Last Updated: 15 Nov 2020
https://docs.confluent.io/current/ksql/docs/tutorials/examples.html#ksql-examples

1. Installing Kafka - Quick Start (Local)

Demonstrates both the basic and most powerful capabilities of Confluent Platform, including using Control Center for topic management and event stream processing using KSQL. In this quick start you create Apache Kafka® topics, use Kafka Connect to generate mock data to those topics, and create KSQL streaming queries on those topics. You then go to Control Center to monitor and analyze the streaming queries.
Start the VM using VM player and Logon to the server using telnet or directly in the VM console. Enter the root credentials to logon.
You need to install java before installing zookeeper and Kafka. All the necessary software will be in the /Software folder. If its not there ensure to copy it using winscp.exe from the windows desktop to /Software folder. You can create /Software folder using mkdir /Software.
Installing Java
#tar -xvf jdk-8u45-linux-x64.tar.gz -C /opt
Set in the path variable and JAVA_HOME
Include in the profile as follow

```
[root@tos opt]# more ~/.bashrc
# .bashrc

# User specific aliases and functions

alias rm='rm -i'
alias cp='cp -i'
alias mv='mv -i'

export JAVA_HOME=/opt/jdk1.8.0_45

export PATH=$PATH:$JAVA_HOME/bin
# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
[root@tos opt]#
```

Installing a Kafka Broker
The following example installs Confluence Kafka in /apps, configured to use the
Zookeeper server started previously and to store the message log segments stored in /tmp/kafka-logs:
tar -zxf kafka_2.12-1.1.0.tgz –C /apps
mv kafka_2.12-1.1.0 /opt/kafka
mkdir /opt/data/kafka-logs
Decompress the file. You should have these directories:

```
(base) [root@tos confluent]#
(base) [root@tos confluent]# pwd
/apps/confluent
(base) [root@tos confluent]# ls -ltr
total 16
drwxr-xr-x.  3 life life   21 Jun  5 10:11 lib
drwxr-xr-x.  7 life life  106 Jun  5 10:42 share
drwxr-xr-x. 23 life life 4096 Jun  5 10:42 etc
drwxr-xr-x.  3 life life 4096 Jun  5 10:42 bin
drwxr-xr-x.  2 life life  178 Jun  5 11:17 src
-rw-r--r--.  1 life life  871 Jun  5 11:17 README
drwxr-xr-x.  2 root root 4096 Jul  7 02:01 logs
(base) [root@tos confluent]#
```

/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties

```
[root@tos opt]# /opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/s
erver.properties
[root@tos opt]# jps
3476 Kafka
3499 Jps
2895 QuorumPeerMain
[root@tos opt]#
```

#mkdir /opt/scripts
All the common execution scripts will be stored in the above folder.
The following scripts will start a zookeeper along with a broker. Create the follwong file and update with the following scripts. It will start the zookeeper and kafka broker using the mention script.

#vi startABroker.sh
/## Scripts Begin
###################
#!/usr/bin/env bash

Start Zookeeper.
/opt/zookeeper/bin/zkServer.sh start

#Start Kafka Server
/opt/kafka/bin/kafka-server-start.sh -daemon /opt/kafka/config/server.properties

echo "Started Successfully"

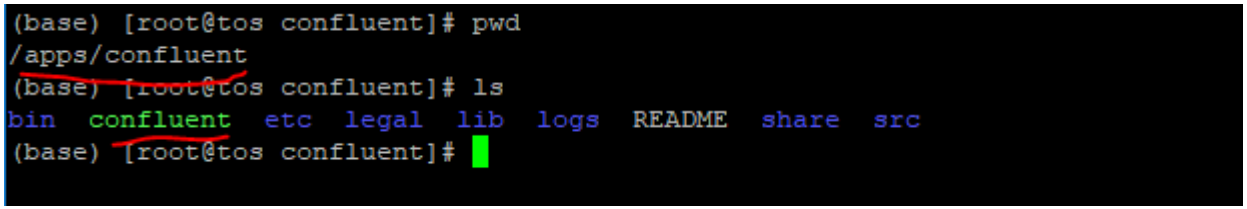
//## Scripts End
###################
To shutdown the Broker , find the process and kill.
ps -eaf | grep java
To Stop Zookeeper create the following script.
#vi /opt/scripts/stopZookeeper.sh
Update the following commands in the above script and save it.

#!/usr/bin/env bash
Stop Zookeeper.
/opt/zookeeper/bin/zkServer.sh stop
echo "Stop zookeeper Successfully"

Installing and Configuring the CLI

#tar -xvf confluent_latest_linux_amd64.tar.gz -C /apps

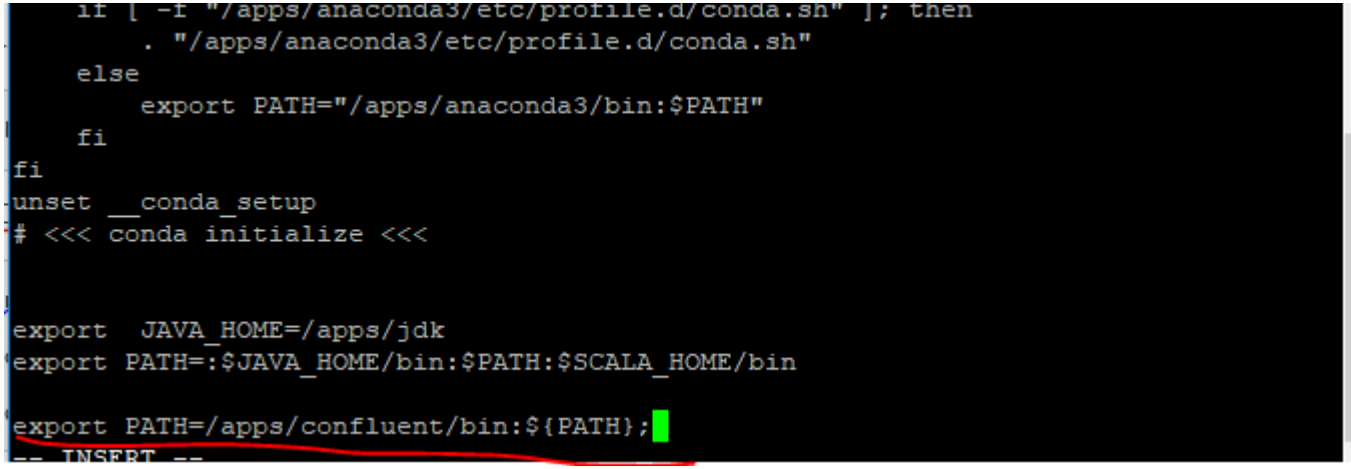
Run this script to install the Confluent CLI. This command creates a `bin` directory in your designated location (`<path-to-directory>/bin`). The location must be in your PATH (e.g. `/usr/local/bin`).



Set your PATH variable:

vi ~/.bashrc

```
export PATH=/apps/confluent/bin:${PATH};
```



If RBAC is enabled, you must log in to the Confluent CLI with the Metadata service URL specified (`<url>`). For more information, see confluent login.

```
confluent login --url <url>
Enter your Confluent credentials:
Email:
Password:
```

Your output should resemble:

```
Logged in as user
```

```
curl -L https://cnfl.io/cli | sh -s -- -b /<path-to-directory>/bin
```

Install the Kafka Connect Datagen source connector using the Confluent Hub client. This connector generates mock data for demonstration purposes and is not suitable for production. Confluent Hub is an online library of pre-packaged and ready-to-install extensions or add-ons for Confluent Platform and Kafka.
confluent-hub install --no-prompt confluentinc/kafka-connect-datagen:latest

```
(base) [root@tos ~]# cd /apps
(base) [root@tos apps]# confluent-hub install --no-prompt confluentinc/kafka-con
nect-datagen:latest
Running in a "--no-prompt" mode
Implicit acceptance of the license below:
Apache License 2.0
https://www.apache.org/licenses/LICENSE-2.0
Downloading component Kafka Connect Datagen 0.1.3, provided by Confluent, Inc. f
rom Confluent Hub and installing into /apps/confluent/share/confluent-hub-compon
ents
Adding installation directory to plugin path in the following files:
  /apps/confluent/etc/kafka/connect-distributed.properties
  /apps/confluent/etc/kafka/connect-standalone.properties
  /apps/confluent/etc/schema-registry/connect-avro-distributed.properties
  /apps/confluent/etc/schema-registry/connect-avro-standalone.properties
  /tmp/confluent.8A2Ii7O4/connect/connect.properties

Completed
(base) [root@tos apps]#
```

Add the install location of the Confluent `bin` directory to your PATH.

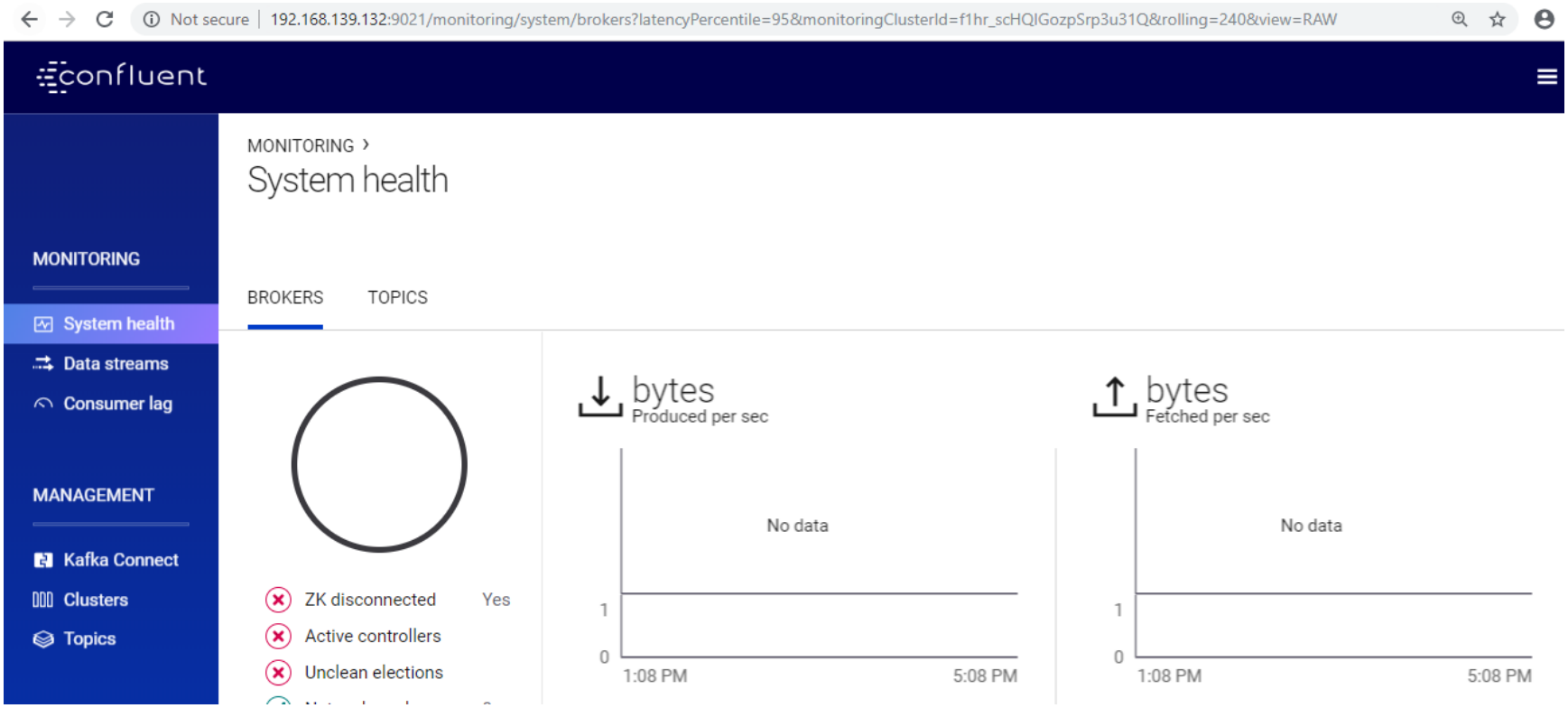
```
export PATH=/apps/confluent/bin:$PATH
```

```
unset __conda_setup
# <<< conda initialize <<<


export  JAVA_HOME=/apps/jdk
export PATH=:$JAVA_HOME/bin:$PATH:$SCALA_HOME/bin

export PATH=/apps/confluent/bin:${PATH};
export PATH=/apps/confluent/bin:$PAT
"~/.bashrc" 34L, 786C written
```

Start Confluent Platform using the Confluent CLI confluent local start command. This command starts all of the Confluent Platform components; including Kafka, ZooKeeper, Schema Registry, HTTP REST Proxy for Kafka, Kafka Connect, KSQL, and Control Center.
/apps/confluent/bin/confluent start

```
(base) [root@tos bin]# confluent start
This CLI is intended for development only, not for production
https://docs.confluent.io/current/cli/index.html

Using CONFLUENT_CURRENT: /tmp/confluent.8A2Ii7O4
Starting zookeeper
zookeeper is [UP]
Starting kafka
kafka is [UP]
Starting schema-registry
schema-registry is [UP]
Starting kafka-rest
kafka-rest is [UP]
Starting connect
connect is [UP]
Starting ksql-server
ksql-server is [UP]
Starting control-center
control-center is [UP]
(base) [root@tos bin]#
```

Navigate to the Control Center web interface at http://localhost:9021/.



Set the Data directory

```
export CONFLUENT_CURRENT=/opt/data/local
```

Install a Kafka Connector and Generate Sample Data
In this step, you use Kafka Connect to run a demo source connector called kafka-connect-datagen that creates sample data for the Kafka topics pageviews and users.

Run one instance of the Kafka Connect Datagen connector to produce Kafka data to the `pageviews` topic in AVRO format.

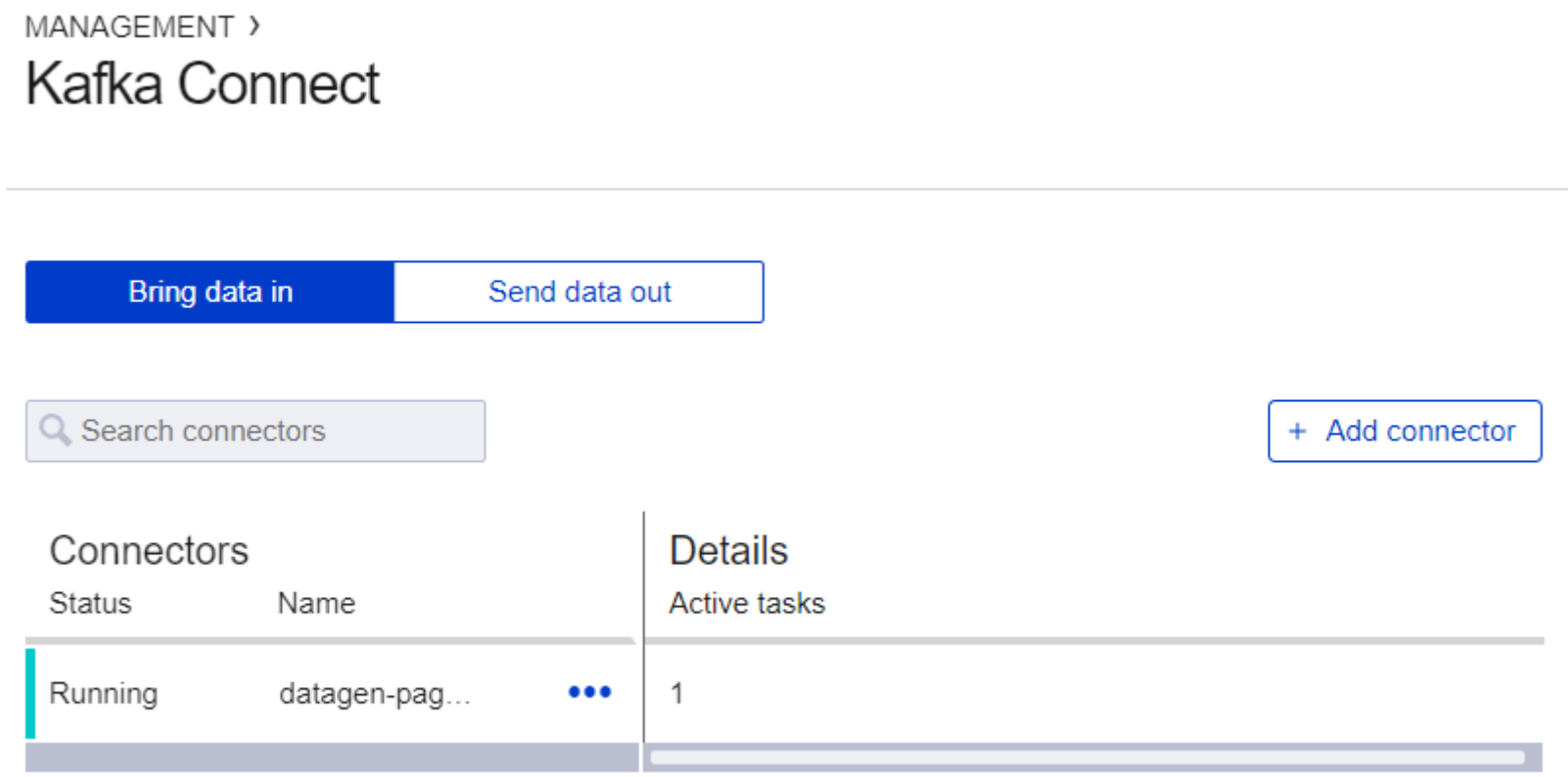
Management → **Add connector**.

Find the DatagenConnector tile and click **Connect**.

Name the connector `datagen-pageviews`. After naming the connector, new fields appear. Scroll down and specify the following configuration values:

- In the **Key converter class** field, type `org.apache.kafka.connect.storage.StringConverter`.
- In the **kafka.topic** field, type `pageviews`.
- In the **max.interval** field, type `100`.
- In the **iterations** field, type `1000000000`.
- In the **quickstart** field, type `pageviews`.

1. Click **Continue**.
2. Review the connector configuration and click **Launch**.



Run another instance of the Kafka Connect Datagen connector to produce Kafka data to the `users` topic in AVRO format.

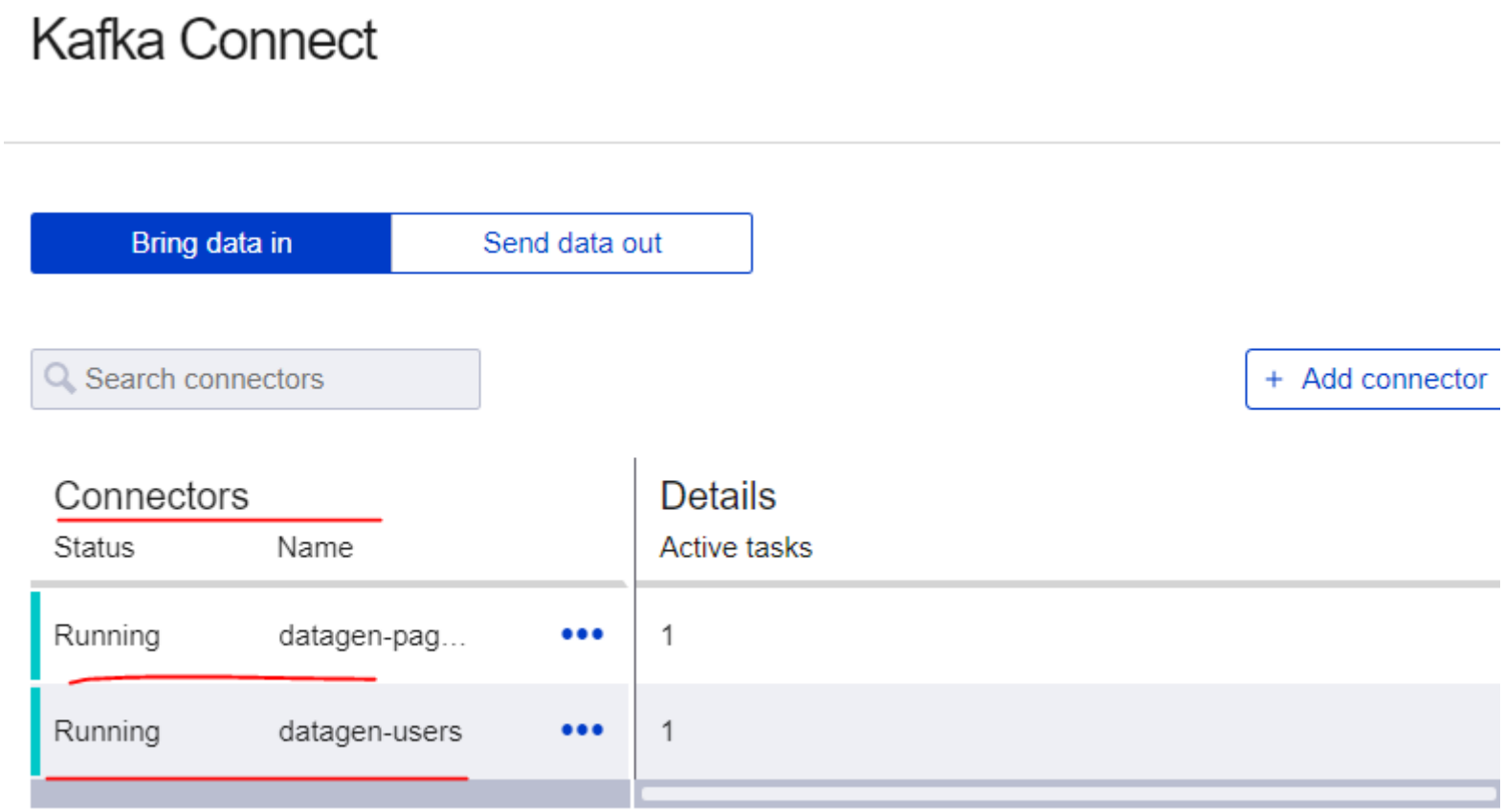
Click **Add connector**.

Find the DatagenConnector tile and click **Connect**.

Name the connector `datagen-users`. After naming the connector, new fields appear. Scroll down and specify the following configuration values:

- In the **Key converter class** field, type `org.apache.kafka.connect.storage.StringConverter`.
- In the **kafka.topic** field, type `users`.
- In the **max.interval** field, type `1000`.
- In the **iterations** field, type `1000000000`.
- In the **quickstart** field, type `users`.
  - Click **Continue**.
  - Review the connector configuration and click **Launch**.

At the end of this.


Kafka Connect
Bring data in
Send data out
Search connectors
+ Add connector
Connectors
Status
Name
Details
Active tasks
Running
datagen-pag...
1
Running
datagen-users
1

----------Lab Installation completes End here. ----------

2. Basic Kafka Operations - Command-line Tools - Partially

You need to start the broker using startABroker.sh. The script should be in /opt/scripts folder
#sh startABroker.sh
#jps

```
[root@tos scripts]# sh startABroker.sh
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
Started Successfully
[root@tos scripts]# jps
11665 Jps
11646 Kafka
11375 QuorumPeerMain
[root@tos scripts]#
```

Once the Kafka broker is started, we can verify that it is working by performing some simple operations against the broker; creating a test topic, producing some messages, and consuming the same messages.
Create and verify a details on topic:

/opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic test
/opt/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --describe --topic test

```
[root@tos opt]# /opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:21
81 --replication-factor 1 --partitions 1 --topic test
Created topic "test".
[root@tos opt]# /opt/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --desc
ribe --topic test
Topic:test      PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: test     Partition: 0    Leader: 0       Replicas: 0     Isr: 0
[root@tos opt]#
```

Produce messages to a test topic: It will open a console to send message to the topic, test. Enter some text as shown below.

/opt/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
Test Message 1
Test Message 2
^D
#

```
[root@tos config]# /opt/kafka/bin/kafka-console-producer.sh --broker-list localh
ost:9092 --topic test
>hi
>Hello
>TEst message
>[root@tos config]#
```

Consume messages from a test topic: As soon as you enter the following script in a separate terminal, You should be able to consume the messages that we have type in the producer console.

/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning

```
[root@tos config]# /opt/kafka/bin/kafka-console-consumer.sh -zookeeper localhost
:2181 --topic test --from-beginning
Using the ConsoleConsumer with old consumer is deprecated and will be removed in
 a future major release. Consider using the new consumer by passing [bootstrap-s
erver] instead of [zookeeper].
hi
Hello
TEst message
^CProcessed a total of 3 messages
```

Create, list and describe topics.

#/opt/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 -list

```
[root@tos ~]# /opt/kafka/bin/kafka-topics.sh --zookeeper localhost:2181 --list
__consumer_offsets
my-failsafe-topic
test
```

List and describe Topics

What does the tool do?

This tool lists the information for a given list of topics. If no topics are provided in the command line, the tool queries zookeeper to get all the topics and lists the information for them. The fields that the tool displays are - topic name, partition, leader, replicas, isr. Two optional arguments can be provided to the tool. If "under-replicated-partitions" is specified, the tool only provides information for those topic / partitions which have replicas that are under replicated. If "unavailable-partitions" is specified, the tool only provides information for those topic/partitions whose leader is not available.

How to use the tool?

List only single topic named "test" (prints only topic name)

/opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181 --topic test

List all topics (prints only topic names)

#/opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181

```
[root@tos scripts]# jps
12960 Jps
12314 Kafka
12043 QuorumPeerMain
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --list --zookeeper tos.master.com:2181 --topic
 test
test
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
CustomerCountry
__consumer_offsets
henry-topic
my-failsafe-topic
my-kafka-topic
my-kafka-topicl
test
test-topic
```

Describe only single topic named "test" (prints details about the topic)
#/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test

Describe all topics (prints details about the topics)
#/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181

```
[root@tos scripts]#
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic
test
Topic:test      PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: test     Partition: 0    Leader: 0       Replicas: 0     Isr: 0
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181
Topic:CustomerCountry   PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: CustomerCountry  Partition: 0    Leader: 1       Replicas: 1     Isr: 1
Topic:__consumer_offsets        PartitionCount:50       ReplicationFactor:1     Configs:segment.b
ytes=104857600,cleanup.policy=compact,compression.type=producer
        Topic: __consumer_offsets       Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 1    Leader: 2       Replicas: 2     Isr: 2
```

We will understand in details later in the tutorials.
Create Topics
What does the tool do?
By default, Kafka auto creates topic if "auto.create.topics.enable" is set to true on the server. This creates a topic with a default number of partitions, replication factor and uses Kafka's default scheme to do replica assignment. Sometimes, it may be required that we would like to customize a topic while creating it. This tool helps to create a topic and also specify the number of partitions, replication factor and replica assignment list for the topic.
How to use the tool?
create topic with default settings
/opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic topic1 --partitions 2 --replication-factor 1

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic to
picl --partitions 2 --replication-factor 1
Created topic "topicl".
[root@tos scripts]#
```

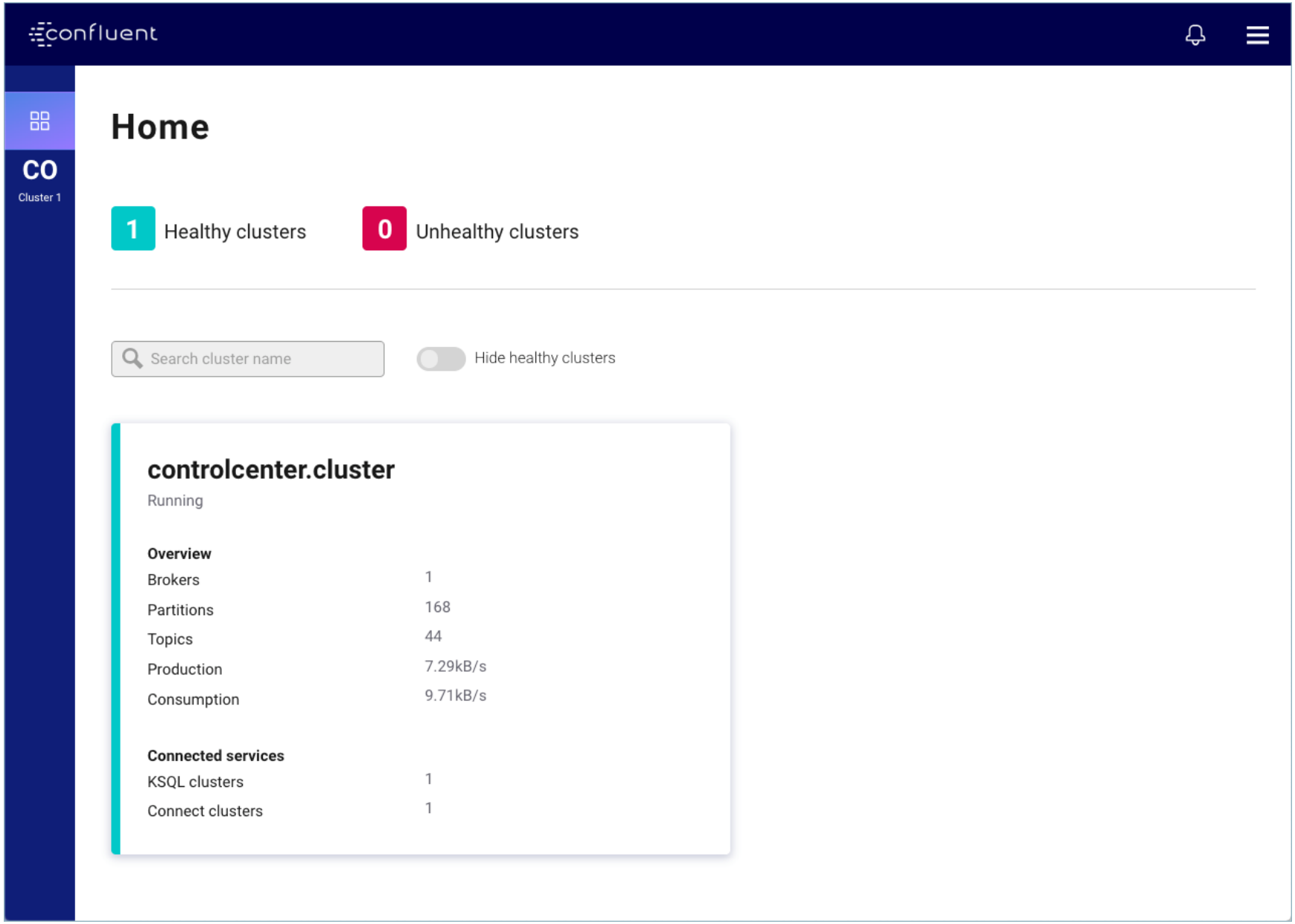
Create Kafka Topics

In this step, you create Kafka topics by using the Confluent Control Center. Confluent Control Center provides the functionality for building and monitoring production data pipelines and event streaming applications.

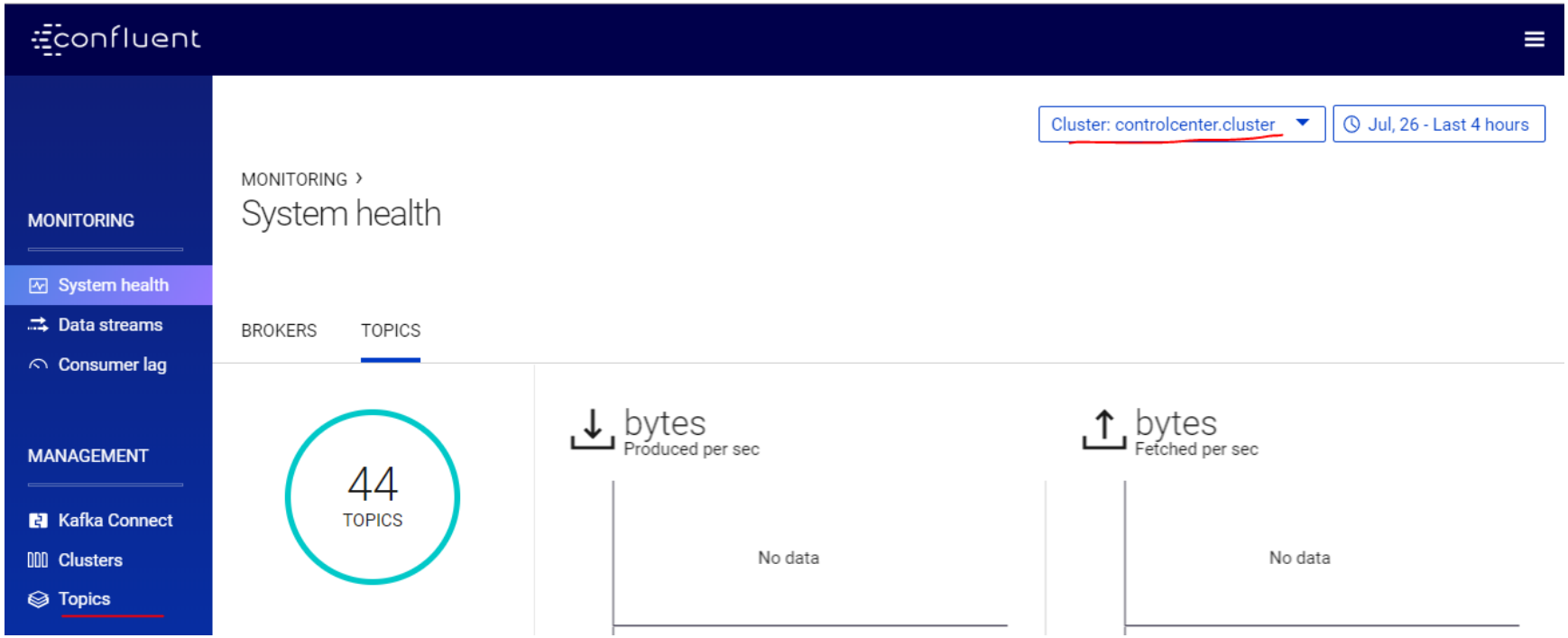
1. Navigate to the Control Center web interface at http://localhost:9021/.

Important

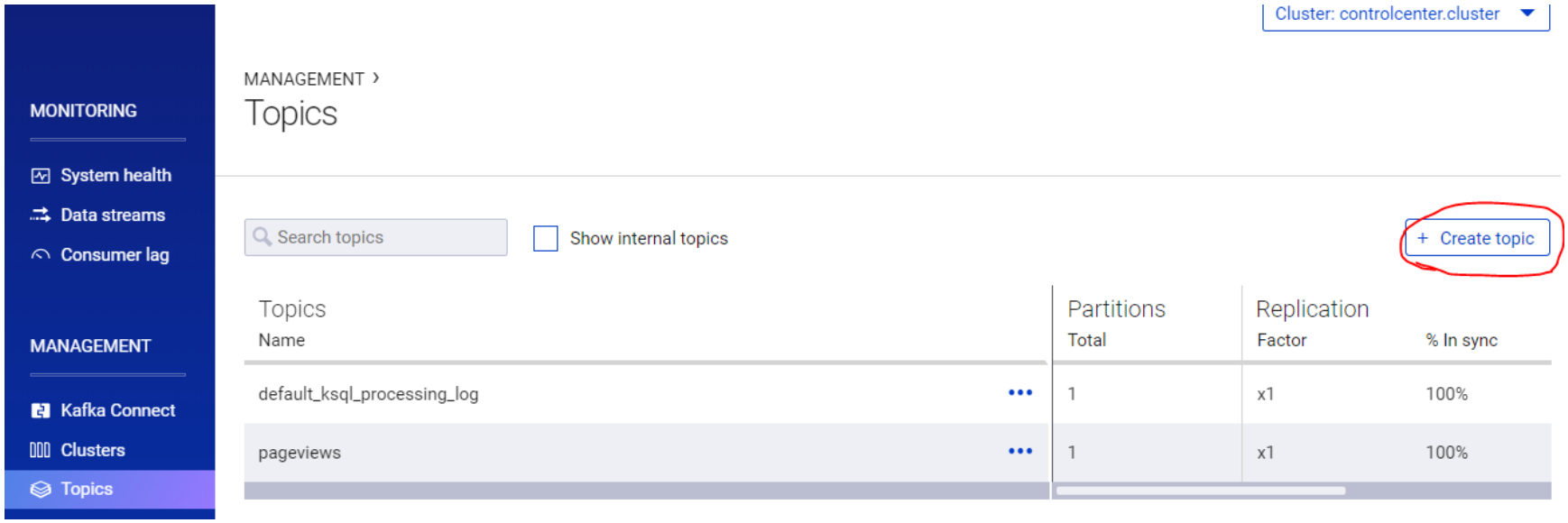
It may take a minute or two for Control Center to come online.


confluent
Home
CO
Cluster 1
1 Healthy clusters
0 Unhealthy clusters
Search cluster name
Hide healthy clusters
controlcenter.cluster
Running
Overview
Brokers 1
Partitions 168
Topics 44
Production 7.29kB/s
Consumption 9.71kB/s
Connected services
KSQL clusters 1
Connect clusters 1

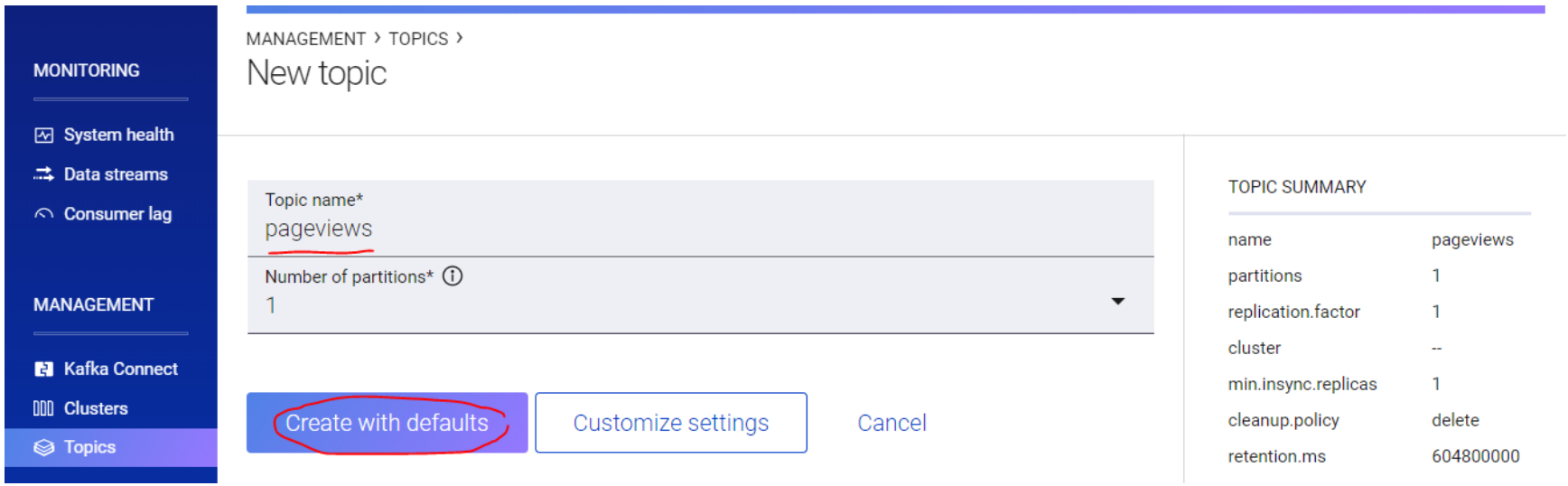
2. Select your cluster name.



3. Select **Topics** from the cluster submenu and click **Create topic**.



4. Create a topic named pageviews and click **Create with defaults**.



5. Repeat the previous steps and create a topic named `users` and click **Create with defaults**.

Lab CLI completes End here.

3. Writing a Kafka Producer in Java

In this tutorial, we are going to create a simple Java example that creates a Kafka producer. You need to create a new replicated Kafka topic called my-kafka-topic, then you will create a Kafka producer using Java API that uses this topic to send records. You will send records with the Kafka producer. You will send records synchronously.
You need to start the zookeeper and three nodes brokers before going ahead.

#cd /opt/scripts
#sh start3Brokers.sh

Here, as shown above three Kafka broker services and ZK service need to be started.

Create Replicated Kafka Topic
Create topic
#/opt/kafka/bin/kafka-topics.sh --create --replication-factor 3 --partitions 13 --topic my-kafka-topic --zookeeper localhost:2181

Above we created a topic named my-kafka-topic with 13 partitions and a replication factor of 3. Then we list the Kafka topics.

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create     --replication-fa
ctor 3    --partitions 13    --topic my-kafka-topic     --zookeeper  localhost:2
181
Created topic "my-kafka-topic".
[root@tos scripts]#
```

List created topics, you can verify the topic now
/opt/kafka/bin/kafka-topics.sh --list   --zookeeper localhost:2181

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --list    --zookeeper localhost:2181
__consumer_offsets
my-failsafe-topic
my-kafka-topic
test
[root@tos scripts]#
```

We will use maven to create the java project. (You can refer the Annexure I – How to create Maven project).

Maven java project Details:
Group ID: com.tos.kafka
Artifact ID: my-kafka-producer.

After you create a maven java project, include the following dependency.

```
<dependencies>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-clients</artifactId>
               <version>2.6.0</version>
          </dependency>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-streams</artifactId>
               <version>2.6.0</version>
          </dependency>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-streams-scala_2.13</artifactId>
               <version>2.6.0</version>
          </dependency>
   <dependency>
               <groupId>io.advantageous.boon</groupId>
               <artifactId>boon-json</artifactId>
               <version>0.6.6</version>
          </dependency>

          <dependency>
               <groupId>org.slf4j</groupId>
               <artifactId>slf4j-ext</artifactId>
               <version>1.7.13</version>
          </dependency>

          </dependencies>
   <build>
          <plugins>
               <plugin>
```

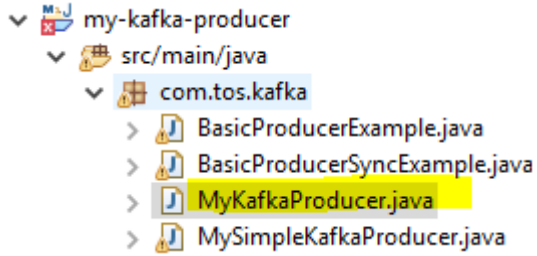
```
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

        </plugins>


</build>
```

Construct a Kafka Producer
To create a Kafka producer, you will need to pass it a list of bootstrap servers (a list of Kafka brokers). You will also specify a client.id that uniquely identifies this Producer client. In this example, we are going to send messages with ids. The message body is a string, so we need a record value serializer as we will send the message body in the Kafka's records value field. The message id (long), will be sent as the Kafka's records key. You will need to specify a Key serializer and a value serializer, which Kafka will use to encode the message id as a Kafka record key, and the message body as the Kafka record value.

Create a java class with the following package name:

Class : MyKafkaProducer
Package Name : com.tos.kafka

At the end of this lab, you will have a project structure as shown below:



Next, we will import the Kafka packages and define a constant for the topic and a constant to define the list of bootstrap servers that the producer will connect.

Add the following imports in your code.

```
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.LongSerializer;
import org.apache.kafka.common.serialization.StringSerializer;
import java.util.Properties;
```

Notice that we have imports LongSerializer which gets configured as the Kafka record key serializer, and imports StringSerializer which gets configured as the record value serializer.

Then, let us define some constant variables as stated below,
BOOTSTRAP_SERVERS is set to tos.master.com:9092, tos.master.com:9093, tos.master.com:9094 which is the three Kafka servers that we started up in the last lesson. Go ahead and make sure all three Kafka servers are running.

The constant TOPIC is set to the replicated Kafka topic that we just created.

Add the following variables in your code.

```
private final static String TOPIC = "my-kafka-topic";
  private final static String BOOTSTRAP_SERVERS =
        "tos.master.com:9092,tos.master.com:9093,tos.master.com:9094";
```

Create Kafka Producer to send records
Now, that we imported the Kafka classes and defined some constants, let's create a Kafka producer.
Add the following function in the code.

```
private static Producer<Long, String> createProducer() {
      Properties props = new Properties();

props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,BOOTSTRAP_SERVERS);
      props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");

props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,LongSerializer.class.getName());
      props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                                 StringSerializer.class.getName());
      return new KafkaProducer<>(props);
   }
```

To create a Kafka producer, you use **java.util.Properties** and define certain properties that we pass to the constructor of a **KafkaProducer**.

Above **createProducer** sets the **BOOTSTRAP_SERVERS_CONFIG** ("bootstrap.servers) property to the list of broker addresses we defined earlier.
BOOTSTRAP_SERVERS_CONFIG value is a comma separated list of host/port pairs

that the Producer uses to establish an initial connection to the Kafka cluster. The producer uses of all servers in the cluster no matter which ones we list here. This list only specifies the initial Kafka brokers used to discover the full set of servers of the Kafka cluster. If a server in this list is down, the producer will just go to the next broker in the list to discover the full topology of the Kafka cluster.

The CLIENT_ID_CONFIG ("client.id") is an id to pass to the server when making requests so the server can track the source of requests beyond just IP/port by passing a producer name for things like server-side request logging.

The **KEY_SERIALIZER_CLASS_CONFIG** ("key.serializer") is a Kafka **Serializer** class for Kafka record keys that implements the Kafka Serializer interface. Notice that we set this to **LongSerializer** as the message ids in our example are longs.

The **VALUE_SERIALIZER_CLASS_CONFIG** ("value.serializer") is a Kafka **Serializer** class for Kafka record values that implements the Kafka **Serializer** interface. Notice that we set this to **StringSerializer** as the message body in our example are strings.

Send records synchronously with Kafka Producer
Kafka provides a synchronous send method to send a record to a topic. Let's use this method to send some message ids and messages to the Kafka topic we created earlier.

Add the following in your code.

```
static void runProducer(final int sendMessageCount) throws Exception {
        final Producer<Long, String> producer = createProducer();
        long time = System.currentTimeMillis();

        try {
            for (long index = time; index < time + sendMessageCount; index++) {
                final ProducerRecord<Long, String> record = new
ProducerRecord<>(TOPIC, index, "Hello Kafka " + index);

                RecordMetadata metadata = producer.send(record).get();

            long elapsedTime = System.currentTimeMillis() - time;
                System.out.printf("Sent record(key=%s value=%s) " +
"meta(partition=%d, offset=%d) time=%d\n",
                        record.key(), record.value(), metadata.partition(),
metadata.offset(), elapsedTime);


            }
        } catch (Exception e) {

            e.printStackTrace();
```

```
        } finally {
            producer.flush();
            producer.close();
        }
    }
```

The above just iterates through a for loop, creating a ProducerRecord sending an example message ("Hello Kafka " + index) as the record value and the for loop index as the record key. For each iteration, runProducer calls the send method of the producer (RecordMetadata metadata = producer.send(record).get()). The send method returns a Java Future.

The response RecordMetadata has 'partition' where the record was written and the 'offset' of the record in that partition.

Notice the call to flush and close. Kafka will auto flush on its own, but you can also call flush explicitly which will send the accumulated records now. It is polite to close the connection when we are done.

Running the Kafka Producer.

Next you define the main method.

Add the following imports in your code.

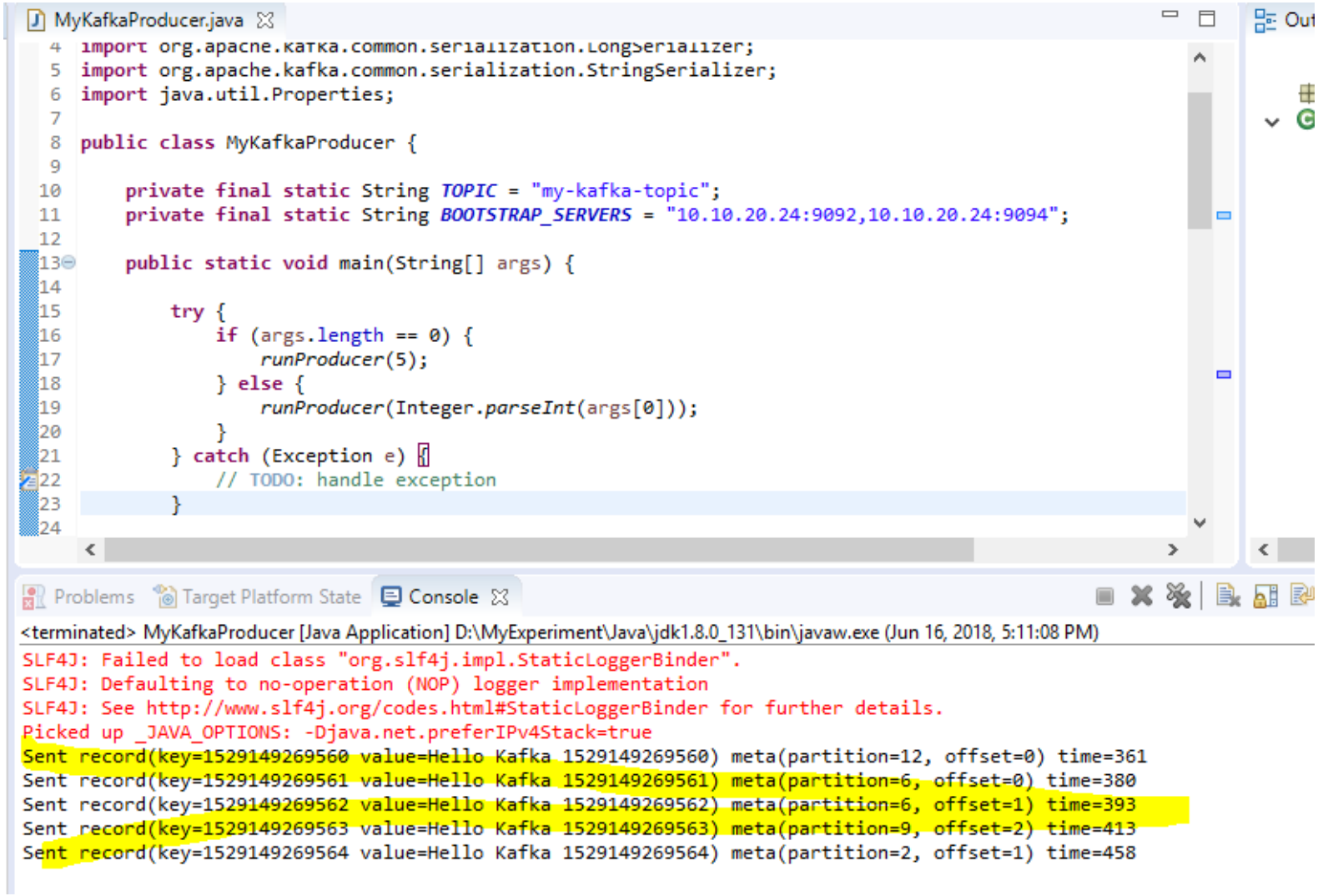
```
public static void main(String[] args) {

        try {
            if (args.length == 0) {
                runProducer(5);
            } else {
                runProducer(Integer.parseInt(args[0]));
            }
        } catch (Exception e) {
            // TODO: handle exception
        }

    }
```

The main method just calls runProducer.

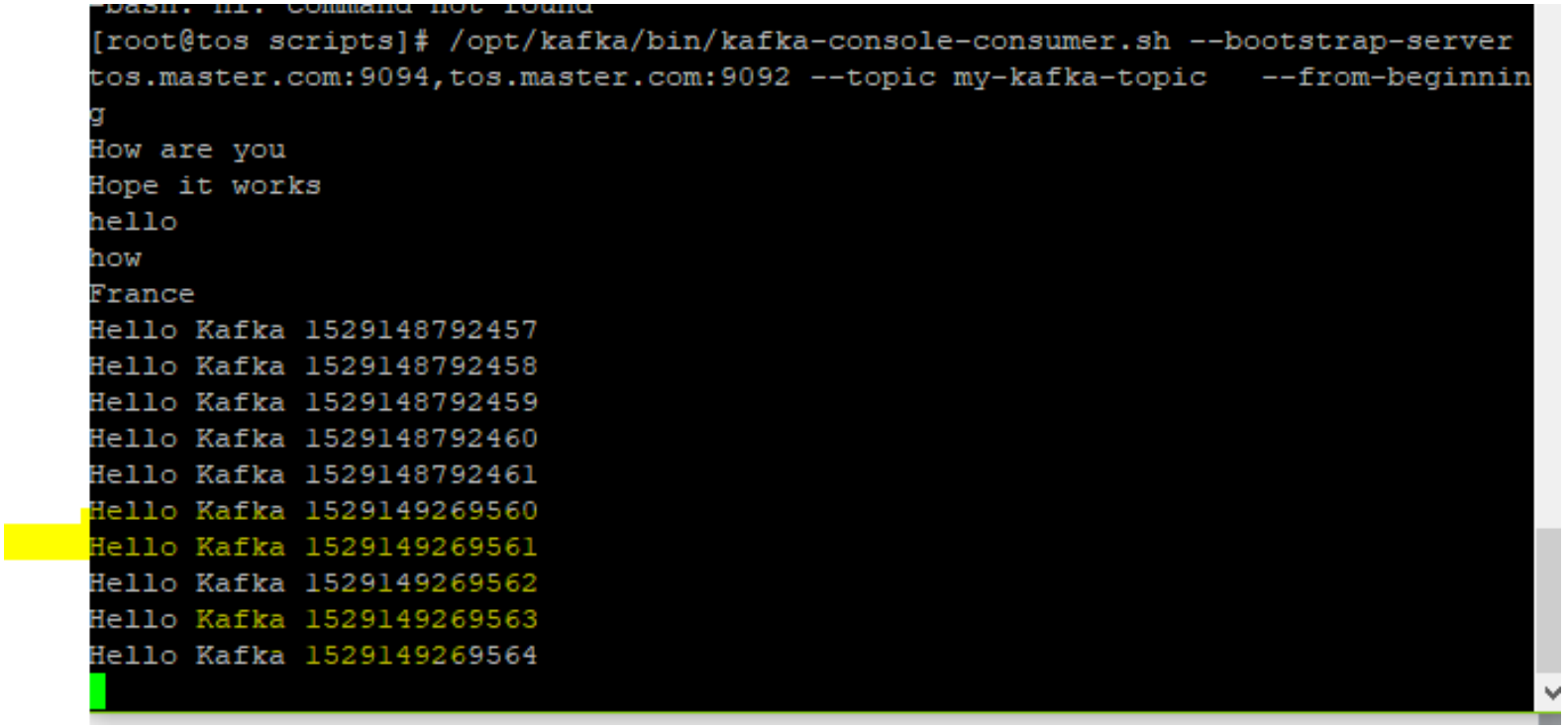
Start a consumer console so that you can consume the message sent by this producer.
#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server tos.master.com:9094,localhost:9092 --topic my-kafka-topic --from-beginning

```
[root@tos scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server
tos.master.com:9094,tos.master.com:9092 --topic my-kafka-topic   --from-beginnin
g
```

Execute the main program.



You should be able to view the message as shown above.
You can verify from the consumer console that whatever messages that were sent from the producer was consumed in the consumer console as shown below. In the next lab we will consume this from a Java client



Conclusion Kafka Producer example
We created a simple example that creates a Kafka Producer. First, we created a new replicated Kafka topic; then we created Kafka Producer in Java that uses the Kafka

replicated topic to send records. We sent records with the Kafka Producer using sync send method.

-------- ---------- ----------

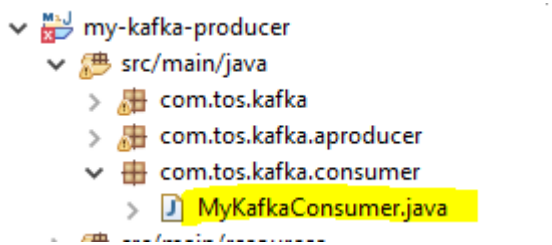
Lab End here

4. Consumer – Java API

In this tutorial, you are going to create simple *Kafka Consumer*. This consumer consumes messages from the Kafka Producer you wrote in the last tutorial. This tutorial demonstrates how to process records from a *Kafka topic* with a *Kafka Consumer*.
This tutorial describes how *Kafka Consumers* in the same group divide up and share partitions while each *consumer group* appears to get its own copy of the same data.
In the last tutorial, we created simple Java example that creates a Kafka producer. We also created replicated Kafka topic called my-example-topic, then you used the Kafka producer to send records (synchronously). Now, the consumer you create will consume those messages.
Construct a Kafka Consumer
Just like we did with the producer, you need to specify bootstrap servers. You also need to define a group.id that identifies which consumer group this consumer belongs. Then you need to designate a Kafka record key deserializer and a record value deserializer. Then you need to subscribe the consumer to the topic you created in the producer tutorial.
Kafka Consumer imports and constants

Next, you import the Kafka packages and define a constant for the topic and a constant to set the list of bootstrap servers that the consumer will connect.

KafkaConsumerExample.java - imports and constants
You can use the earlier java project. In that create a separate package and create the following class and package.
Package name : com.tos.kafka.consumer
MyKafkaConsumer.java




```
package com.tos.kafka.consumer;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.common.serialization.LongDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;
import java.util.Collections;
import java.util.Properties;

public class MyKafkaConsumer {
   private final static String TOPIC = "my-kafka-topic";
   private final static String BOOTSTRAP_SERVERS =
         "10.10.20.24:9092,10.10.20.24:9093,10.10.20.24:9094";

}
```

Notice that KafkaConsumerExample imports LongDeserializer which gets configured as the Kafka record key deserializer, and imports StringDeserializer which gets set up as the record value

deserializer. The constant BOOTSTRAP_SERVERS gets set to localhost:9092,localhost:9093,localhost:9094 which is the three Kafka servers that we started up in the last lesson. Go ahead and make sure all three Kafka servers are running. The constant TOPIC gets set to the replicated Kafka topic that you created in the last tutorial.

Create Kafka Consumer consuming Topic to Receive Records
Now, that you imported the Kafka classes and defined some constants, let's create the Kafka consumer.
KafkaConsumerExample.java - Create Consumer to process Records
Add the following method that will initialize the consumer parameters.

```
 private static Consumer<Long, String> createConsumer() {
              final Properties props = new Properties();
              props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
BOOTSTRAP_SERVERS);
              props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleConsumer");
              props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
LongDeserializer.class.getName());
              props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());

              // Create the consumer using props.
              final Consumer<Long, String> consumer = new KafkaConsumer<>(props);

              // Subscribe to the topic.
              consumer.subscribe(Collections.singletonList(TOPIC));
              return consumer;
}
```

To create a Kafka consumer, you use java.util.Properties and define certain properties that we pass to the constructor of a KafkaConsumer.
Above KafkaConsumerExample.createConsumer sets the BOOTSTRAP_SERVERS_CONFIG ("bootstrap.servers") property to the list of broker addresses we defined earlier. BOOTSTRAP_SERVERS_CONFIG value is a comma separated list of host/port pairs that the Consumer uses to establish an initial connection to the Kafka cluster. Just like the producer, the consumer uses of all servers in the cluster no matter which ones we list here.
The GROUP_ID_CONFIG identifies the consumer group of this consumer.
The KEY_DESERIALIZER_CLASS_CONFIG ("key.deserializer") is a Kafka Deserializer class for Kafka record keys that implements the Kafka Deserializer interface. Notice that we set this to LongDeserializer as the message ids in our example are longs.
The VALUE_DESERIALIZER_CLASS_CONFIG ("value.deserializer") is a Kafka Serializer class for Kafka record values that implements the Kafka Deserializer interface. Notice that we set this to StringDeserializer as the message body in our example are strings.
Important notice that you need to subscribe the consumer to the topic consumer.subscribe(Collections.singletonList(TOPIC));. The subscribe method takes a list of topics to subscribe to, and this list will replace the current subscriptions if any.
Process messages from Kafka with Consumer
Now, let's process some records with our Kafka Producer.

Add the following code that will process the message from the topic;

```
    static void runConsumer() throws InterruptedException {
            final Consumer<Long, String> consumer = createConsumer();

            final int giveUp = 100;
            int noRecordsCount = 0;

            while (true) {
                final ConsumerRecords<Long, String> consumerRecords = consumer.poll(1000);

                if (consumerRecords.count() == 0) {
                    noRecordsCount++;
                    if (noRecordsCount > giveUp)
                        break;
                    else
                        continue;
                }

                consumerRecords.forEach(record -> {
                    System.out.printf("Consumer Record:(%d, %s, %d, %d)\n", record.key(),
record.value(),
                                    record.partition(), record.offset());
                });

                consumer.commitAsync();
            }
            consumer.close();
```

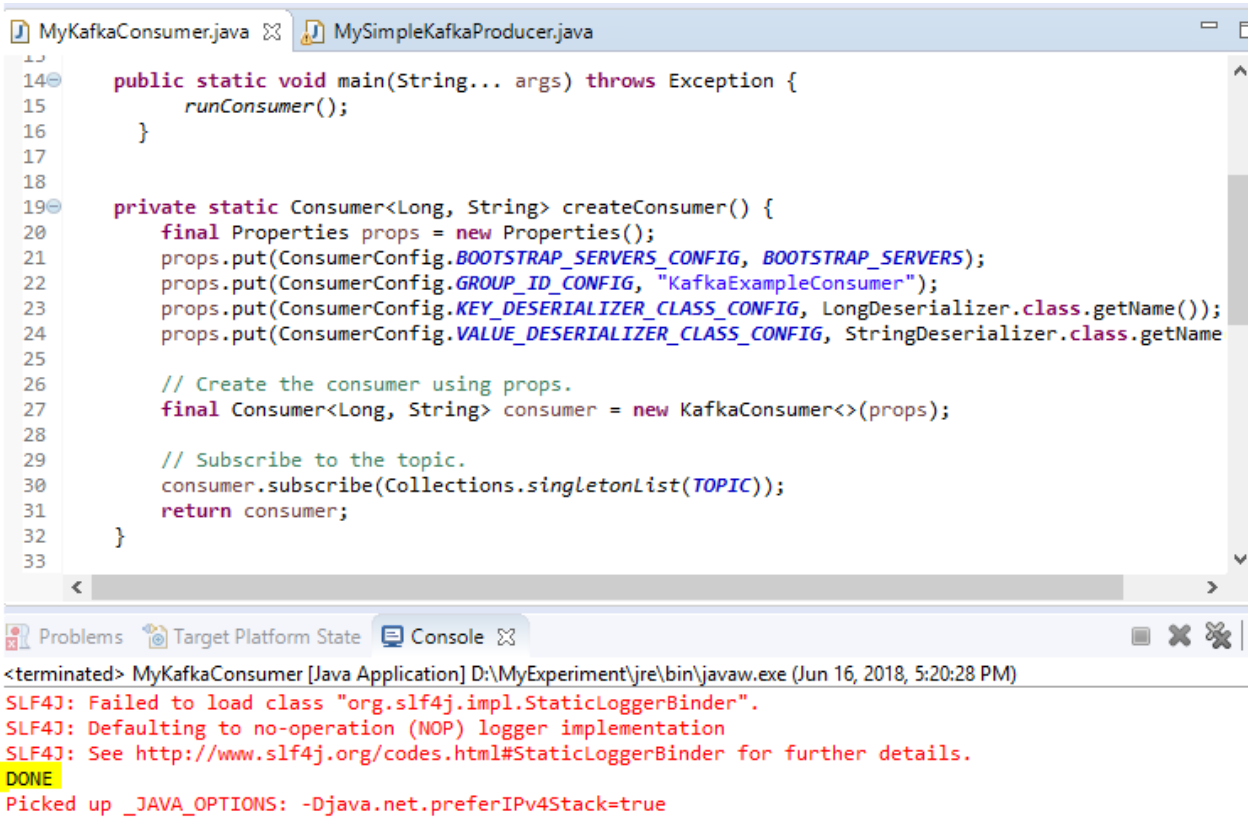
```
                System.out.println("DONE");
        }

}
```

Notice you use ConsumerRecords which is a group of records from a Kafka topic partition.
The ConsumerRecords class is a container that holds a list of ConsumerRecord(s) per partition for a particular topic. There is one ConsumerRecord list for every topic partition returned by a the consumer.poll().
Notice if you receive records (consumerRecords.count()!=0), then runConsumer method calls consumer.commitAsync() which commit offsets returned on the last call to consumer.poll(…) for all the subscribed list of topic partitions.
Kafka Consumer Poll method
The poll method returns fetched records based on current partition offset. The poll method is a blocking method waiting for specified time in seconds. If no records are available after the time period specified, the poll method returns an empty ConsumerRecords.
When new records become available, the poll method returns straight away.
You can control the maximum records returned by the poll() with props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);. The poll method is not thread safe and is not meant to get called from multiple threads.

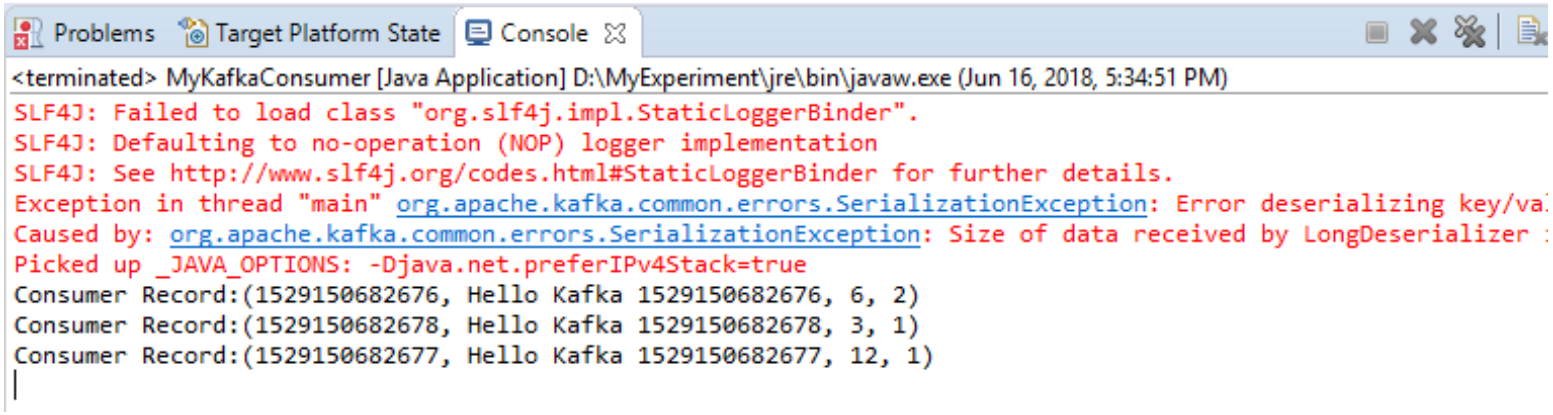
Running the Kafka Consumer
Next you define the main method.

```
public class KafkaConsumerExample {

  public static void main(String... args) throws Exception {
     runConsumer();
  }
}
```

The main method just calls runConsumer.
Try running the consumer and producer.
Run the consumer from your IDE. Then run the producer from the last tutorial from your IDE. You should see the consumer get the records that the producer sent.



As you can see there are no records. Why?? It was already consumed by the terminal windows which offset is committed.
Stop the consumer console.
Execute the Producer and then execute the consumer



Logging set up for Kafka
If you don't set up logging well, it might be hard to see the consumer get the messages.
Kafka like most Java libs these days uses sl4j. You can use Kafka with Log4j, Logback or JDK logging. We used logback in our maven build (compile 'ch.qos.logback:logback-classic:1.2.2').

src/main/resources/logback.xml

```
<configuration>

        <appender name="STDOUT"
                class="ch.qos.logback.core.ConsoleAppender">
                <encoder>
                        <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
                        </pattern>
                </encoder>
        </appender>
```

```
    <logger name="org.apache.kafka" level="INFO" />
    <logger name="org.apache.kafka.common.metrics" level="INFO" />

    <root level="debug">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Notice that we set org.apache.kafka to INFO, otherwise we will get a lot of log messages. You should run it set to debug and read through the log messages. It gives you a flavor of what Kafka is doing under the covers. Leave org.apache.kafka.common.metrics or what Kafka is doing under the covers is drowned by metrics logging.

Lab End Here

5. Java API – Producer – II

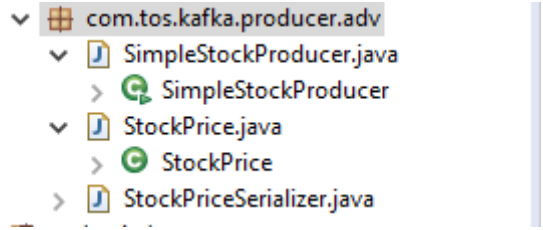
In this lab, we will be sending a JSON object using a custom serialize and in async mode.
The Stock Price Producer example has the following classes:

- StockPrice - holds a stock price has a name, dollar, and cents
- SimpleStockProducer - Configures and creates KafkaProducer, StockSender list, ThreadPool (ExecutorService), starts StockSender runnable into thread pool
- StockPriceSerializer - can serialize a StockPrice into byte[]

You can use the same maven project we have created in the producer example. In order to categorize the tutorials, let us create a package com.tos.kafka.aproducer, all the classes created for this lab will be stored inside this package.

Package Name:
com.tos.kafka.producer.adv

At the end of this lab we will have the following class.



Define all dependencies in the pom.xml as shown below:

```
<properties>
		<maven.compiler.target>1.8</maven.compiler.target>
		<maven.compiler.source>1.8</maven.compiler.source>
	</properties>
	<dependencies>
		<dependency>
			<groupId>org.apache.kafka</groupId>
			<artifactId>kafka-clients</artifactId>
			<version>1.1.0</version>
			<scope>compile</scope>
		</dependency>
		<!-- https://mvnrepository.com/artifact/io.advantageous.boon/boon-json -->
		<dependency>
			<groupId>io.advantageous.boon</groupId>
			<artifactId>boon-json</artifactId>
			<version>0.6.6</version>
		</dependency>

		<dependency>
			<groupId>org.slf4j</groupId>
			<artifactId>slf4j-ext</artifactId>
			<version>1.7.13</version>
		</dependency>
	</dependencies>

	<build>
		<plugins>
```

```
<plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.6.1</version>
        <configuration>
                <source>1.8</source>
                <target>1.8</target>
        </configuration>
</plugin>
<plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
                <archive>
                        <manifest>
                                <mainClass>fully.qualified.MainClass</mainClass>
                        </manifest>
                </archive>
                <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
        </configuration>
        <executions>
                <execution>
                        <id>make-assembly</id> <!-- this is used for inheritance merges -->
                        <phase>package</phase> <!-- bind to the packaging phase -->
                        <goals>
                                <goal>single</goal>
                        </goals>
                </execution>
```

```
                </executions>
            </plugin>
        </plugins>
    </build>
```

StockPrice
The StockPrice is a simple domain object that holds a stock price has a name, dollar, and cents. The StockPrice knows how to convert itself into a JSON string.

```
package com.tos.kafka.producer.adv;
import io.advantageous.boon.json.JsonFactory;

public class StockPrice {

      private final int dollars;
   private final int cents;
   private final String name;

   public StockPrice(final String json) {
      this(JsonFactory.fromJson(json, StockPrice.class));
   }


   public StockPrice() {
      dollars = 0;
      cents = 0;
      name = "";
   }

   public StockPrice(final String name, final int dollars, final int cents) {
      this.dollars = dollars;
      this.cents = cents;
      this.name = name;
   }
```

```
public StockPrice(final StockPrice stockPrice) {
   this.cents = stockPrice.cents;
   this.dollars = stockPrice.dollars;
   this.name = stockPrice.name;
}


public int getDollars() {
   return dollars;
}


public int getCents() {
   return cents;
}


public String getName() {
   return name;
}


@Override
public String toString() {
   return "StockPrice{" +
         "dollars=" + dollars +
         ", cents=" + cents +
         ", name='" + name + '\'' +
```

```
        '}';
}


@Override
public boolean equals(Object o) {
   if (this == o) return true;
   if (o == null || getClass() != o.getClass()) return false;

   StockPrice that = (StockPrice) o;

   if (dollars != that.dollars) return false;
   if (cents != that.cents) return false;
   return name != null ? name.equals(that.name) : that.name == null;
}

@Override
public int hashCode() {
   int result = dollars;
   result = 31 * result + cents;
   result = 31 * result + (name != null ? name.hashCode() : 0);
   return result;
}


public String toJson() {
   return "{" +
         "\"dollars\": " + dollars +
         ", \"cents\": " + cents +
```

```
            ", \"name\": \"" + name + '\"' +
            '}';
    }
}
```

Create a class SimpleStockProducer, which will be our main program that will send records to Broker. Here we will demonstrate the pushing of JSON document using serialize component. SimpleStockProducer import classes and sets up a logger. It has a main() method to create a KafkaProducer instance. It has a initProducer() method to initialize bootstrap servers, client id, key serializer and custom serializer (StockPriceSerializer). It has a main() method that creates the producer, creates a StockSender list passing each instance the producer, and then it runs each stockSender in its own thread.

public class SimpleStockProducer

Add the following import in the above class.

```
package com.tos.kafka.producer.adv;

import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.TimeUnit;

import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;
```

Add the following method to initialize the properties of the Producer.

```
public final static String TOPIC = "stock-prices";

    public static Properties initProducer() {
        // Assign topicName to string variable

        // create instance for properties to access producer configs
        Properties props = new Properties();

        // Assign localhost id
        props.put("bootstrap.servers", "192.168.139.129:9093");
        // props.put("bootstrap.servers", "localhost:9092");

        // Set acknowledgements for producer requests.
        // props.put("acks", "all");
        props.put("enable.auto.commit", "true");
        // If the request fails, the producer can automatically retry,
        props.put("retries", 1);

        // Specify buffer size in config
        props.put("batch.size", 1);

        // Reduce the no of requests less than 0
        props.put("linger.ms", 100);

        // The buffer.memory controls the total amount of memory available to the
```

```
        // producer for buffering.
        props.put("buffer.memory", 302);

        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        props.put("value.serializer", StockPriceSerializer.class.getName());
        // props.put("transactional.id", "my-transactional-id");
        return props;

    }
```

The main method that will create the producer and send message to Broker

```
    public static void main(String[] args) {
        Properties props = initProducer();
        Producer<String, StockPrice> producer = new KafkaProducer<String, StockPrice>(props);
        //sendMessage(producer, TOPIC);
        //sendMessageAsync(producer, TOPIC);

    }
```

Add the following function in the main class to send records synchronously to Kafka Producer.

```
public static void sendMessage(Producer<String, StockPrice> producer, String topic) {

        for (int i = 0; i < 2; i++) {
            try {
```

```
                    ProducerRecord <String, StockPrice> record =  createRandomRecord(i);
                    RecordMetadata future = producer.send(record).get();
                    System.out.println(">>>" + future.topic() + " Offset:" + future.offset());
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (ExecutionException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                producer.flush();
                System.out.println("Message sent successfully");
            }
            producer.close();
            // test(producer);
            System.out.println("Message sent successfully & Close");
        }
```

Method that will create Two Producer records using StockPrice class

```
    private static ProducerRecord<String, StockPrice> createRandomRecord(int i) {
        final int dollarAmount = i* 200;
        final int centAmount = i * 1000;
        final StockPrice stockPrice = new StockPrice(" STOCK"+ i, dollarAmount, centAmount);
        return new ProducerRecord<>(TOPIC, stockPrice.getName(), stockPrice);
    }
```

To send records asynchronously with Kafka Producer the following method needs to be added.

Kafka provides an asynchronous send method to send a record to a topic. Let's use this method to send some message ids and messages to the Kafka topic we created earlier. The big difference here will be that we use a lambda expression to define a callback.

```
// Sending Message Asynchronously
        public static void sendMessageAsync(Producer<String, StockPrice> producer, String topic) {
                final CountDownLatch countDownLatch = new CountDownLatch(2);
                for (int i = 0; i < 2; i++) {
                        try {
                                ProducerRecord <String, StockPrice> record =  createRandomAsyncRecord(i);

                                long time = System.currentTimeMillis();
                                  // Register a call back.
                                        producer.send(record, (metadata, exception) -> {
                                    long elapsedTime = System.currentTimeMillis() - time;
                                    if (metadata != null) {
                                       System.out.printf("sent record(key=%s value=%s) " +
                                                "meta(partition=%d, offset=%d) time=%d\n",
                                            record.key(), record.value(), metadata.partition(),
                                            metadata.offset(), elapsedTime);
                                    } else {
                                       exception.printStackTrace();
                                    }
                                    countDownLatch.countDown();
                                  });

                                    countDownLatch.await(25, TimeUnit.SECONDS);

                        } catch (InterruptedException e) {
```

```
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                producer.flush();
                System.out.println("Message sent successfully");
            }
            producer.close();
            // test(producer);
            System.out.println("Message sent successfully & Close");
        }
```

Notice the use of a CountDownLatch so we can send all N messages and then wait for them all to send.

Async Interface Callback and Async Send Method

Kafka defines a Callback interface that you use for asynchronous operations. The callback interface allows code to execute when the request is complete. The callback executes in a background I/O thread so it should be fast (don't block it). The onCompletion(RecordMetadata metadata, Exception exception) gets called when the asynchronous operation completes. The metadata gets set (not null) if the operation was a success, and the exception gets set (not null) if the operation had an error.

The async send method is used to send a record to a topic, and the provided callback gets called when the send is acknowledged. The send method is asynchronous, and when called returns immediately once the record gets stored in the buffer of records waiting to post to the Kafka broker. The send method allows sending many records in parallel without blocking to wait for the response after each one.

Since the send call is asynchronous it returns a Future for the RecordMetadata that will be assigned to this record. Invoking get() on this future will block until the associated request completes and then return the metadata for the record or throw any exception that occurred while sending the record. KafkaProducer

In the above code, the following line overrides the onCompletion(RecordMetadata metadata, Exception exception) method.

```
producer.send(record, (metadata, exception)
-------- ------------------ ------------------- --------------------
```

Add the following code that will create Record for sending to broker.

```
private static ProducerRecord<String, StockPrice> createRandomAsyncRecord(int i) {
                final int dollarAmount = i* 200;
                final int centAmount = i * 1000;
                final StockPrice stockPrice = new StockPrice(" Async STOCK "+ i, dollarAmount,
centAmount);
                return new ProducerRecord<>(TOPIC, stockPrice.getName(), stockPrice);
        }
```

Custom Serializers

We are not using built-in Kafka serializers. We are writing your own custom serializer. You just need to be able to convert your custom keys and values using the serializer convert to and convert from byte arrays (byte[]). Serializers work for keys and values, and you set them up with the Kafka Producer properties value.serializer, and key.serializer. The StockPriceSerializer will serialize StockPrice into bytes.

Create the following Custom Serializer class

```
package com.tos.kafka.producer.adv;


import java.nio.charset.StandardCharsets;
import java.util.Map;

import org.apache.kafka.common.serialization.Serializer;



public class StockPriceSerializer implements Serializer<StockPrice> {

   @Override
   public byte[] serialize(String topic, StockPrice data) {
      return data.toJson().getBytes(StandardCharsets.UTF_8);
   }

   @Override
   public void configure(Map<String, ?> configs, boolean isKey) {
   }

   @Override
   public void close() {
   }
}
```

You need to override the serialize method to convert JSON to bytes so that it can stream the stock price json document to broker.
Let us test the application now.

Start the broker and create the topic if not created ()

#sh start3Brokers.sh

```
[root@tos scripts]# sh start3Brokers.sh
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
Start zookeeper & 3 Brokers Successfully
[root@tos scripts]# jps
3025 QuorumPeerMain
3041 Kafka
3042 Kafka
3043 Kafka
3854 Jps
[root@tos scripts]# jps
```

/opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic stock-prices

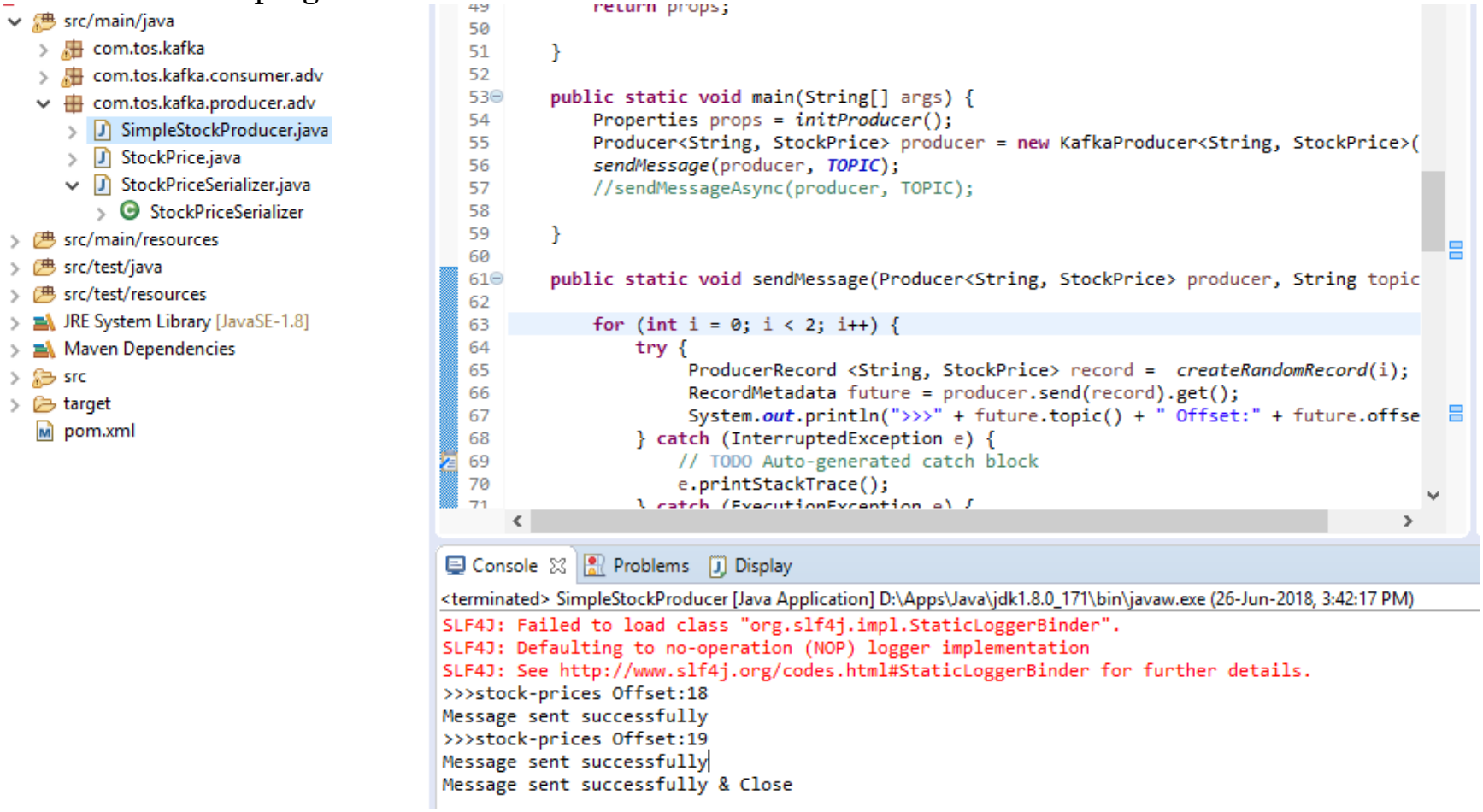
We will send the JSON message synchronously.

In the main method ensure that you have uncommented *sendMessage() and commented sendMessageAsync()*

```
public static void main(String[] args) {
    Properties props = initProducer();
    Producer<String, StockPrice> producer = new KafkaProducer<String, StockPrice>(
    sendMessage(producer, TOPIC);
    //sendMessageAsync(producer, TOPIC);

}
```

Open a terminal and execute the following command to consume the message:
#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic stock-prices

Execute the main program.



The offset of the message will be printed in the console of the producer. (Ensure that kafka server IP and port are mention accordingly to your setting)
You should be able to view 2 json message in the consumer console.

```
[root@tos scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server
tos.master.com:9093 --topic stock-prices
{"dollars": 0, "cents": 0, "name": " STOCK0"}
{"dollars": 200, "cents": 1000, "name": " STOCK1"}
```

Let us execute the send message in ascync mode. For this comment the sync method and uncomment async as shown below.

```
    public static void main(String[] args) {
        Properties props = initProducer();
        Producer<String, StockPrice> producer = new KafkaProducer<String, StockPrice>
        //sendMessage(producer, TOPIC);
        sendMessageAsync(producer, TOPIC);

    }
```

Execute the main program now. After a couple of seconds you should have console as shown below:

```
<terminated> SimpleStockProducer [Java Application] D:\Apps\Java\jdk1.8.0_171\bin\javaw.exe (26-Jun-2018, 3:44:57 PM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
sent record(key= Async STOCK 0 value=StockPrice{dollars=0, cents=0, name=' Async STOCK 0'}) meta(partition=0, offset=20) time=261
Message sent successfully
sent record(key= Async STOCK 1 value=StockPrice{dollars=200, cents=1000, name=' Async STOCK 1'}) meta(partition=0, offset=21) time=105
Message sent successfully
Message sent successfully & Close
```

We have printed the meta data of the record in the call back method. Here it mentions the partition also. You can verify from the consume console also. Here you can see two asnyc stock being consume from the topic, stock-prices.

```
[root@tos scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server
tos.master.com:9093 --topic stock-prices
{"dollars": 0, "cents": 0, "name": " STOCK0"}
{"dollars": 200, "cents": 1000, "name": " STOCK1"}
{"dollars": 0, "cents": 0, "name": " Async STOCK 0"}
{"dollars": 200, "cents": 1000, "name": " Async STOCK 1"}
```

Let us verify the Partitioning of message too.
Create a topic with 4 partitions as shown below:
/opt/kafka/bin/kafka-topics.sh --create --zookeeper tos.master.com:2181 --replication-factor 2 --partitions 4 --topic mystock-prices

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create --zookeeper tos.mast
er.com:2181 --replication-factor 2 --partitions 4 --topic mystock-prices
Created topic "mystock-prices".
```

Update the topic name as shown below.

```
14
15  public class SimpleStockProducer {
16
17      public final static String TOPIC = "mystock-prices";
18
19      public static Properties initProducer() {
```

We have created the topi, mystock-prices with 4 partitions so that message will be distributed across the partition.

Ensure that the following method is not commented.

```
public static void main(String[] args) {
    Properties props = initProducer();
    Producer<String, StockPrice> producer = new KafkaProducer<String, StockPrice
    //sendMessage(producer, TOPIC);
    sendMessageAsync(producer, TOPIC);

}
```

We will send 4 messages as shown, updated the counter to 4.

```
// Sending Message Asynchronously
public static void sendMessageAsync(Producer<String, StockPrice> producer, String
    final CountDownLatch countDownLatch = new CountDownLatch(2);
    for (int i = 0; i < 4; i++) {
        try {
            ProducerRecord <String, StockPrice> record =  createRandomAsyncRecord

            long time = System.currentTimeMillis();
                // Register a call back.
                producer.send(record, (metadata, exception) -> {
                    long elapsedTime = System.currentTimeMillis() - time;
                    if (metadata != null) {
                        System.out.printf("sent record(key=%s value=%s) "
                                "meta(partition=%d, offset=%d) tim
                                record.key(), record.value(), metadata.par
                                metadata.offset(), elapsedTime);
                    } else {
                        exception.printStackTrace();
                    }
```

Execute the main program.

As you can see from the console, there are distributed across different partition.

```
<terminated> SimpleStockProducer [Java Application] D:\Apps\Java\jdk1.8.0_171\bin\javaw.exe (26-Jun-2018, 4:58:35 PM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
sent record(key= Async STOCK 0 value=StockPrice{dollars=0, cents=0, name=' Async STOCK 0'}) meta(partition=2, offset=0) time=340
Message sent successfully
sent record(key= Async STOCK 1 value=StockPrice{dollars=200, cents=1000, name=' Async STOCK 1'}) meta(partition=3, offset=0) time=158
Message sent successfully
sent record(key= Async STOCK 2 value=StockPrice{dollars=400, cents=2000, name=' Async STOCK 2'}) meta(partition=2, offset=1) time=10
Message sent successfully
sent record(key= Async STOCK 3 value=StockPrice{dollars=600, cents=3000, name=' Async STOCK 3'}) meta(partition=1, offset=0) time=20
Message sent successfully
Message sent successfully & Close
```

You can describe the partition as shown below:

/opt/kafka/bin/kafka-topics.sh --describe --zookeeper tos.master.com:2181 --topic mystock-prices

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper tos.master.com:2181  --topic mystock-pric
es
Topic:mystock-prices    PartitionCount:4        ReplicationFactor:2     Configs:
        Topic: mystock-prices   Partition: 0    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: mystock-prices   Partition: 1    Leader: 1       Replicas: 1,2   Isr: 1,2
        Topic: mystock-prices   Partition: 2    Leader: 2       Replicas: 2,0   Isr: 2,0
        Topic: mystock-prices   Partition: 3    Leader: 0       Replicas: 0,2   Isr: 0,2
[root@tos scripts]#
```

Consumer:

#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server tos.master.com:9093 --topic mystock-prices

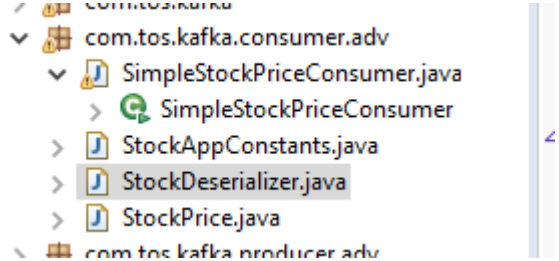
```
[root@tos scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server
tos.master.com:9093 --topic mystock-prices
{"dollars": 0, "cents": 0, "name": " Async STOCK 0"}
{"dollars": 200, "cents": 1000, "name": " Async STOCK 1"}
{"dollars": 400, "cents": 2000, "name": " Async STOCK 2"}
{"dollars": 600, "cents": 3000, "name": " Async STOCK 3"}
{"dollars": 0, "cents": 0, "name": " Async STOCK 0"}
{"dollars": 200, "cents": 1000, "name": " Async STOCK 1"}
{"dollars": 400, "cents": 2000, "name": " Async STOCK 2"}
{"dollars": 600, "cents": 3000, "name": " Async STOCK 3"}
{"dollars": 800, "cents": 4000, "name": " Async STOCK 4"}
{"dollars": 1000, "cents": 5000, "name": " Async STOCK 5"}
```

Lab Ends here

6. Java API – Consumer – II

We will use the same Maven project that was created earlier. Let us create a separate package for storing all the classes that will be created in this tutorial.
Package : com.tos.kafka.consumer.adv

We will demonstrate how to use JSON deserialization and retry option.
At the end of the lab you should have the project structure as shown below:



Update the pom.xml with the following dependencies.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-
4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.tos.kafka</groupId>
  <artifactId>my-kafka-producer</artifactId>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-clients</artifactId>
               <version>2.6.0</version>
          </dependency>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-streams</artifactId>
               <version>2.6.0</version>
          </dependency>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-streams-scala_2.13</artifactId>
               <version>2.6.0</version>
          </dependency>
   <dependency>
               <groupId>io.advantageous.boon</groupId>
               <artifactId>boon-json</artifactId>
               <version>0.6.6</version>
          </dependency>
          <dependency>
       <groupId>com.google.code.gson</groupId>
```

```
        <artifactId>gson</artifactId>
        <version>2.6.2</version>
    </dependency>

            <dependency>
                <groupId>org.slf4j</groupId>
                <artifactId>slf4j-ext</artifactId>
                <version>1.7.13</version>
            </dependency>

            </dependencies>
    <build>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <version>3.6.1</version>
                    <configuration>
                        <source>1.8</source>
                        <target>1.8</target>
                    </configuration>
                </plugin>

            </plugins>


</build>


</project>
```

The Stock Price Consumer example has the following classes:

- StockPrice - holds a stock price has a name, dollar, and cents
- SimpleStockPriceConsumer - consumes StockPrices and display batch lengths for poll
- StockAppConstants - holds topic and broker list
- StockDeserializer - can deserialize a StockPrice from byte[]

StockDeserializer

The StockDeserializer just calls the JSON parser to parse JSON in bytes to a StockPrice object. Create the following class.

```
package com.tos.kafka.consumer.adv;

import org.apache.kafka.common.serialization.Deserializer;

import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockDeserializer implements Deserializer<StockPrice> {

   @Override
   public StockPrice deserialize(final String topic, final byte[] data) {
      return new StockPrice(new String(data, StandardCharsets.UTF_8));
   }
```

```
    @Override
    public void configure(Map<String, ?> configs, boolean isKey) {
    }

    @Override
    public void close() {
    }
}
```

We just need to override the deserialize() that convert the JSON bytes to Stock Price.

Create the following class which will be our data or domain value object.

```
package com.tos.kafka.consumer.adv;


import com.google.gson.Gson;

public class StockPrice {

   private final int dollars ;
    private final int cents ;
    private final String name ;
    static Gson gson = new Gson();

    public StockPrice(final String json) {
          this(gson.fromJson(json, StockPrice.class));
    }


    public StockPrice() {
        dollars = 0;
        cents = 0;
        name = "";
    }

    public StockPrice(final String name, final int dollars, final int cents) {
        this.dollars = dollars;
        this.cents = cents;
        this.name = name;
    }


    public StockPrice(final StockPrice stockPrice) {
```

```
        this.cents = stockPrice.cents;
        this.dollars = stockPrice.dollars;
        this.name = stockPrice.name;
    }


    public int getDollars() {
        return dollars;
    }


    public int getCents() {
        return cents;
    }


    public String getName() {
        return name;
    }


    @Override
    public String toString() {
        return "StockPrice{" +
                "dollars=" + dollars +
                ", cents=" + cents +
                ", name='" + name + '\'' +
                '}';
    }


    @Override
    public boolean equals(Object o) {
```

```
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        StockPrice that = (StockPrice) o;

        if (dollars != that.dollars) return false;
        if (cents != that.cents) return false;
        return name != null ? name.equals(that.name) : that.name == null;
    }

    @Override
    public int hashCode() {
        int result = dollars;
        result = 31 * result + cents;
        result = 31 * result + (name != null ? name.hashCode() : 0);
        return result;
    }


    public String toJson() {
        return "{" +
                "\"dollars\": " + dollars +
                ", \"cents\": " + cents +
                ", \"name\": \"" + name + '\"' +
                '}';
    }
}
```

The application constants that define the servers that we will be connecting by our consumer. Ensure that you change your system broker IP accordingly. Update the code with that of the following.

```
package com.tos.kafka.consumer.adv;

public class StockAppConstants {
        public final static String TOPIC = "stock-prices";
          public final static String BOOTSTRAP_SERVERS =
                "localhost:9092";

}
```

SimpleStockPriceConsumer uses createConsumer method to create a KafkaConsumer instance, subscribes to stock-prices topics and has a custom deserializer.
It has a runConsumer() method that drains topic, creates List of current stocks and calls displayRecordsStatsAndStocks() method.
The method displayRecordsStatsAndStocks() prints out size of each partition read and total record count and prints out each stock at its current price.

Create a class SimpleStockPriceConsumer and import the following class.

```
package com.tos.kafka.consumer.adv;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.UUID;

import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
```

Add the following method that will create KafkaConsumer.

```
private static Consumer<String, StockPrice> createConsumer() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
StockAppConstants.BOOTSTRAP_SERVERS);
        // props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleAdvConsumer");
        props.put(ConsumerConfig.GROUP_ID_CONFIG, "KA" +
UUID.randomUUID().toString());
        // props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
        // "KafkaExampleAdvConsumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StockDeserializer.class.getName());
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        // Create the consumer using props.
        final Consumer<String, StockPrice> consumer = new KafkaConsumer<>(props);
        // Subscribe to the topic.
        consumer.subscribe(Collections.singletonList(StockAppConstants.TOPIC));
        return consumer;
    }
```

Add the following method that will call the poll method that will pull the message from the broker

```
static void runConsumer() throws InterruptedException {
        final Consumer<String, StockPrice> consumer = createConsumer();
        final List<StockPrice> map = new ArrayList<StockPrice>();
        try {
```

```
                final int giveUp = 1;
                int noRecordsCount = 0;
                while (true) {
                        consumer.seekToBeginning(consumer.assignment());
                        ConsumerRecords<String, StockPrice> consumerRecords = consumer.poll(100);
                        if (consumerRecords.count() == 0) {
                                noRecordsCount++;
                                if (noRecordsCount > giveUp)
                                        break;
                                else
                                        continue;
                        }
                        consumerRecords.forEach(record -> {
                                map.add(record.value());
                        });
                        displayRecordsStatsAndStocks(map, consumerRecords);
                        consumer.commitAsync();
                }
        } finally {
                consumer.close();
        }
        System.out.println("DONE");
}
```

The following method will print the records fetch from the Topic.

```
        private static void displayRecordsStatsAndStocks(final List<StockPrice> stockPriceMap,
                        final ConsumerRecords<String, StockPrice> consumerRecords) {
                System.out.printf("New ConsumerRecords partition count: %d Message count: %d\n",
                                consumerRecords.partitions().size(), consumerRecords.count());
                stockPriceMap.forEach((stockPrice) -> System.out.printf("ticker %s price %d.%d \n",
stockPrice.getName(),
                                stockPrice.getDollars(), stockPrice.getCents()));
                System.out.println();
        }
```

Finally update the following method to call the method that consume the record.

```
public static void main(String[] args) {
                try {
                        runConsumer();
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
```

You can execute the main method now. Ensure that you have started the broker before going ahead.
#start3Brokers.sh

```
[root@tos scripts]#
[root@tos scripts]# sh start3Brokers.sh
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
Start zookeeper & 3 Brokers Successfully
[root@tos scripts]# jps
2675 Kafka
2676 Kafka
2661 QuorumPeerMain
3429 -- process information unavailable
3386 Jps
[root@tos scripts]#
```

Execute the main program

```
Console  Problems  Display
<terminated> SimpleStockPriceConsumer [Java Application] D:\Apps\Java\jdk1.8.0_171\bin\javaw.exe (27-Jun-2018, 12:53:17 PM)
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
New ConsumerRecords partition count: 1 Message count: 22
ticker  STOCK0 price 0.0
ticker  STOCK1 price 200.1000
ticker  STOCK0 price 0.0
ticker  STOCK1 price 200.1000
ticker  STOCK0 price 0.0
ticker  STOCK1 price 200.1000
ticker  STOCK0 price 0.0
ticker  STOCK1 price 200.1000
ticker  STOCK0 price 0.0
```

As you can see above, it printed the number of partition count and the number of messages. It may vary depending on the number of messages that was pushed by your producer.

---------- ----------Lab Ends here ----------

7. Streaming – Exactly Once API

Create a maven Jav Project:

com.ostech:exactly-once

Update the pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.ostech</groupId>
  <artifactId>exactly-once</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <properties>
        <!-- Keep versions as properties to allow easy modification -->
        <java.version>8</java.version>
        <avro.version>1.10.0</avro.version>
        <gson.version>2.2.4</gson.version>
        <!-- Maven properties for compilation -->
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8
        </project.reporting.outputEncoding>
        <checkstyle.suppressions.location>checkstyle/suppressions.xml
        </checkstyle.suppressions.location>
        <confluent.version>5.3.0</confluent.version>
    </properties>
  <dependencies>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>2.6.0</version>
```

```
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams-scala_2.13</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.avro</groupId>
            <artifactId>avro</artifactId>
            <version>${avro.version}</version>
        </dependency>
        <dependency>
            <groupId>io.confluent</groupId>
            <artifactId>kafka-streams-avro-serde</artifactId>
            <version>5.2.0</version>
        </dependency>
        <dependency>
      <groupId>io.confluent</groupId>
      <artifactId>kafka-avro-serializer</artifactId>
      <version>${confluent.version}</version>
    </dependency>
    </dependencies>
</project>
```

Copy the following two classes.

```
package com.ostech.transaction;

import static java.time.Duration.ofSeconds;
import static java.util.Collections.singleton;
import static
org.apache.kafka.clients.consumer.ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG;
import static
org.apache.kafka.clients.consumer.ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG;
import static
org.apache.kafka.clients.consumer.ConsumerConfig.GROUP_ID_CONFIG;
import static
org.apache.kafka.clients.consumer.ConsumerConfig.ISOLATION_LEVEL_CONFIG;
import static
org.apache.kafka.clients.consumer.ConsumerConfig.KEY_DESERIALIZER_CLASS_CONF
IG;
import static
org.apache.kafka.clients.consumer.ConsumerConfig.VALUE_DESERIALIZER_CLASS_CO
NFIG;
import static
org.apache.kafka.clients.producer.ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG;
import static
org.apache.kafka.clients.producer.ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG
;
import static
```

```
org.apache.kafka.clients.producer.ProducerConfig.TRANSACTIONAL_ID_CONFIG;
import static
org.apache.kafka.clients.producer.ProducerConfig.VALUE_SERIALIZER_CLASS_CONF
IG;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.stream.Collectors;
import java.util.stream.Stream;

import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.OffsetAndMetadata;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.KafkaException;
import org.apache.kafka.common.TopicPartition;

public class TransactionalWordCount {

    private static final String CONSUMER_GROUP_ID = "my-group-id";
    private static final String OUTPUT_TOPIC = "output";
    private static final String INPUT_TOPIC = "input";

    public static void main(String[] args) {
```

```
        KafkaConsumer<String, String> consumer = createKafkaConsumer();
        KafkaProducer<String, String> producer = createKafkaProducer();

        producer.initTransactions();

        try {

            while (true) {

                ConsumerRecords<String, String> records =
consumer.poll(ofSeconds(60));

                Map<String, Integer> wordCountMap = records.records(new
TopicPartition(INPUT_TOPIC, 0))
                        .stream()
                        .flatMap(record -> Stream.of(record.value().split("
")))
                        .map(word -> Tuple.of(word, 1))
                        .collect(Collectors.toMap(tuple -> tuple.getKey(),
t1 -> t1.getValue(), (v1, v2) -> v1 + v2));

                producer.beginTransaction();

                wordCountMap.forEach((key, value) -> producer.send(new
ProducerRecord<String, String>(OUTPUT_TOPIC, key, value.toString())));

                Map<TopicPartition, OffsetAndMetadata> offsetsToCommit = new
```

```
HashMap<>();

                for (TopicPartition partition : records.partitions()) {
                    List<ConsumerRecord<String, String>> partitionedRecords
= records.records(partition);

                    long offset =
partitionedRecords.get(partitionedRecords.size() - 1).offset();
                    System.out.println(">>>" + offset + " - " +
partition.partition());
                    offsetsToCommit.put(partition, new
OffsetAndMetadata(offset + 1));
                }

                boolean flag = false;

                producer.sendOffsetsToTransaction(offsetsToCommit,
CONSUMER_GROUP_ID);

                if (flag == true) {
                 throw new Exception();
                }

                producer.commitTransaction();

            }

        } catch (Exception e) {
```

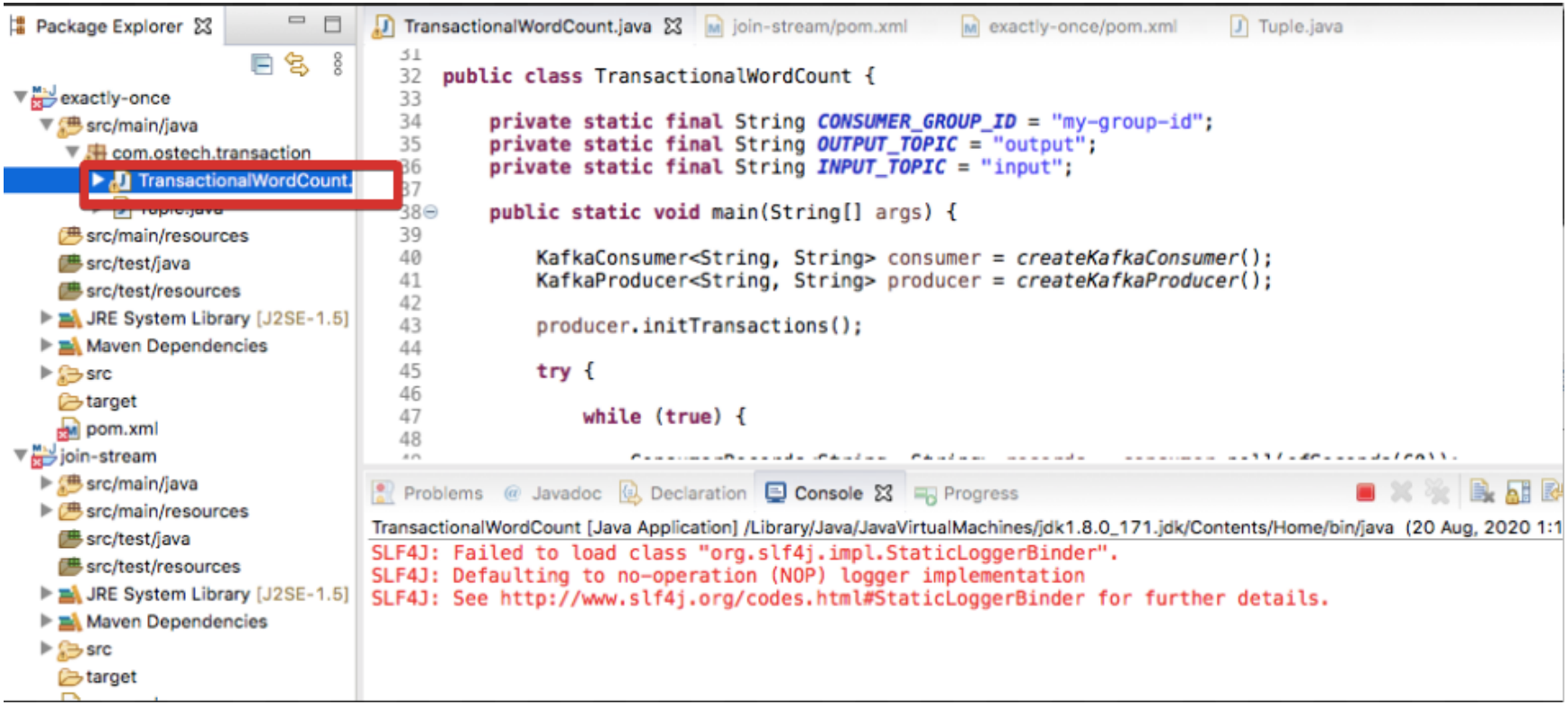
```
                producer.abortTransaction();

            }


        }

        private static KafkaConsumer<String, String> createKafkaConsumer() {
            Properties props = new Properties();
            props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(GROUP_ID_CONFIG, CONSUMER_GROUP_ID);
            props.put(ENABLE_AUTO_COMMIT_CONFIG, "false");
            props.put(ISOLATION_LEVEL_CONFIG, "read_committed");
            props.put(KEY_DESERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringDeserializer");
            props.put(VALUE_DESERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringDeserializer");

            KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
            consumer.subscribe(singleton(INPUT_TOPIC));
            return consumer;
        }

        private static KafkaProducer<String, String> createKafkaProducer() {

            Properties props = new Properties();
            props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

```
        props.put(ENABLE_IDEMPOTENCE_CONFIG, "true");
        props.put(TRANSACTIONAL_ID_CONFIG, "prod-1");
        props.put(KEY_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");
        props.put(VALUE_SERIALIZER_CLASS_CONFIG,
"org.apache.kafka.common.serialization.StringSerializer");

        return new KafkaProducer(props);

    }

}
```

```
package com.ostech.transaction;

public class Tuple {

    private String key;
    private Integer value;

    private Tuple(String key, Integer value) {
        this.key = key;
        this.value = value;
    }
```

```
    public static Tuple of(String key, Integer value){
        return new Tuple(key,value);
    }

    public String getKey() {
        return key;
    }

    public Integer getValue() {
        return value;
    }
}
```

Execute the Program:



Sent some records to the **input** topic. You don't need to explicitly create the topic as auto enable in the broker and will be created when the program will get started.

Execute the following from a separate terminal:

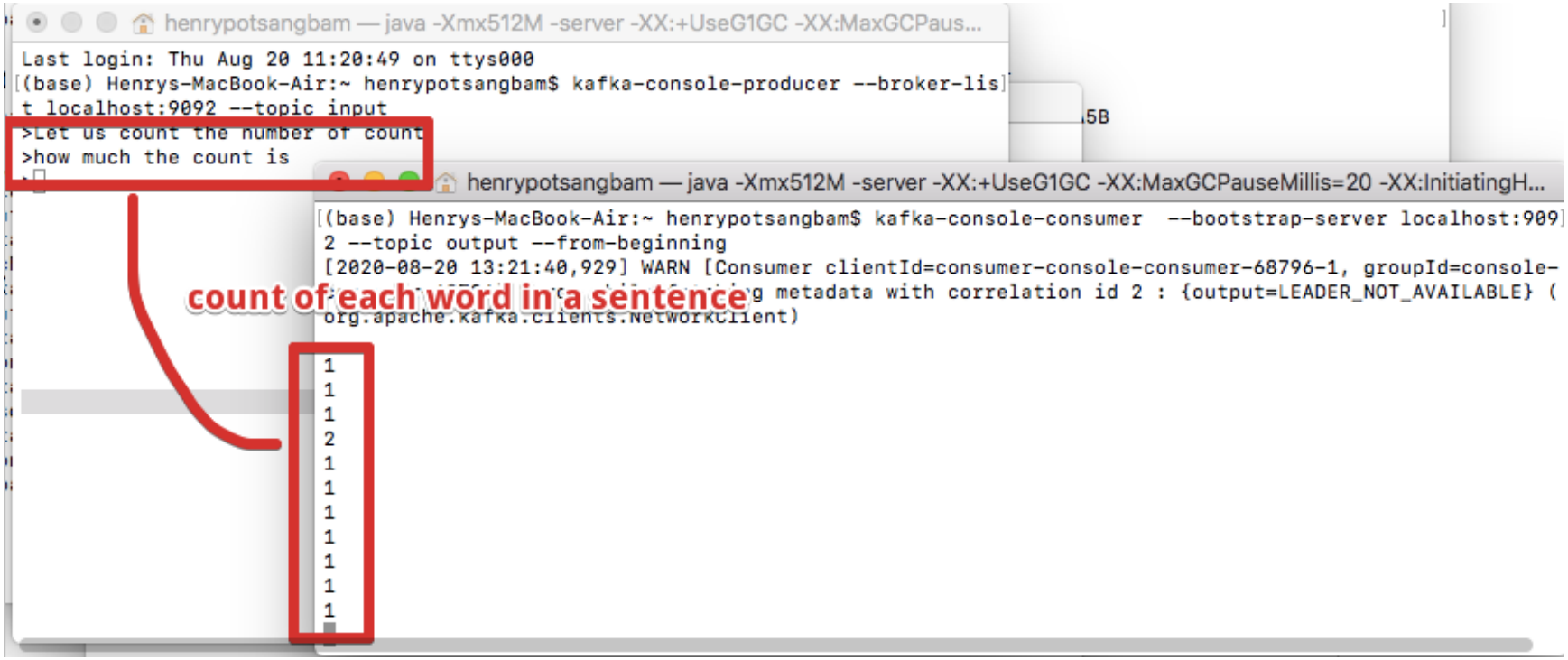
```
# kafka-console-producer --broker-list localhost:9092 --topic input
```

Enter the following sentences:

```
Let us count the number of count
how much the count is
```

Consume from the **output** topic.

```
# kafka-console-consumer  --bootstrap-server localhost:9092 --topic output --from-beginning
```



Enable the following Flag:

```
58
59                  wordCountMap.forEach((key, value) -> producer.send(new ProducerRecord<String, String>(OU
60
61                  Map<TopicPartition, OffsetAndMetadata> offsetsToCommit = new HashMap<>();
62
63                  for (TopicPartition partition : records.partitions()) {
64                      List<ConsumerRecord<String, String>> partitionedRecords = records.records(partition)
65
66                      long offset = partitionedRecords.get(partitionedRecords.size() - 1).offset();
67                      System.out.println(">>>" + offset + " - " + partition.partition());
68                      offsetsToCommit.put(partition, new OffsetAndMetadata(offset + 1));
69                  }
70
71                  boolean flag = true;
72
73                  producer.sendOffsetsToTransaction(offsetsToCommit, CONSUMER_GROUP_ID);
74
75                  if (flag == true) {
76                      throw new Exception();
77                  }
78
79                  producer.commitTransaction();
```

Stop the program and start it again

And type a sentence on the producer terminal:

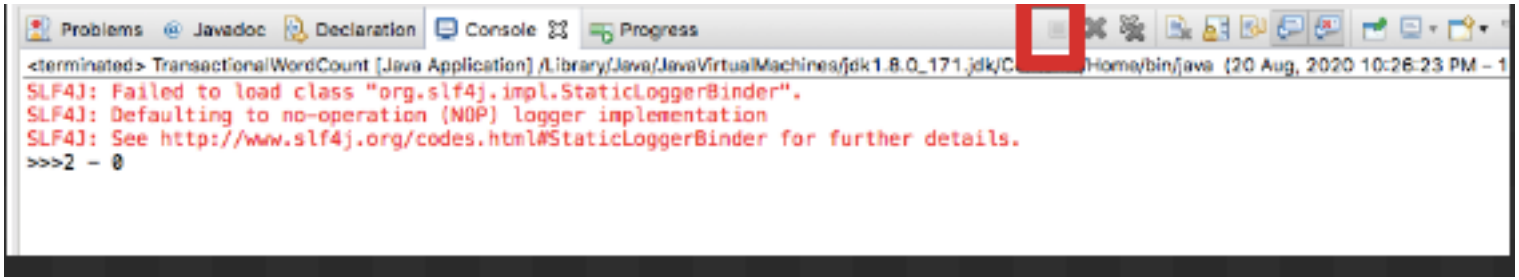
```
(base) Henrys-MacBook-Air:~ henrypotsangbam$ kafka-console-producer --broker-list localhost:9092 --topic input
>Let us count the number of count
>transaction works
```

Verify the console on the consumer and the execution of the program.

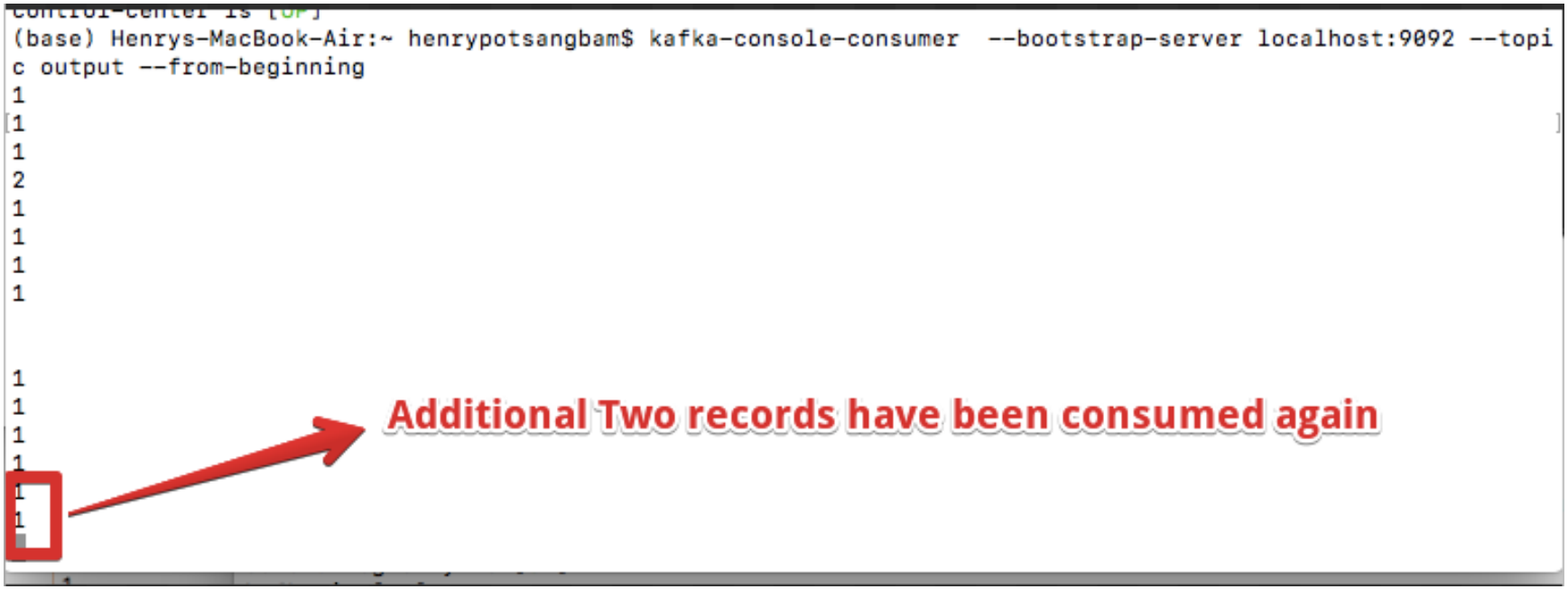
No changes in the console.

```
control-center is [UP]
(base) Henrys-MacBook-Air:~ henrypotsangbam$ kafka-console-consumer  --bootstrap-server localhost:9092 --topi
c output --from-beginning
1
1
1
2
1
1
1
1


1
1
1
1
```

The program stop execution:



changes the flag to false and execute the program now.



As observe because of the transaction, the messages are sent to the output topic once only.

---------- Lab Ends Here ----------

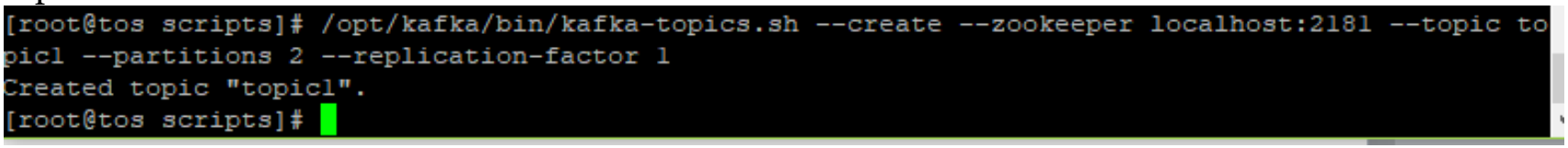
8. Manging Kafka Using CLI - Operations

You need to start the first broker using startABroker.sh
#sh startABroker.sh
#jps

```
[root@tos scripts]# sh startABroker.sh
ZooKeeper JMX enabled by default
Using config: /opt/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
Started Successfully
[root@tos scripts]# jps
11665 Jps
11646 Kafka
11375 QuorumPeerMain
[root@tos scripts]#
```

List and describe Topics
What does the tool do?
This tool lists the information for a given list of topics. If no topics are provided in the command line, the tool queries zookeeper to get all the topics and lists the information for them. The fields that the tool displays are - topic name, partition, leader, replicas, isr. Two optional arguments can be provided to the tool. If "under-replicated-partitions" is specified, the tool only provides information for those topic / partitions which have replicas that are under replicated. If "unavailable-partitions" is specified, the tool only provides information for those topic/partitions whose leader is not available.
How to use the tool?
2.2 Create Topics
What does the tool do?

By default, Kafka auto creates topic if "auto.create.topics.enable" is set to true on the server. This creates a topic with a default number of partitions, replication factor and uses Kafka's default scheme to do replica assignment. Sometimes, it may be required that we would like to customize a topic while creating it. This tool helps to create a topic and also specify the number of partitions, replication factor and replica assignment list for the topic.
How to use the tool?
Create topic with default settings
/opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic topic1 --partitions 2 --replication-factor 1



List only single topic named "test" (prints only topic name), if you have not created it. Execute the following script to create it.
#/opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic test --partitions 2 --replication-factor 1
#/opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181 --topic test

List all topics (prints only topic names)
#/opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181

```
[root@tos scripts]# jps
12960 Jps
12314 Kafka
12043 QuorumPeerMain
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --list --zookeeper tos.master.com:2181 --topic
 test
test
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
CustomerCountry
__consumer_offsets
henry-topic
my-failsafe-topic
my-kafka-topic
my-kafka-topic1
test
test-topic
```

Describe only single topic named "test" (prints details about the topic)
#/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic test

Describe all topics (prints details about the topics)
#/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181

```
[root@tos scripts]#
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic
test
Topic:test      PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: test     Partition: 0    Leader: 0       Replicas: 0     Isr: 0
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181
Topic:CustomerCountry   PartitionCount:1        ReplicationFactor:1     Configs:
        Topic: CustomerCountry  Partition: 0    Leader: 1       Replicas: 1     Isr: 1
Topic:__consumer_offsets        PartitionCount:50       ReplicationFactor:1     Configs:segment.b
ytes=104857600,cleanup.policy=compact,compression.type=producer
        Topic: __consumer_offsets       Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 1    Leader: 2       Replicas: 2     Isr: 2
```

List info for topics which have under replicated count
/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --under-replicated-partitions

List info for topics whose leader for a partition is not available
/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --unavailable-partitions

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --unavai
lable-partitions
        Topic: CustomerCountry  Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 0    Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 1    Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 3    Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 4    Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 6    Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 7    Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 9    Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 10   Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 12   Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 13   Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 15   Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 16   Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 18   Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 19   Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 21   Leader: 1       Replicas: 1     Isr: 1
        Topic: __consumer_offsets       Partition: 22   Leader: 2       Replicas: 2     Isr: 2
        Topic: __consumer_offsets       Partition: 24   Leader: 1       Replicas: 1     Isr: 1
```

Add Partition to Topic

What does the tool do?

In Kafka partitions act as the unit of parallelism: messages of a single topic are distributed to multiple partitions that can be stored and served on different servers. Upon creation of a topic, the number of partitions for this topic has to be specified. Later on more partitions may be needed for this topic when the volume of this topic increases. This tool helps to add more partitions for a specific topic and also allow manual replica assignment of the added partitions.

How to use the tool?

Increase number of partitions for topic

/opt/kafka/bin/kafka-topics.sh --alter --zookeeper localhost:2181 --topic topic1 --partitions 4

/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic topic1

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --alter --zookeeper localhost:2181 --topic top
ic1 --partitions 4
WARNING: If partitions are increased for a topic that has a key, the partition logic or ordering
of the messages will be affected
Adding partitions succeeded!
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic
topic1
Topic:topic1    PartitionCount:4        ReplicationFactor:1     Configs:
        Topic: topic1   Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: topic1   Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: topic1   Partition: 2    Leader: 0       Replicas: 0     Isr: 0
        Topic: topic1   Partition: 3    Leader: 0       Replicas: 0     Isr: 0
[root@tos scripts]#
```

2.4 Delete Topic
What does the tool do?
When topic deletion is enabled in the broker (delete.topic.enable), topics can be deleted using the Kafka Topics tool.

How to use the tool?
Delete topic named topic1
/opt/kafka/bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic topic1
/opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --delete --zookeeper localhost:2181 --topic to
picl
Topic topicl is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set to true.
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --list --zookeeper localhost:2181
CustomerCountry
__consumer_offsets
henry-topic
my-failsafe-topic
my-kafka-topic
my-kafka-topicl
test
test-topic
[root@tos scripts]#
```

Change topic configuration
What does the tool do?
Kafka Confings tool can be used to modify topic configuration:

Add new config options
Change existing config options
Remove config options
How to use the tool?
Create a topic and verify it before going ahead
#/opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic topic1 --partitions 2 --replication-factor 1
#/opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic topic1

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181 --topic to
picl --partitions 2 --replication-factor 1
Created topic "topicl".
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper localhost:2181 --topic
topicl
Topic:topicl    PartitionCount:2        ReplicationFactor:1     Configs:
        Topic: topicl   Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: topicl   Partition: 1    Leader: 0       Replicas: 0     Isr: 0
[root@tos scripts]#
```

Add new option or change exsiting option
/opt/kafka/bin/kafka-configs.sh --alter --zookeeper localhost:2181 --entity-name topic1 --entity-type topics --add-config cleanup.policy=compact

Remove exsiting option
/opt/kafka /bin/kafka-configs.sh --alter --zookeeper localhost:2181 --entity-name topic1 --entity-type topics --delete-config cleanup.policy

```
[root@tos scripts]# /opt/kafka/bin/kafka-configs.sh --alter --zookeeper localhost:2181 --entity-n
ame topicl --entity-type topics --add-config cleanup.policy=compact
Completed Updating config for entity: topic 'topicl'.
[root@tos scripts]# /opt/kafka/bin/kafka-configs.sh --alter --zookeeper localhost:2181 --entity-n
ame topicl --entity-type topics --delete-config cleanup.policy
Completed Updating config for entity: topic 'topicl'.
[root@tos scripts]#
```

Determine the consumer group details.
Start the single broker in case its stop else skip this step.
#sh startABroker.sh
Create a producer and consume from two separate consoles.
Let us create a topic, benchmark-consumer for this lab.
/opt/kafka/bin/kafka-topics.sh --create \
--zookeeper tos.master.com:2181 \
--replication-factor 1 \
--partitions 1 \
--topic benchmark-consumer

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create \
> --zookeeper tos.master.com:2181 \
> --replication-factor 1 \
> --partitions 1 \
> --topic benchmark-consumer
Created topic "benchmark-consumer".
[root@tos scripts]#
```

/opt/kafka/bin/kafka-producer-perf-test.sh --topic benchmark-consumer \
--num-records 45 \
--record-size 100 \
--throughput 15 \
--producer-props \
acks=1 \
bootstrap.servers=tos.master.com:9092 \
buffer.memory=674 \
compression.type=none \
batch.size=8

```
[root@tos scripts]# /opt/kafka/bin/kafka-producer-perf-test.sh --topic benchmark
-consumer \
> --num-records 45 \
> --record-size 100 \
> --throughput 15 \
> --producer-props \
> acks=1 \
> bootstrap.servers=tos.master.com:9092 \
> buffer.memory=674 \
> compression.type=none \
> batch.size=8
45 records sent, 14.985015 records/sec (0.00 MB/sec), 24.04 ms avg latency, 343.
00 ms max latency, 6 ms 50th, 140 ms 95th, 343 ms 99th, 343 ms 99.9th.
[root@tos scripts]#
```

Open 2 console and start consuming from the above topic, ensure that consumer groups are different for each consumer terminal.
Create the following two properties.
#cd /opt/scripts
#vi consumer.properties
group.id=tos-1
#vi consumer1.properties
group.id=tos-2

#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server tos.master.com:9092 --topic benchmark-consumer --from-beginning --consumer.config consumer.properties
#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server tos.master.com:9092 --topic benchmark-consumer --from-beginning --consumer.config consumer1.properties
For example, list new consumer groups:
/opt/kafka/bin/kafka-consumer-groups.sh --bootstrap-server tos.master.com:9092 –list

For any group listed, you can get more details by changing the --list parameter to --describe and adding the --group parameter. This will list all the topics that the group is consuming, as well as the offsets for each topic partition.

#/opt/kafka/bin/kafka-consumer-groups.sh --bootstrap-server tos.master.com:9092 --describe --group console-consumer-90417

```
[root@tos ~]# /opt/kafka/bin/kafka-consumer-groups.sh  --bootstrap-server tos.ma
ster.com:9092 --list
Note: This will not show information about old Zookeeper-based consumers.
console-consumer-90417
console-consumer-45224
[root@tos ~]# /opt/kafka/bin/kafka-consumer-groups.sh  --bootstrap-server tos.ma
ster.com:9092 --describe  --group console-consumer-90417
Note: This will not show information about old Zookeeper-based consumers.

TOPIC              PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CO
NSUMER-ID                                     HOST            CLIENT-ID
benchmark-consumer 0          94              94              0               co
nsumer-1-ca134c4f-ad6a-4380-a62d-07c314ffldaf /192.168.139.129 consumer-1
[root@tos ~]# /opt/kafka/bin/kafka-consumer-groups.sh  --bootstrap-server tos.master.com:9092 --descr
ibe  --group console-consumer-90417
Note: This will not show information about old Zookeeper-based consumers.

TOPIC              PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG             CONSUMER-ID
                        HOST            CLIENT-ID
benchmark-consumer 0          94              94              0               consumer-1-ca134c4f-ad6
a-4380-a62d-07c314ffldaf /192.168.139.129 consumer-1
[root@tos ~]#
```

Lab End Here

9. Understand Kafka Stream - Topology

Kafka Streams is a client library for building mission-critical real-time applications and microservices, where the input and/or output data is stored in Kafka clusters. Kafka Streams combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology to make these applications highly scalable, elastic, fault-tolerant, distributed, and much more.
This tutorial will demonstrate how to run a streaming application coded in this library. In this guide we will start from scratch on setting up your own project to write a stream processing application using Kafka Streams.

Create a new maven project (simple project) for this as shown below:

```
<groupId>com.tos.kstream</groupId>
 <artifactId>MyStream</artifactId>
 <version>0.0.1-SNAPSHOT</version>
```

We will then define in the pom.xml file the library that need to be included in the project which is the Streams and kafka dependency as shown below;

```
<dependencies>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-streams</artifactId>
               <version>2.3.0</version>
          </dependency>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-clients</artifactId>
               <version>2.3.0</version>
```

```
  </dependency>
  <!-- Optionally include Kafka Streams DSL for Scala for Scala 2.12 -->
  <dependency>
       <groupId>org.apache.kafka</groupId>
       <artifactId>kafka-streams-scala_2.12</artifactId>
       <version>2.3.0</version>
  </dependency>
  <dependency>
       <groupId>io.advantageous.boon</groupId>
       <artifactId>boon-json</artifactId>
       <version>0.6.6</version>
  </dependency>

  <dependency>
       <groupId>org.slf4j</groupId>
       <artifactId>slf4j-ext</artifactId>
       <version>1.7.13</version>
  </dependency>
  <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
  <dependency>
       <groupId>com.google.code.gson</groupId>
       <artifactId>gson</artifactId>
       <version>2.8.5</version>
  </dependency>
 <dependency>
       <groupId>com.fasterxml.jackson.core</groupId>
       <artifactId>jackson-core</artifactId>
       <version>2.7.1</version>
```

```
        </dependency>

        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>2.7.1</version>
        </dependency>
    </dependencies>
```

Add the following plugin dependencies too in the pom.xml

```
<build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>fully.qualified.MainClass</mainClass>
```

```
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
            <executions>
                <execution>
                    <id>make-assembly</id> <!-- this is used for inheritance
merges -->
                    <phase>package</phase> <!-- bind to the packaging phase
-->
                    <goals>
                        <goal>single</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>

</build>
```

Writing a first Streams application: Pipe

Create a java file under `src/main/java`. Let's name it `Pipe.java`:

```
package com.tos.kafka.stream;

public class Pipe {

	public static void main(String[] args) {
		// TODO Auto-generated method stub

	}

}
```

We are going to write all the code in the main function in this pipe program.
Import the following class.

```
import java.util.Arrays;
import java.util.Locale;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.utils.Bytes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.kstream.Produced;
```

```
import org.apache.kafka.streams.state.KeyValueStore;
```

The first step to write a Streams application is to create a java.util.Properties map to specify different Streams execution configuration values as defined in StreamsConfig. A couple of important configuration values you need to set are: StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, which specifies a list of host/port pairs to use for establishing the initial connection to the Kafka cluster, and StreamsConfig.APPLICATION_ID_CONFIG, which gives the unique identifier of your Streams application to distinguish itself with other applications talking to the same Kafka cluster:
In addition, you can customize other configurations in the same map, for example, default serialization and deserialization libraries for the record key-value pairs:
Add the following properties code.

```
// Initialize the properties
                Properties props = new Properties();
                props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-pipe");
                props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.139.129:9092");
                props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
                props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
                props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
```

Next we will define the computational logic of our Streams application. In Kafka Streams this computational logic is defined as a topology of connected processor nodes. We can use a topology builder to construct such a topology,
And then create a source stream from a Kafka topic named streams-plaintext-input using this topology builder:

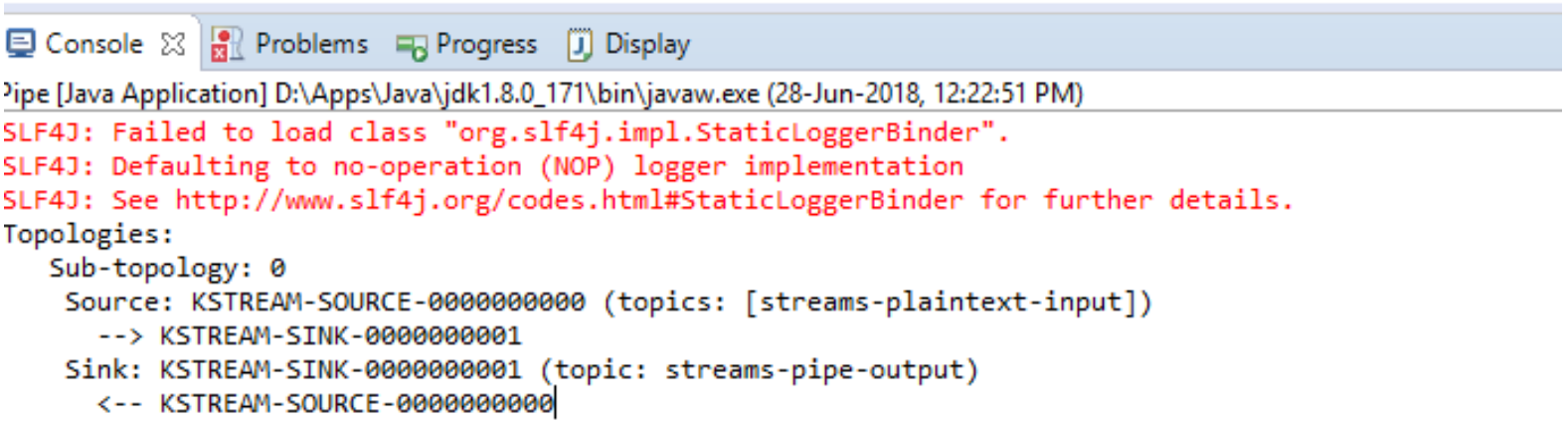
Now we get a KStream that is continuously generating records from its source Kafka topic streams-plaintext-input. The records are organized as String typed key-value pairs. The simplest thing we can do with this stream is to write it into another Kafka topic, say it's named streams-pipe-output. Note that we can also concatenate the above two lines into a single line. We can then inspect what kind of topology is created from this builder and print its description to standard output.

Copy the following code inside the main class.

```
        final StreamsBuilder builder = new StreamsBuilder();

        builder.stream("streams-plaintext-input").to("streams-pipe-output");

        final Topology topology = builder.build();

        final KafkaStreams streams = new KafkaStreams(topology, props);
        final CountDownLatch latch = new CountDownLatch(1);

    System.out.println(topology.describe());

        // attach shutdown handler to catch control-c
        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });
```

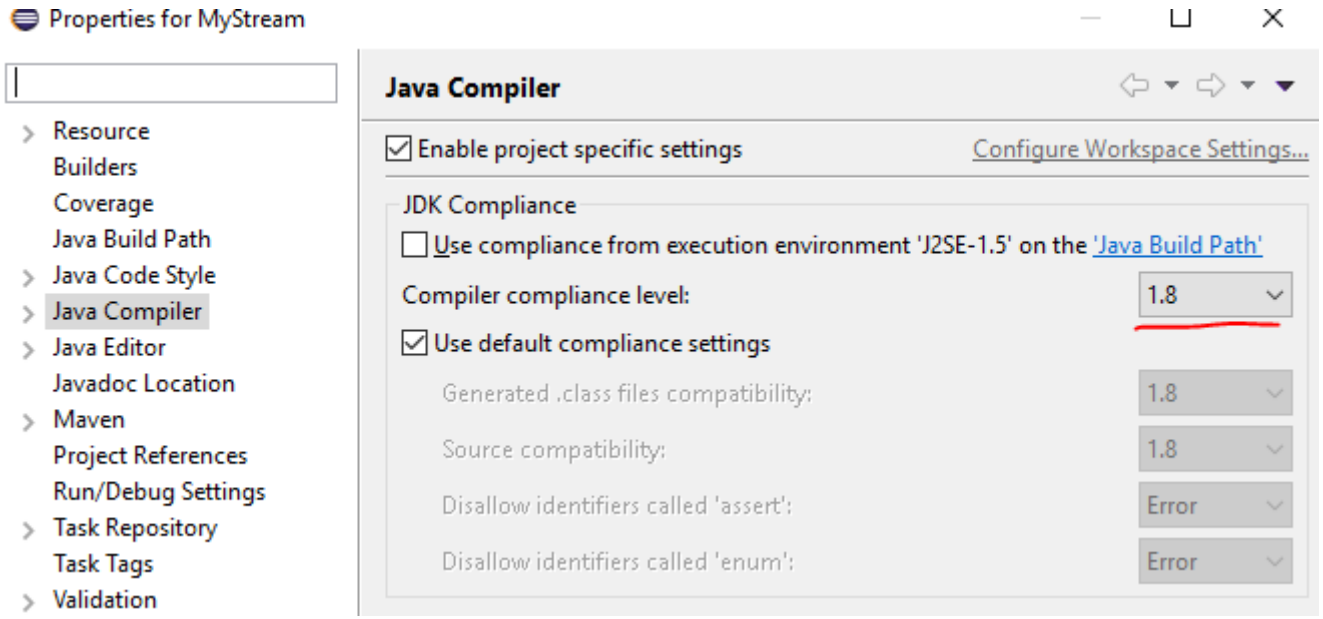
```
        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
```

If we just stop here, compile and run the program, it will output the following information:
Run Maven install, then execute the main program. Ensure that you have started the Broker.



As shown above, it illustrates that the constructed topology has two processor nodes, a source node KSTREAM-SOURCE-0000000000 and a sink node KSTREAM-SINK-0000000001. KSTREAM-SOURCE-0000000000 continuously read records from Kafka topic streams-plaintext-input and pipe them to its downstream node KSTREAM-SINK-0000000001; KSTREAM-SINK-0000000001 will write each of its received record in order to another Kafka topic streams-pipe-output (the --> and <-- arrows dictates the downstream and

upstream processor nodes of this node, i.e. "children" and "parents" within the topology graph). It also illustrates that this simple topology has no global state stores associated with it
You may need to configure project to support 1.8 java in case you have not done it before.



---------- Labs End Here ----------

10. Schema registry

In the theory – Explain Avro ah bit.

This tutorial provides a step-by-step workflow for using Confluent Schema Registry. You will learn how to enable client applications to read and write Avro data and use Confluent Control Center, which has integrated capabilities with Schema Registry.

Create the transactions topic

For the exercises in this tutorial, you will be producing to and consuming from a topic called transactions. Create this topic in Control Center.

1. Navigate to the Control Center web interface at http://localhost:9021/.

Click into the cluster, select **Topics** and click **Add a topic**.
Name the topic transactions and click **Create with defaults**.
The new topic is displayed.

Schema Definition

The first thing developers need to do is agree on a basic schema for data. Client applications form a contract:

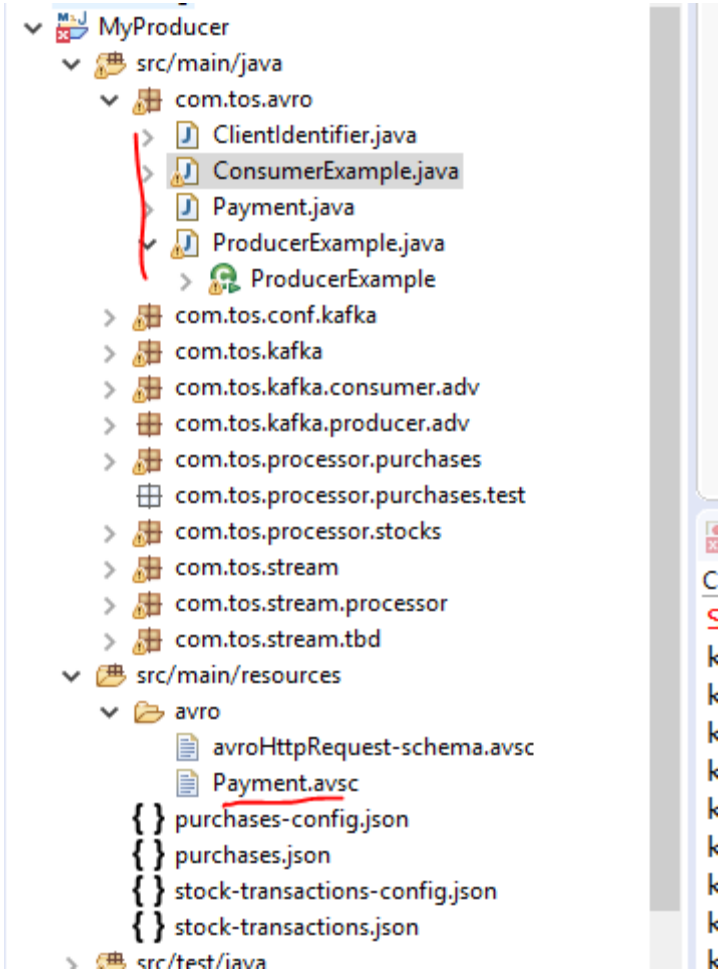
- producers will write data in a schema
- consumers will be able to read that data

Consider the original Payment schema.

```
{"namespace": "io.tos.examples.clients.basicavro",
 "type": "record",
 "name": "Payment",
 "fields": [
    {"name": "id", "type": "string"},
    {"name": "amount", "type": "double"}
 ]
}
```

Here is a break-down of what this schema defines:

- `namespace`: a fully qualified name that avoids schema naming conflicts
- `type`: Avro data type, for example, `record`, `enum`, `union`, `array`, `map`, or `fixed`
- `name`: unique schema name in this namespace
- `fields`: one or more simple or complex data types for a `record`. The first field in this record is called *id*, and it is of type *string*. The second field in this record is called *amount*, and it is of type *double*.



Update pom.xml

- Dependencies `org.apache.avro.avro` and `io.confluent.kafka-avro-serializer` to serialize data as Avro
- Plugin `avro-maven-plugin` to generate Java class files from the source schema

The `pom.xml` file may also include:

- Plugin `kafka-schema-registry-maven-plugin` to check compatibility of evolving schemas

https://docs.confluent.io/current/schema-registry/schema_registry_tutorial.html

Configuring Avro

Kafka applications using Avro data and Schema Registry need to specify at least two configuration parameters:

- Avro serializer or deserializer
- Properties to connect to Schema Registry

There are two basic types of Avro records that your application can use:

- a specific code-generated class, or
- a generic record

The examples in this tutorial demonstrate how to use the specific `Payment` class. Using a specific code-generated class requires you to define and compile a Java class for your schema, but it easier to work with in your code.

Java Producers

Within the application, Java producers need to configure the Avro serializer for the Apache Kafka® value (or Kafka key) and URL to Schema Registry. Then the producer can write records where the Kafka value is of `Payment` class.

Example Producer Code

When constructing the producer, configure the message value class to use the application's code-generated `Payment` class. For example:

```
// Code Begins Here
package com.tos.avro;

import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.errors.SerializationException;
import org.apache.kafka.common.serialization.StringSerializer;
import io.confluent.kafka.serializers.AbstractKafkaAvroSerDeConfig;
import io.confluent.kafka.serializers.KafkaAvroSerializer;
public class ProducerExample {

    private static final String TOPIC = "transactions";

    @SuppressWarnings("InfiniteLoopStatement")

    public static void main(final String[] args) {

        final Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.139.132:9092");
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 0);
```

```
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
KafkaAvroSerializer.class);
        props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://192.168.139.132:8081");

        try (KafkaProducer<String, Payment> producer = new KafkaProducer<String,
Payment>(props)) {

            for (long i = 0; i < 10; i++) {
                final String orderId = "id" + Long.toString(i);
                final Payment payment = new Payment(orderId, 1000.00d);
                final ProducerRecord<String, Payment> record = new ProducerRecord<String,
Payment>(TOPIC,
                        payment.getId().toString(), payment);

                producer.send(record);
                Thread.sleep(1000L);
            }

            producer.flush();
            System.out.printf("Successfully produced 10 messages to a topic called %s%n",
TOPIC);

        } catch (final SerializationException e) {
            e.printStackTrace();
        } catch (final InterruptedException e) {
            e.printStackTrace();
```

```
            }

        }

}
// Code Ends Here
```

Run the Producer

Run the following build commands in a shell in `/examples/clients/avro`.

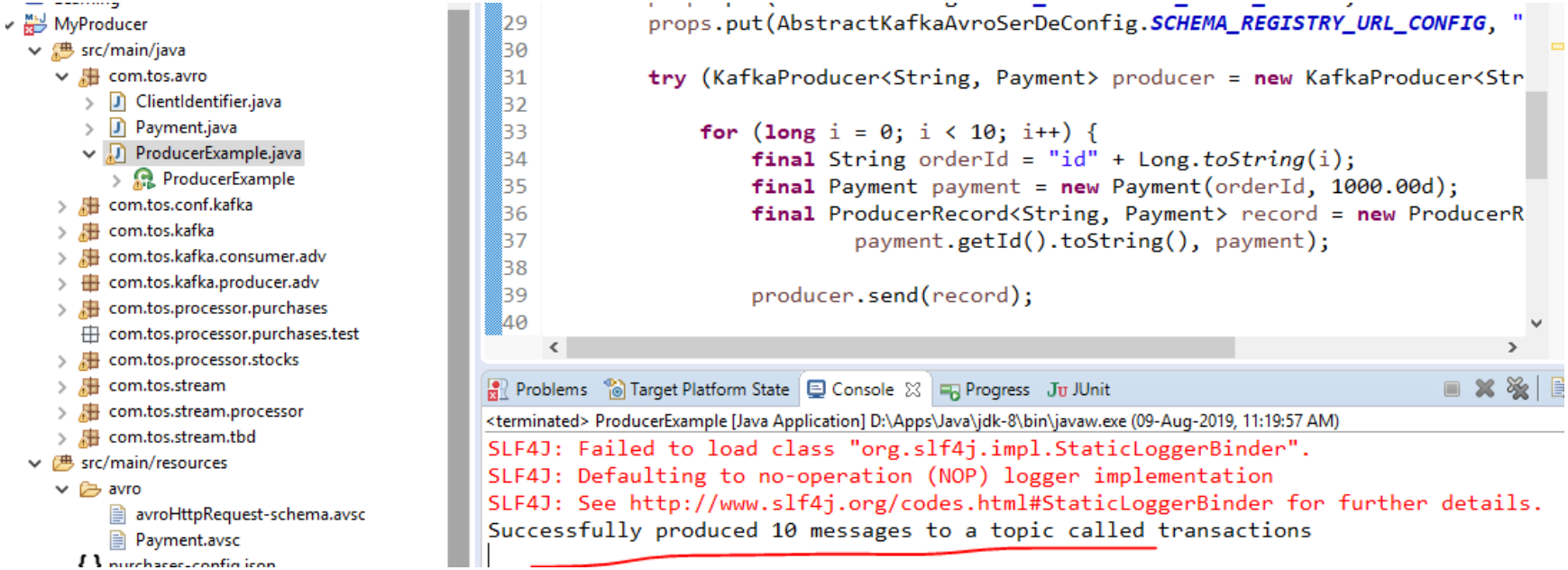
1. To run this producer, first compile the project:

```
2. mvn clean compile package
```

3. From the Control Center navigation menu at http://localhost:9021/, make sure the cluster is selected, and click **Management -> Topics**.

   Next, click the `transactions` topic and go to the **Messages** tab.

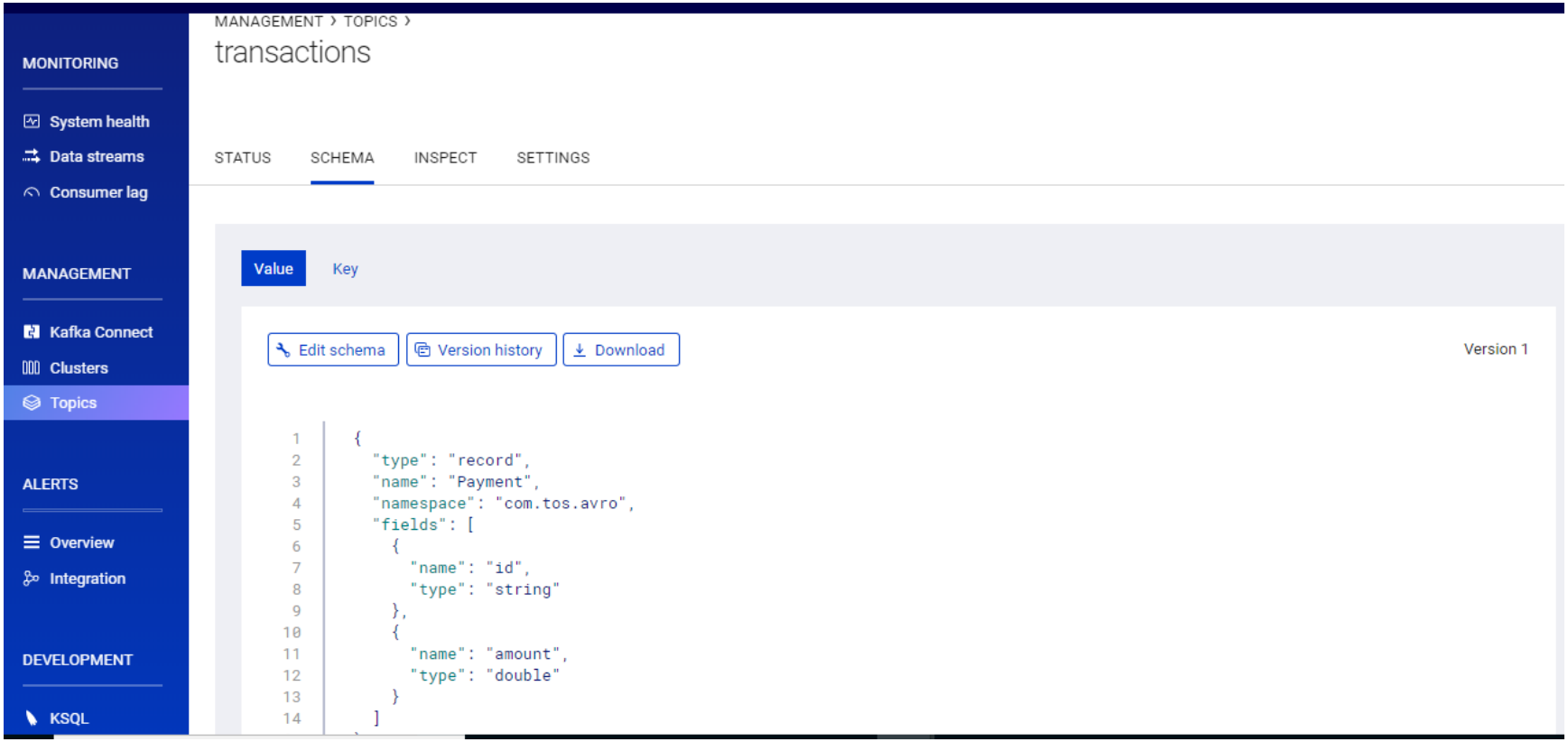
   You should see no messages because no messages have been produced to this topic yet.
4. Run `ProducerExample`, which produces Avro-formatted messages to the `transactions` topic.



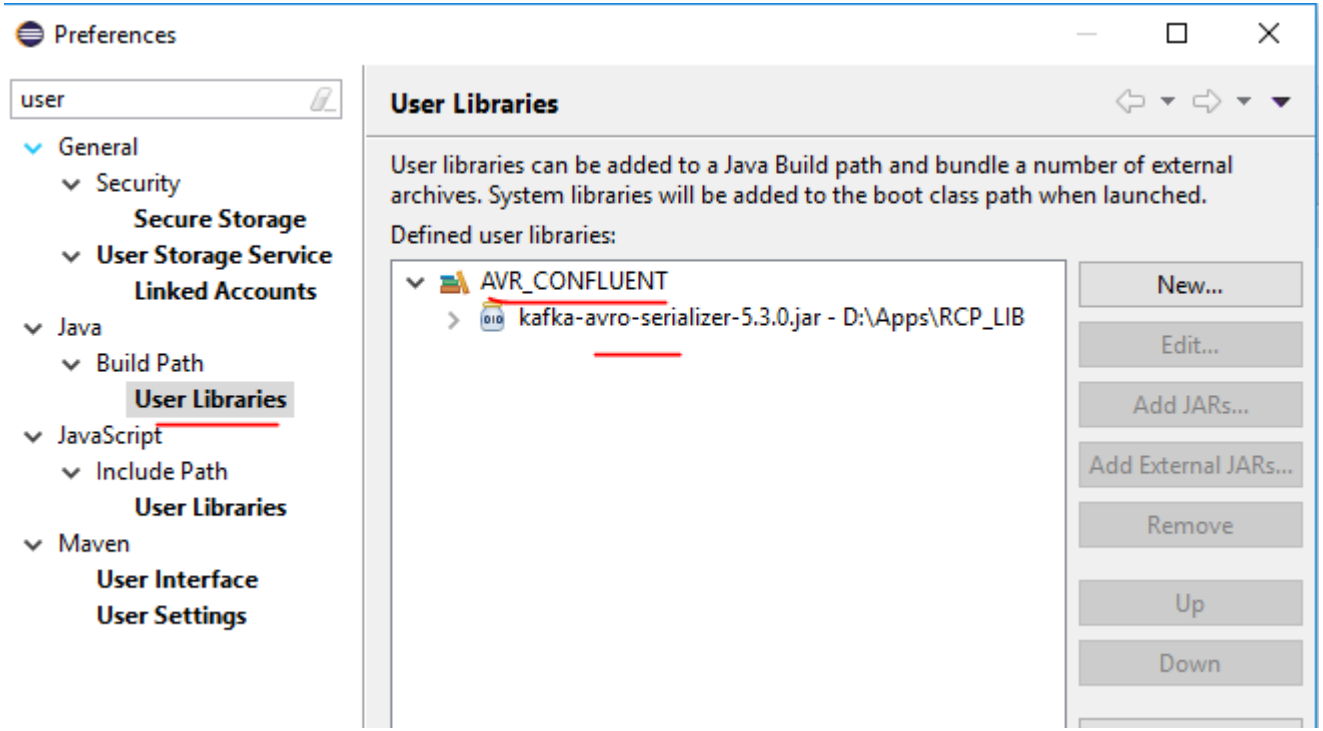
Now you should be able to see messages in Control Center by inspecting the `transactions` topic as it dynamically deserializes the newly arriving data that was serialized as Avro.

At http://localhost:9021/, click into the cluster on the left, then go to **Topics** -> `transactions` -> **Messages**.

Verify the schema too from the control Center.



If the compilation issue is there include the following jar in the project separately.



Java Consumers

Within the client application, Java consumers need to configure the Avro deserializer for the Kafka value (or Kafka key) and URL to Schema Registry. Then the consumer can read records where the Kafka value is of `Payment` class.

Example Consumer Code

By default, each record is deserialized into an Avro `GenericRecord`, but in this tutorial the record should be deserialized using the application's code-generated `Payment` class.

Therefore, configure the deserializer to use Avro `SpecificRecord`, i.e., `SPECIFIC_AVRO_READER_CONFIG` should be set to `true`. For example:

```
// Code Begins Here
package com.tos.avro;

import io.confluent.kafka.serializers.AbstractKafkaAvroSerDeConfig;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;

import org.apache.kafka.clients.consumer.KafkaConsumer;

import io.confluent.kafka.serializers.KafkaAvroDeserializer;

import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;

import org.apache.kafka.common.serialization.StringDeserializer;



import java.time.Duration;
import java.util.Arrays;
import java.util.Collections;
import java.util.Properties;
public class ConsumerExample {
   private static final String TOPIC = "transactions";
```

```
  @SuppressWarnings("InfiniteLoopStatement")
  public static void main(final String[] args) {
    final Properties props = new Properties();
    props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.139.132:9092");
    props.put(ConsumerConfig.GROUP_ID_CONFIG, "test-payments");
    props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
    props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
    props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
"http://192.168.139.132:8081");
    props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class);
    props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
KafkaAvroDeserializer.class);
    props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, true);

    try (final KafkaConsumer<String, Payment> consumer = new KafkaConsumer<>(props)) {

      consumer.subscribe(Collections.singletonList(TOPIC));
      while (true) {
        final ConsumerRecords<String, Payment> records = consumer.poll(100);
        for (final ConsumerRecord<String, Payment> record : records) {
          final String key = record.key();
          final Payment value = record.value();
          System.out.printf("key = %s, value = %s%n", key, value);

        }

      }
```

```
    }
  }
}
// Code Ends Here
```

Run the Consumer

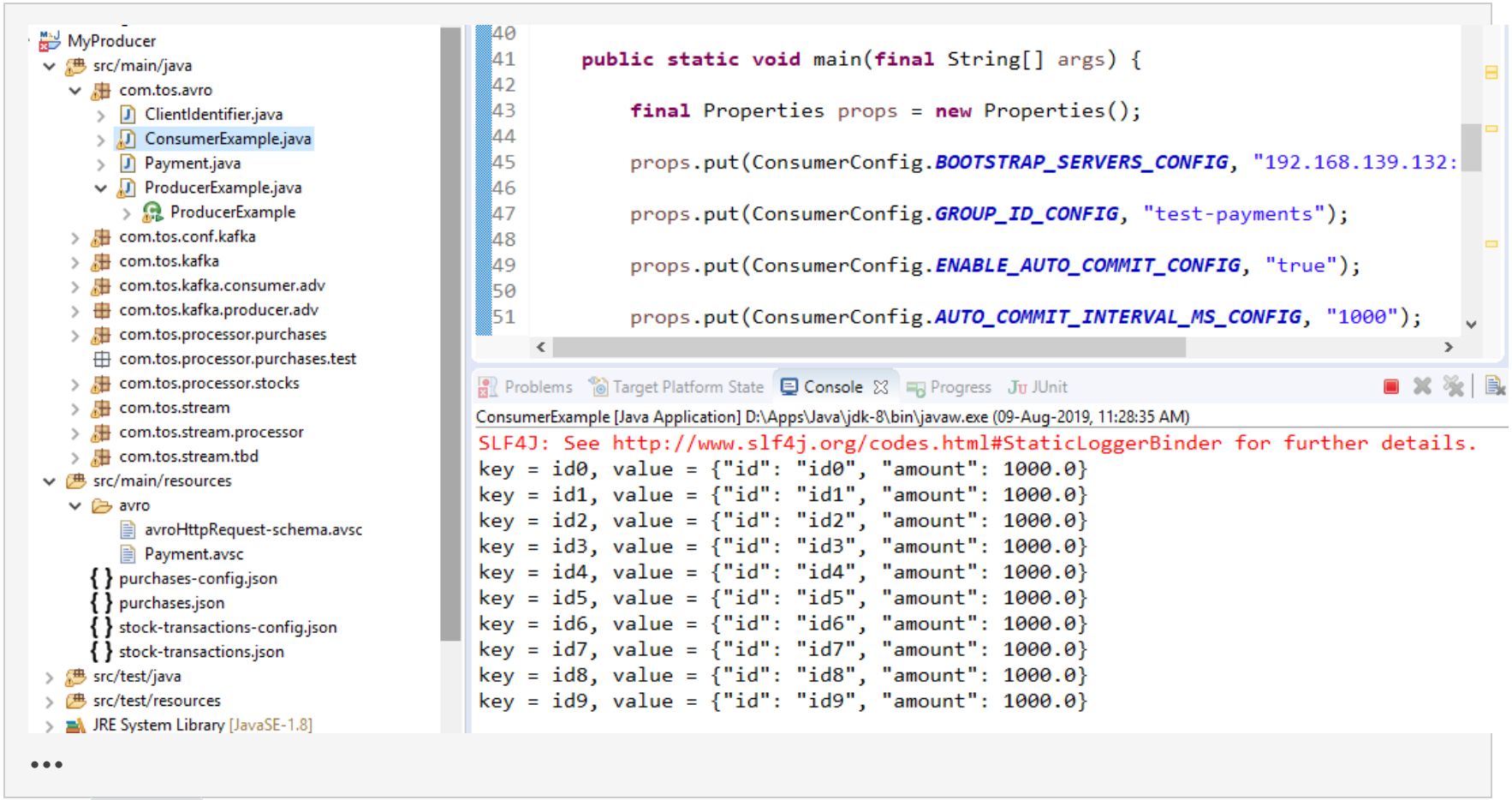
1. To run this consumer, first compile the project.

2. mvn clean compile package

3. Then run `ConsumerExample` (assuming you already ran the `ProducerExample` above).

4. mvn exec:java -Dexec.mainClass=io.confluent.examples.clients.basicavro.ConsumerExample

   You should see:



5. Hit Ctrl-C to stop.

------ ---------- Lab Ends Here ----------

11. DSL - Transform a stream of events

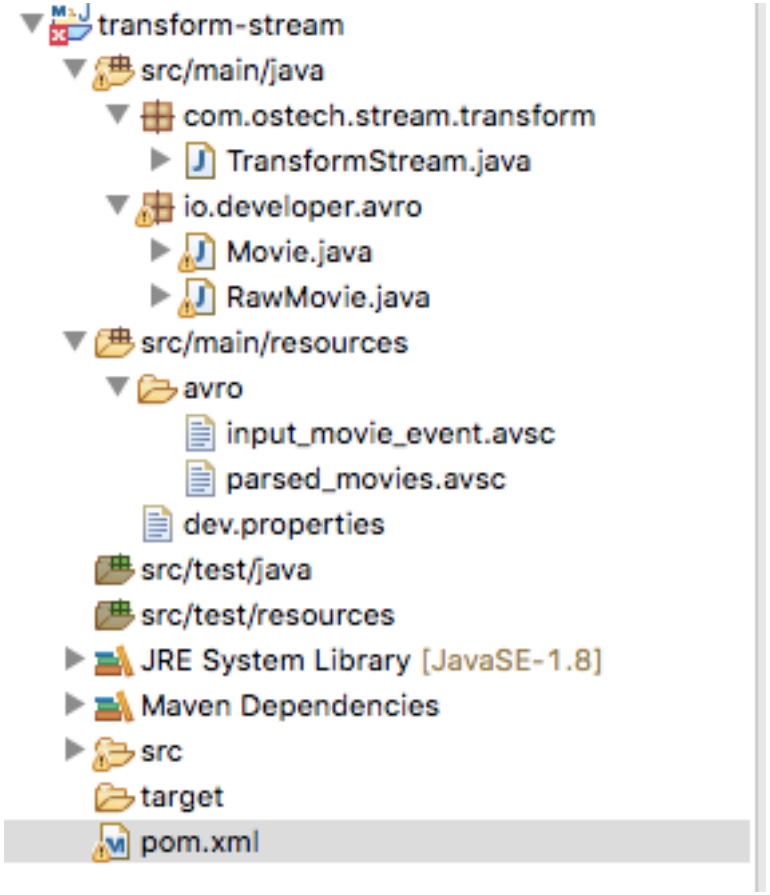
https://kafka-tutorials.confluent.io/transform-a-stream-of-events/kstreams.html

In this lab, the following topic will be implemented:

- Use Schema Registry with Avro
- Map Transformation and KeyValue Pair
- Create Topic with AdminClient API

Consider a topic with events that represent movies. Each event has a single attribute that combines its title and its release year into a string. In this tutorial, we'll write a program that creates a new topic with the title and release date turned into their own attributes.

Create a Maven Project:
com.ostech:transform-stream
At the end you should have the project structure as shown below:


transform-stream
src/main/java
com.ostech.stream.transform
TransformStream.java
io.developer.avro
Movie.java
RawMovie.java
src/main/resources
avro
input_movie_event.avsc
parsed_movies.avsc
dev.properties
src/test/java
src/test/resources
JRE System Library [JavaSE-1.8]
Maven Dependencies
src
target
pom.xml

Update the pom.xml with the following entry.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.ostech</groupId>
    <artifactId>transform-stream</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <properties>
        <!-- Keep versions as properties to allow easy modification -->
        <java.version>8</java.version>
        <avro.version>1.10.0</avro.version>
        <gson.version>2.2.4</gson.version>
        <!-- Maven properties for compilation -->
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8
        </project.reporting.outputEncoding>
        <checkstyle.suppressions.location>checkstyle/suppressions.xml
        </checkstyle.suppressions.location>
        <schemaRegistryUrl>http://localhost:8081</schemaRegistryUrl>
        <schemaRegistryBasicAuthUserInfo></schemaRegistryBasicAuthUserInfo>
        <confluent.version>5.3.0</confluent.version>
    </properties>
    <dependencies>
        <dependency>
```

```
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-clients</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams-scala_2.13</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.avro</groupId>
            <artifactId>avro</artifactId>
            <version>${avro.version}</version>
        </dependency>
        <dependency>
            <groupId>io.confluent</groupId>
            <artifactId>kafka-streams-avro-serde</artifactId>
            <version>5.2.0</version>
        </dependency>
         <dependency>
    <groupId>io.confluent</groupId>
    <artifactId>kafka-avro-serializer</artifactId>
    <version>${confluent.version}</version>
```

```
</dependency>
</dependencies>
<repositories>
    <repository>
        <id>confluent-repo</id>
        <url>https://packages.confluent.io/maven</url>
    </repository>
</repositories>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>3.8.1</version>
            <configuration>
                <source>${java.version}</source>
                <target>${java.version}</target>
                <compilerArgs>
                    <arg>-Xlint:all</arg>
                </compilerArgs>
            </configuration>
        </plugin>
        <plugin>
            <groupId>org.apache.avro</groupId>
            <artifactId>avro-maven-plugin</artifactId>
            <version>${avro.version}</version>
            <executions>
                <execution>
```

```
                        <phase>generate-sources</phase>
                        <goals>
                            <goal>schema</goal>
                        </goals>
                        <configuration>

<sourceDirectory>${project.basedir}/src/main/resources/avro/
                            </sourceDirectory>
                            <includes>
                                <include>input_movie_event.avsc</include>
                                <include>parsed_movies.avsc</include>
                            </includes>

<outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>io.confluent</groupId>
                <artifactId>kafka-schema-registry-maven-plugin</artifactId>
                <version>${confluent.version}</version>
                <configuration>
                    <schemaRegistryUrls>
                        <param>${schemaRegistryUrl}</param>
                    </schemaRegistryUrls>

<userInfoConfig>${schemaRegistryBasicAuthUserInfo}</userInfoConfig>
```

```
                        <subjects>
                            <transactions-
value>src/main/resources/avro/Payment2a.avsc
                            </transactions-value>
                        </subjects>
                    </configuration>
                    <goals>
                        <goal>test-compatibility</goal>
                    </goals>
                </plugin>
            </plugins>
        </build>

</project>
```

Then create a development file at configuration src/main/resources/dev.properties:

```
application.id=transforming-app
bootstrap.servers=localhost:9092
schema.registry.url=http://localhost:8081

input.topic.name=raw-movies
input.topic.partitions=1
input.topic.replication.factor=1

output.topic.name=movies
output.topic.partitions=1
output.topic.replication.factor=1
```

Create a schema for the events
Create a directory for the schemas that represent the events in the stream:
src/main/resources/avro

Then create the following Avro schema file at `src/main/resources/avro/input_movie_event.avsc` for the raw movies:

```
{
  "namespace": "io.developer.avro",
  "type": "record",
  "name": "RawMovie",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "title", "type": "string"},
    {"name": "genre", "type": "string"}
  ]
}
```

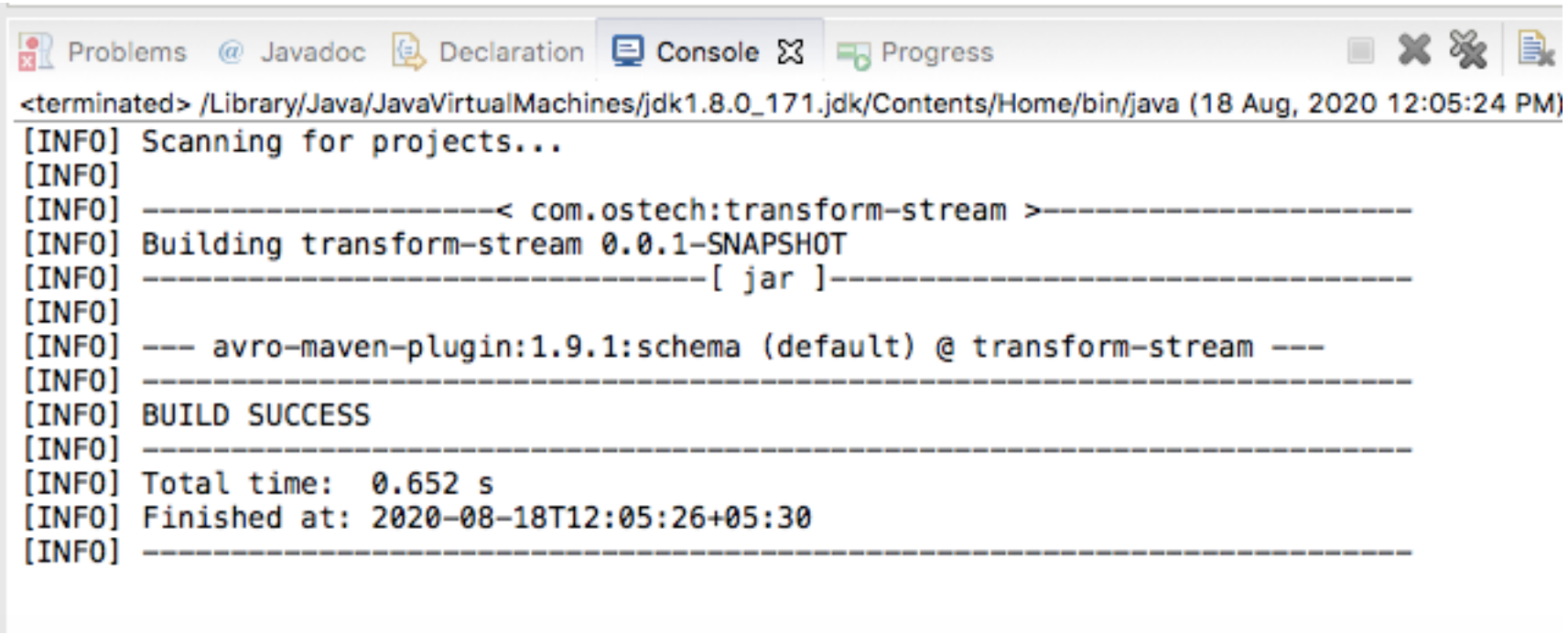
While you're at it, create another Avro schema file at `src/main/resources/avro/parsed_movies.avsc` for the transformed movies:

```
{
  "namespace": "io.developer.avro",
  "type": "record",
  "name": "Movie",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "title", "type": "string"},
```

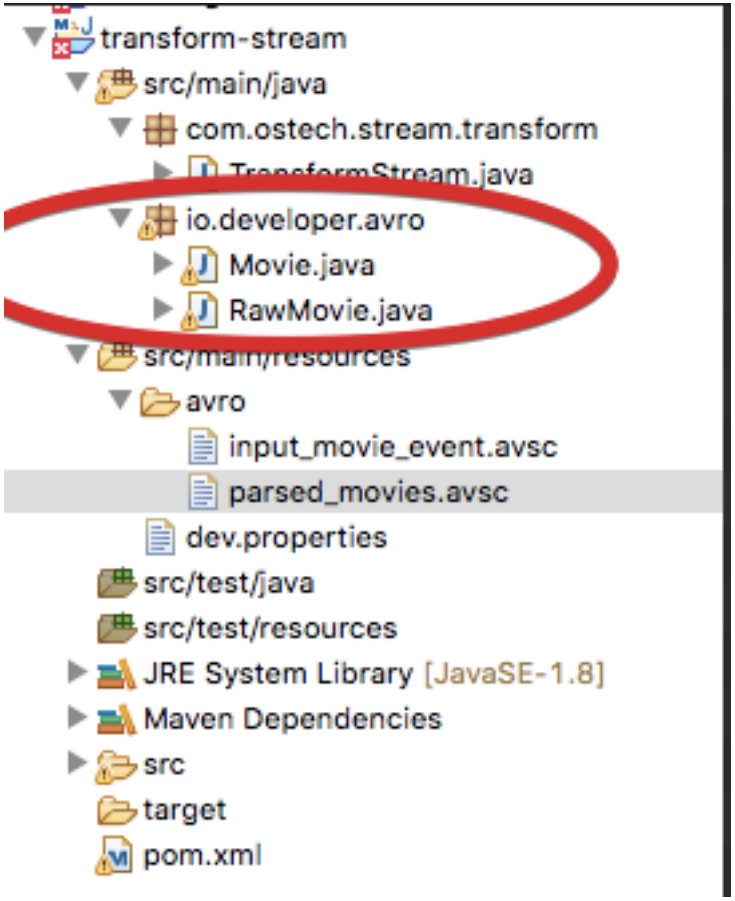
```
    {"name": "release_year", "type": "int"},
    {"name": "genre", "type": "string"}
  ]
}
```

Because we will use this Avro schema in our Java code, we'll need to compile it. The Maven Avro plugin is a part of the build, so it will see your new Avro files, generate Java code for them, and compile those and all other Java sources. Run this command to get it all done:

Maven → Generated-Sources



Ensure that generated classes don't have any compile error.


transform-stream
src/main/java
com.ostech.stream.transform
io.developer.avro
Movie.java
RawMovie.java
src/main/resources
avro
input_movie_event.avsc
parsed_movies.avsc
dev.properties
src/test/java
src/test/resources
JRE System Library [JavaSE-1.8]
Maven Dependencies
src
target
pom.xml

Create the Kafka Streams topology

Then create the following file at `src/main/java/io/confluent/developer/TransformStream.java`

```
package com.ostech.stream.transform;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.NewTopic;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Produced;

import io.confluent.kafka.serializers.AbstractKafkaAvroSerDeConfig;
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;
```

```
import io.developer.avro.Movie;
import io.developer.avro.RawMovie;

public class TransformStream {

    public Properties buildStreamsProperties(Properties envProps) {
        Properties props = new Properties();

        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
envProps.getProperty("application.id"));
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
envProps.getProperty("bootstrap.servers"));
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
SpecificAvroSerde.class);
        props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
envProps.getProperty("schema.registry.url"));

        return props;
    }

    public Topology buildTopology(Properties envProps) {
        final StreamsBuilder builder = new StreamsBuilder();
        final String inputTopic = envProps.getProperty("input.topic.name");

        KStream<String, RawMovie> rawMovies = builder.stream(inputTopic);
        KStream<Long, Movie> movies = rawMovies.map((key, rawMovie) ->
```

```
                new KeyValue<Long, Movie>(rawMovie.getId(),
convertRawMovie(rawMovie)));

      //  movies.to("movies", Produced.with(Serdes.Long(),
movieAvroSerde(envProps)));

        movies.to("movies", Produced.with(Serdes.Long(),
movieAvroSerde(envProps)));
        return builder.build();
    }

    public static Movie convertRawMovie(RawMovie rawMovie) {
        String titleParts[] =  rawMovie.getTitle().toString().split("::");
        String title = titleParts[0];
        int releaseYear = Integer.parseInt(titleParts[1]);
        return new Movie(rawMovie.getId(), title, releaseYear,
rawMovie.getGenre());
    }

    private SpecificAvroSerde<Movie> movieAvroSerde(Properties envProps) {
        SpecificAvroSerde<Movie> movieAvroSerde = new SpecificAvroSerde<>();

        final HashMap<String, String> serdeConfig = new HashMap<>();
;
serdeConfig.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
                envProps.getProperty("schema.registry.url"));

        movieAvroSerde.configure(serdeConfig, false);
```

```
        return movieAvroSerde;
    }

    public void createTopics(Properties envProps) {
        Map<String, Object> config = new HashMap<>();
        config.put("bootstrap.servers",
envProps.getProperty("bootstrap.servers"));
        AdminClient client = AdminClient.create(config);

        List<NewTopic> topics = new ArrayList<>();

        topics.add(new NewTopic(
                envProps.getProperty("input.topic.name"),

Integer.parseInt(envProps.getProperty("input.topic.partitions")),

Short.parseShort(envProps.getProperty("input.topic.replication.factor"))));

        topics.add(new NewTopic(
                envProps.getProperty("output.topic.name"),

Integer.parseInt(envProps.getProperty("output.topic.partitions")),

Short.parseShort(envProps.getProperty("output.topic.replication.factor"))));

        client.createTopics(topics);
        client.close();
    }
```

```
    public Properties loadEnvProperties(String fileName) throws IOException
{
        Properties envProps = new Properties();
        FileInputStream input = new FileInputStream(fileName);
        envProps.load(input);
        input.close();

        return envProps;
    }

    public static void main(String[] args) throws Exception {
       /* if (args.length < 1) {
            throw new IllegalArgumentException("This program takes one
argument: the path to an environment configuration file.");
        }
        */
        String propFile = "/Users/henrypotsangbam/eclipse-
workspace/LearningKafka/transform-stream/src/main/resources/dev.properties";
        TransformStream ts = new TransformStream();
        Properties envProps = ts.loadEnvProperties(propFile);
        Properties streamProps = ts.buildStreamsProperties(envProps);
        Topology topology = ts.buildTopology(envProps);

        ts.createTopics(envProps);

        final KafkaStreams streams = new KafkaStreams(topology,
streamProps);
```

```
        final CountDownLatch latch = new CountDownLatch(1);

        // Attach shutdown handler to catch Control-C.
        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-
hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```

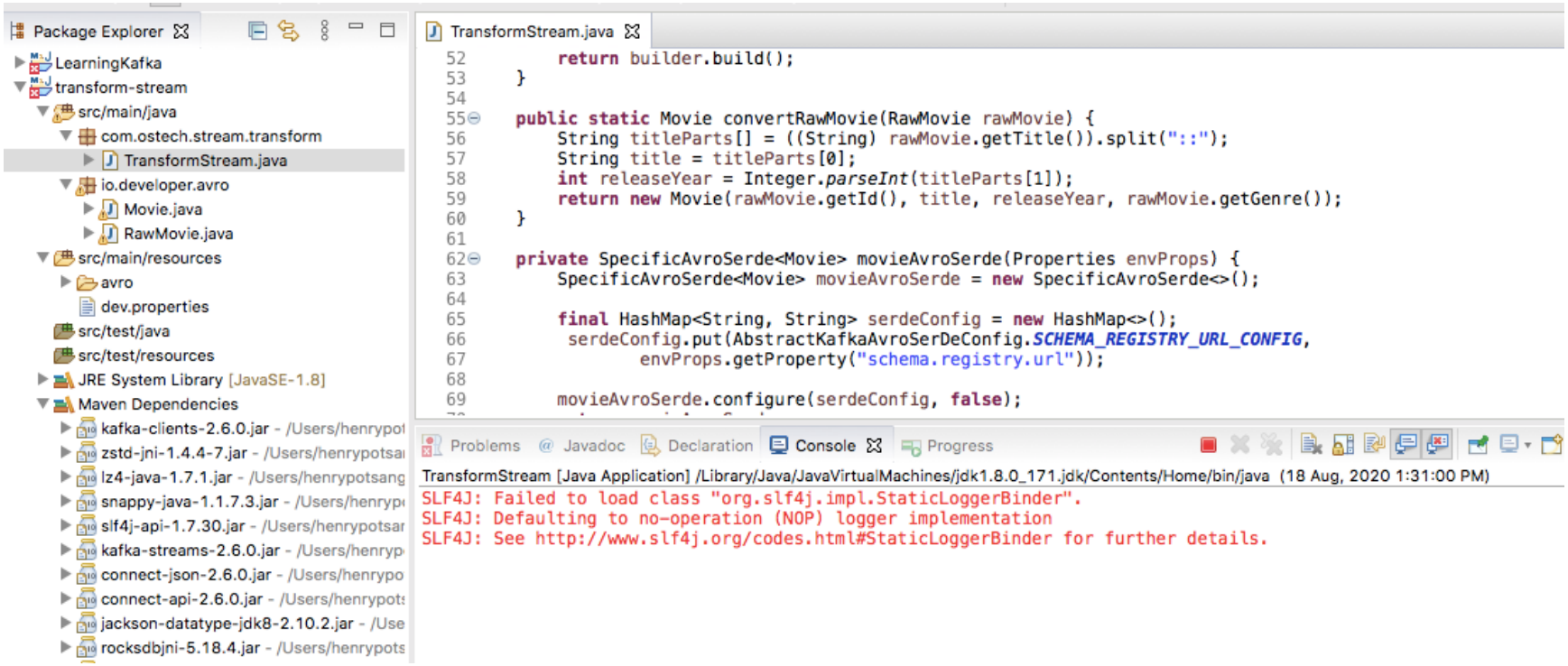
`Description :`

The first thing the method does is create an instance of `StreamsBuilder`, which is the helper object that lets us build our topology. Next we call the `stream()` method, which creates a `KStream` object (called `rawMovies` in this case) out of an underlying Kafka topic. Note the type of that stream is `Long, RawMovie`, because the topic contains the raw movie objects we want to transform. RawMovie's `title` field contains the title and the release year together, which we want to make into separate fields in a new object.

We get that transforming work done with the next line, which is a call to the `map()` method. `map()` takes each input record and creates a new stream with transformed records in it. Its parameter is a single Java Lambda that takes the input key and value and returns an instance of the `KeyValue` class with the new record in it. This does two things. First, it rekeys the incoming stream, using the `movieId` as the key. We don't absolutely need to do that to

accomplish the transformation, but it's easy enough to do at the same time, and it sets a useful key on the output stream, which is generally a good idea.
Second, it calls the `convertRawMovie()` method to turn the `RawMovie` value into a `Movie`. This is the essence of the transformation.
The `convertRawMovie()` method contains the sort of unpleasant string parsing that is a part of many stream processing pipelines, which we are happily able to encapsulate in a single, easily testable method. Any further stages we might build in the pipeline after this point are blissfully unaware that we ever had a string to parse in the first place.
Moreover, it's worth noting that we're calling `map()` and not `mapValues()`:

Compile and run the Kafka Streams program



Start the kafka.

confluent local start

Produce events to the input topic

In a new terminal, run:

Execute from your workspace directory in which Java code exists.

```
kafka-avro-console-producer --topic raw-movies --broker-list localhost:9092 --property
value.schema="$(< src/main/resources/avro/input_movie_event.avsc)"
```

[
./kafka-avro-console-producer --topic raw-movies --broker-list localhost:9092 --property value.schema="$(< /software/input_movie_event.avsc)"
]
When the console producer starts, it will log some messages and hang, waiting for your input. Type in one line at a time and press enter to send it. Each line represents an event. To send all of the events below, paste the following into the prompt and press enter:

```
{"id": 294, "title": "Die Hard::1988", "genre": "action"}
{"id": 354, "title": "Tree of Life::2011", "genre": "drama"}
{"id": 782, "title": "A Walk in the Clouds::1995", "genre": "romance"}
{"id": 128, "title": "The Big Lebowski::1998", "genre": "comedy"}
```

```
(base) Henrys-MacBook-Air:transform-stream henrypotsangbam$ kafka-avro-console-producer --topic r
aw-movies --broker-list localhost:9092 --property value.schema="$(< src/main/resources/avro/input
_movie_event.avsc)"
{"id": 294, "title": "Die Hard::1988", "genre": "action"}
{"id": 354, "title": "Tree of Life::2011", "genre": "drama"}
{"id": 782, "title": "A Walk in the Clouds::1995", "genre": "romance"}
{"id": 128, "title": "The Big Lebowski::1998", "genre": "comedy"}
```

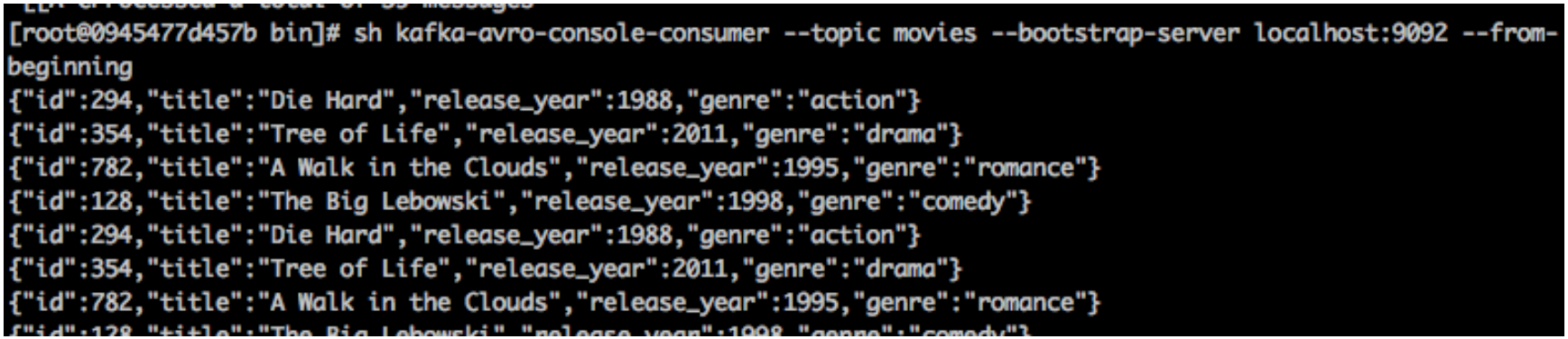
At registry log:

```
[2020-09-28 12:40:20,182] INFO HV000001: Hibernate Validator 6.0.17.Final (org.hibernate.validator.internal.util.Version:21)
[2020-09-28 12:40:20,768] INFO Started o.e.j.s.ServletContextHandler@4c9e38{/,null,AVAILABLE} (org.eclipse.jetty.server.handler.ContextHandl
er:825)
[2020-09-28 12:40:20,805] INFO Started o.e.j.s.ServletContextHandler@27aae97b{/ws,null,AVAILABLE} (org.eclipse.jetty.server.handler.ContextH
andler:825)
[2020-09-28 12:40:20,851] INFO Started NetworkTrafficServerConnector@186f8716{HTTP/1.1,[http/1.1]}{0.0.0.0:8081} (org.eclipse.jetty.server.A
bstractConnector:330)
[2020-09-28 12:40:20,852] INFO Started @7688ms (org.eclipse.jetty.server.Server:399)
[2020-09-28 12:40:20,853] INFO Server started, listening for requests... (io.confluent.kafka.schemaregistry.rest.SchemaRegistryMain:44)
[2020-09-28 12:41:04,660] INFO Registering new schema: subject raw-movies-value, version null, id null, type null (io.confluent.kafka.schema
registry.rest.resources.SubjectVersionsResource:249)
[2020-09-28 12:41:04,819] INFO 127.0.0.1 - - [28/Sep/2020:12:41:04 +0000] "POST /subjects/raw-movies-value/versions HTTP/1.1" 200 8  410 (io
.confluent.rest-utils.requests:62)
```

Observe the transformed movies in the output topic
Leave your original terminal running. To consume the events produced by your Streams application you'll need another terminal open. Use confluent kafka installation bin folder to execute the following command.

First, to consume the events of drama films, run the following:
This should yield the following messages:

```
#sh kafka-avro-console-consumer --topic movies --bootstrap-server localhost:9092 --from-beginning
```



Open another termminal, you can consume the raw-movies as shown below:

```
#sh kafka-avro-console-consumer --topic raw-movies --bootstrap-server localhost:9092 --from-beginning
```

```
(base) Henrys-MacBook-Air:~ henrypotsangbam$ kafka-avro-console-consumer --topic raw-movies --bootstrap-server localhost:9092 --from-beginning
{"id":294,"title":"Die Hard::1988","genre":"action"}
{"id":354,"title":"Tree of Life::2011","genre":"drama"}
{"id":354,"title":"Tree of Life::2011","genre":"drama"}
{"id":294,"title":"Die Hard::1988","genre":"action"}
{"id":294,"title":"Die Hard::1988","genre":"action"}
{"id":354,"title":"Tree of Life::2011","genre":"drama"}
{"id":782,"title":"A Walk in the Clouds::1995","genre":"romance"}
{"id":128,"title":"The Big Lebowski::1998","genre":"comedy"}
{"id":294,"title":"Die Hard::1988","genre":"action"}
{"id":354,"title":"Tree of Life::2011","genre":"drama"}
{"id":782,"title":"A Walk in the Clouds::1995","genre":"romance"}
{"id":128,"title":"The Big Lebowski::1998","genre":"comedy"}
```

---------- Lab Ends Here ----------

12. DSL : stateful transformations - reduce

Demonstrates how to use `reduce` to sum numbers.

Note: Use Full path with .sh extension for Kafka broker else the commands are for confluent kafka. Replace the hostname with

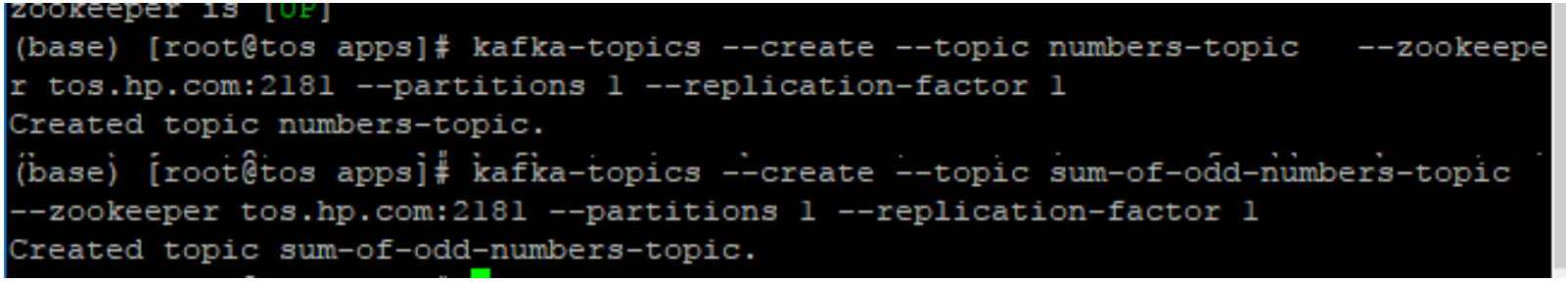
#sh kafka-topics.sh --create --topic numbers-topic --zookeeper tos.hp.com:2181 --partitions 1 --replication-factor 1

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create --topic numbers-topi
c   --zookeeper tos.master.com:2181 --partitions 1 --replication-factor 1
Created topic "numbers-topic".
[root@tos scripts]#
```

#sh kafka-topics.sh --create --topic sum-of-odd-numbers-topic --zookeeper tos.hp.com:2181 --partitions 1 --replication-factor 1

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create --topic sum-of-odd-n
umbers-topic  --zookeeper tos.master.com:2181 --partitions 1 --replication-facto
r 1
Created topic "sum-of-odd-numbers-topic".
[root@tos scripts]#
```

On Confluent Kafka Only:



We will create two java classes in this lab. You can use the transform-stream maven project in this lab.

Create a java class, SumLambdaExample and replace the code which is provided below. It Perform the transformation.

// Code Begins Here.

```
package com.tos.stream.dsl;

import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;

public class SumLambdaExample {

	static final String SUM_OF_ODD_NUMBERS_TOPIC = "sum-of-odd-numbers-topic";
	static final String NUMBERS_TOPIC = "numbers-topic";

	public static void main(final String[] args) {
	  final String bootstrapServers = args.length > 0 ? args[0] :
"192.168.139.132:9092";
	  final Properties streamsConfiguration = new Properties();
```

```
        streamsConfiguration.put(StreamsConfig.APPLICATION_ID_CONFIG, "sum-lambda-
example");
        streamsConfiguration.put(StreamsConfig.CLIENT_ID_CONFIG, "sum-lambda-
example-client");
        streamsConfiguration.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
bootstrapServers);
        streamsConfiguration.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.Integer().getClass().getName());
        streamsConfiguration.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.Integer().getClass().getName());
        streamsConfiguration.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest");
        streamsConfiguration.put(StreamsConfig.STATE_DIR_CONFIG, "\\tmp\\kafka-
streams");
        // Records should be flushed every 10 seconds. This is less than the
default
        // in order to keep this example interactive.
        streamsConfiguration.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 10 *
1000);

        final Topology topology = getTopology();
        final KafkaStreams streams = new KafkaStreams(topology,
streamsConfiguration);
        // Always (and unconditionally) clean local state prior to starting the
processing topology.
        // We opt for this unconditional call here because this will make it easier
for you to play around with the example
```

```
        // when resetting the application for doing a re-run (via the Application
Reset Tool,
        // http://docs.confluent.io/current/streams/developer-
guide.html#application-reset-tool).
        //
        // The drawback of cleaning up local state prior is that your app must
rebuilt its local state from scratch, which
        // will take time and will require reading all the state-relevant data from
the Kafka cluster over the network.
        streams.cleanUp();
        streams.start();

        // Add shutdown hook to respond to SIGTERM and gracefully close Kafka
Streams
        Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
      }

      static Topology getTopology() {
        final StreamsBuilder builder = new StreamsBuilder();
        // We assume the input topic contains records where the values are
Integers.
        // We don't really care about the keys of the input records;  for
simplicity, we assume them
        // to be Integers, too, because we will re-key the stream later on, and the
new key will be
        // of type Integer.
        final KStream<Integer, Integer> input = builder.stream(NUMBERS_TOPIC);
```

```
        final KTable<Integer, Integer> sumOfOddNumbers = input
          // We are only interested in odd numbers.
          .filter((k, v) -> v % 2 != 0)
          // We want to compute the total sum across ALL numbers, so we must re-key
all records to the
          // same key.  This re-keying is required because in Kafka Streams a data
record is always a
          // key-value pair, and KStream aggregations such as `reduce` operate on a
per-key basis.
          // The actual new key (here: `1`) we pick here doesn't matter as long it
is the same across
          // all records.
          .selectKey((k, v) -> 1)
          // no need to specify explicit serdes because the resulting key and value
types match our default serde settings
          .groupByKey()
          // Add the numbers to compute the sum.
          .reduce((v1, v2) -> v1 + v2);

        sumOfOddNumbers.toStream().to(SUM_OF_ODD_NUMBERS_TOPIC);

        return builder.build();
      }

    }
// Code Ends Here.
```

Create another class file: SumLambdaExampleDriver. It will Produce message for the input topic and consume the transform message from the Output topic.

// Code Begins Here.

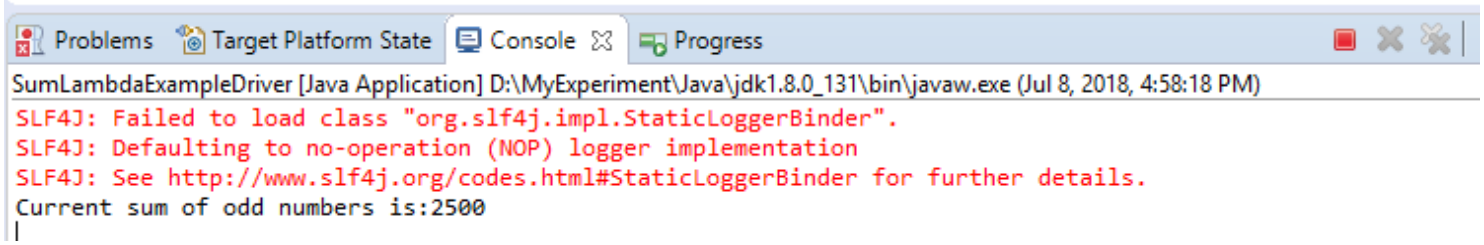
```
package com.tos.stream.dsl;

import java.time.Duration;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Properties;
import java.util.stream.IntStream;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.ConsumerRecord;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.IntegerDeserializer;
import org.apache.kafka.common.serialization.IntegerSerializer;

public class SumLambdaExampleDriver {

       public static void main(final String[] args) {
         final String bootstrapServers = args.length > 0 ? args[0] :
"192.168.139.132:9092";
```

```
        produceInput(bootstrapServers);
        consumeOutput(bootstrapServers);
      }

      private static void consumeOutput(final String bootstrapServers) {
        final Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
IntegerDeserializer.class);
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
IntegerDeserializer.class);
        properties.put(ConsumerConfig.GROUP_ID_CONFIG, "sum-lambda-example-
consumer");
        properties.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        final KafkaConsumer<Integer, Integer> consumer = new
KafkaConsumer<>(properties);

consumer.subscribe(Collections.singleton(SumLambdaExample.SUM_OF_ODD_NUMBERS_TOPIC))
;
        while (true) {
          final ConsumerRecords<Integer, Integer> records =
                  consumer.poll(Duration.ofMillis(Long.MAX_VALUE));

          for (final ConsumerRecord<Integer, Integer> record : records) {
            System.out.println("Current sum of odd numbers is:" + record.value());
          }
        }
      }
```

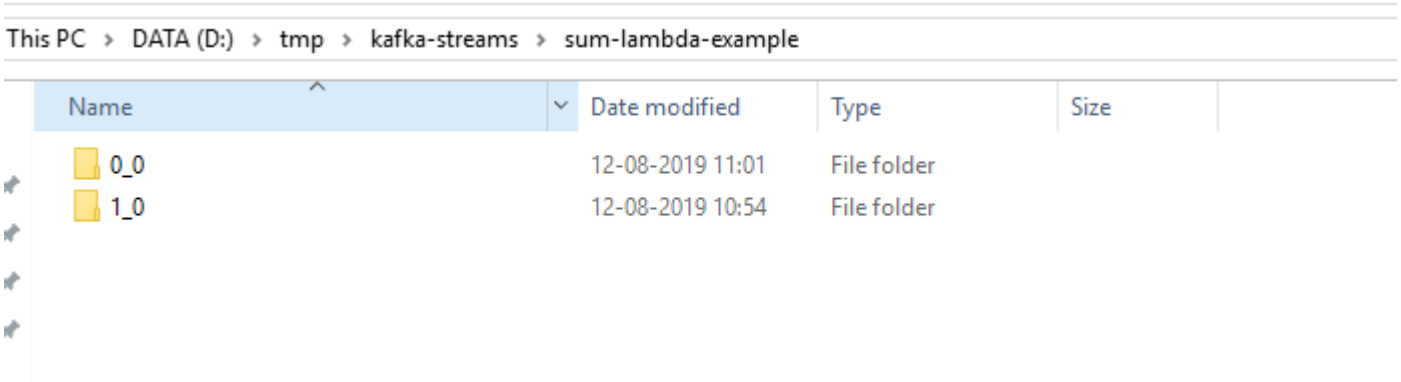
```
        private static void produceInput(final String bootstrapServers) {
          final Properties props = new Properties();
          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, bootstrapServers);
          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);
          props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
IntegerSerializer.class);

          final KafkaProducer<Integer, Integer> producer = new
KafkaProducer<>(props);

          List noList = new ArrayList();
          IntStream.range(0, 100)
                  .mapToObj(val -> new
ProducerRecord<>(SumLambdaExample.NUMBERS_TOPIC, val, val))
                  .forEach(producer::send);

          producer.flush();
        }


    }
// Code Ends Here.
```

Execute the following sequentially.

SumLambdaExample → It will listen to the input topic and perform the transformation and insert the output to the destination topic.
SumLambdaExampleDriver → It will generate message and sent to the i/p topic and consume from the o/p topic and display to the console.



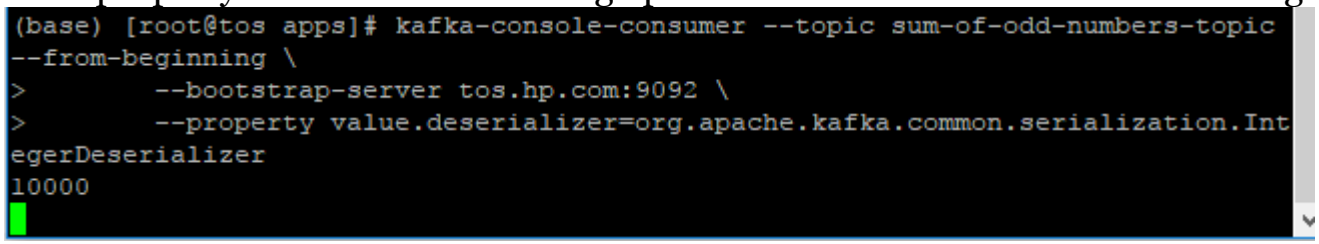
If you get error while running subsequently. Clear the following folder manually.



This is where the message is store for state full calculation of the message .i.e sum.

Inspect the resulting data in the output topics,

```
kafka-console-consumer --topic sum-of-odd-numbers-topic --from-beginning \
    --bootstrap-server tos.hp.com:9092 \
    --property value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer
```



--------- Labs End Here ----------

13. Stream – DSL and Windows

The following features will be demonstrated:

- Window
- Reduce and Filter

Demonstrates, using the high-level KStream DSL, how to implement an IoT demo application which ingests temperature value processing the maximum value in the latest TEMPERATURE_WINDOW_SIZE seconds (which * is 5 seconds) sending a new message if it exceeds the TEMPERATURE_THRESHOLD (which is 20)

In this example, the input stream reads from a topic named "iot-temperature", where the values of messages represent temperature values; using a TEMPERATURE_WINDOW_SIZE seconds "tumbling" window, the maximum value is processed and sent to a topic named "iot-temperature-max" if it exceeds the TEMPERATURE_THRESHOLD.

You can use any of the Project you have created earlier.

Create a java class TemperatureDemo.java in package: `com.ostech.windows`.

Type the following logic in the class file.

```
package com.ostech.windows;

import java.time.Duration;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;
```

```
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Produced;
import org.apache.kafka.streams.kstream.TimeWindows;
import org.apache.kafka.streams.kstream.Windowed;
import org.apache.kafka.streams.kstream.WindowedSerdes;
public class TemperatureDemo {

     // threshold used for filtering max temperature values
    private static final int TEMPERATURE_THRESHOLD = 20;
    // window size within which the filtering is applied
    private static final int TEMPERATURE_WINDOW_SIZE = 5;

    public static void main(String[] args) {

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-temperature");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

        props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);

        StreamsBuilder builder = new StreamsBuilder();
```

```
        KStream<String,String> source = builder.stream("iot-temperature");
      //  source.to("iot-temperature-max");
        final KStream<Windowed<String>, String> max = source
                // temperature values are sent without a key (null), so in order
                // to group and reduce them, a key is needed ("temp" has been chosen)
                .selectKey((key, value) -> {
                    //System.out.print(">>>" + value);
                    return "temp";

                })
                .groupByKey()
                .windowedBy(TimeWindows.of(Duration.ofSeconds(TEMPERATURE_WINDOW_SIZE)))
                .reduce((value1, value2) -> {
                    if (Integer.parseInt((String)value1) >
Integer.parseInt((String)value2)) {
                        return value1;
                    } else {
                        return value2;
                    }
                })
                .toStream()
                .filter((key, value) -> Integer.parseInt((String)(value)) >
TEMPERATURE_THRESHOLD);


        final Serde<Windowed<String>> windowedSerde  =
WindowedSerdes.timeWindowedSerdeFrom(String.class);

        // need to override key serde to Windowed type
        max.to("iot-temperature-max", Produced.with(windowedSerde, Serdes.String()));

        final KafkaStreams streams = new KafkaStreams(builder.build(), props);
```

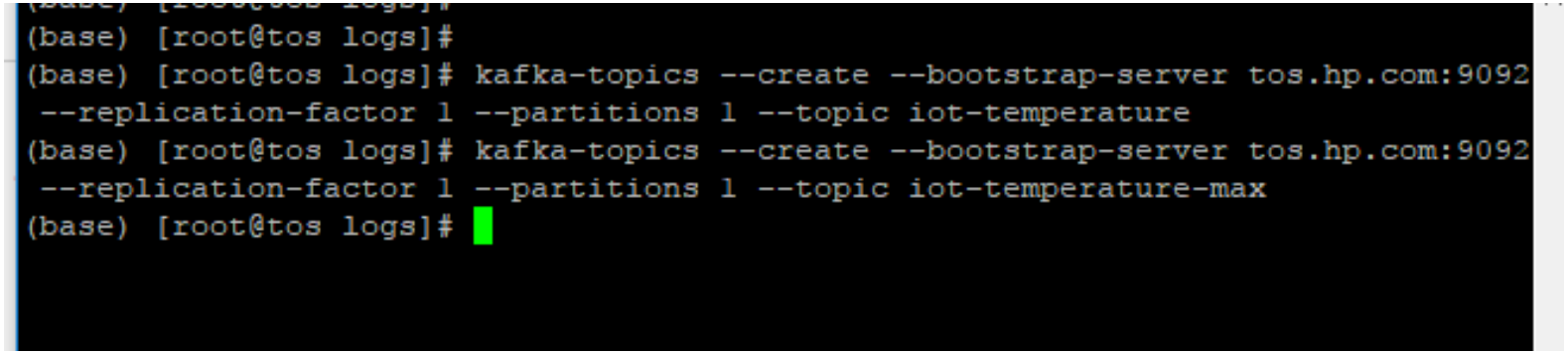
```
        final CountDownLatch latch = new CountDownLatch(1);

        // attach shutdown handler to catch control-c
        Runtime.getRuntime().addShutdownHook(new Thread("streams-temperature-shutdown-
hook") {
            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }
        System.exit(0);
    }
}
```

Execution:
Before running this example, you must create the input topic for temperature values in
Using the following command:

#sh kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic iot-temperature

And at same time create the output topic for filtered values:

#sh kafka-topics.sh --create --bootstrap-server localhost:9092 --replication-factor 1 --partitions 1 --topic iot-temperature-max



After that, a console consumer can be started in order to read filtered values from the "iot-temperature-max" topic:

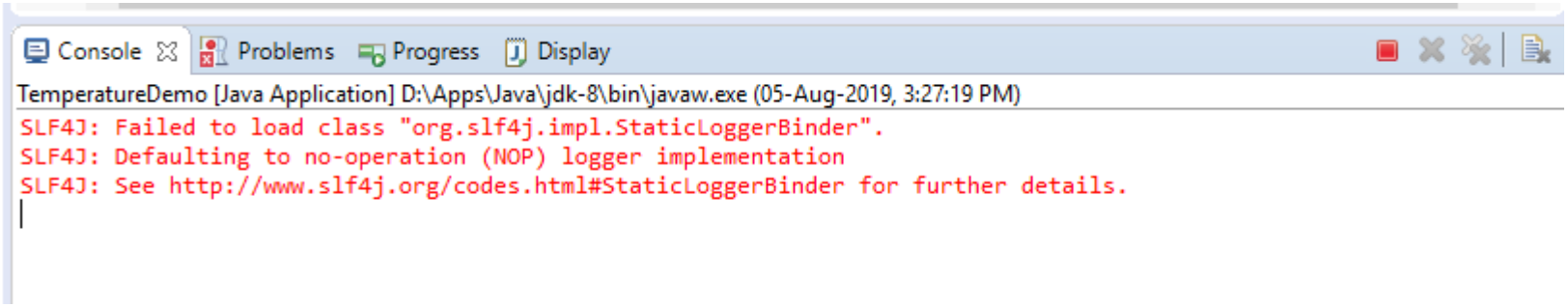
#sh kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic iot-temperature-max --from-beginning

On the other side, a console producer can be used for sending temperature values (which needs to be integers) to "iot-temperature" typing them on the console :

#sh kafka-console-producer.sh --broker-list localhost:9092 --topic iot-temperature
> 10
> 15
> 22

```
-bash: kafka-console-producer.sh: command not found
(base) [root@tos ~]# kafka-console-producer --broker-list tos.hp.com:9092 --topi
c iot-temperature
>10
>15
>22
```

Execute the java program.



You will observe the max temperature in the Consumer console.

```
(base) [root@tos logs]#  kafka-console-consumer --bootstrap-server tos.hp.com:90
92 --topic iot-temperature-max --from-beginning
22
35
```

---------- Lab Ends Here ----------

14. Stream Using – Ktable Stateless

In this lab, following actions will be performed.

- Kstream creation from the Topic
- Perform transformation using flatMap function
- Perform aggregation method
- Conversion from stream to KTable.

Logic:
Stream from the input topic, streams-plaintext-input is created and perform some transformation to it and the output is written to the topic, streams-wordcount-output.
Transformation logic:
Parse the text inserted in the input topic and count the occurrence of each word.

Create a class file with the following specification.

Package: com.ostech.ktable
Class: WordCountLambdaExample.java

Update the following with the IP address of your broker.
final String bootstrapServers = "192.168.139.132:9092";

// Update with the following code in the class file.

```
package com.ostech.ktable;

import java.util.Arrays;
```

```
import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import java.util.regex.Pattern;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Produced;

public class WordCountLambdaExample {

     final String bootstrapServers =  "localhost:9092";
     final static String inputTopic =  "streams-plaintext-input";
     final static String outputTopic =  "streams-wordcount-output";

     public static void main(String args[]) {

               Properties props = new Properties();
              props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-temperature");
              props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
              props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
              props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

              props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
              props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);

              StreamsBuilder builder = new StreamsBuilder();
```

```
          // Update with the following code in createWordCountStream method body.
            final KStream<String, String> textLines = builder.stream(inputTopic);

             final Pattern pattern = Pattern.compile("\\W+",
Pattern.UNICODE_CHARACTER_CLASS);

             final KTable<String, Long> wordCounts = textLines
             .flatMapValues(value -> Arrays.asList(pattern.split(value.toLowerCase())))
             .groupBy((keyIgnored, word) -> word)
             .count();

             // Write the `KTable<String, Long>` to the output topic.
             wordCounts.toStream().to(outputTopic, Produced.with(Serdes.String(),
Serdes.Long()));
          // Update till Here

             final KafkaStreams streams = new KafkaStreams(builder.build(), props);
            final CountDownLatch latch = new CountDownLatch(1);
             // attach shutdown handler to catch control-c
            Runtime.getRuntime().addShutdownHook(new Thread("streams-temperature-
shutdown-hook") {
                @Override
                public void run() {
                    streams.close();
                    latch.countDown();
                }
            });

            try {
                streams.start();
                latch.await();
            } catch (Throwable e) {
```

```
                System.exit(1);
            }
            System.exit(0);


    }

}
```

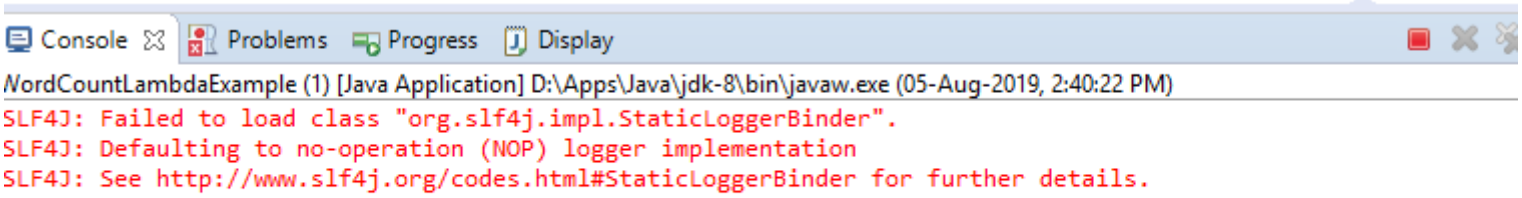
// Update till Here

Executing the Application:

Create the input and output topics used by this example. Execute the Kafka topic command from the bin folder of kafka installation folder.

```
# sh kafka-topics.sh --create --topic streams-plaintext-input \
            --zookeeper tos.hp.com:2181 --partitions 1 --replication-factor 1
# sh kafka-topics.sh --create --topic streams-wordcount-output \
            --zookeeper tos.hp.com:2181 --partitions 1 --replication-factor 1
```

Start this example application either in your IDE or in the command line.



Start the console producer. You can then enter input data by writing some line of text, followed by ENTER:

```
 * #
 * #   hello kafka streams<ENTER>
 * #   all streams lead to kafka<ENTER>
 * #   join kafka summit<ENTER>
 * #
 * # Every line you enter will become the value of a single Kafka message.
```

```
# sh kafka-console-producer.sh --broker-list tos.hp.com:9092 --topic streams-plaintext-input
```

```
(base) [root@tos ~]# kafka-console-producer --broker-list tos.hp.com:9092 --topi
c streams-plaintext-inputl
>Hello
>finally its seems to be workinhg
>Hey
>is it working now
>Great It working
>
```

Inspect the resulting data in the output topic. * You should see output data similar to below. Please note that the exact output * sequence will depend on how fast you type the above sentences. If you type them
* slowly, you are likely to get each count update, e.g., kafka 1, kafka 2, kafka 3.
* If you type them quickly, you are likely to get fewer count updates, e.g., just kafka 3.
* This is because the commit interval is set to 10 seconds. Anything typed within
* that interval will be compacted in memory.

```
# sh kafka-console-consumer.sh  --topic streams-wordcount-output --from-beginning \
                    --bootstrap-server tos.hp.com:9092 \
                   --property print.key=true \
                    --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

```
(base) [root@tos logs]# kafka-console-consumer --topic streams-wordcount-output1
 --from-beginning \
>                             --bootstrap-server tos.hp.com:9092 \
>                            --property print.key=true \
>                              --property value.deserializer=org.apache.kafka.co
mmon.serialization.LongDeserializer
hello   1
finally 1
its     1
seems   1
to      1
be      1
workinhg        1
hey     1
is      1
it      1
working 1
now     1
great   1
it      2
working 2
```

Once you're done with your experiments, you can stop this example via {@code Ctrl-C}.

---------- Lab Ends Here ----------

15. DSL - join a stream and a table together

Suppose you have a set of movies that have been released and a stream of ratings from movie-goers about how entertaining they are. In this tutorial, we'll write a program that joins each rating with content about the movie.

Initialize a maven project
com.ostech:join-stream

Update the pom.xml with the following content.

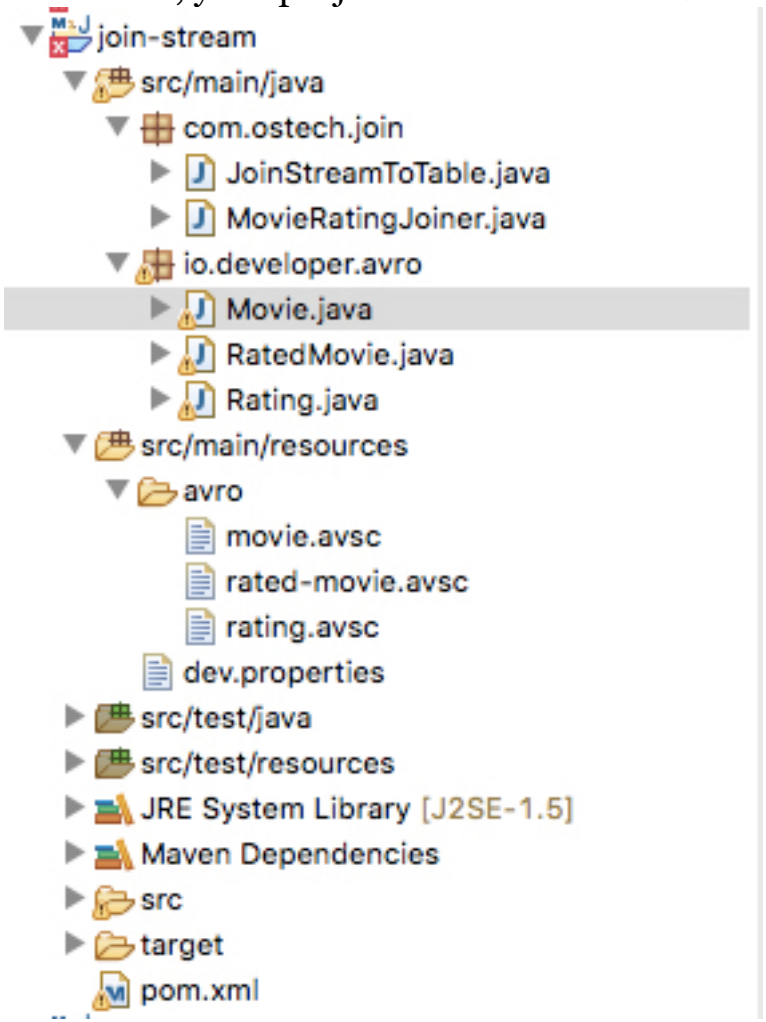
```
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.ostech</groupId>
     <artifactId>transform-stream</artifactId>
     <version>0.0.1-SNAPSHOT</version>

     <properties>
          <!-- Keep versions as properties to allow easy modification -->
          <java.version>8</java.version>
          <avro.version>1.10.0</avro.version>
          <gson.version>2.2.4</gson.version>
          <!-- Maven properties for compilation -->
          <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
          <project.reporting.outputEncoding>UTF-8
          </project.reporting.outputEncoding>
          <checkstyle.suppressions.location>checkstyle/suppressions.xml
          </checkstyle.suppressions.location>
          <schemaRegistryUrl>http://localhost:8081</schemaRegistryUrl>
          <schemaRegistryBasicAuthUserInfo></schemaRegistryBasicAuthUserInfo>
          <confluent.version>5.3.0</confluent.version>
     </properties>
     <dependencies>
          <dependency>
               <groupId>org.apache.kafka</groupId>
               <artifactId>kafka-clients</artifactId>
               <version>2.6.0</version>
          </dependency>
          <dependency>
```

```
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams</artifactId>
            <version>2.5.1</version>
        </dependency>
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams-scala_2.13</artifactId>
            <version>2.6.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.avro</groupId>
            <artifactId>avro</artifactId>
            <version>${avro.version}</version>
        </dependency>
        <dependency>
            <groupId>io.confluent</groupId>
            <artifactId>kafka-streams-avro-serde</artifactId>
            <version>5.2.0</version>
        </dependency>
       <dependency>
  <groupId>io.confluent</groupId>
  <artifactId>kafka-avro-serializer</artifactId>
  <version>${confluent.version}</version>
</dependency>
 </dependencies>
 <repositories>
      <repository>
          <id>confluent-repo</id>
          <url>https://packages.confluent.io/maven</url>
      </repository>
 </repositories>
 <build>
      <plugins>
```

```
                <plugin>
                     <groupId>org.apache.maven.plugins</groupId>
                     <artifactId>maven-compiler-plugin</artifactId>
                     <version>3.8.1</version>
                     <configuration>
                          <source>${java.version}</source>
                          <target>${java.version}</target>
                          <compilerArgs>
                               <arg>-Xlint:all</arg>
                          </compilerArgs>
                     </configuration>
                </plugin>
                <plugin>
                     <groupId>org.apache.avro</groupId>
                     <artifactId>avro-maven-plugin</artifactId>
                     <version>${avro.version}</version>
                     <executions>
                          <execution>
                               <phase>generate-sources</phase>
                               <goals>
                                    <goal>schema</goal>
                               </goals>
                               <configuration>

    <sourceDirectory>${project.basedir}/src/main/resources/avro/
                                    </sourceDirectory>
                                    <includes>
                                         <include>*.avsc</include>
                                    </includes>

    <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
                               </configuration>
                          </execution>
```

```
                    </executions>
               </plugin>
               <plugin>
                    <groupId>io.confluent</groupId>
                    <artifactId>kafka-schema-registry-maven-plugin</artifactId>
                    <version>${confluent.version}</version>
                    <configuration>
                         <schemaRegistryUrls>
                              <param>${schemaRegistryUrl}</param>
                         </schemaRegistryUrls>

     <userInfoConfig>${schemaRegistryBasicAuthUserInfo}</userInfoConfig>
                         <subjects>
                              <transactions-value>
                              </transactions-value>
                         </subjects>
                    </configuration>
                    <goals>
                         <goal>test-compatibility</goal>
                    </goals>
               </plugin>
          </plugins>
     </build>

</project>
```

At the end, your project structure should be as shown below:



Create a development file at src/main/resources/dev.properties

```
application.id=joining-app
bootstrap.servers=localhost:9092
schema.registry.url=http://localhost:8081

movie.topic.name=movies
movie.topic.partitions=1
movie.topic.replication.factor=1

rekeyed.movie.topic.name=rekeyed-movies
rekeyed.movie.topic.partitions=1
rekeyed.movie.topic.replication.factor=1

rating.topic.name=ratings
rating.topic.partitions=1
rating.topic.replication.factor=1

rated.movies.topic.name=rated-movies
rated.movies.topic.partitions=1
rated.movies.topic.replication.factor=1
```

Create a schema for the events

This tutorial uses three streams: one called movies that holds movie reference data, one called ratings that holds a stream of inbound movie ratings, and one called rated-movies that holds the result of the join between ratings and movies. Let's create schemas for all three.
Create a directory for the schemas that represent the events in the stream:

Src/java/main/resources/avro

Then create the following Avro schema file at `src/main/resources/avro/movie.avsc` for the movies lookup table:

```
{
  "namespace": "io.confluent.developer.avro",
  "type": "record",
  "name": "Movie",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "title", "type": "string"},
    {"name": "release_year", "type": "int"}
  ]
}
```

Next, create another Avro schema file at `src/main/resources/avro/rating.avsc` for the stream of ratings:

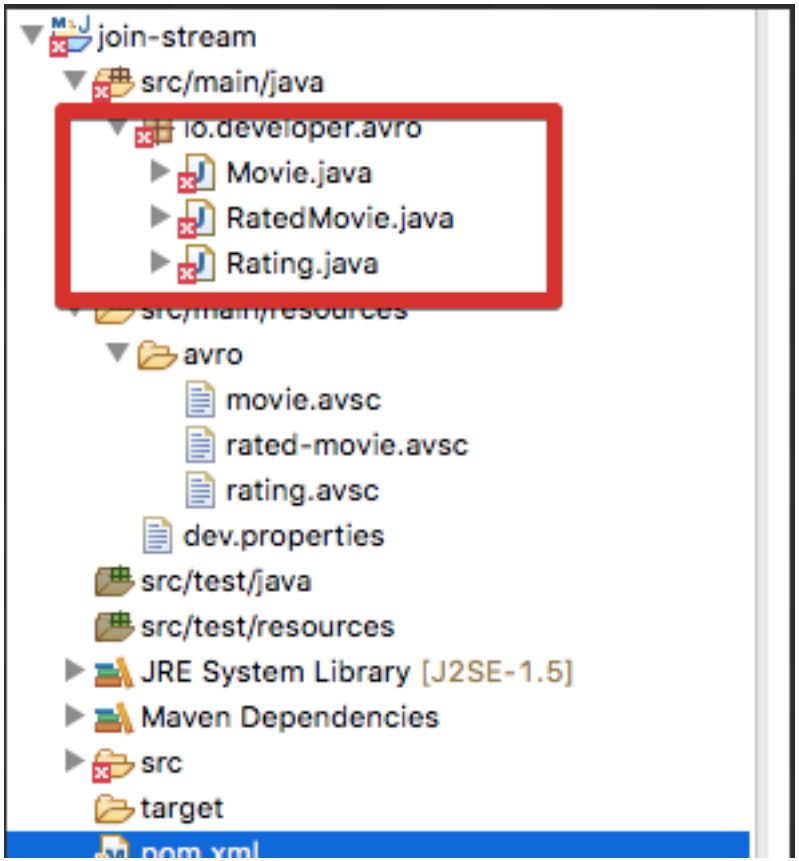
```
{
  "namespace": "io.developer.avro",
  "type": "record",
  "name": "Rating",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "rating", "type": "double"}
  ]
}
```

And finally, create another Avro schema file at `src/main/resources/avro/rated-movie.avsc` for the result of the join:

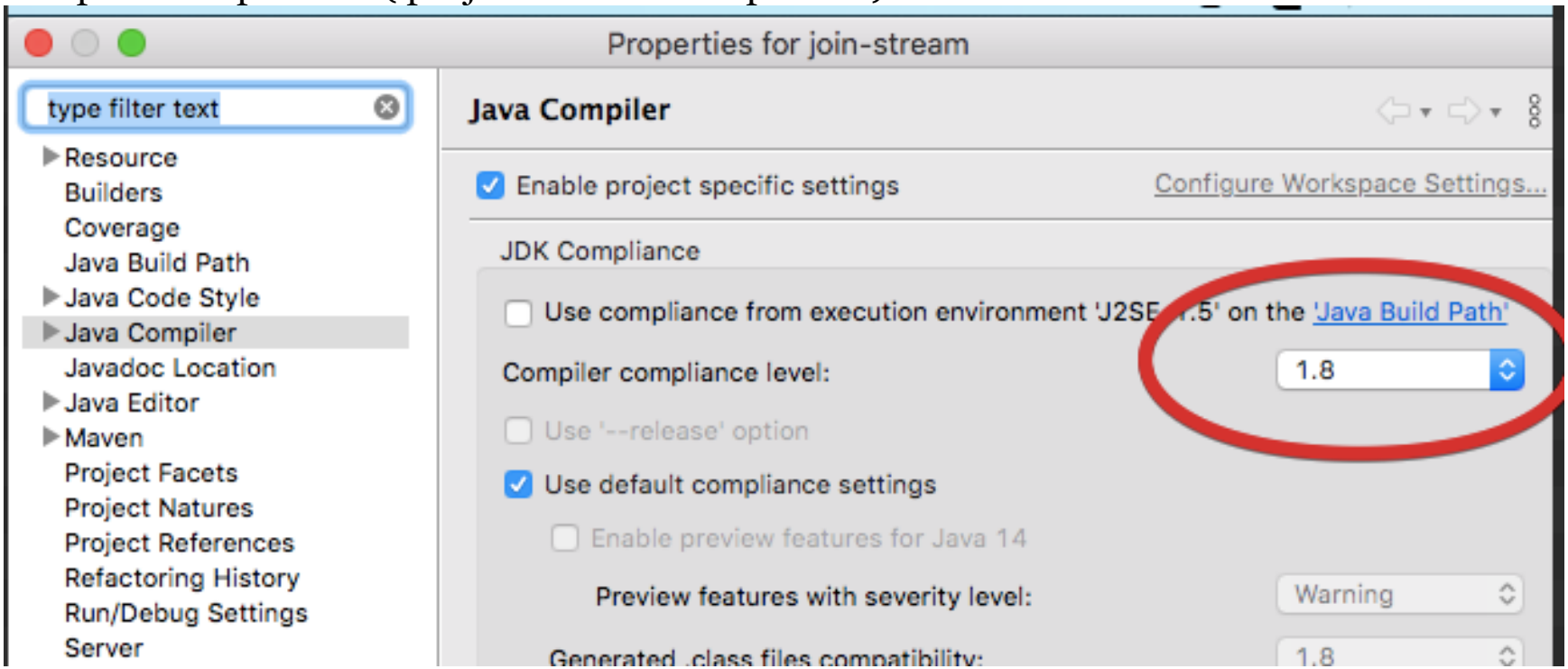
```
{
  "namespace": "io.developer.avro",
  "type": "record",
  "name": "RatedMovie",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "title", "type": "string"},
    {"name": "release_year", "type": "int"},
    {"name": "rating", "type": "double"}
  ]
}
```

Because we will use this Avro schema in our Java code, we'll need to compile it. The Gradle Maven plugin is a part of the build, so it will see your new Avro files, generate Java code for them, and compile those and all other Java sources. Run this command to get it all done:

```
# mvn generated-sources
```



If you observe any compile error like above, ensure to convert the project into java 1.8 compiler compatible. (project name → Properties)



Create the Kafka Streams topology.(Package Name/ Java Class)

`com.ostech.join/JoinStreamToTable.java`

```
package com.ostech.join;

import java.io.FileInputStream;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.CountDownLatch;

import org.apache.kafka.clients.admin.AdminClient;
import org.apache.kafka.clients.admin.NewTopic;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Produced;

import io.confluent.kafka.serializers.AbstractKafkaAvroSerDeConfig;
```

```
import io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde;
import io.developer.avro.Movie;
import io.developer.avro.RatedMovie;
import io.developer.avro.Rating;

public class JoinStreamToTable {

    public Properties buildStreamsProperties(Properties envProps) {
        Properties props = new Properties();

        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
envProps.getProperty("application.id"));
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
envProps.getProperty("bootstrap.servers"));
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
SpecificAvroSerde.class);
        props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
envProps.getProperty("schema.registry.url"));

        return props;
    }

    public Topology buildTopology(Properties envProps) {
        final StreamsBuilder builder = new StreamsBuilder();
        final String movieTopic = envProps.getProperty("movie.topic.name");
        final String rekeyedMovieTopic =
```

```
envProps.getProperty("rekeyed.movie.topic.name");
        final String ratingTopic =
envProps.getProperty("rating.topic.name");
        final String ratedMoviesTopic =
envProps.getProperty("rated.movies.topic.name");
        final MovieRatingJoiner joiner = new MovieRatingJoiner();

        KStream<String, Movie> movieStream = builder.<String,
Movie>stream(movieTopic)
                .map((key, movie) -> new
KeyValue<>(Long.valueOf(movie.getId()).toString(), movie));

        movieStream.to(rekeyedMovieTopic);

        KTable<String, Movie> movies = builder.table(rekeyedMovieTopic);

        KStream<String, Rating> ratings = builder.<String,
Rating>stream(ratingTopic)
                .map((key, rating) -> new
KeyValue<>(Long.valueOf(rating.getId()).toString(), rating));

        KStream<String, RatedMovie> ratedMovie = ratings.join(movies,
joiner);

        // System.out.println(">>>"+ ratedMovie);
        ratedMovie.to(ratedMoviesTopic, Produced.with(Serdes.String(),
ratedMovieAvroSerde(envProps)));
```

```
        return builder.build();
    }

    private SpecificAvroSerde<RatedMovie> ratedMovieAvroSerde(Properties
envProps) {
        SpecificAvroSerde<RatedMovie> movieAvroSerde = new
SpecificAvroSerde<>();

        final HashMap<String, String> serdeConfig = new HashMap<>();

    serdeConfig.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG,
                envProps.getProperty("schema.registry.url"));

        movieAvroSerde.configure(serdeConfig, false);
        return movieAvroSerde;
    }

    public void createTopics(Properties envProps) {
        Map<String, Object> config = new HashMap<>();
        config.put("bootstrap.servers",
envProps.getProperty("bootstrap.servers"));
        AdminClient client = AdminClient.create(config);

        List<NewTopic> topics = new ArrayList<>();

        topics.add(new NewTopic(envProps.getProperty("movie.topic.name"),

    Integer.parseInt(envProps.getProperty("movie.topic.partitions")),
```

```
    Short.parseShort(envProps.getProperty("movie.topic.replication.factor"))
));

        topics.add(new
NewTopic(envProps.getProperty("rekeyed.movie.topic.name"),

    Integer.parseInt(envProps.getProperty("rekeyed.movie.topic.partitions"))
,

    Short.parseShort(envProps.getProperty("rekeyed.movie.topic.replication.f
actor"))));

        topics.add(new NewTopic(envProps.getProperty("rating.topic.name"),

    Integer.parseInt(envProps.getProperty("rating.topic.partitions")),

    Short.parseShort(envProps.getProperty("rating.topic.replication.factor")
)));

        topics.add(new
NewTopic(envProps.getProperty("rated.movies.topic.name"),

    Integer.parseInt(envProps.getProperty("rated.movies.topic.partitions")),

    Short.parseShort(envProps.getProperty("rated.movies.topic.replication.fa
ctor"))));
```

```
        client.createTopics(topics);
        client.close();
    }

    public Properties loadEnvProperties(String fileName) throws IOException
{
        Properties envProps = new Properties();
        FileInputStream input = new FileInputStream(fileName);
        envProps.load(input);
        input.close();

        return envProps;
    }

    public static void main(String[] args) throws Exception {

        String propFile = "/Users/henrypotsangbam/eclipse-
workspace/LearningKafka/join-stream/src/main/resources/dev.properties";

        JoinStreamToTable ts = new JoinStreamToTable();
        Properties envProps = ts.loadEnvProperties(propFile);
        Properties streamProps = ts.buildStreamsProperties(envProps);
        Topology topology = ts.buildTopology(envProps);

        ts.createTopics(envProps);

        final KafkaStreams streams = new KafkaStreams(topology,
streamProps);
```

```
        final CountDownLatch latch = new CountDownLatch(1);

        // Attach shutdown handler to catch Control-C.
        Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-
hook1") {

            @Override
            public void run() {
                streams.close();
                latch.countDown();
            }
        });

        try {
            streams.start();
            latch.await();
        } catch (Throwable e) {
            System.exit(1);
        }

        System.exit(0);

    }
}
```

The first thing the method does is create an instance of StreamsBuilder, which is the helper object that lets us build our topology. With our builder in hand, there are three things we need to do. First, we call the stream() method to create a KStream<String, Movie> object. The problem is that we can't make any assumptions about the key of this stream, so we have to repartition it explicitly. We use the map() method for that, creating a new KeyValue instance for each record, using the movie ID as the new key.
The movies start their life in a stream, but fundamentally, movies are entities that belong in a table. To turn them into a table, we first emit the rekeyed stream to a Kafka topic using the to() method. We can then use the builder.table() method to create a KTable<String,Movie>. We have successfully turned a topic full of movie entities into a scalable, key-addressable table of Movie objects. With that, we're ready to move on to ratings.
Creating the KStream<String,Rating> of ratings looks just like our first step with the movies: we create a stream from the topic, then repartition it with the map() method. Note that we must choose the same key—movie ID—for our join to work.
With the ratings stream and the movie table in hand, all that remains is to join them using the join() method. It's a wonderfully simply one-liner, but we have concealed a bit of complexity in the form of the MovieRatingJoiner class.

Implement a ValueJoiner class

For the ValueJoiner class, create the following file MovieRatingJoiner.java

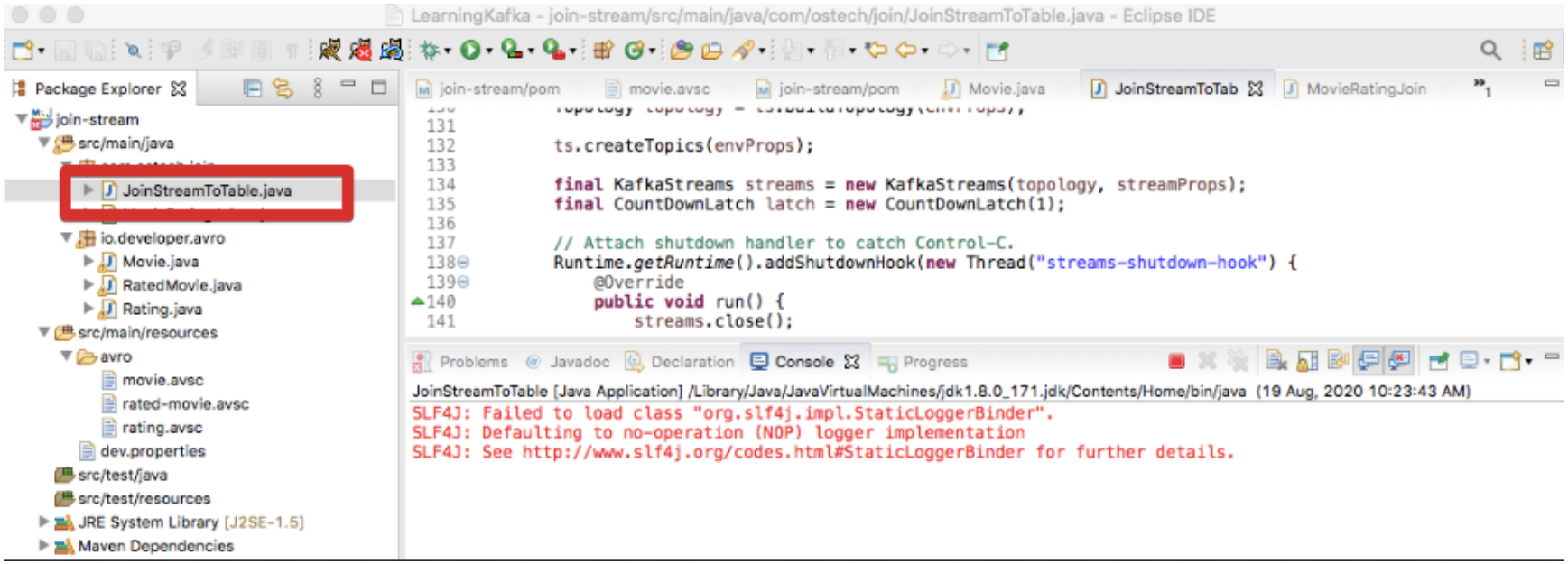
```
package com.ostech.join;

import org.apache.kafka.streams.kstream.ValueJoiner;

import io.developer.avro.Movie;
import io.developer.avro.RatedMovie;
import io.developer.avro.Rating;

public class MovieRatingJoiner implements ValueJoiner<Rating, Movie,
RatedMovie> {
    public RatedMovie apply(Rating rating, Movie movie) {
        return RatedMovie.newBuilder()
                .setId(movie.getId())
                .setTitle(movie.getTitle())
                .setReleaseYear(movie.getReleaseYear())
                .setRating(rating.getRating())
                .build();
    }
}
```

When you join two tables in a relational database, by default you get a new table containing all of the columns of the left table plus all of the columns of the right table. When you join a stream and a table, you get a new stream, but you must be explicit about the value of that

stream—the combination between the value in the stream and the assocaited value in the table. The ValueJoiner interface in the Streams API does this work. The single apply() method takes the stream and table values as parameters, and returns the value of the joined stream as output. (Their keys are not a part of the equation, because they are equal by definition and do not change in the result.) As you can see here, this is just a matter of creating a RatedMovie object and populating it with the relevant fields of the input movie and rating.

You can do this in a Java Lambda in the call to the join() method where you're building the stream topology, but the joining logic may become complex, and breaking it off into its own trivially testable class is a good move.

Compile and run the Kafka Streams program



Load in some movie reference data

Start your zookeeper, kafka broker and Registry server for this.

Create a movie.avsc schema file in /software folder with the following content.

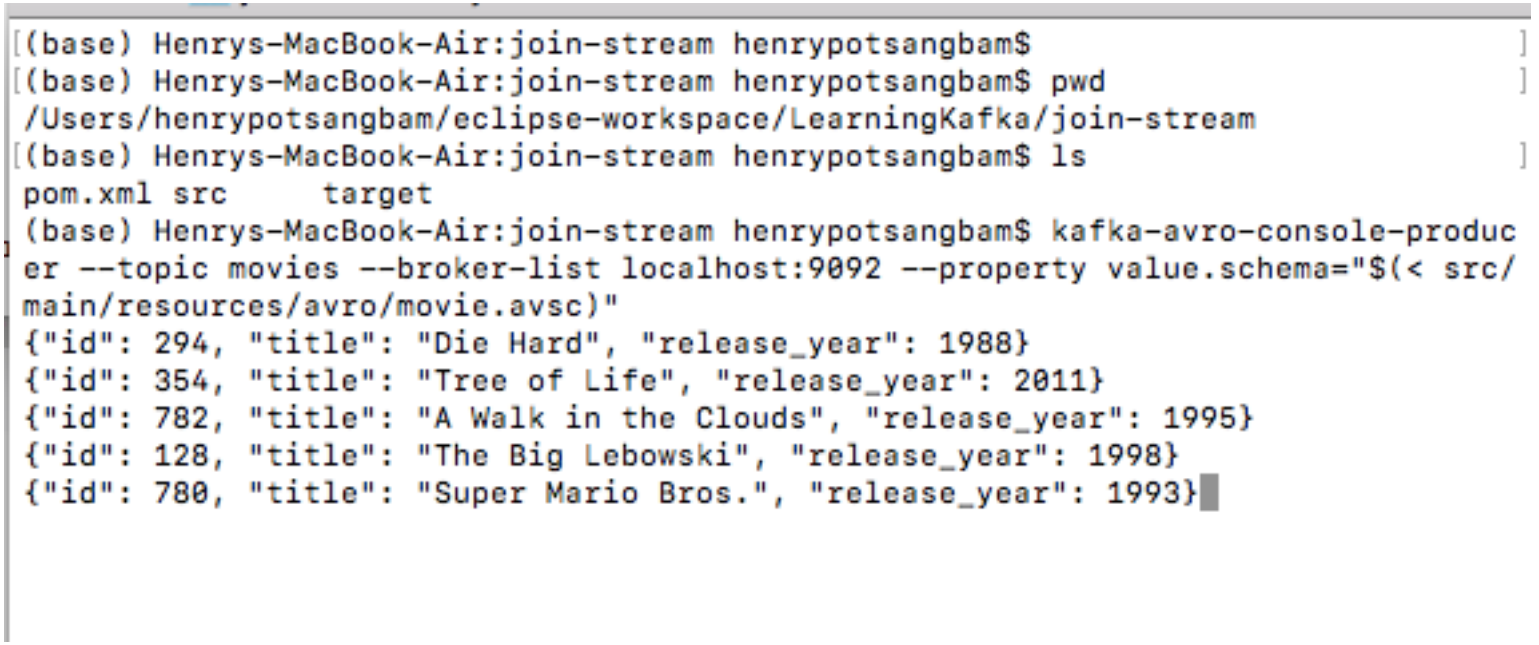
```
{
  "namespace": "io.developer.avro",
  "type": "record",
  "name": "Movie",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "title", "type": "string"},
    {"name": "release_year", "type": "int"}
  ]
}
```

In a new terminal (Use confluent bin folder), run:

```
# sh kafka-avro-console-producer --topic movies --broker-list localhost:9092 --property value.schema="$(< /software/movie.avsc)"
```

When the console producer starts, it will log some messages and hang, waiting for your input. Copy and paste one line at a time and press enter to send it. Note that these lines contain hard tabs between the key and the value, so retyping them without the tab will not work.
Each line represents a movie we will be able to rate. To send all of the events below, paste the following into the prompt and press enter:

```
{"id": 294, "title": "Die Hard", "release_year": 1988}
{"id": 354, "title": "Tree of Life", "release_year": 2011}
{"id": 782, "title": "A Walk in the Clouds", "release_year": 1995}
{"id": 128, "title": "The Big Lebowski", "release_year": 1998}
{"id": 780, "title": "Super Mario Bros.", "release_year": 1993}
```



Get ready to observe the rated movies in the output topic

Before you start producing ratings, it's a good idea to set up the consumer on the output topic. This way, as soon as you produce ratings (and they're joined to movies), you'll see the results right away. Run this to get ready to consume the rated movies:

```
# sh kafka-avro-console-consumer --topic rated-movies --bootstrap-server localhost:9092 --from-beginning
```

```
Last login: Wed Aug 19 10:48:49 on ttys001
(base) Henrys-MacBook-Air:~ henrypotsangbam$ kafka-avro-console-consumer --topic
 rated-movies --bootstrap-server localhost:9092 --from-beginning
```

You won't see any results until the next step.

Produce some ratings to the input topic

Run the following in a new terminal window. This process is the most fun if you can see this and the previous terminal (which is consuming the rated movies) at the same time. If your terminal program lets you do horizontal split panes, try it that way:

Create a rating.avsc schema file in /software folder.

```
{
  "namespace": "io.developer.avro",
  "type": "record",
  "name": "Rating",
  "fields": [
    {"name": "id", "type": "long"},
    {"name": "rating", "type": "double"}
  ]
}
```

```
#sh kafka-avro-console-producer --topic ratings --broker-list localhost:9092 --property
value.schema="$(< /software/rating.avsc)"
```

When the producer starts up, copy and paste these lines into the terminal. Try doing them one at a time, observing the results in the consumer terminal:

```
{"id": 294, "rating": 8.2}
{"id": 294, "rating": 8.5}
{"id": 354, "rating": 9.9}
{"id": 354, "rating": 9.7}
{"id": 782, "rating": 7.8}
{"id": 782, "rating": 7.7}
{"id": 128, "rating": 8.7}
```

```
{"id": 128, "rating": 8.4}
{"id": 780, "rating": 2.1}
```

Speaking of that consumer terminal, these are the results you should see there if you paste in all the movies and ratings as shown in this tutorial:

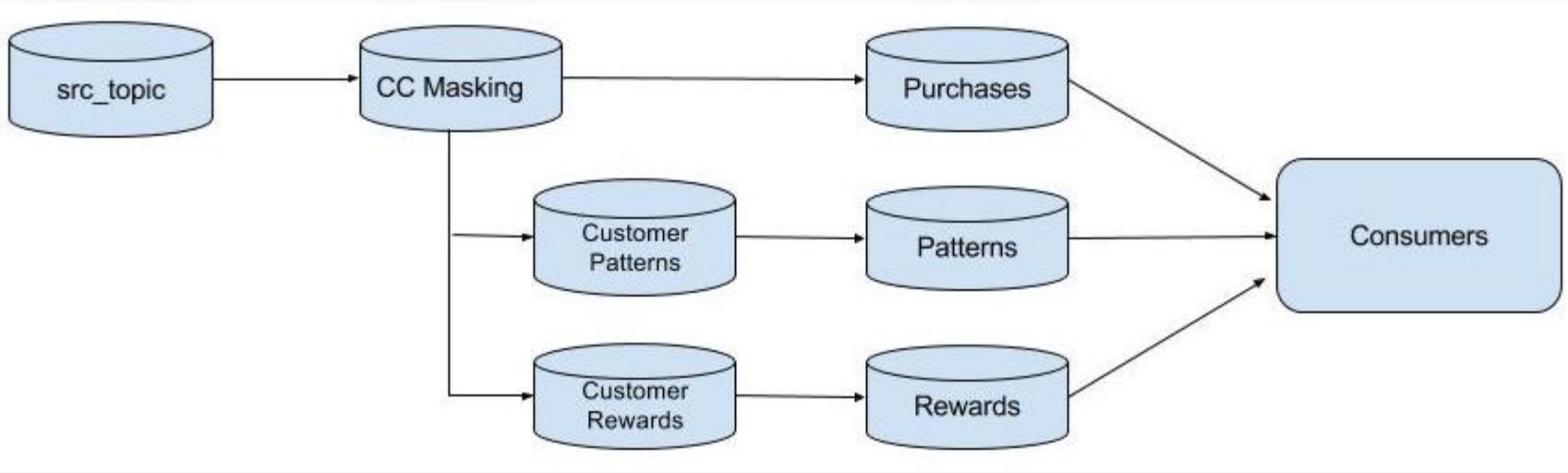
```
{"id":294,"title":"Die Hard","release_year":1988,"rating":8.2}
{"id":294,"title":"Die Hard","release_year":1988,"rating":8.5}
{"id":354,"title":"Tree of Life","release_year":2011,"rating":9.9}
{"id":354,"title":"Tree of Life","release_year":2011,"rating":9.7}
{"id":782,"title":"A Walk in the Clouds","release_year":1995,"rating":7.8}
{"id":782,"title":"A Walk in the Clouds","release_year":1995,"rating":7.7}
{"id":128,"title":"The Big Lebowski","release_year":1998,"rating":8.7}
{"id":128,"title":"The Big Lebowski","release_year":1998,"rating":8.4}
{"id":780,"title":"Super Mario Bros.","release_year":1993,"rating":2.1}
```

---------- Lab Ends Here ----------

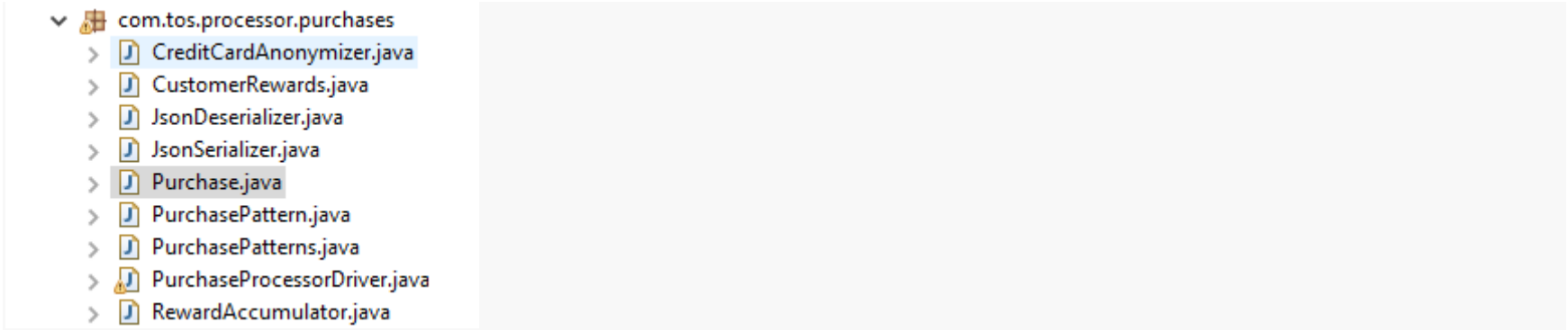
17. Stream API – Stateless Custom Processor

In this tutorial, we'll transform fictitious customer purchase data and perform the following actions:

1. A processor to mask credit card numbers.
2. A processor to collect the customer name and the amount spent to use in a rewards program.
3. A processor to collect the zip code and the item purchased to help determine shopping patterns.



Here's the code for our 3 processors, Import the following class in your Project. You need to dump all classes in the com.tos.processor.purchases package. At the end you should have the following classes.


com.tos.processor.purchases
CreditCardAnonymizer.java
CustomerRewards.java
JsonDeserializer.java
JsonSerializer.java
Purchase.java
PurchasePattern.java
PurchasePatterns.java
PurchaseProcessorDriver.java
RewardAccumulator.java

CreditCardAnonymizer – Processor

```
// Code Start Here ******************************************
package com.tos.processor.purchases;
import org.apache.kafka.streams.processor.AbstractProcessor;


public class CreditCardAnonymizer extends AbstractProcessor<String, Purchase> {

    private static final String CC_NUMBER_REPLACEMENT="xxxx-xxxx-xxxx-";

    @Override
    public void process(String key, Purchase purchase) {
          String last4Digits = purchase.getCreditCardNumber().split("-")[3];
          Purchase updated =
Purchase.builder(purchase).creditCardNumber(CC_NUMBER_REPLACEMENT+last4Digits).build
();
          context().forward(key,updated);
          context().commit();
    }
}
// Code Ends Here ******************************************
```

CustomerRewards - Processor

```
// Code Start Here ********************************************

package com.tos.processor.purchases;


import org.apache.kafka.streams.processor.AbstractProcessor;


public class CustomerRewards extends AbstractProcessor<String,Purchase> {

	@Override
	public void process(String key, Purchase value) {
		RewardAccumulator accumulator = RewardAccumulator.builder(value).build();
		context().forward(key,accumulator);
		context().commit();

	}
}
// Code Ends Here **********************************************
```

Purchase Processor.

```
// Code Start Here ********************************************

package com.tos.processor.purchases;
import org.apache.kafka.streams.processor.AbstractProcessor;
public class PurchasePatterns extends AbstractProcessor<String, Purchase> {

    @Override
    public void process(String key, Purchase value) {
        PurchasePattern purchasePattern =
PurchasePattern.newBuilder().date(value.getPurchaseDate())
                .item(value.getItemPurchased())
                .zipCode(value.getZipCode()).build();
        context().forward(key, purchasePattern);
        context().commit();
    }
}
// Code Ends Here **********************************************
```

Let's briefly describe the structure of the processor objects. All three processors extend the AbstractProcessor class, which provides no-op overrides for the `punctuate` and `close` methods. In this example we just need to implement the `process` method, where the action is performed on each message. After work is completed, the `context().forward` method is called which forwards the modified/new key-value pair to downstream consumers. (The `context()` method retrieves the `context` instance variable initialized in the parent class by the `init` method).
Then `context().commit` method is called, committing the current state of the stream including the message offset.

Building the Graph of Processors

Now we need to define the DAG to determine the flow of messages. To build our graph of processing nodes we use the Toplogy. Although our messages are json, we need to define Serializer and Desrializer instances since the processors work with types. Here's the code from the PurchaseProcessorDriver that builds the graph topology and serializers/deserializers.

PurchaseProcessorDriver

```
// Code Start Here ********************************************

package com.tos.processor.purchases;

import java.util.Properties;

import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.processor.WallclockTimestampExtractor;

public class PurchaseProcessorDriver {

     public static void main(String[] args) throws Exception {


          JsonDeserializer<Purchase> purchaseJsonDeserializer = new
JsonDeserializer<>(Purchase.class);
          JsonSerializer<Purchase> purchaseJsonSerializer = new JsonSerializer<>();

          JsonSerializer<RewardAccumulator> rewardAccumulatorJsonSerializer = new
JsonSerializer<>();


          JsonSerializer<PurchasePattern> purchasePatternJsonSerializer = new
JsonSerializer<>();
```

```
        StringDeserializer stringDeserializer = new StringDeserializer();
        StringSerializer stringSerializer = new StringSerializer();

        Topology topologyBuilder = new Topology();

       topologyBuilder.addSource("SOURCE", stringDeserializer,
purchaseJsonDeserializer, "src-topic")

                .addProcessor("PROCESS", CreditCardAnonymizer::new, "SOURCE")
                .addProcessor("PROCESS2", PurchasePatterns::new, "PROCESS")
               .addProcessor("PROCESS3", CustomerRewards::new, "PROCESS")

                .addSink("SINK", "patterns", stringSerializer,
purchasePatternJsonSerializer, "PROCESS2")
               .addSink("SINK2", "rewards", stringSerializer,
rewardAccumulatorJsonSerializer, "PROCESS3")
               .addSink("SINK3", "purchases", stringSerializer,
purchaseJsonSerializer, "PROCESS");

        System.out.println("Starting PurchaseProcessor Example");

        KafkaStreams streaming = new KafkaStreams(topologyBuilder, getProperties());
        streaming.setUncaughtExceptionHandler((Thread thread, Throwable throwable) -> {
            // here you should examine the throwable/exception and perform an
appropriate
            // action!
            System.out.println(" Error :" + throwable);
        });
        streaming.start();
        System.out.println("Now started PurchaseProcessor Example");

        // Stop the Kafka Streams threads
        // Add shutdown hook to stop the Kafka Streams threads.
```

```
            // You can optionally provide a timeout to `close`.
            Runtime.getRuntime().addShutdownHook(new Thread(streaming::close));

        }

        private static Properties getProperties() {
            Properties props = new Properties();
            props.put(StreamsConfig.CLIENT_ID_CONFIG, "Example-Processor-Job");
            props.put("group.id", "test-consumer-group");
            props.put(StreamsConfig.APPLICATION_ID_CONFIG, "testing-processor-api");
            props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
            props.put(StreamsConfig.REPLICATION_FACTOR_CONFIG, 1);

            props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
WallclockTimestampExtractor.class);
        //props.put("default.deserialization.exception.handler",
LogAndContinueExceptionHandler.class);
            return props;
        }


}

// Code Ends Here **********************************************
```

There's several steps here, so let's do a quick walk through

1. On line 11 we add a source node named "SOURCE" with a StringDeserializer for the keys and JsonSerializer genericized to work with Purchase objects and one to N number of topics that will feed this source node. In this case we are using input from one topic, "src-topic".

2. Next we start adding processor nodes. The `addProcessor` method takes a String for the name, a ProcessorSupplier and one to N number of parent nodes. Here the first processor is a child of the "SOURCE" node, but is a parent of the next two processors. A quick note here about the syntax for our ProcessorSupplier. The code is leveraging method handles which can used as lambda expressions for Supplier instances in Java 8. The code goes on to define two more processors in a similar manner.
3. Finally we add sinks (output topics) to complete our messaging pipeline. The `addSink` method takes a String name, the name of a topic, a serializer for the key, a serializer for the value and one to N number of parent nodes. In the 3 `addSink` methods we can see the JsonDeserializer objects that were created earlier in the code.

Import the remaining Class in the project.

- `JsonDeserializer`
- `JsonSerializer`
- `Purchase`
- `PurchasePattern`
- `RewardAccumulator`

// Code Begin

```
package com.tos.processor.purchases;
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import com.google.gson.JsonSyntaxException;

import org.apache.kafka.common.serialization.Deserializer;

import java.util.Map;
public class JsonDeserializer<T> implements Deserializer<T> {

    private Gson gson = new GsonBuilder().setDateFormat("yyyy-MM-
dd'T'HH:mm:ss").create();;
    //new Gson();

    private Class<T> deserializedClass;

    public JsonDeserializer(Class<T> deserializedClass) {
        this.deserializedClass = deserializedClass;
    }

    public JsonDeserializer() {

    }

    @Override
    @SuppressWarnings("unchecked")
    public void configure(Map<String, ?> map, boolean b) {
        if(deserializedClass == null) {
            deserializedClass = (Class<T>) map.get("serializedClass");
        }
    }
```

```
    @Override
    public T deserialize(String s, byte[] bytes) {
      T obj = null;
         try {
               if(bytes == null){
                    return null;
                }

                obj = gson.fromJson(new String(bytes),deserializedClass);
          } catch (JsonSyntaxException e) {
               // TODO Auto-generated catch block
               System.out.println(">>>>>" + new String(bytes) + " -- " + s);
               e.printStackTrace();
          }
         return obj;
    }

    @Override
    public void close() {

    }
}
```

```
package com.tos.processor.purchases;
import com.google.gson.Gson;
import org.apache.kafka.common.serialization.Serializer;

import java.nio.charset.Charset;
import java.util.Map;
public class JsonSerializer<T> implements Serializer<T> {
```

```
    private Gson gson = new Gson();

    @Override
    public void configure(Map<String, ?> map, boolean b) {

    }

    @Override
    public byte[] serialize(String topic, T t) {
        return gson.toJson(t).getBytes(Charset.forName("UTF-8"));
    }

    @Override
    public void close() {

    }
}
```

```
package com.tos.processor.purchases;
import java.util.Date;
import java.util.Objects;
public class Purchase {

     private String firstName;
    private String lastName;
    private String creditCardNumber;
    private String itemPurchased;
    int quantity;
    double price;
    private Date purchaseDate;
```

```
  private String zipCode;

private Purchase(Builder builder) {
    firstName = builder.firstName;
    lastName = builder.lastName;
    creditCardNumber = builder.creditCardNumber;
    itemPurchased = builder.itemPurchased;
    quantity = builder.quanity;
    price = builder.price;
    purchaseDate = builder.purchaseDate;
    zipCode = builder.zipCode;
}

public static Builder builder() {
    return new Builder();
}

public static Builder builder(Purchase copy) {
    Builder builder = new Builder();
    builder.firstName = copy.firstName;
    builder.lastName = copy.lastName;
    builder.creditCardNumber = copy.creditCardNumber;
    builder.itemPurchased = copy.itemPurchased;
    builder.quanity = copy.quantity;
    builder.price = copy.price;
    builder.purchaseDate = copy.purchaseDate;
    builder.zipCode = copy.zipCode;
    return builder;
}


public String getFirstName() {
    return firstName;
```

```
    }

    public String getLastName() {
        return lastName;
    }

    public String getCreditCardNumber() {
        return creditCardNumber;
    }

    public String getItemPurchased() {
        return itemPurchased;
    }

    public int getQuantity() {
        return quantity;
    }

    public double getPrice() {
        return price;
    }

    public Date getPurchaseDate() {
        return purchaseDate;
    }

    public String getZipCode() {
        return zipCode;
    }

    @Override
    public String toString() {
        return "Purchase{" +
```

```
                "firstName='" + firstName + '\'' +
                ", lastName='" + lastName + '\'' +
                ", creditCardNumber='" + creditCardNumber + '\'' +
                ", itemPurchased='" + itemPurchased + '\'' +
                ", quantity=" + quantity +
                ", price=" + price +
                ", purchaseDate=" + purchaseDate +
                ", zipCode='" + zipCode + '\'' +
                '}';
    }

    public static final class Builder {
        private String firstName;
        private String lastName;
        private String creditCardNumber;
        private String itemPurchased;
        private int quanity;
        private double price;
        private Date purchaseDate;
        private String zipCode;

        private static final String CC_NUMBER_REPLACEMENT="xxxx-xxxx-xxxx-";

        private Builder() {
        }

        public Builder firstName(String val) {
            firstName = val;
            return this;
        }

        public Builder lastName(String val) {
            lastName = val;
```

```
        return this;
    }


    public Builder maskCreditCard(){
        Objects.requireNonNull(this.creditCardNumber, "Credit Card can't be null");
        String last4Digits = this.creditCardNumber.split("-")[3];
        this.creditCardNumber = CC_NUMBER_REPLACEMENT+last4Digits;
        return this;
    }

    public Builder creditCardNumber(String val) {
        creditCardNumber = val;
        return this;
    }

    public Builder itemPurchased(String val) {
        itemPurchased = val;
        return this;
    }

    public Builder quanity(int val) {
        quanity = val;
        return this;
    }

    public Builder price(double val) {
        price = val;
        return this;
    }

    public Builder purchaseDate(Date val) {
        purchaseDate = val;
```

```
            return this;
        }

        public Builder zipCode(String val) {
            zipCode = val;
            return this;
        }

        public Purchase build() {
            return new Purchase(this);
        }
    }
}
```

```
package com.tos.processor.purchases;

import java.util.Date;

public class PurchasePattern {
     private String zipCode;
    private String item;
    private Date date;

    private PurchasePattern(Builder builder) {
        zipCode = builder.zipCode;
        item = builder.item;
        date = builder.date;
    }

    public static Builder newBuilder() {
        return new Builder();
    }
```

```
    public static Builder builder(Purchase purchase){
        return new Builder(purchase);

    }
    public String getZipCode() {
        return zipCode;
    }

    public String getItem() {
        return item;
    }

    public Date getDate() {
        return date;
    }


    @Override
    public String toString() {
        return "PurchasePattern{" +
                "zipCode='" + zipCode + '\'' +
                ", item='" + item + '\'' +
                ", date=" + date +
                '}';
    }

    public static final class Builder {
        private String zipCode;
        private String item;
        private Date date;

        private  Builder() {
```

```
        }

        private Builder(Purchase purchase) {
            this.zipCode = purchase.getZipCode();
            this.item = purchase.getItemPurchased();
            this.date = purchase.getPurchaseDate();
        }

        public Builder zipCode(String val) {
            zipCode = val;
            return this;
        }

        public Builder item(String val) {
            item = val;
            return this;
        }

        public Builder date(Date val) {
            date = val;
            return this;
        }

        public PurchasePattern build() {
            return new PurchasePattern(this);
        }
    }

}
```

```
package com.tos.processor.purchases;
```

```
public class RewardAccumulator {

      private String customerName;
         private double purchaseTotal;

         private RewardAccumulator(String customerName, double purchaseTotal) {
             this.customerName = customerName;
             this.purchaseTotal = purchaseTotal;
         }

         public String getCustomerName() {
             return customerName;
         }

         public double getPurchaseTotal() {
             return purchaseTotal;
         }

         @Override
         public String toString() {
             return "RewardAccumulator{" +
                     "customerName='" + customerName + '\'' +
                     ", purchaseTotal=" + purchaseTotal +
                     '}';
         }

         public static Builder builder(Purchase purchase){return new Builder(purchase);}

         public static final class Builder {
             private String customerName;
             private double purchaseTotal;

             private Builder(Purchase purchase){
```

```
                this.customerName = purchase.getLastName()+","+purchase.getFirstName();
                this.purchaseTotal = purchase.getPrice() * purchase.getQuantity();
            }


            public RewardAccumulator build(){
                return new RewardAccumulator(customerName,purchaseTotal);
            }

        }
}
```

// Code Ends.

Execution steps:
Install the Json-Data-Generator:
https://github.com/everwatchsolutions/json-data-generator/releases

Then copy the json config files to json generator conf directory
stock-transactions-config.json

```
{
  "workflows": [
    {
      "workflowName": "purchases",
      "workflowFilename": "purchases.json"
    }
  ],
  "producers": [
    {
      "type": "kafka",
      "broker.server": "127.0.0.1",
      "broker.port": 9092,
      "topic": "src-topic",
      "flatten" : false,
      "sync": false
    },
    {
      "type": "logger"
    }
  ]
}
```

purchases-config.json

```
{
          "workflows": [
            {
              "workflowName": "purchases",
              "workflowFilename": "purchases.json"
            }
          ],
          "producers": [
            {
              "type": "kafka",
              "broker.server": "127.0.0.1",
              "broker.port": 9092,
              "topic": "src-topic",
              "flatten" : false,
              "sync": false
            },
            {
              "type": "logger"
            }

          ]
        }
```

purchases.json

```
{
  "eventFrequency": 400,
  "varyEventFrequency": true,
  "repeatWorkflow": true,
  "timeBetweenRepeat": 1500,
  "varyRepeatFrequency": true,
  "steps": [
    {
      "config": [
        {
          "lastName" : "lastName()",
          "firstName" : "firstName()",
          "creditCardNumber": "random('4929-3813-3266-4295', '5370-4638-8881-3020','4916-4811-5814-
8111','4916-4034-9269-8783','5299-1561-5689-1938','5293-8502-0071-3058') ",
          "itemPurchased": "random('batteries','eggs','diapers','shampoo','shaving
cream','doughnuts','beer')",
          "quantity": "integer(1,4)",
          "price": "double(3.95,14.99)",
          "purchaseDate": "date(\"2016/02/12T00:00:00\",\"2016/02/19T23:59:59\")" ,
          "zipCode" : "random('20841','20852','19971','10005','21842')"
        }
      ],
      "duration": 0
    }
  ]
}
```

stock-transactions-config.json

```
{
 "workflows": [
  {
   "workflowName": "stock-transactions",
   "workflowFilename": "stock-transactions.json"
  }
 ],
 "producers": [
  {
   "type": "kafka",
   "broker.server": "127.0.0.1",
   "broker.port": 9092,
   "topic": "stocks",
   "flatten" :false,
   "sync": false
  },
  {
   "type": "logger"
  }
 ]
}
```

stock-transactions.json

```
{
  "eventFrequency": 400,
  "varyEventFrequency": true,
  "repeatWorkflow": true,
  "timeBetweenRepeat": 1500,
  "varyRepeatFrequency": true,
  "steps": [
    {
      "config": [
        {
          "timestamp": "now()",
          "symbol": "random('GOOG','MSFT','AAPL','YHOO','TWTR')",
          "amount": "double(100.0,60000.0)",
          "shares": "integer(100,1000)",
          "type": "random('purchase','sell')"
        }
      ],
      "duration": 0
    }
  ]
}
```

```
/apps/json-gen/conf
(base) [root@tos conf]# cp /mnt/hgfs/Software/data/*json .
(base) [root@tos conf]# ls -lt
total 304
-rwxr-xr-x. 1 root root    470 Aug  7 11:28 stock-transactions.json
-rwxr-xr-x. 1 root root    358 Aug  7 11:28 stock-transactions-config.json
-rwxr-xr-x. 1 root root    838 Aug  7 11:28 purchases.json
-rwxr-xr-x. 1 root root    344 Aug  7 11:28 purchases-config.json
-rwxr-xr-x. 1 root root 191778 Aug  7 11:28 complaints.json
-rw-r--r--. 1 root root    182 Apr 23  2018 jsonWebLogConfig.json
-rw-r--r--. 1 root root  67078 Apr 23  2018 jsonWebLogWorkflow.json
-rw-r--r--. 1 root root    306 Sep 19  2017 iothubExampleSimConfig.json
-rw-r--r--. 1 root root   2297 Jun  2  2015 kitchesSinkExample.json
-rw-r--r--. 1 root root    653 May 21  2015 jackieChanWorkflow.json
-rw-r--r--. 1 root root    347 May 21  2015 jackieChanSimConfig.json
-rw-r--r--. 1 root root   1891 May  6  2015 kitchenSinkOutput.json
-rw-r--r--. 1 root root   2200 Apr 29  2015 exampleWorkflow.json
-rw-r--r--. 1 root root    178 Apr 29  2015 exampleSimConfig.json
(base) [root@tos conf]# pwd
/apps/json-gen/conf
(base) [root@tos conf]#
```

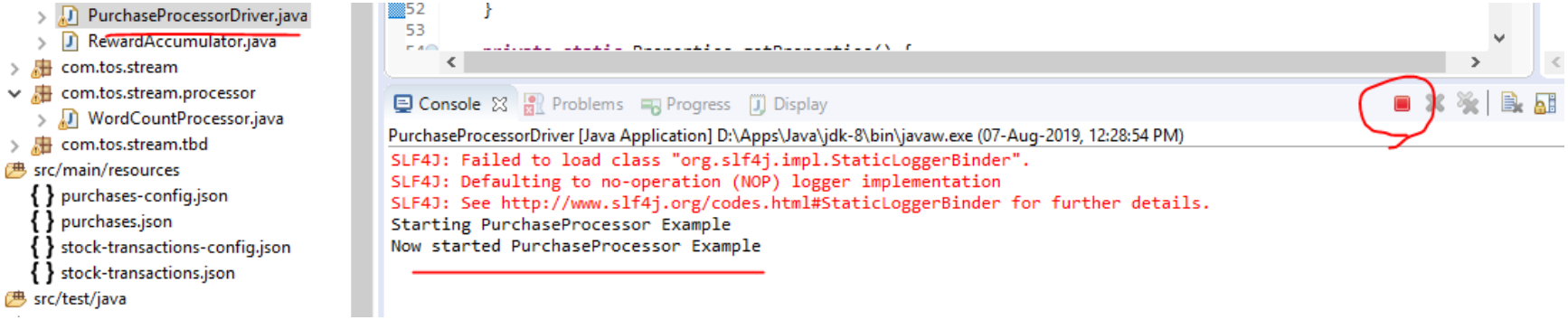
Create all the topics required by the examples using CLI:
src-topic , patterns , rewards , purchases

Running the Purchase Processor API KStreams API Examples

```
cd <dir>/json-gen/
java -jar json-data-generator-1.4.1.jar purchases-config.json
```

```
016-02-15T02:22:03.000Z","zipCode":"19971"} ]
2019-08-07 12:27:26,575 INFO data-logger [Thread-1] {"lastName":"Doe","firstName
":"Tara","creditCardNumber":"4916-4811-5814-8111","itemPurchased":"shampoo","qua
ntity":1,"price":8.0803,"purchaseDate":"2016-02-15T02:22:03.000Z","zipCode":"199
71"}
2019-08-07 12:27:27,729 DEBUG n.a.d.j.g.l.KafkaLogger [Thread-1] Sending event t
o Kafka: [ {"lastName":"Loxly","firstName":"Bob","creditCardNumber":"4929-3813-3
266-4295","itemPurchased":"doughnuts","quantity":2,"price":5.4363,"purchaseDate"
:"2016-02-15T23:02:48.000Z","zipCode":"19971"} ]
2019-08-07 12:27:27,732 INFO data-logger [Thread-1] {"lastName":"Loxly","firstNa
me":"Bob","creditCardNumber":"4929-3813-3266-4295","itemPurchased":"doughnuts","
quantity":2,"price":5.4363,"purchaseDate":"2016-02-15T23:02:48.000Z","zipCode":"
19971"}
2019-08-07 12:27:28,018 TRACE n.a.d.j.g.EventGenerator [Thread-1] Generator( pur
chases ) generated 2.0 events/sec
2019-08-07 12:27:29,374 DEBUG n.a.d.j.g.l.KafkaLogger [Thread-1] Sending event t
o Kafka: [ {"lastName":"Baggins","firstName":"Tara","creditCardNumber":"5299-156
1-5689-1938","itemPurchased":"diapers","quantity":1,"price":12.2558,"purchaseDat
e":"2016-02-17T07:38:47.000Z","zipCode":"20841"} ]
2019-08-07 12:27:29,374 INFO data-logger [Thread-1] {"lastName":"Baggins","first
Name":"Tara","creditCardNumber":"5299-1561-5689-1938","itemPurchased":"diapers",
"quantity":1,"price":12.2558,"purchaseDate":"2016-02-17T07:38:47.000Z","zipCode"
:"20841"}
```

Execute the following grey Class file.



Verify the message that are in the sink processor i.e patterns , rewards , purchases. You can verify with the CLI console.

```
#cd /opt/kafka/bin
./kafka-console-consumer.sh --topic patterns --bootstrap-server localhost:9092
./kafka-console-consumer.sh --topic rewards --bootstrap-server localhost:9092
./kafka-console-consumer.sh --topic purchases --bootstrap-server localhost:9092
```

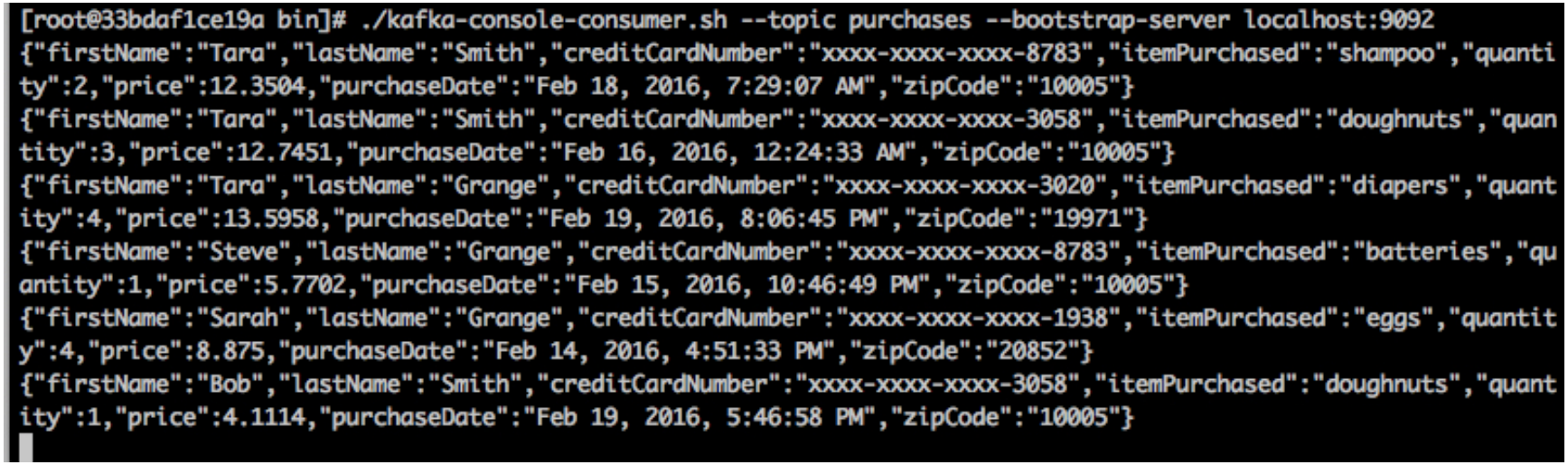
Topic - patterns :

```
[root@33bdaf1ce19a bin]# ./kafka-console-consumer.sh --topic patterns --bootstrap-server localhost:9092
{"zipCode":"20852","item":"beer","date":"Feb 17, 2016, 6:58:39 PM"}
{"zipCode":"10005","item":"shaving cream","date":"Feb 18, 2016, 3:17:44 PM"}
{"zipCode":"20852","item":"eggs","date":"Feb 19, 2016, 2:50:24 AM"}
{"zipCode":"20841","item":"shaving cream","date":"Feb 12, 2016, 11:46:56 PM"}
{"zipCode":"19971","item":"eggs","date":"Feb 17, 2016, 4:58:55 PM"}
{"zipCode":"20841","item":"batteries","date":"Feb 13, 2016, 2:35:46 PM"}
```

Observe that Message is displayed in the format of Pattern class as we have transformed the message to Pattern and sink to the patterns topics. Transformation is done using Processor API.
Likewise you can verify for rewards and purchases object.
>print ‘rewards’;

```
^CProcessed a total of 22 messages
[root@33bdaf1ce19a bin]# ./kafka-console-consumer.sh --topic rewards --bootstrap-server localhost:9092
{"customerName":"Loxly,Bob","purchaseTotal":43.596000000000004}
{"customerName":"Black,Steve","purchaseTotal":23.1122}
{"customerName":"Baggins,Eric","purchaseTotal":33.8424}
{"customerName":"Doe,Steve","purchaseTotal":44.8756}
{"customerName":"Smith,Eric","purchaseTotal":13.3642}
{"customerName":"Grange,Eric","purchaseTotal":23.6484}
{"customerName":"Smith,Andrew","purchaseTotal":41.358900000000006}
{"customerName":"Black,Steve","purchaseTotal":54.2284}
```

>print ‘purchases’;



https://github.com/bbejeck/kafka-streams

---------- Lab Ends Here ----------

18. Stream API - Stateful Processor

This lab will demonstrate the basic use of state in the processor API, it will be based in stock trading. In this example the processor will capture results for each trade and store aggregate information by ticker symbol. The aggregate information is then published periodically.

// Code Start Here – Processor

```
package com.tos.processor.stocks;

import java.time.Duration;
import java.util.Objects;

import org.apache.kafka.streams.processor.AbstractProcessor;
import org.apache.kafka.streams.processor.ProcessorContext;
import org.apache.kafka.streams.processor.PunctuationType;
import org.apache.kafka.streams.state.KeyValueIterator;
import org.apache.kafka.streams.state.KeyValueStore;



@SuppressWarnings("unchecked")
public class StockSummaryProcessor extends AbstractProcessor<String,
StockTransaction> {

    private KeyValueStore<String, StockTransactionSummary> summaryStore;
    private ProcessorContext context;


    public void process(String key, StockTransaction stockTransaction) {
```

```
        String currentSymbol = stockTransaction.getSymbol();
        StockTransactionSummary transactionSummary =
summaryStore.get(currentSymbol);
        if (transactionSummary == null) {
            transactionSummary =
StockTransactionSummary.fromTransaction(stockTransaction);
        } else {
            transactionSummary.update(stockTransaction);
        }
        summaryStore.put(currentSymbol, transactionSummary);

        this.context.commit();
    }


    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        this.context = context;
      //  this.context.schedule(10000);
        summaryStore = (KeyValueStore<String, StockTransactionSummary>)
this.context.getStateStore("stock-transactions");
        Objects.requireNonNull(summaryStore, "State store can't be null");

        // schedule a punctuate() method every second based on event-time
        this.context.schedule(Duration.ofSeconds(10), PunctuationType.STREAM_TIME,
(timestamp) -> {
```

```
            KeyValueIterator<String, StockTransactionSummary> iter =
this.summaryStore.all();
            long currentTime = System.currentTimeMillis();
            while (iter.hasNext()) {
              StockTransactionSummary summary = iter.next().value;
                if (summary.updatedWithinLastMillis(currentTime, 11000)) {
                    this.context.forward(summary.tickerSymbol, summary);
                }
            }
            iter.close();

            // commit the current processing progress
            context.commit();
        });

    }
```

// Code Ends Here – Processor

In this example we see the `process` method along with two overridden methods: `init` and `punctuate`. The `process` method extracts the stock symbol, updates/creates the trade information then places the summary results in the store.

In the `init` method we are:

1. Setting the ProcessorContext reference.
2. Calling the `ProcessorContext.schedule` method which controls how frquently the `punctuate` method is executed. In this case it's every 10 seconds.
3. Getting a reference to the state store created when constructing the `Topology`.

The `punctuate` method iterates over all the values in the store and if they have been updated with the last 11 seconds, the StockTransactionSummary object is sent to consumers.

Constructing a Topology with a State Store

Here's the section from the source code that creates our `Topology` including a `KeyValueStore`:

```
// Code Begin Here --  Driver Topology
package com.tos.processor.stocks;

import java.util.Properties;

import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.processor.WallclockTimestampExtractor;
import org.apache.kafka.streams.state.KeyValueStore;
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;

import com.tos.processor.purchases.JsonDeserializer;
import com.tos.processor.purchases.JsonSerializer;

public class StockSummaryStatefulProcessorDriver {

     public static void main(String[] args) {

          org.apache.kafka.streams.Topology builder = new
org.apache.kafka.streams.Topology();
```

```
        JsonSerializer<StockTransactionSummary> stockTxnSummarySerializer = new
JsonSerializer<>();
        JsonDeserializer<StockTransactionSummary> stockTxnSummaryDeserializer =
new JsonDeserializer<>(
                StockTransactionSummary.class);
        JsonDeserializer<StockTransaction> stockTxnDeserializer = new
JsonDeserializer<>(StockTransaction.class);
        JsonSerializer<StockTransaction> stockTxnJsonSerializer = new
JsonSerializer<>();
        StringSerializer stringSerializer = new StringSerializer();
        StringDeserializer stringDeserializer = new StringDeserializer();

        Serde<StockTransactionSummary> stockTransactionSummarySerde =
Serdes.serdeFrom(stockTxnSummarySerializer,
                stockTxnSummaryDeserializer);
        StoreBuilder<KeyValueStore<String, StockTransactionSummary>>
countStoreSupplier = Stores.keyValueStoreBuilder(
                Stores.inMemoryKeyValueStore("stock-transactions"),
Serdes.String(), stockTransactionSummarySerde);
        // KeyValueStore<String, Long> countStore = countStoreSupplier.build();

        builder.addSource("stocks-source", stringDeserializer,
stockTxnDeserializer, "stocks")
                .addProcessor("summary", StockSummaryProcessor::new, "stocks-
source")
                .addStateStore(countStoreSupplier, "summary")
```

```
                    .addSink("sink", "stocks-out", stringSerializer,
stockTxnJsonSerializer, "stocks-source")
                    .addSink("sink-2", "transaction-summary", stringSerializer,
stockTxnSummarySerializer, "summary");

          System.out.println("Starting StockSummaryStatefulProcessor Example");
          KafkaStreams streaming = new KafkaStreams(builder, getProperties());
           streaming.setUncaughtExceptionHandler((Thread thread, Throwable
throwable) -> {
                 // here you should examine the throwable/exception and perform an
appropriate action!
               System.out.println(" Error :" + throwable);
               });
          streaming.start();
          System.out.println("StockSummaryStatefulProcessor Example now started");
          // Stop the Kafka Streams threads
          // Add shutdown hook to stop the Kafka Streams threads.
          // You can optionally provide a timeout to `close`.
          Runtime.getRuntime().addShutdownHook(new Thread(streaming::close));
     }

     private static Properties getProperties() {
          Properties props = new Properties();
          props.put(StreamsConfig.CLIENT_ID_CONFIG, "Sample-Stateful-Processor");
          props.put("group.id", "test-consumer-group");
          props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stateful_processor_id");
          props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.139.132:9092");
          props.put(StreamsConfig.REPLICATION_FACTOR_CONFIG, 1);
```

```
            props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
WallclockTimestampExtractor.class);
            return props;
        }
}
// Code Ends Here --  Driver Topology
```

For the most part, this is very similar code in terms of creating serializers, deserializers and the topology builder. But there is one difference - creating an in-memory state store (named "`stock-transactions`") to be used by the processor. The name passed to the `Stores.create` method is the same we used in the processor `init` method to retieve the store. When specifying the keys we can use the convenience method `Stores.withStringKeys()` that requires no argments since Strings are a supported type. But since we are using a typed value, the `withValues` method is used and provides serializer and deserializer instances.

StockTransaction – Model Class

```
// Code Begin Here --  StockTransaction
package com.tos.processor.stocks;

import java.util.Date;


public class StockTransaction {

    private String symbol;
    private String type;
    private double shares;
    private double amount;
    private Date timeStamp;
```

```
public String getSymbol() {
    return symbol;
}

public void setSymbol(String symbol) {
    this.symbol = symbol;
}

public String getType() {
    return type;
}

public void setType(String type) {
    this.type = type;
}

public double getShares() {
    return shares;
}

public void setShares(double shares) {
    this.shares = shares;
}

public double getAmount() {
    return amount;
```

```
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    public Date getTimeStamp() {
        return timeStamp;
    }

    public void setTimeStamp(Date timeStamp) {
        this.timeStamp = timeStamp;
    }

    @Override
    public String toString() {
        return "StockTransaction{" +
                "symbol='" + symbol + '\'' +
                ", type='" + type + '\'' +
                ", shares=" + shares +
                ", amount=" + amount +
                ", timeStamp=" + timeStamp +
                '}';
    }
}
```

// Code Ends Here -- StockTransaction

StockTransactionSummary Class

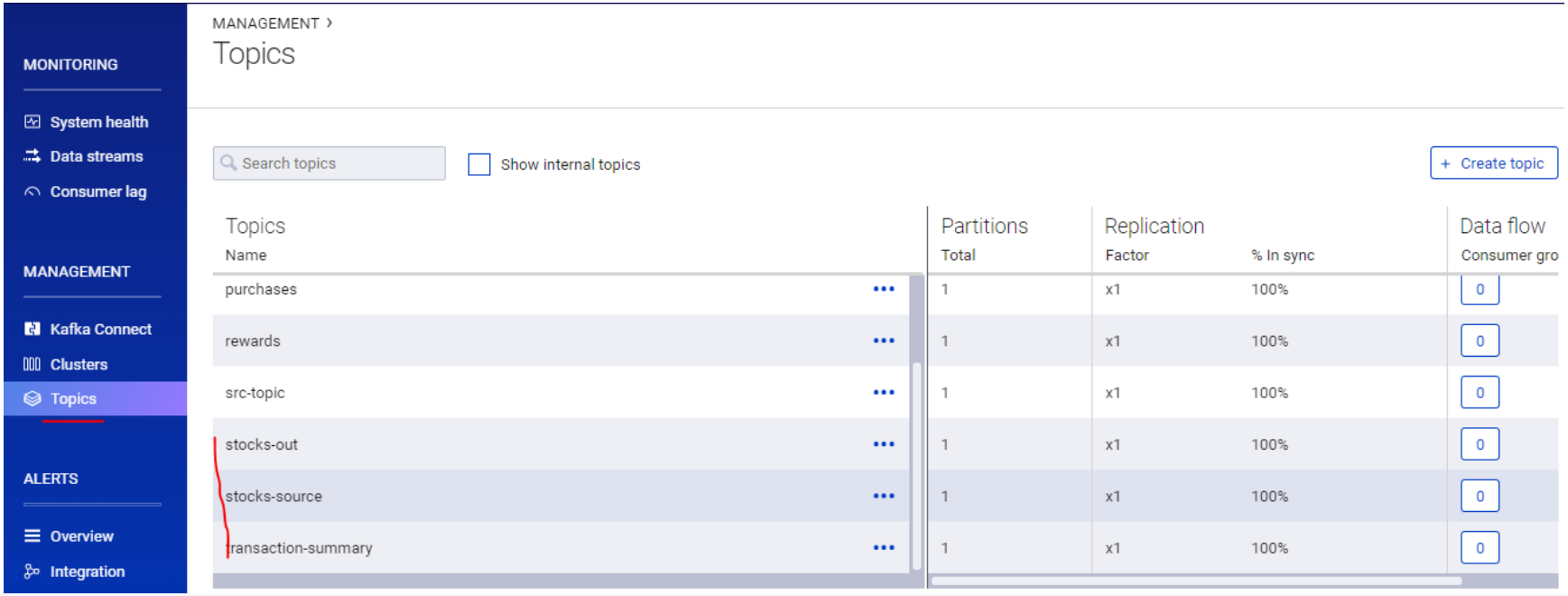
// Code Begin Here -- StockTransactionSummary

```
package com.tos.processor.stocks;

public class StockTransactionSummary {

    public double amount;
    public String tickerSymbol;
    public int sharesPurchased;
    public int sharesSold;
    private long lastUpdatedTime;



    public void update(StockTransaction transaction){
          this.amount += transaction.getAmount();
          if(transaction.getType().equalsIgnoreCase("purchase")){
              this.sharesPurchased += transaction.getShares();
          } else{
              this.sharesSold += transaction.getShares();
          }
        this.lastUpdatedTime = System.currentTimeMillis();
    }

    public boolean updatedWithinLastMillis(long currentTime, long limit){
         return currentTime - this.lastUpdatedTime <= limit;
    }
```

```
    public static StockTransactionSummary fromTransaction(StockTransaction
transaction){
                StockTransactionSummary summary = new StockTransactionSummary();
                summary.tickerSymbol = transaction.getSymbol();
                summary.update(transaction);
                return summary;
    }
}
// Code Ends Here --  StockTransactionSummary
```

Execution Steps:
Create the required Topics using the Control Center.

- stocks-source
- stocks-out
- transaction-summary




```
cd /apps/json-gen/
 java -jar json-data-generator-1.4.0.jar stock-transactions-config.json
```

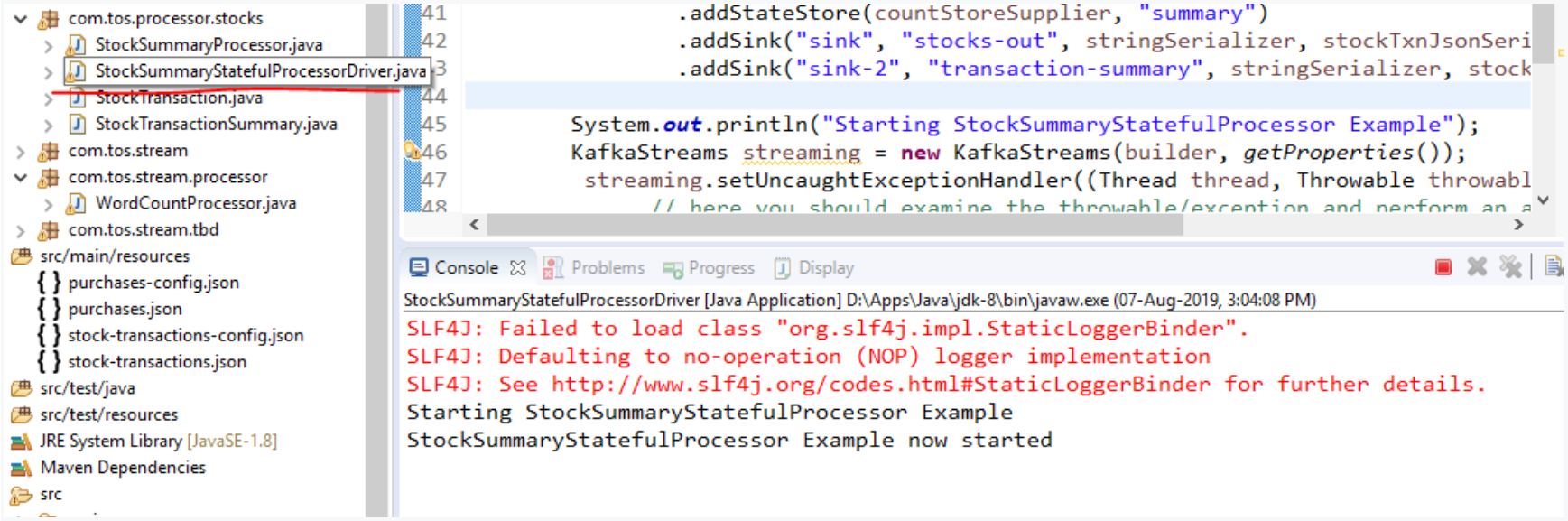
```
ed TypeHandler [ date,net.acesinc.data.json.generator.types.DateType ]
2019-08-07 14:59:34,668 DEBUG n.a.d.j.g.t.TypeHandlerFactory [Thread-1] Discover
ed TypeHandler [ randomIncrementLong,net.acesinc.data.json.generator.types.Rando
mIncrementLongType ]
2019-08-07 14:59:34,668 DEBUG n.a.d.j.g.t.TypeHandlerFactory [Thread-1] Discover
ed TypeHandler [ alphaNumeric,net.acesinc.data.json.generator.types.AlphaNumeric
Type ]
2019-08-07 14:59:34,669 DEBUG n.a.d.j.g.t.TypeHandlerFactory [Thread-1] Discover
ed TypeHandler [ counter,net.acesinc.data.json.generator.types.CounterType ]
2019-08-07 14:59:34,669 DEBUG n.a.d.j.g.t.TypeHandlerFactory [Thread-1] Discover
ed TypeHandler [ stringMerge,net.acesinc.data.json.generator.types.StringMergeTy
pe ]
2019-08-07 14:59:34,669 DEBUG n.a.d.j.g.t.TypeHandlerFactory [Thread-1] Discover
ed TypeHandler [ alpha,net.acesinc.data.json.generator.types.AlphaType ]
2019-08-07 14:59:34,669 DEBUG n.a.d.j.g.t.TypeHandlerFactory [Thread-1] Discover
ed TypeHandler [ firstName,net.acesinc.data.json.generator.types.FirstName ]
2019-08-07 14:59:34,670 DEBUG n.a.d.j.g.t.TypeHandlerFactory [Thread-1] Discover
ed TypeHandler [ timestamp,net.acesinc.data.json.generator.types.TimestampType ]
2019-08-07 14:59:34,691 DEBUG n.a.d.j.g.l.KafkaLogger [Thread-1] Sending event t
o Kafka: [ {"timestamp":"2019-08-07T14:59:34.672Z","symbol":"TWTR","amount":2021
6.6381,"shares":313,"type":"purchase"} ]
2019-08-07 14:59:35,130 INFO data-logger [Thread-1] {"timestamp":"2019-08-07T14:
59:34.672Z","symbol":"TWTR","amount":20216.6381,"shares":313,"type":"purchase"}
```

Verify that whether STOCK is being pushed to stocks topic from KSQL CLI

>print 'stocks' limit 3;

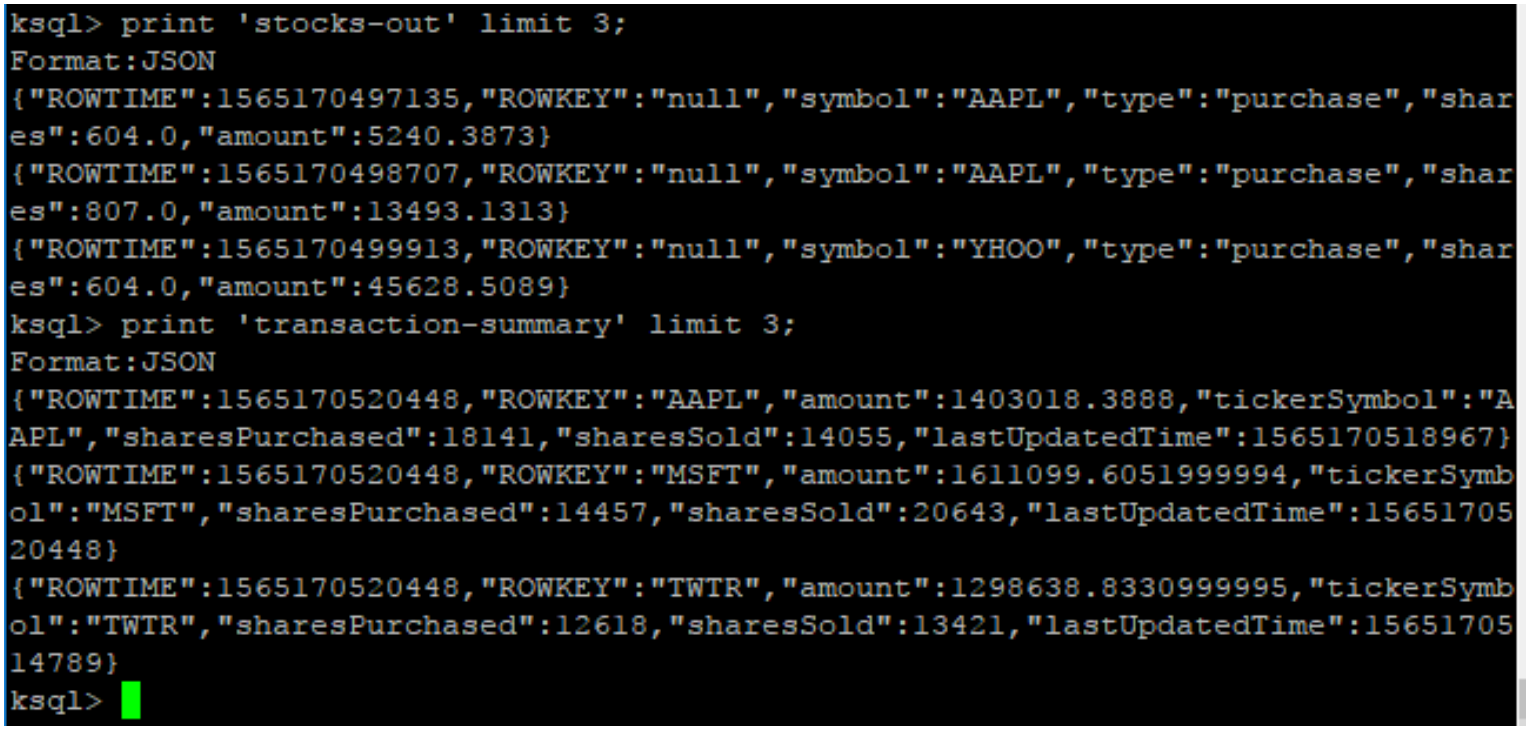
```
ksql> print 'stocks' limit 3;
Format:JSON
{"ROWTIME":1565170307820,"ROWKEY":"null","timestamp":"2019-08-07T15:01:47.819Z",
"symbol":"AAPL","amount":14651.8356,"shares":717,"type":"sell"}
{"ROWTIME":1565170309443,"ROWKEY":"null","timestamp":"2019-08-07T15:01:49.442Z",
"symbol":"MSFT","amount":45917.6004,"shares":741,"type":"sell"}
{"ROWTIME":1565170310905,"ROWKEY":"null","timestamp":"2019-08-07T15:01:50.903Z",
"symbol":"MSFT","amount":25746.4825,"shares":883,"type":"sell"}
ksql>
```

Execute the Stock Driver Program.



Verify the topics and determine that aggregated stocks are being displayed in it as shown below:

```
>print 'stocks-out' limit 3;
> print 'transaction-summary' limit 3;
```



---------- Lab Ends Here ----------

19. Stream API – Testing - TBD (Output Verifier)

To test a Kafka Streams application, Kafka provides a test-utils artifact that can be added as regular dependency to your test code base. Example pom.xml snippet when using Maven:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams-test-utils</artifactId>
  <version>2.3.0</version>
  <scope>test</scope>
</dependency>
```

Prerequisite : You need to complete the Stream API –Stateless Processor API before executing the test case.

Streams provides the `TopologyTestDriver` in the `kafka-streams-test-utils` package as a drop-in replacement for the `KafkaStreams` class. Use the `ConsumerRecordFactory` to generate records by providing regular Java types for key and values and the corresponding serializers. `ProducerRecord` contains all the record metadata in addition to the key and value, which can make it awkward to use with test frameworks' equality checks.

Test Class is as shown below:
// Test Class – Code Begins Here

```
package com.tos.processor.purchases.test;

import static org.junit.Assert.assertEquals;

import java.util.Calendar;
```

```
import java.util.Properties;

import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.apache.kafka.common.serialization.StringSerializer;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.TopologyTestDriver;
import org.apache.kafka.streams.state.KeyValueStore;
import org.apache.kafka.streams.test.ConsumerRecordFactory;
import org.apache.kafka.streams.test.OutputVerifier;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.tos.processor.purchases.CreditCardAnonymizer;
import com.tos.processor.purchases.CustomerRewards;
import com.tos.processor.purchases.JsonDeserializer;
import com.tos.processor.purchases.JsonSerializer;
import com.tos.processor.purchases.Purchase;
import com.tos.processor.purchases.RewardAccumulator;


public class PurchaseTest {
    private static final String inputTopic = "src-topic";
    private static final String outputTopic = "rewards";
```

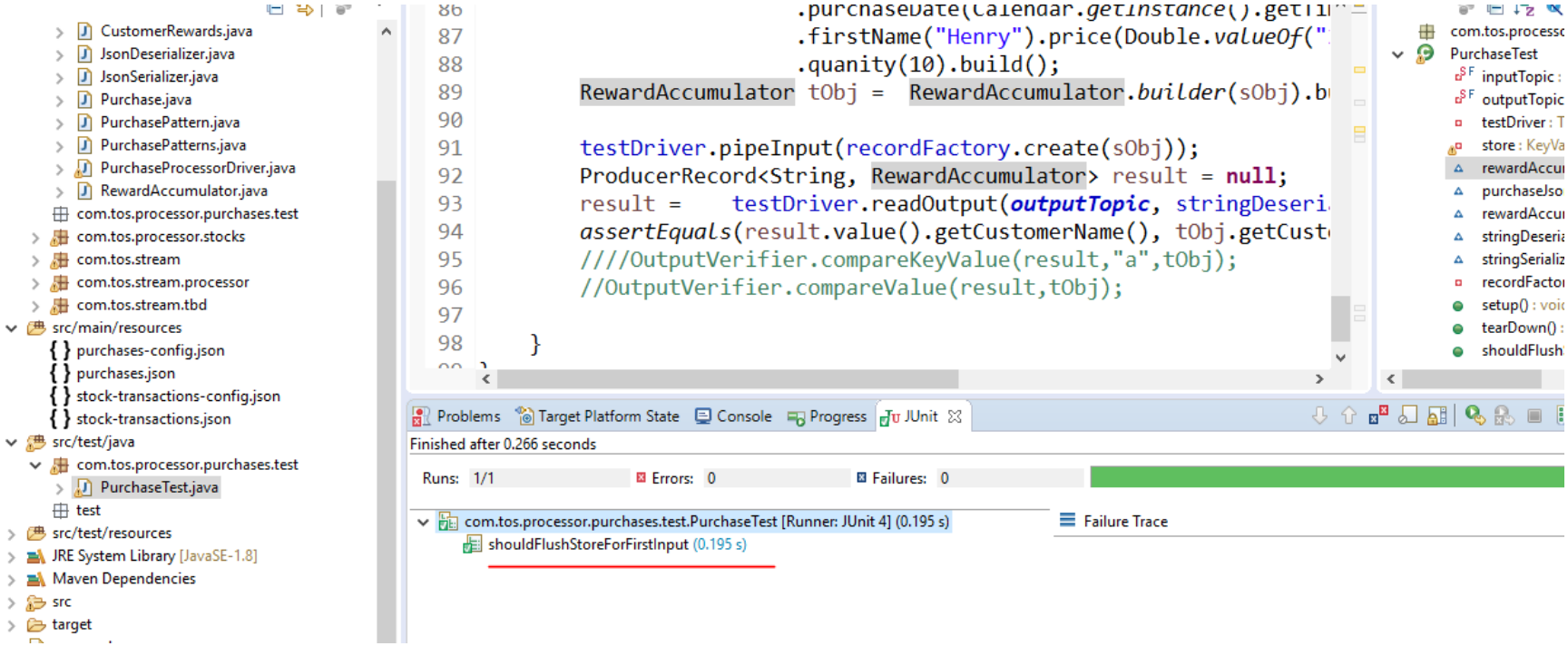
```
    private TopologyTestDriver testDriver;
    private KeyValueStore<String, Long> store;
    JsonSerializer<RewardAccumulator> rewardAccumulatorJsonSerializer = new
JsonSerializer<>();
    JsonDeserializer<Purchase> purchaseJsonDeserializer = new
JsonDeserializer<>(Purchase.class);
    JsonDeserializer<RewardAccumulator> rewardAccumulatorJsonDeserializer = new
JsonDeserializer<>(RewardAccumulator.class);
    StringDeserializer stringDeserializer = new StringDeserializer();
   StringSerializer stringSerializer = new StringSerializer();

    private ConsumerRecordFactory<String, Purchase> recordFactory =
             new ConsumerRecordFactory<>(inputTopic,new StringSerializer(),
                                                                  new
JsonSerializer<Purchase>());


    @Before
    public void setup() {
        StringDeserializer stringDeserializer = new StringDeserializer();
        StringSerializer stringSerializer = new StringSerializer();
        System.out.println(" >>>>> " + "Test");
        Topology topology = new Topology();
        topology.addSource("SOURCE", stringDeserializer, purchaseJsonDeserializer
,inputTopic);
        topology.addProcessor("PROCESS", CreditCardAnonymizer::new, "SOURCE");
        topology.addProcessor("PROCESS3", CustomerRewards::new, "PROCESS");
        /*topology.addStateStore(
```

```
                    Stores.keyValueStoreBuilder(
                            Stores.inMemoryKeyValueStore("aggStore"),
                            Serdes.String(),
                            Serdes.Long()).withLoggingDisabled(), // need to disable
logging to allow store pre-populating
                    "PROCESS3");*/
            topology.addSink("SINK2", outputTopic,stringSerializer,
rewardAccumulatorJsonSerializer, "PROCESS3");

            // setup test driver
            Properties props = new Properties();
            props.setProperty(StreamsConfig.APPLICATION_ID_CONFIG, "maxPurchase");
            props.setProperty(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"192.168.139.1321:9092");
            /*props.setProperty(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
            props.setProperty(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.Long().getClass().getName());*/
            props.setProperty(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
            testDriver = new TopologyTestDriver(topology, props);

            /*// pre-populate store
            store = testDriver.getKeyValueStore("aggStore");
            store.put("a", 21L);*/
        }

        @After
        public void tearDown() {
```

```
            testDriver.close();
        }

        @Test
        public void shouldFlushStoreForFirstInput() {
            Purchase sObj =  Purchase.builder().creditCardNumber("9456-7896-5432-
1091")

.purchaseDate(Calendar.getInstance().getTime()).lastName("P")
                                .firstName("Henry").price(Double.valueOf("20.00"))
                                .quanity(10).build();
            RewardAccumulator tObj =  RewardAccumulator.builder(sObj).build();

            testDriver.pipeInput(recordFactory.create(sObj));
            ProducerRecord<String, RewardAccumulator> result = null;
            result = testDriver.readOutput(outputTopic, stringDeserializer,
rewardAccumulatorJsonDeserializer);
           assertEquals(result.value().getCustomerName(), tObj.getCustomerName());
            ////OutputVerifier.compareKeyValue(result,"a",tObj);
            //OutputVerifier.compareValue(result,tObj);

        }
}
```

// Test Class – Code Ends Here. Try changing the result output and verify it.

Execute the test Unit from the IDE.


CustomerRewards.java
JsonDeserializer.java
JsonSerializer.java
Purchase.java
PurchasePattern.java
PurchasePatterns.java
PurchaseProcessorDriver.java
RewardAccumulator.java
com.tos.processor.purchases.test
com.tos.processor.stocks
com.tos.stream
com.tos.stream.processor
com.tos.stream.tbd
src/main/resources
purchases-config.json
purchases.json
stock-transactions-config.json
stock-transactions.json
src/test/java
com.tos.processor.purchases.test
PurchaseTest.java
test
src/test/resources
JRE System Library [JavaSE-1.8]
Maven Dependencies
src
target
.firstName("Henry").price(Double.valueOf("
.quanity(10).build();
RewardAccumulator tObj = RewardAccumulator.builder(sObj).b
testDriver.pipeInput(recordFactory.create(sObj));
ProducerRecord<String, RewardAccumulator> result = null;
result = testDriver.readOutput(outputTopic, stringDeseri
assertEquals(result.value().getCustomerName(), tObj.getCust
////OutputVerifier.compareKeyValue(result,"a",tObj);
//OutputVerifier.compareValue(result,tObj);
}
com.tos.processc
PurchaseTest
inputTopic :
outputTopic
testDriver : T
store : KeyVa
rewardAccu
purchaseJso
rewardAccu
stringDeseria
stringSerializ
recordFactor
setup() : voi
tearDown() :
shouldFlush
Problems
Target Platform State
Console
Progress
JUnit
Finished after 0.266 seconds
Runs: 1/1
Errors: 0
Failures: 0
com.tos.processor.purchases.test.PurchaseTest [Runner: JUnit 4] (0.195 s)
shouldFlushStoreForFirstInput (0.195 s)
Failure Trace

20. Kafka - UDAF

In this lab we will learn how to to write UDAF and consume in KSQL. UDAFs can be used for computing aggregates against multiple rows of data.
It depends on: KSQL - Kafka Aggregation

This UDAF may seem complicated at first, but it's really just performing some basic math and adding the computations to a Map object. Returning a Map is one method for returning multiple values from a KSQL function. Using the example above for your own UDAF, take note of the following methods:

- initialize: used to specify the initial value of your aggregation
- aggregate: performs the actual aggregation by looking at the current row's value (i.e., the currentValue argument), as well as the current aggregation value (i.e., aggregateValue argument), and generates a new aggregate
- merge: describes how to merge two aggregations into one (e.g., when using session windows)

<!---------- POM Begins Here---------->

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
     <modelVersion>4.0.0</modelVersion>
     <groupId>com.tos.kafka</groupId>
     <artifactId>MyProducer</artifactId>
     <version>0.0.1</version>
```

```
<properties>
     <maven.compiler.target>1.8</maven.compiler.target>
     <maven.compiler.source>1.8</maven.compiler.source>
     <avro.version>1.8.2</avro.version>
     <confluent.version>5.3.0</confluent.version>
</properties>
<dependencies>
     <dependency>
          <groupId>org.apache.kafka</groupId>
          <artifactId>kafka-streams</artifactId>
          <version>2.3.0</version>
     </dependency>
     <dependency>
          <groupId>org.apache.kafka</groupId>
          <artifactId>kafka-clients</artifactId>
          <version>2.3.0</version>
     </dependency>
     <!-- Optionally include Kafka Streams DSL for Scala for Scala 2.12 -->
     <dependency>
          <groupId>org.apache.kafka</groupId>
          <artifactId>kafka-streams-scala_2.12</artifactId>
          <version>2.3.0</version>
     </dependency>
     <dependency>
          <groupId>io.advantageous.boon</groupId>
          <artifactId>boon-json</artifactId>
          <version>0.6.6</version>
     </dependency>
```

```
  <!-- KSQL dependency is needed to write your own UDF -->
<dependency>
    <groupId>io.confluent.ksql</groupId>
    <artifactId>ksql-udf</artifactId>
    <version>${confluent.version}</version>
</dependency>

 <dependency>
      <groupId>org.slf4j</groupId>
      <artifactId>slf4j-ext</artifactId>
      <version>1.7.13</version>
 </dependency>
 <!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
 <dependency>
      <groupId>com.google.code.gson</groupId>
      <artifactId>gson</artifactId>
      <version>2.8.5</version>
 </dependency>
 <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-core</artifactId>
      <version>2.7.1</version>
 </dependency>

 <dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
      <version>2.7.1</version>
```

```
</dependency>
<dependency>
     <groupId>org.apache.kafka</groupId>
     <artifactId>kafka-streams-test-utils</artifactId>
     <version>2.3.0</version>
     <scope>test</scope>
</dependency>
<!-- Test dependencies -->
<dependency>
     <groupId>junit</groupId>
     <artifactId>junit</artifactId>
     <version>4.12</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>org.assertj</groupId>
     <artifactId>assertj-core</artifactId>
     <version>3.3.0</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>org.apache.avro</groupId>
     <artifactId>avro</artifactId>
     <version>${avro.version}</version>
</dependency>
<dependency>
     <groupId>org.apache.avro</groupId>
     <artifactId>avro-compiler</artifactId>
```

```
            <version>1.8.2</version>
        </dependency>
        <dependency>
            <groupId>org.apache.avro</groupId>
            <artifactId>avro-maven-plugin</artifactId>
            <version>1.8.2</version>
        </dependency>
        <dependency>
            <groupId>io.confluent</groupId>
            <artifactId>kafka-avro-serializer</artifactId>
            <version>${confluent.version}</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.apache.kafka/kafka-streams-
test-utils -->
        <dependency>
            <groupId>org.apache.kafka</groupId>
            <artifactId>kafka-streams-test-utils</artifactId>
            <version>2.3.0</version>
            <scope>test</scope>
        </dependency>



    </dependencies>

    <!-- <repositories> <repository> <id>confluent</id> <name>Confluent</name>
        <url>http://maven.icm.edu.pl/artifactory/repo/</url> </repository>
</repositories> -->
```

```
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.6.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <mainClass>fully.qualified.MainClass</mainClass>
                        </manifest>
                    </archive>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                </configuration>
                <executions>
                    <execution>
                        <id>make-assembly</id> <!-- this is used for inheritance
merges -->
```

```
                        <phase>package</phase> <!-- bind to the packaging phase
-->
                        <goals>
                            <goal>single</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.avro</groupId>
                <artifactId>avro-maven-plugin</artifactId>
                <version>${avro.version}</version>
                <executions>
                    <execution>
                        <id>schemas</id>
                        <phase>generate-sources</phase>
                        <goals>
                            <goal>schema</goal>
                            <goal>protocol</goal>
                            <goal>idl-protocol</goal>
                        </goals>
                        <configuration>

    <sourceDirectory>${project.basedir}/src/main/resources/avro/</sourceDirectory>

    <outputDirectory>${project.basedir}/src/main/java/</outputDirectory>
                        </configuration>
                    </execution>
```

```
                    </executions>
                </plugin>
            </plugins>
        </build>
</project>
```

<!-------------------------- POM Ends Here -->

<!-------------------------- UDAF Begins Here--->

```
package com.tos.kafka.udf;

import java.util.HashMap;
import java.util.Map;

import io.confluent.ksql.function.udaf.Udaf;
import io.confluent.ksql.function.udaf.UdafDescription;
import io.confluent.ksql.function.udaf.UdafFactory;

/**
 * In this example, we implement a UDAF for computing some summary statistics
 * for a stream of doubles.
 * </pre>
 */
@UdafDescription(name = "summary_stats_big", description = "Example UDAF that
computes some summary stats for a stream of BigInt", version = "1.0", author =
"Henry P")
public final class SummaryStatsUdaf {
```

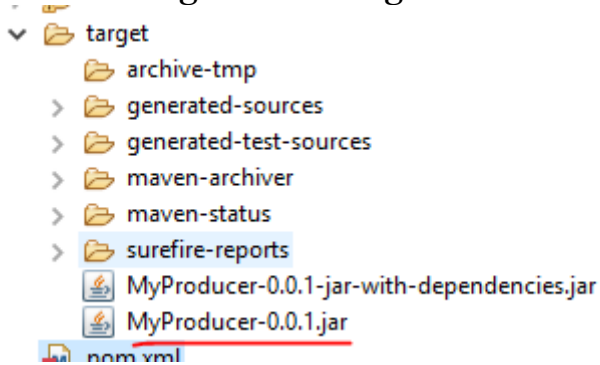
```
    private SummaryStatsUdaf() {
    }

    @UdafFactory(description = "compute summary stats for BigInt")
    // Can be used with stream aggregations. The input of our aggregation will be
    // doubles,
    // and the output will be a map
    public static Udaf<Long, Map<String, Long>> createUdaf() {

        return new Udaf<Long, Map<String, Long>>() {

            /**
             * Specify an initial value for our aggregation
             *
             * @return the initial state of the aggregate.
             */
            @Override
            public Map<String, Long> initialize() {
                final Map<String, Long> stats = new HashMap<>();
                stats.put("mean", Long.valueOf(0));
                stats.put("sample_size", Long.valueOf(0));
                stats.put("sum", Long.valueOf(0));
                return stats;
            }

            /**
             * Perform the aggregation whenever a new record appears in our
stream.
```

```
                 *
                 * @param newValue
                 *            the new value to add to the {@code aggregateValue}.
                 * @param aggregateValue
                 *            the current aggregate.
                 * @return the new aggregate value.
                 */
                @Override
                public Map<String, Long> aggregate(final Long newValue, final
Map<String, Long> aggregateValue) {
                    final Long sampleSize = Long.valueOf(1) +
(aggregateValue.getOrDefault
                                                          ("sample_size",
Long.valueOf(0)));

                    final Long sum = newValue + (aggregateValue.getOrDefault("sum",
Long.valueOf(0)));

                    // calculate the new aggregate
                    aggregateValue.put("mean", sum / (sampleSize));
                    aggregateValue.put("sample_size", sampleSize);
                    aggregateValue.put("sum", sum);
                    return aggregateValue;
                }

                /**
                 * Called to merge two aggregates together.
                 *
```

```
                 * @param aggOne
                 *            the first aggregate
                 * @param aggTwo
                 *            the second aggregate
                 * @return the merged result
                 */
                @Override
                public Map<String, Long> merge(final Map<String, Long> aggOne, final
Map<String, Long> aggTwo) {
                    final Long sampleSize = aggOne.getOrDefault("sample_size",
Long.valueOf(0)) + (
                                aggTwo.getOrDefault("sample_size", Long.valueOf(0)));
                    final Long sum = aggOne.getOrDefault("sum", Long.valueOf(0)) +
                                        (aggTwo.getOrDefault("sum", Long.valueOf(0)));
                    // calculate the new aggregate
                    final Map<String, Long> newAggregate = new HashMap<>();
                    newAggregate.put("mean", sum / (sampleSize));
                    newAggregate.put("sample_size", sampleSize);
                    newAggregate.put("sum", sum);
                    return newAggregate;
                }
            };
        }
}
```

<!---------- UDAF Ends Here ----------→

Once the UDAF logic is ready, then it's time to deploy your KSQL functions to a KSQL server. To begin, build the project by running the following command in the project root directory:
Maven → Run As →Maven Install
The following file will be generated and need to be deployed in the kafka cluster.



Now, simply copy this JAR file to the KSQL extension directory (see the ksql.extension.dir property in the ksql-server.properties file) and restart your KSQL server so that it can pick up the new JAR containing your custom KSQL function. If the entry is not there; configure it by entering the following line in the ksql-server.properties file.
ksql.extension.dir=/apps/confluent/etc/ksql/ext

```
root@tos:/apps/confluent/etc/ksql
# this file except in compliance with the License.  You may obtain a copy of the
# License at
#
# http://www.confluent.io/confluent-community-license
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
# WARRANTIES OF ANY KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations under the License.
#

ksql.extension.dir=/apps/confluent/etc/ksql/ext
#------ Endpoint config -------

### HTTP  ###
# The URL the KSQL server will listen on:
listeners=http://tos.hp.com:8088
```

Create the folder structure and grant the access.

```
(base) [root@tos ksql]# pwd
/apps/confluent/etc/ksql
(base) [root@tos ksql]# mkdir ext
(base) [root@tos ksql]# chmod 777 ext
(base) [root@tos ksql]#
```

Hints[mkdir ext , chmod 777 ext]

Copy the jar.

```
(base) [root@tos ksql]# cd ext
(base) [root@tos ext]# ls
(base) [root@tos ext]# cp /mnt/hgfs/Software/data/MyProducer-0.0.1.jar .
(base) [root@tos ext]# ls -lt
total 76
-rwxr-xr-x. 1 root root 74652 Aug 22 11:03 MyProducer-0.0.1.jar
(base) [root@tos ext]#
```

Restart the confluent
Once KSQL has finished restarting and has connected to a running Apache Kafka® cluster, you can verify that the new functions exist by running the DESCRIBE FUNCTION command from the CLI:
DESCRIBE FUNCTION SUMMARY_STATS_BIG ;

```
ksql> DESCRIBE FUNCTION summary_stats_big;

Name        : SUMMARY_STATS_BIG
Author      : Henry P
Version     : 1.0
Overview    : Example UDAF that computes some summary stats for a stream of
              BigInt
Type        : aggregate
Jar         : /apps/confluent/etc/ksql/ext/MyProducer-0.0.1.jar
Variations  :

        Variation   : SUMMARY_STATS_BIG(INT)
        Returns     : MAP<VARCHAR,INT>
        Description : compute summary stats for BigInt
ksql>
```

Let us generate data for using this UDAF in our query.
Use the following avro schema to generate the data.
Create impressions.avro file in /apps folder and execute the following command from that folder only.
<!----------Schema File : impressions.avro Begins---------->

```
{
      "namespace": "streams",
      "name": "impressions",
      "type": "record",
      "fields": [
            {"name": "impresssiontime", "type": {
               "type": "long",
               "format_as_time" : "unix_long",
```

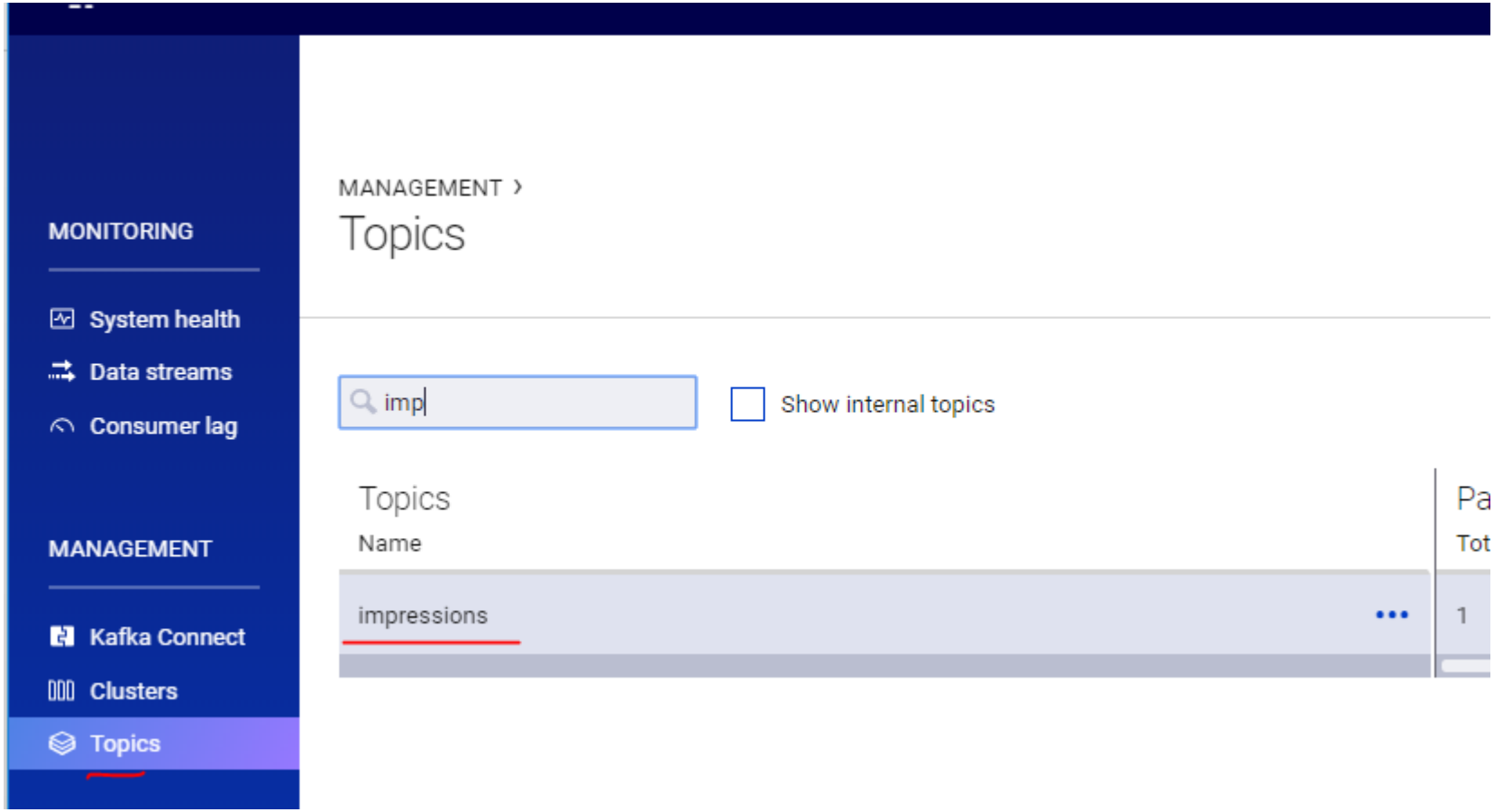
```
        "arg.properties": {
          "iteration": { "start": 1, "step": 10}
        }
      }},
      {"name": "impressionid", "type": {
        "type": "string",
        "arg.properties": {
          "regex": "impression_[1-9][0-9][0-9]"
        }
      }},
      {"name": "userid", "type": {
        "type": "string",
        "arg.properties": {
          "regex": "user_[1-9][0-9]?"
        }
      }},
      {"name": "adid", "type": {
        "type": "string",
        "arg.properties": {
          "regex": "ad_[1-9][0-9]?"
        }
      }},
      {"name": "impresssionscore", "type": {
        "type": "long",
        "arg.properties": {
          "iteration": { "start": 1, "step": 2}
        }
      }}
]
```

}
<!---------- Schema File : impressions.avro Ends ----------→

#cd /apps
ksql-datagen schema=impressions.avro format=avro topic=impressions key=impressionid
When you have a custom schema registered, you can generate test data that's made up of random values that satisfy the schema requirements. In the impressions schema, advertisement identifiers are two-digit random numbers between 10 and 99, as specified by the regular expression ad_[1-9][0-9]. It has impresssionscore which is the score of impression for the advertisement. We will calculate stats on this column using the UDAF we have define earlier.
After a few startup messages, your output should resemble:

```
impression_162 --> ([ 1566555549459 | 'impression_162' | 'user_81' | 'ad_38' | 575 ]) ts:156655
5549459
impression_712 --> ([ 1566555549895 | 'impression_712' | 'user_33' | 'ad_26' | 577 ]) ts:156655
5549895
impression_279 --> ([ 1566555549920 | 'impression_279' | 'user_91' | 'ad_92' | 579 ]) ts:156655
5549920
impression_306 --> ([ 1566555550302 | 'impression_306' | 'user_34' | 'ad_78' | 581 ]) ts:156655
5550303
impression_522 --> ([ 1566555550475 | 'impression_522' | 'user_40' | 'ad_96' | 583 ]) ts:156655
5550476
impression_733 --> ([ 1566555550636 | 'impression_733' | 'user_80' | 'ad_31' | 585 ]) ts:156655
5550637
impression_318 --> ([ 1566555551108 | 'impression_318' | 'user_91' | 'ad_17' | 587 ]) ts:156655
5551109
impression_632 --> ([ 1566555551319 | 'impression_632' | 'user_25' | 'ad_80' | 589 ]) ts:156655
5551319
impression_959 --> ([ 1566555551366 | 'impression_959' | 'user_28' | 'ad_65' | 591 ]) ts:156655
5551366
```

By now, you should have a topic by the name, impressions as shown below.



Consume the Test Data Stream

In the KSQL CLI or in Control Center, register the impressions stream:

CREATE STREAM impressions (viewtime BIGINT, key VARCHAR, userid VARCHAR, adid VARCHAR , impresssionscore BIGINT) WITH (KAFKA_TOPIC='impressions', VALUE_FORMAT='avro');

You can query the stream now.

select * from impressions;

```
1566555480855 | impression_410 | null | null | user_74 | ad_22 | 1
1566555480926 | impression_631 | null | null | user_50 | ad_10 | 3
1566555481142 | impression_705 | null | null | user_31 | ad_12 | 5
1566555481574 | impression_365 | null | null | user_91 | ad_72 | 7
1566555481894 | impression_610 | null | null | user_91 | ad_14 | 9
1566555481970 | impression_526 | null | null | user_45 | ad_88 | 11
1566555482437 | impression_663 | null | null | user_15 | ad_24 | 13
1566555482725 | impression_939 | null | null | user_26 | ad_95 | 15
1566555482856 | impression_654 | null | null | user_39 | ad_73 | 17
```

At this point, invoking our UDF/UDAF is simply a matter of adding it to our KSQL query:

```
SELECT userid , summary_stats_big ( impresssionscore )
FROM impressions   GROUP BY userid;
```

```
user_64 | {sample_size=1, mean=1223, sum=1223}
user_12 | {sample_size=1, mean=1225, sum=1225}
user_14 | {sample_size=1, mean=1227, sum=1227}
user_60 | {sample_size=1, mean=1229, sum=1229}
user_99 | {sample_size=1, mean=1231, sum=1231}
user_23 | {sample_size=1, mean=1233, sum=1233}
user_18 | {sample_size=1, mean=1237, sum=1237}
user_57 | {sample_size=2, mean=1237, sum=2474}
user_29 | {sample_size=1, mean=1241, sum=1241}
user_96 | {sample_size=1, mean=1243, sum=1243}
user_47 | {sample_size=1, mean=1245, sum=1245}
user_15 | {sample_size=1, mean=1247, sum=1247}
user_96 | {sample_size=2, mean=1246, sum=2492}
user_19 | {sample_size=1, mean=1251, sum=1251}
user_24 | {sample_size=1, mean=1253, sum=1253}
```

As you can observe for each userid it returns a stats calculated using the UDAF we have define earlier.

---------- Lab Ends Here ----------

21. KSQL Rest API – TBR

Here's an example request that returns the results from the `LIST STREAMS` command:

```
curl -X "POST" "http://localhost:8088/ksql" \
     -H "Content-Type: application/vnd.ksql.v1+json; charset=utf-8" \
     -d $'{
  "ksql": "LIST STREAMS;",
  "streamsProperties": {}
}'
```

Here's an example request that retrieves streaming data from `TEST_STREAM`:

```
curl -X "POST" "http://localhost:8088/query" \
     -H "Content-Type: application/vnd.ksql.v1+json; charset=utf-8" \
     -d $'{
  "ksql": "SELECT * FROM users_ws;",
  "streamsProperties": {}
}'
```

Example request

```
POST /ksql HTTP/1.1
Accept: application/vnd.ksql.v1+json
Content-Type: application/vnd.ksql.v1+json

{
  "ksql": "CREATE STREAM pageviews_home AS SELECT * FROM pageviews_original WHERE pageid='home'; CREATE STREAM pageviews_al
ice AS SELECT * FROM pageviews_original WHERE userid='alice';",
  "streamsProperties": {
    "ksql.streams.auto.offset.reset": "earliest"
  }
}
```

Copy

Example response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.ksql.v1+json

[
  {
    "statementText":"CREATE STREAM pageviews_home AS SELECT * FROM pageviews_original WHERE pageid='home';",
    "commandId":"stream/PAGEVIEWS_HOME/create",
    "commandStatus": {
      "status":"SUCCESS",
      "message":"Stream created and running"
    },
    "commandSequenceNumber":10
  },
  {
    "statementText":"CREATE STREAM pageviews_alice AS SELECT * FROM pageviews_original WHERE userid='alice';",
    "commandId":"stream/PAGEVIEWS_ALICE/create",
    "commandStatus": {
      "status":"SUCCESS",
      "message":"Stream created and running"
    },
    "commandSequenceNumber":11
  }
]
```

Copy

Example request

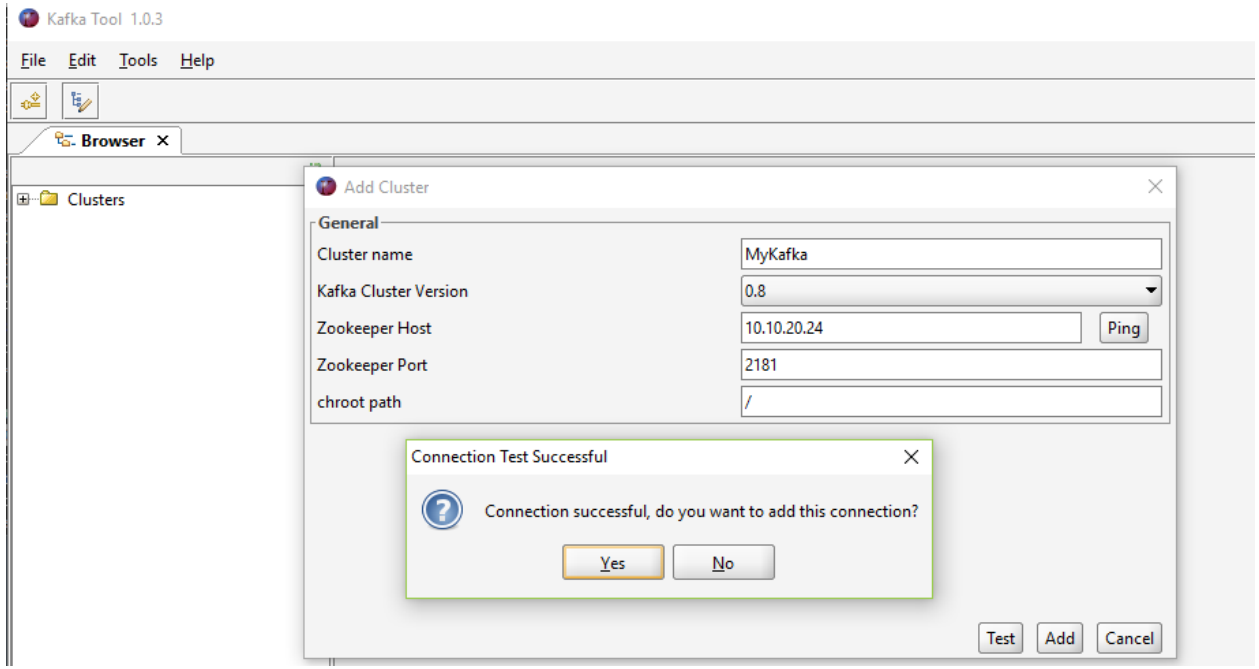
```
POST /query HTTP/1.1
Accept: application/vnd.ksql.v1+json
Content-Type: application/vnd.ksql.v1+json

{
  "ksql": "SELECT * FROM pageviews;",
  "streamsProperties": {
    "ksql.streams.auto.offset.reset": "earliest"
  }
}
```

Copy

Example response

```
HTTP/1.1 200 OK
Content-Type: application/vnd.ksql.v1+json
Transfer-Encoding: chunked

...
{"row":{"columns":[1524760769983,"1",1524760769747,"alice","home"]},"errorMessage":null}
...
```

22. Kafkatools



23. Errors

1. LEADER_NOT_AVAILABLE

{test=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkClient)

```
[2018-05-15 23:46:40,132] WARN [Producer clientId=console-producer] Error while
fetching metadata with correlation id 14 : {test=LEADER_NOT_AVAILABLE} (org.apac
he.kafka.clients.NetworkClient)
[2018-05-15 23:46:40,266] WARN [Producer clientId=console-producer] Error while
fetching metadata with correlation id 15 : {test=LEADER_NOT_AVAILABLE} (org.apac
he.kafka.clients.NetworkClient)
^C[2018-05-15 23:46:40,394] WARN [Producer clientId=console-producer] Error whil
e fetching metadata with correlation id 16 : {test=LEADER_NOT_AVAILABLE} (org.ap
ache.kafka.clients.NetworkClient)
[root@tos opt]#  {test=LEADER_NOT_AVAILABLE} (org.apache.kafka.clients.NetworkCl
ient)
bash: syntax error near unexpected token `org.apache.kafka.clients.NetworkClient
'
```

Solutions: /opt/kafka/config/server.properties

Update the following information.

```
# it uses the value for "listeners" if configured.  Otherwise, it will use the v
alue
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://localhost:9092
```

java.util.concurrent.ExecutionException:

org.apache.kafka.common.errors.TimeoutException: Expiring 1 record(s) for my-kafka-topic-6: 30037 ms has passed since batch creation plus linger time

at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.valueOrError(FutureRecordMetadata.java:94)

at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:64)

at
org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:29)
at com.tos.kafka.MyKafkaProducer.runProducer(MyKafkaProducer.java:97)
at com.tos.kafka.MyKafkaProducer.main(MyKafkaProducer.java:18)
Caused by: org.apache.kafka.common.errors.TimeoutException: Expiring 1 record(s) for my-kafka-topic-6: 30037 ms has passed since batch creation plus linger time.
Solution:
Update the following in all the server properties: /opt/kafka/config/server.properties

```
#    listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://tos.master.com:9093

# Hostname and port the broker will advertise to producers and consumers. If not
 set,
# it uses the value for "listeners" if configured.  Otherwise, it will use the v
alue
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://tos.master.com:9093

# Maps listener names to security protocols, the default is for them to be the s
ame. See the config documentation for more details
#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_
PLAINTEXT,SASL_SSL:SASL_SSL
```

Its should be updated with your hostname and restart the broker
Changes in the following file, if the hostname is to be changed.
//kafka/ Server.properties and control center
/apps/confluent/etc/confluent-control-center/control-center-dev.properties
/apps/confluent/etc/ksql/ksql-server.properties
/tmp/confluent.8A2Ii7O4/connect/connect.properties

Update localhost to resolve to the ip in /etc/hosts.

In case the hostname doesn't started, updated with ip address and restart the broker.

24. Annexure Code:

2. DumplogSegment

/opt/kafka/bin/kafka-run-class.sh kafka.tools.DumpLogSegments --deep-iteration --print-data-log --files \
/tmp/kafka-logs/my-kafka-connect-0/oooooooooooooooooooo.log | head -n 4

```
[root@tos test-topic-0]# more 00000000000000000000.log
[root@tos test-topic-0]# cd ../
[root@tos kafka-logs]# cd my-kafka-connect-0/
[root@tos my-kafka-connect-0]# ls
00000000000000000000.index      00000000000000000011.snapshot
00000000000000000000.log        leader-epoch-checkpoint
00000000000000000000.timeindex
[root@tos my-kafka-connect-0]# more *log
\▒▒▒▒▒▒▒▒afka Connector.--More--(53%)

[root@tos my-kafka-connect-0]# pwd
/tmp/kafka-logs/my-kafka-connect-0
[root@tos my-kafka-connect-0]# /opt/kafka/bin/kafka-run-class.sh kafka.tools.Dum
pLogSegments --deep-iteration --print-data-log --files \
> /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log | head -n 4
Dumping /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log
Starting offset: 0
offset: 0 position: 0 CreateTime: 1530552634675 isvalid: true keysize: -1 values
ize: 31 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -1 sequence:
-1 isTransactional: false headerKeys: [] payload: This Message is from Test File
.
offset: 1 position: 0 CreateTime: 1530552634677 isvalid: true keysize: -1 values
ize: 43 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -1 sequence:
-1 isTransactional: false headerKeys: [] payload: It will be consumed by the Kaf
ka Connector.
```

3. Data Generator – JSON

Streaming Json Data Generator

Downloading the generator

You can always find the most recent release over on github where you can download the bundle file that contains the runnable application and example configurations. Head there now and download a release to get started!

Configuration

The generator runs a Simulation which you get to define. The Simulation can specify one or many Workflows that will be run as part of your Simulation. The Workflows then generates Events and these Events are then sent somewhere. You will also need to define Producers that are used to send the Events generated by your Workflows to some destination. These destinations could be a log file, or something more complicated like a Kafka Queue.

You define the configuration for the json-data-generator using two configuration files. The first is a Simulation Config. The Simulation Config defines the Workflows that should be run and different Producers that events should be sent to. The second is a Workflow configuration (of which you can have multiple). The Workflow defines the frequency of Events and Steps that the Workflow uses to generate the Events. It is the Workflow that defines the format and content of your Events as well.

For our example, we are going to pretend that we have a programmable Jackie Chan robot. We can command Jackie Chan though a programmable interface that happens to take json

as an input via a Kafka queue and you can command him to perform different fighting moves in different martial arts styles. A Jackie Chan command might look like this:

```
{
 "timestamp":"2015-05-20T22:05:44.789Z",
 "style":"DRUNKEN_BOXING",
 "action":"PUNCH",
 "weapon":"CHAIR",
 "target":"ARMS",
 "strength":8.3433
}
```

view rawexampleJackieChanCommand.json hosted with by GitHub

Now, we want to have some fun with our awesome Jackie Chan robot, so we are going to make him do random moves using our json-data-generator! First we need to define a Simulation Config and then a Workflow that Jackie will use.

SIMULATION CONFIG

Let's take a look at our example Simulation Config:

```
{
   "workflows": [{
        "workflowName": "jackieChan",
        "workflowFilename": "jackieChanWorkflow.json"
     }],
   "producers": [{
      "type": "kafka",
```

```
            "broker.server": "192.168.59.103",
            "broker.port": 9092,
            "topic": "jackieChanCommand",
            "flatten": false,
            "sync": false


        }]
    }
```

view rawjackieChanSimConfig.json hosted with by GitHub

As you can see, there are two main parts to the Simulation Config. The Workflows name and list the workflow configurations you want to use. The Producers are where the Generator will send the events to. At the time of writing this, we have three supported Producers:

- A Logger that sends events to log files
- A Kafka Producer that will send events to your specified Kafka Broker
- A Tranquility Producer that will send events to a Druid cluster.

You can find the full configuration options for each on the github page. We used a Kafka producer because that is how you command our Jackie Chan robot.

WORKFLOW CONFIG

The Simulation Config above specifies that it will use a Workflow called jackieChanWorkflow.json. This is where the meat of your configuration would live. Let's

take a look at the example Workflow config and see how we are going to control Jackie Chan:

```
{
   "eventFrequency": 400,
   "varyEventFrequency": true,
   "repeatWorkflow": true,
   "timeBetweenRepeat": 1500,
   "varyRepeatFrequency": true,
   "steps": [{
         "config": [{
               "timestamp": "now()",
               "style": "random('KUNG_FU','WUSHU','DRUNKEN_BOXING')",
               "action": "random('KICK','PUNCH','BLOCK','JUMP')",
               "weapon": "random('BROAD_SWORD','STAFF','CHAIR','ROPE')",
               "target": "random('HEAD','BODY','LEGS','ARMS')",
               "strength": "double(1.0,10.0)"
            }
         ],
         "duration": 0
      }]
}
```

view rawjackieChanWorkflow.json hosted with by GitHub

The Workflow defines many things that are all defined on the github page, but here is a summary:

- At the top are the properties that define how often events should be generated and if / when this workflow should be repeated. So this is like saying we want Jackie Chan to do a martial arts move every 400 milliseconds (he's FAST!), then take a break for 1.5 seconds, and do another one.
- Next, are the Steps that this Workflow defines. Each Step has a config and a duration. The duration specifies how long to run this step. The config is where it gets interesting!

WORKFLOW STEP CONFIG

The Step Config is your specific definition of a json event. This can be any kind of json object you want. In our example, we want to generate a Jackie Chan command message that will be sent to his control unit via Kafka. So we define the command message in our config, and since we want this to be fun, we are going to randomly generate what kind of style, move, weapon, and target he will use.

You'll notice that the values for each of the object properties look a bit funny. These are special Functions that we have created that allow us to generate values for each of the properties. For instance, the "random('KICK','PUNCH','BLOCK','JUMP')" function will randomly choose one of the values and output it as the value of the "action" property in the command message. The "now()" function will output the current date in an ISO8601 date formatted string. The "double(1.0,10.0)" will generate a random double between 1 and 10 to determine the strength of the action that Jackie Chan will perform. If we wanted to, we could make Jackie Chan perform combo moves by defining a number of Steps that will be executed in order.

There are many more Functions available in the generator with everything from random string generation, counters, random number generation, dates, and even support for

randomly generating arrays of data. We also support the ability to reference other randomly generated values. For more info, please check out the full documentation on the github page.

Once we have defined the Workflow, we can run it using the json-data-generator. To do this, do the following:

1. If you have not already, go ahead and download the most recent release of the json-data-generator.
2. Unpack the file you downloaded to a directory.

   (tar -xvf json-data-generator-1.4.0-bin.tar -C /apps)

3. Copy your custom configs into the conf directory
4. Then run the generator like so:
    1. java -jar json-data-generator-1.4.0.jar jackieChanSimConfig.json

You will see logging in your console showing the events as they are being generated. The jackieChanSimConfig.json generates events like these:

{"timestamp":"2015-05-20T22:21:18.036Z","style":"WUSHU","action":"BLOCK","weapon":"CHAIR","target":"BODY","strength":4.7912}
{"timestamp":"2015-05-20T22:21:19.247Z","style":"DRUNKEN_BOXING","action":"PUNCH","weapon":"BROAD_SWORD","target":"ARMS","strength":3.0248}
{"timestamp":"2015-05-20T22:21:20.947Z","style":"DRUNKEN_BOXING","action":"BLOCK","weapon":"ROPE","target":"H

EAD","strength":6.7571}
{"timestamp":"2015-05-20T22:21:22.715Z","style":"WUSHU","action":"KICK","weapon":"BROAD_SWORD","target":"ARMS","strength":9.2062}
{"timestamp":"2015-05-20T22:21:23.852Z","style":"KUNG_FU","action":"PUNCH","weapon":"BROAD_SWORD","target":"HEAD","strength":4.6202}
{"timestamp":"2015-05-20T22:21:25.195Z","style":"KUNG_FU","action":"JUMP","weapon":"ROPE","target":"ARMS","strength":7.5303}
{"timestamp":"2015-05-20T22:21:26.492Z","style":"DRUNKEN_BOXING","action":"PUNCH","weapon":"STAFF","target":"HEAD","strength":1.1247}
{"timestamp":"2015-05-20T22:21:28.042Z","style":"WUSHU","action":"BLOCK","weapon":"STAFF","target":"ARMS","strength":5.5976}
{"timestamp":"2015-05-20T22:21:29.422Z","style":"KUNG_FU","action":"BLOCK","weapon":"ROPE","target":"ARMS","strength":2.152}
{"timestamp":"2015-05-20T22:21:30.782Z","style":"DRUNKEN_BOXING","action":"BLOCK","weapon":"STAFF","target":"ARMS","strength":6.2686}
{"timestamp":"2015-05-20T22:21:32.128Z","style":"KUNG_FU","action":"KICK","weapon":"BROAD_SWORD","target":"BODY","strength":2.3534}

If you specified to repeat your Workflow, then the generator will continue to output events and send them to your Producer simulating a real world client, or in our case, continue to

make Jackie Chan show off his awesome skills. If you also had a Chuck Norris robot, you could add another Workflow config to your Simulation and have the two robots fight it out! Just another example of how you can use the generator to simulate real world situations.

4. Resources

https://developer.ibm.com/hadoop/2017/04/10/kafka-security-mechanism-saslplain/
https://sharebigdata.wordpress.com/2018/01/21/implementing-sasl-plain/
https://developer.ibm.com/code/howtos/kafka-authn-authz
https://github.com/confluentinc/kafka-streams-examples/tree/4.1.x/
https://github.com/spring-cloud/spring-cloud-stream-samples/blob/master/kafka-streams-samples/kafka-streams-table-join/src/main/java/kafka/streams/table/join/KafkaStreamsTableJoin.java