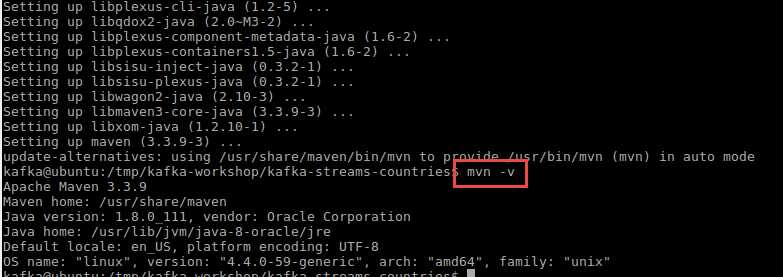
sudo apt-get install maven

Confirm that you want to continue.

Quite a few files are downloaded and installed. This process will take a few minutes.

We can verify whether Maven is installed successfully or not by type command:

mvn –v



## DIY Streams in node.js

Streaming analysis entails analyzing – filtering, aggregating, enriching, routing – events as they stream in. Hanging on to what is still required, letting go of what can be discarded.

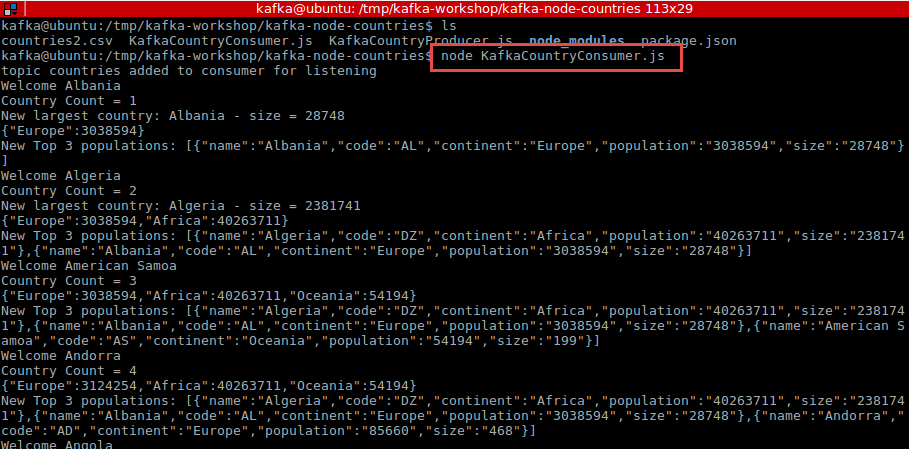
In a new terminal window in the VM, go to directory kafka-workshop/kafka-node-countries.

Run the KafkaCountryConsumer:

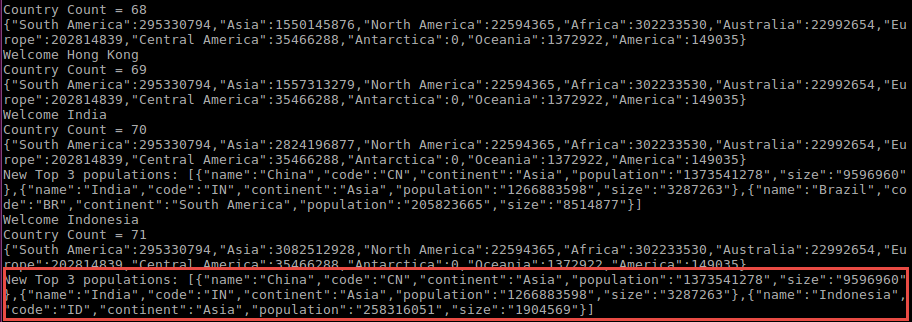
node KafkaCountryConsumer.js

The Node module runs and will start printing the results of streaming analysis – running counts and top-N evaluations – to the console. The country messages that are produced by the KafkaCountryProducer application are consumed and evaluated.

The screenshot shows an overall country count of 4, the population sums thusfar for the continents for which countries have been processed and the current top 3 – after each new country that may upset the previous standings – of most populous countries in the world. Until now, Algeria has the largest population.



After some time, a new Top 3 has been determined:



You can take a look at the source code of KafkaCountryConsumer – to see how this streaming analysis has been implemented, in a DIY way.



It is not very complex. It is a little clumsy. It can quickly get complicated when we start adding evaluations, enrichments, filters and more. There is no easy recognition of patterns and now easy reuse of code for similar operations. Kafka Streams will offer most what is lacking in this implementation.

See the next section for an introduction to Kafka Streams.

## Run Kafka Streams Application

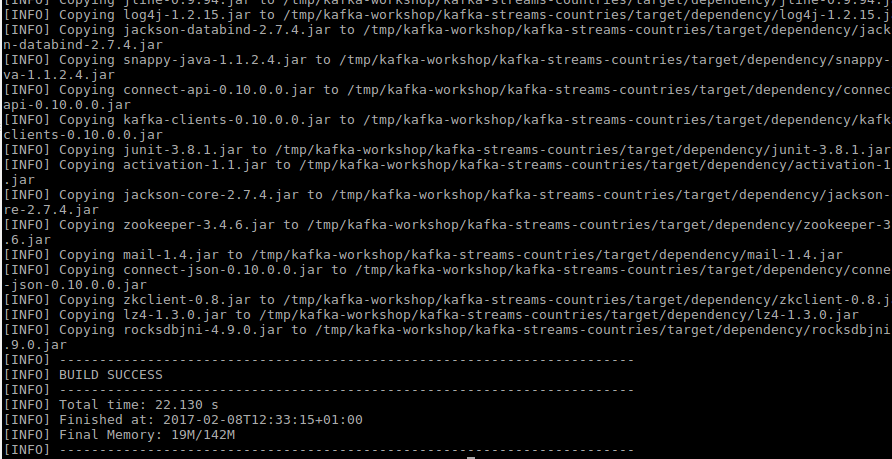
In a different terminal window in the VM, go to directory kafka-workshop/kafka-streams-countries.



Type

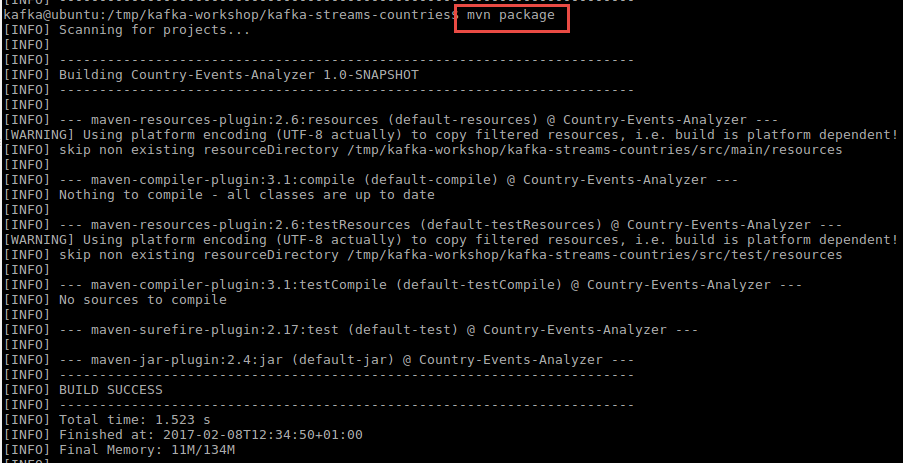
mvn install dependency:copy-dependencies

This will install a number of JAR files for the libraries and framework the Kafka Streams application depends upon.



Then type:

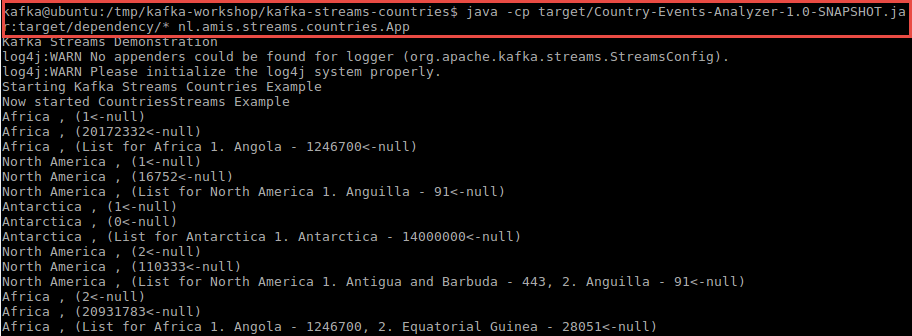
mvn package



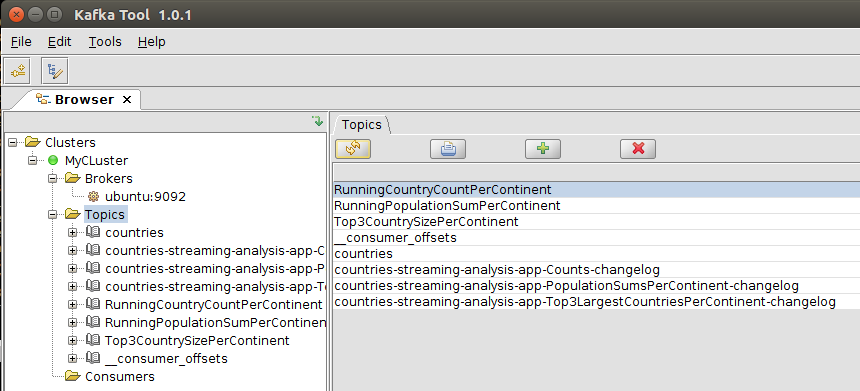
This will build the application – creating the JAR file with the runnable class.

To run the application, type:

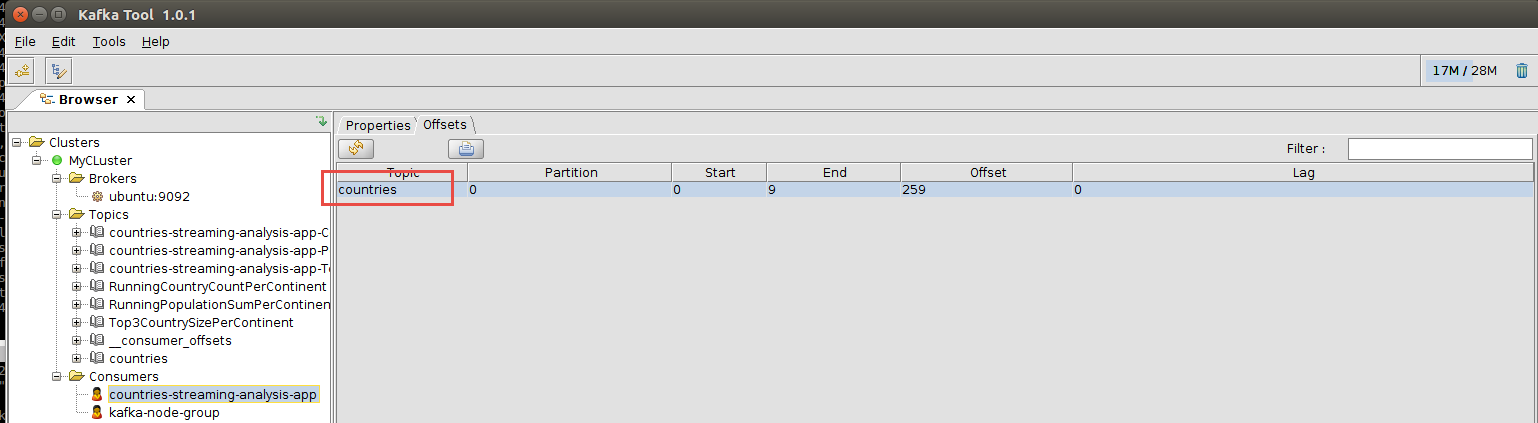
java -cp target/Country-Events-Analyzer-1.0-SNAPSHOT.jar:target/dependency/\* nl.amis.streams.countries.App



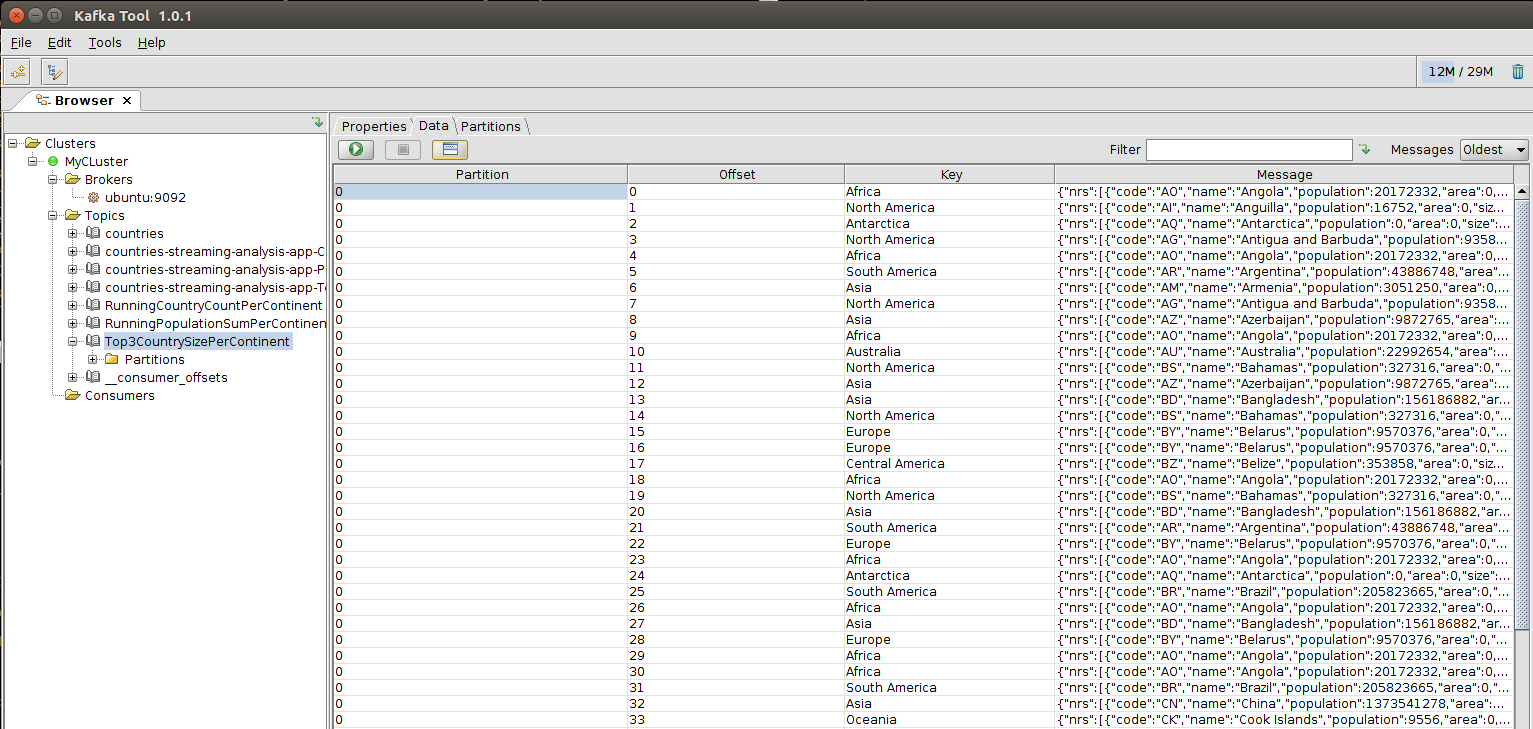
Check in Kafka Tool what happened in terms of topics. Note: Kafka Streams uses Kafka Topics as the intermediate stages for streams.



Also check the node Consumers. You will find a new consumer for the countries-streaming-analysis-app (this label is defined in the Java application). Kafka keeps track of the offset per partition for this consumer, even though that is the responsibility and the prerogative of the consumer as well.



You can inspect the messages published for example to Top3CountrySizePerContinent (both key and message are of type String):



You will see messages that represent the latest status of the Top3 per continent. Each time a message appears for Africa, something has changed in the Top 3 for Africa – or: a country message was published for a country that was large enough to make it into the top3 until now. Note that this country may not be in the top3 in the near future, when even bigger countries make an appearance.

Now it is interesting to inspect the Java code that constitutes this Kafka Streams application. What is done to make the logic rock.

When you strip it down, the core logic for extracting the running country count per continent is not much more than:

KStreamBuilder kStreamBuilder = new KStreamBuilder();

// the source of all streaming analysis is the topic with country messages  
KStream<String, CountryMessage> countriesStream = kStreamBuilder.stream(stringSerde, countryMessageSerde, "countries");

// running count of countries per continent, published in topic RunningCountryCountPerContinent  
KTable<String,Long> runningCountriesCountPerContinent =   
 countriesStream  
.selectKey((k, country) -> country.continent)  
.countByKey("Counts")  
;  
// publish running count to Kafka topic RunningCountryCountPerContinent  
runningCountriesCountPerContinent.toStream().to(stringSerde, longSerde, "RunningCountryCountPerContinent");  
// print results to the condole  
runningCountriesCountPerContinent.print(stringSerde, longSerde);

The running sum of combined populations per continent is calculated and published to Kafka Topic *RunningPopulationSumPerContinent*.

// running sum of population sizes per continent, published in topic RunningPopulationSumPerContinent

// selectKey assigns the key to group by to each record, in this case continent

// the aggregation itself is a simple summation of populations (per continent)

KTable<String,Integer> runningPopulationSumPerContinent = countriesStream

.selectKey((k, country) -> country.continent)

.aggregateByKey(

() -> { return 0;}

, (continent, countryMsg, aggregate) -> {

return aggregate + countryMsg.population;

}

, stringSerde, integerSerde

, "PopulationSumsPerContinent"

);

runningPopulationSumPerContinent.print(stringSerde,integerSerde);

runningPopulationSumPerContinent.toStream().to(stringSerde, integerSerde, "RunningPopulationSumPerContinent");

The Kafka Streams logic to derive the (running) Top3 biggest countries per continent is little more complex, but not a whole lot.:

static public class CountryTop3 {

public CountryMessage[] nrs = new CountryMessage[4] ;

public CountryTop3() {}

}   
  
// top 3 largest countries per continent, published top topic , published to topic Top3CountrySizePerContinent

KTable<String,CountryTop3> top3PerContinent = countriesStream

.selectKey((k, country) -> country.continent)

.aggregateByKey(

CountryTop3::new

, (continent, countryMsg, top3) -> {

top3.nrs[3]=countryMsg;

// sort the array by country size

Arrays.sort(

top3.nrs, (a, b) -> {

if (a==null) return 1;

if (b==null) return -1;

return Integer.compare(b.size, a.size);

}

);

// lose nr 4, only top 3 is relevant

top3.nrs[3]=null;

return (top3);

}

, stringSerde, countryTop3Serde

, "Top3LargestCountriesPerContinent");

// publish to Kafka Topic  
top3PerContinent.to(stringSerde, countryTop3Serde, "Top3CountrySizePerContinent");  
  
// and print to the console:  
top3PerContinent.<String>mapValues((top3) -> {

String rank = " 1. "+top3.nrs[0].name+" - "+top3.nrs[0].size

+ ((top3.nrs[1]!=null)? ", 2. "+top3.nrs[1].name+" - "+top3.nrs[1].size:"")

+ ((top3.nrs[2]!=null) ? ", 3. "+top3.nrs[2].name+" - "+top3.nrs[2].size:"")

;

return "List for "+ top3.nrs[0].continent +rank;

}

)

.print(stringSerde,stringSerde);

### Streams Resources

<http://codingjunkie.net/kafka-streams-part2/>

Example: <https://github.com/gwenshap/kafka-examples/blob/master/KafkaStreamsAvg/src/main/java/com/shapira/examples/kstreamavg/StreamingAvg.java>

<https://github.com/JohnReedLOL/kafka-streams/blob/master/src/main/java/io/confluent/examples/streams/WordCountLambdaExample.java>

KStream Documentation: <https://kafka.apache.org/0100/javadoc/org/apache/kafka/streams/kstream/KStream.html>

# References

Kafka OSA:

<https://www.rittmanmead.com/blog/2016/07/stream-analytics-processing-kafka-oracle-stream-analytics/>

Network access

<https://technology.amis.nl/2017/01/29/network-access-to-ubuntu-virtual-box-vm-from-host-laptop/>