

1.	Producer in Java – 60 Minutes .....	3
2.	Consumer – Java API – 45 Minutes .....	18
3.	Java API – Producer – JSON Object - 60 Minutes .....	28
4.	Java API – Consumer – Json Deserialization - 60 Minutes .....	55
5.	Streaming – Exactly Once API – 60 Minutes.....	70
6.	Schema registry – 90 Minutes.....	96
7.	DSL - Transform a stream of events – 60 Minutes.....	116
8.	Stream Using – FMV Stateless – 45 Minutes.....	138
9.	DSL: stateful transformations – reduce – 45 Minutes .....	146
10.	Running your first Kafka Streams Application: WordCount – 60 minutes .....	159
11.	Stream – DSL and Windows – 45 Minutes .....	177
12.	DSL - join a stream and a table together – 90 Minutes .....	185
14.	Processor API integration – 60 Minutes .....	211
15.	Kafka - UDAF .....	224
16.	KSQL Rest API – TBR.....	245
17.	Kafkatoools.....	248
18.	Errors.....	249

## 2 Kafka – Dev Ops

1.	LEADER_NOT_AVAILABLE.....	249
	java.util.concurrent.ExecutionException: .....	249
19.	Annexure Code:.....	252
2.	DumplogSegment.....	252
3.	Data Generator – JSON .....	253
20.	Pom.xml (Standalone) .....	261
21.	pom.xml (Spring boot).....	267
4.	Resources.....	270

Last Updated: 30 March 2022

## 1. Producer in Java – 60 Minutes

Learning outcome:

- Sending Message Synchronously
- Understanding RecordMetadata
- Using ProducerRecord of Java API

In this tutorial, we are going to create a simple Java Kafka producer. You need to create a new replicated Kafka topic called **my-kafka-topic**, then you will develop code for Kafka producer using Java API that send records to this topic.

You will send records by the Kafka producer synchronously.

You need to start the zookeeper and three nodes brokers before going ahead.

```
[root@tos scripts]# jps
4880 Kafka
4881 Kafka
4882 Kafka
6022 Jps
4845 QuorumPeerMain
[root@tos scripts]#
```

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

## 4 Kafka – Dev Ops

Create Replicated/Unreplicated Kafka Topic. If you have three nodes cluster, change the replication factor to 3 else 1.

Create topic

```
#/opt/kafka/bin/kafka-topics.sh --create --replication-factor 1 --partitions 13 --topic my-kafka-topic --bootstrap-server kafka0:9092
```

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-factor 3 --partitions 13 --topic my-kafka-topic --zookeeper localhost:2181
Created topic "my-kafka-topic".
[root@tos scripts]#
```

## List created topics, you can verify the topic now

```
/opt/kafka/bin/kafka-topics.sh --list --bootstrap-server kafka0:9092
```

```
[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]#
```

## 5 Kafka – Dev Ops

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.

## 6 Kafka – Dev Ops

After you create a maven java project, include the dependency in pom.xml.

### 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.

Note: If you are using docker replace the BOOTSTRAP\_SERVERS with “localhost:8081” – which is the advertise listeners for the broker from outside the container.

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

```
public class MyKafkaProducer
```

Add the following variables in your code.

```
private final static String TOPIC = "my-kafka-topic";
private final static String BOOTSTRAP SERVERS =
    "localhost:8082,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 method 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
```

```
bash: .: command not found  
[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-beginning
```

Execute the main program.

The screenshot shows an IDE interface with the following components:

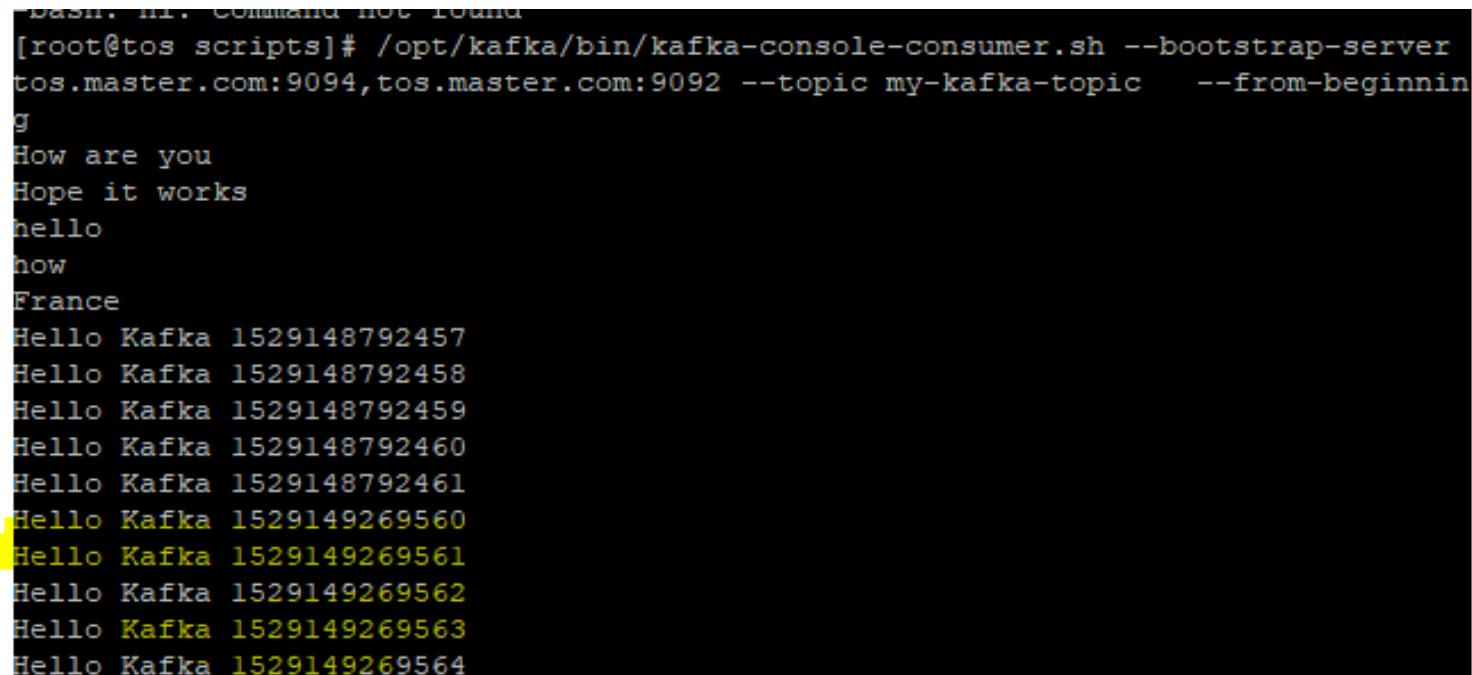
- Editor Area:** Displays the code for `MyKafkaProducer.java`. The code defines a class `MyKafkaProducer` with a `main` method. It sets up a topic named "my-kafka-topic" and bootstrap servers at "10.10.20.24:9092,10.10.20.24:9094". The `main` method checks for command-line arguments and calls `runProducer` with either a default value of 5 or the provided argument.
- Console Tab:** Shows the execution output of the application. The output indicates that SLF4J failed to load its logger binder and defaulted to a no-operation logger. It also shows the configuration of `_JAVA_OPTIONS` and the successful sending of five records to the Kafka topic. The records are sent with different partitions and offsets, and each record includes a timestamp.

```
4 import org.apache.kafka.common.serialization.LongSerializer;
5 import org.apache.kafka.common.serialization.StringSerializer;
6 import java.util.Properties;
7
8 public class MyKafkaProducer {
9
10    private final static String TOPIC = "my-kafka-topic";
11    private final static String BOOTSTRAP_SERVERS = "10.10.20.24:9092,10.10.20.24:9094";
12
13    public static void main(String[] args) {
14
15        try {
16            if (args.length == 0) {
17                runProducer(5);
18            } else {
19                runProducer(Integer.parseInt(args[0]));
20            }
21        } catch (Exception e) {
22            // TODO: handle exception
23        }
24    }
}
<terminated> MyKafkaProducer [Java Application] D:\MyExperiment\Java\jdk1.8.0_131\bin\javaw.exe (Jun 16, 2018, 5:11:08 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.
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
Sent record(key=1529149269560 value=Hello Kafka 1529149269560) meta(partition=12, offset=0) time=361
Sent record(key=1529149269561 value=Hello Kafka 1529149269561) meta(partition=6, offset=0) time=380
Sent record(key=1529149269562 value=Hello Kafka 1529149269562) meta(partition=6, offset=1) time=393
Sent record(key=1529149269563 value=Hello Kafka 1529149269563) meta(partition=9, offset=2) time=413
Sent record(key=1529149269564 value=Hello Kafka 1529149269564) meta(partition=2, offset=1) time=458
```

You should be able to view the messages 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.

```
#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server  
localhost:8082:9094,localhost:9092 --topic my-kafka-topic --from-beginning
```



A screenshot of a terminal window showing the output of a Kafka consumer command. The command is:

```
[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-beginning
```

The output shows several messages being printed:

```
How are you  
Hope it works  
hello  
how  
France  
Hello Kafka 1529148792457  
Hello Kafka 1529148792458  
Hello Kafka 1529148792459  
Hello Kafka 1529148792460  
Hello Kafka 1529148792461  
Hello Kafka 1529149269560  
Hello Kafka 1529149269561  
Hello Kafka 1529149269562  
Hello Kafka 1529149269563  
Hello Kafka 1529149269564
```

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

## 2. Consumer – Java API – 45 Minutes

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.

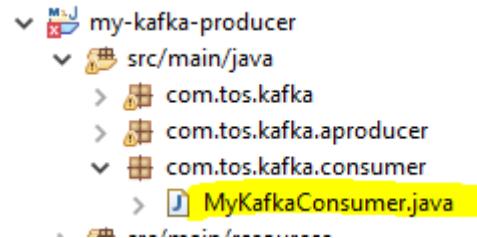
#### KafkaConsumerExample.java - imports and constants

You can use the earlier java project.

In that create a separate package and the following class.

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.time.Duration;
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 =
        "localhost:8082";
```

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());  
//props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
  
    // 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(Duration.ofMillis(1000));

        if (consumerRecords.count() == 0) {
            noRecordsCount++;
            if (noRecordsCount > giveUp)
```

```
        break;
    }  
  
    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

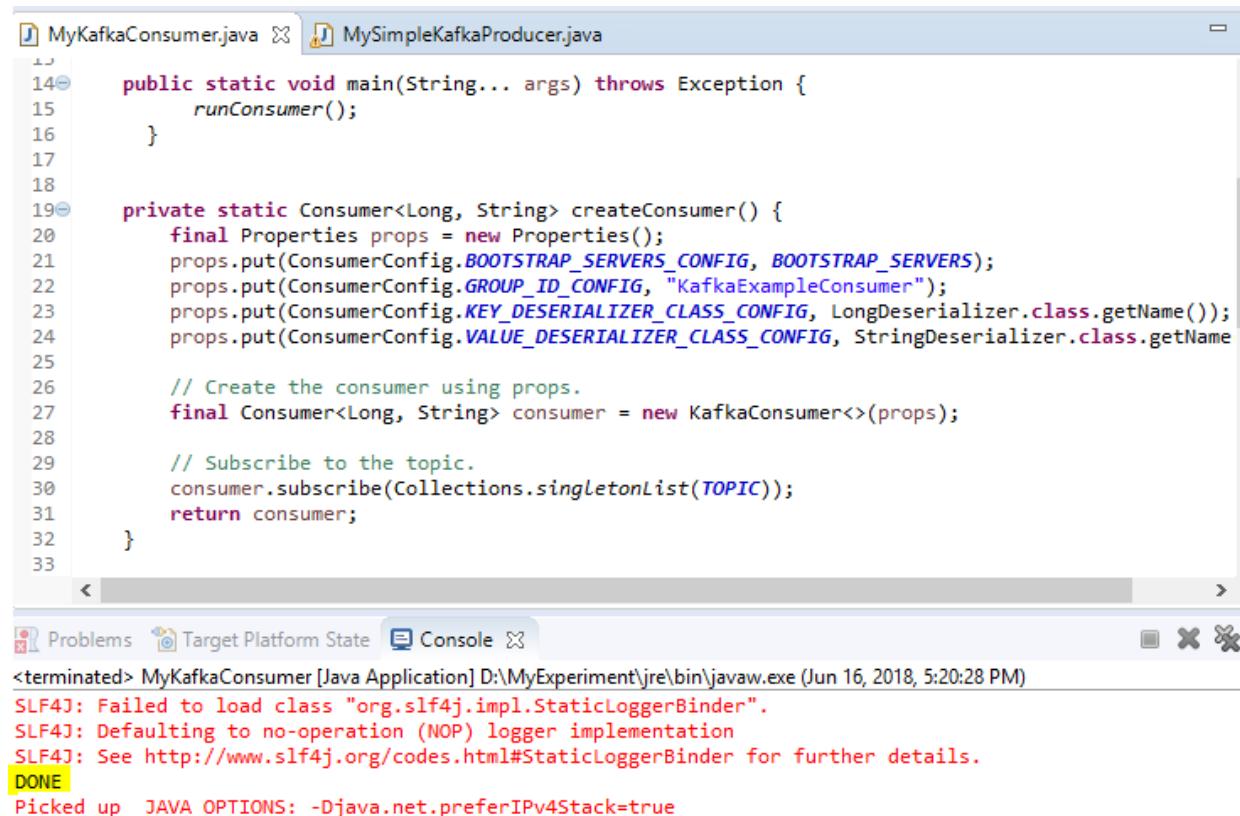
Next you define the `main` method.

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

The `main` method just invokes `runConsumer`.

Try running the consumer program.

Run the consumer from your IDE.

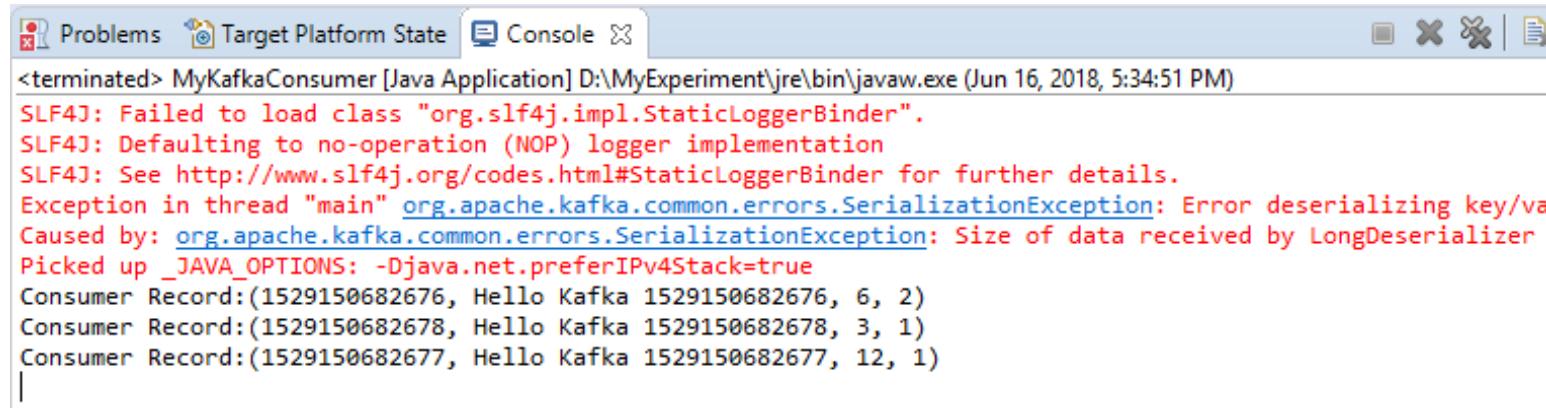


The screenshot shows an IDE interface with two tabs at the top: "MyKafkaConsumer.java" and "MySimpleKafkaProducer.java". The "MyKafkaConsumer.java" tab is active, displaying Java code for a Kafka consumer. The code defines a static `main` method that calls `runConsumer`. It also contains a private static method `createConsumer` which sets up consumer properties, including bootstrap servers, group ID, and deserializer classes. The code then creates a consumer using these properties and subscribes it to a topic. The "Console" tab at the bottom shows the application's output:

```
<terminated> MyKafkaConsumer [Java Application] D:\MyExperiment\jre\bin\javaw.exe (Jun 16, 2018, 5:20:28 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.
DONE
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
```

As you can observe there are no records. Why?? Messages were already consumed by the terminal windows which offset were committed.

Execute the consumer and Producer, then you will observe messages in the consumer console.

A screenshot of an IDE's Console tab. The title bar shows 'Problems', 'Target Platform State', and 'Console'. The console output is as follows:

```
<terminated> MyKafkaConsumer [Java Application] D:\MyExperiment\jre\bin\javaw.exe (Jun 16, 2018, 5:34:51 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.
Exception in thread "main" org.apache.kafka.common.errors.SerializationException: Error deserializing key/value for ConsumerRecord@1529150682676.
Caused by: org.apache.kafka.common.errors.SerializationException: Size of data received by LongDeserializer: 12 >= _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
Consumer Record:(1529150682676, Hello Kafka 1529150682676, 6, 2)
Consumer Record:(1529150682678, Hello Kafka 1529150682678, 3, 1)
Consumer Record:(1529150682677, Hello Kafka 1529150682677, 12, 1)
```

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="STDOOUT"
    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="STDOOUT" />
</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 -----**

### 3. Java API – Producer – JSON Object - 60 Minutes

Learning Outcomes:

- JSON Messages Serialization
- Async and Sync Mode.
- Callback method

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.

## 29 Kafka – Dev Ops

```
com.tos.kafka.producer.adv
  SimpleStockProducer.java
  SimpleStockProducer
  StockPrice.java
  StockPrice
  StockPriceSerializer.java
```

Define all dependencies in the pom.xml:

### 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 com.google.gson.Gson;
public class StockPrice {

    private final int dollars;
    private final int cents;
    private final String name;
    static Gson jfactory = new Gson();
    public StockPrice(final String json) {

        this(jfactory.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.

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

```
public class SimpleStockProducer
```

Add the following import in the above class.

```
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", "localhost:8082");
    // 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 o
    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) {

    // invokeSync()
    // invokeAsync()

}
```

```
public static void invokeSync () {
```

```
Properties props = initProducer();
Producer<String, StockPrice> producer = new KafkaProducer<String,
StockPrice>(props);
sendMessage(producer, TOPIC);
}
```

```
public static void invokeAsync () {
    Properties props = initProducer();
    Producer<String, StockPrice> producer = new KafkaProducer<String,
StockPrice>(props);

    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 ()

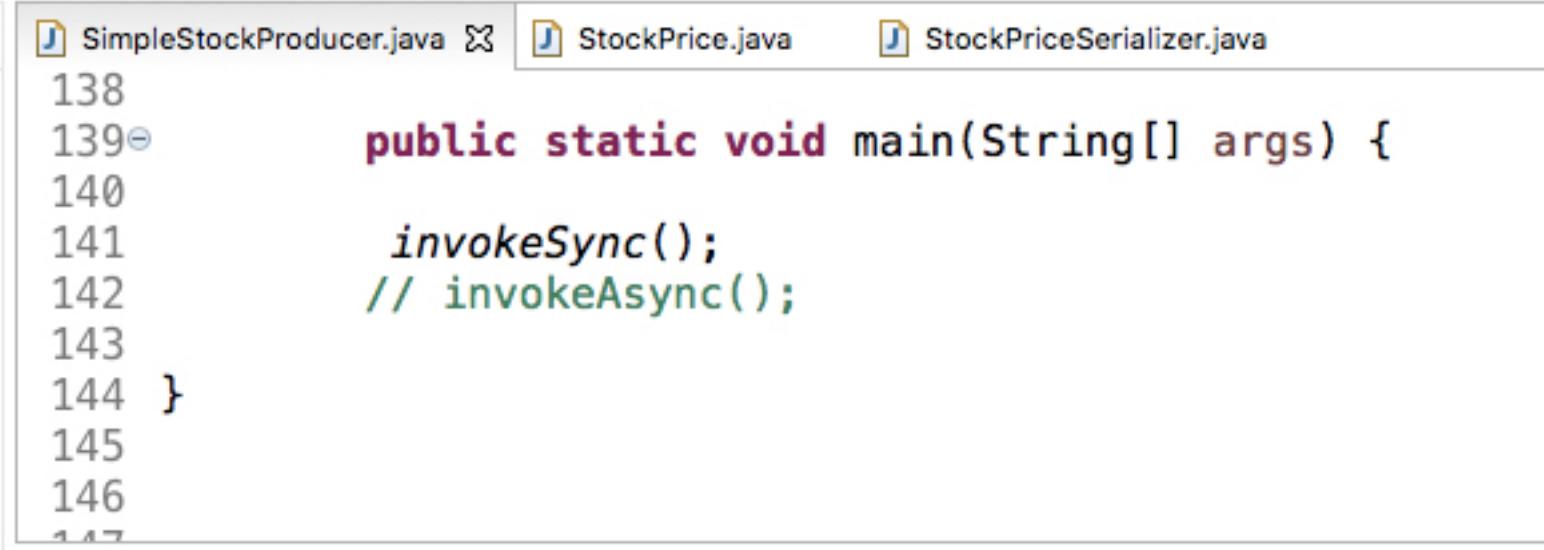
```
#sh start3Brokers.sh
```

```
[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 --bootstrap-server kafka:9092 --replication-factor 1 --partitions 2 --topic stock-prices
```

We will send the JSON message synchronously.

In the main method of `SimpleStockProducer` ensure that you have uncommented `invokeSync()` and commented `invokeAsync()`



```
138
139     public static void main(String[] args) {
140
141         invokeSync();
142         // invokeAsync();
143
144     }
145
146
147
```

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 screenshot shows the Eclipse IDE interface with the following components visible:

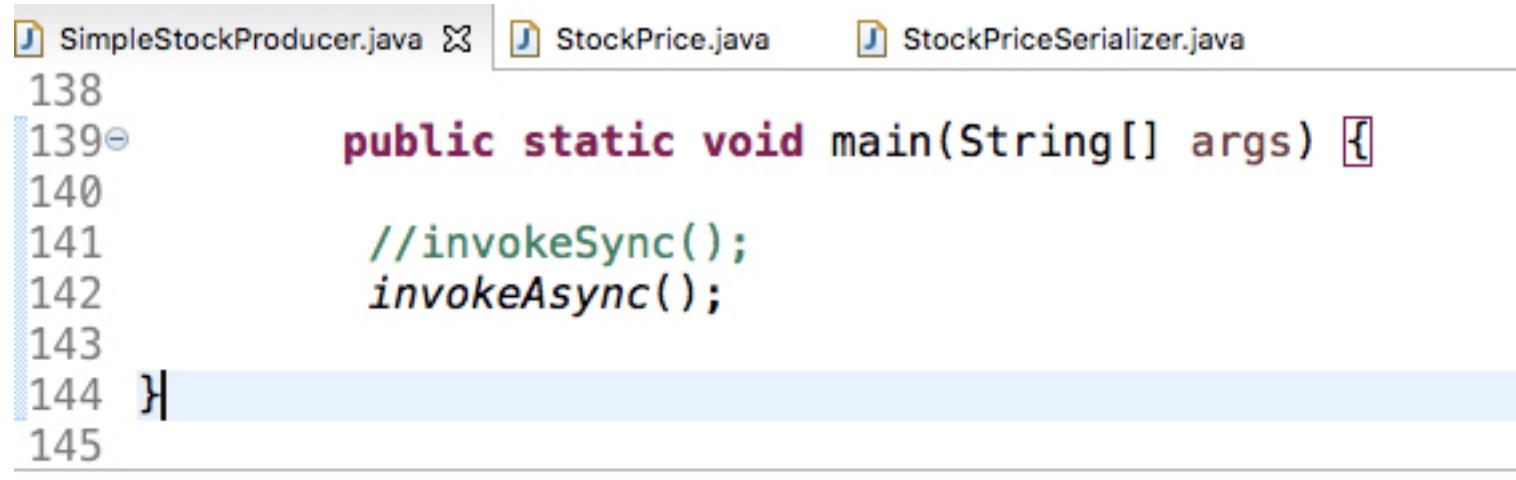
- Package Explorer:** Shows the project structure with files like MyKafkaConsumer.java, SimpleStockProducer.java, StockPrice.java, and StockPriceSerializer.java.
- Editor:** Displays the code for SimpleStockProducer.java, specifically the main method which contains a comment for invokeSync() and another for invokeAsync().
- Console:** Shows the execution output:
  - 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.
  - >>>stock-prices Offset:2
  - Message sent successfully
  - >>>stock-prices Offset:3
  - Message sent successfully
  - Message sent successfully & Close

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.

```
4398 ops
[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.



```
SimpleStockProducer.java StockPrice.java StockPriceSerializer.java
138
139     public static void main(String[] args) {
140
141         //invokeSync();
142         invokeAsync();
143
144     }
145
```

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

```
<terminated> SimpleStockProducer (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (30-Mar-2022, 2:03:24 PM – 2:03:58 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=0) time=1356  
Message sent successfully  
sent record(key= Async STOCK 1 value=StockPrice{dollars=200, cents=1000, name=' Async STOCK 1'})  
meta(partition=1, offset=4) time=142  
Message sent successfully  
sent record(key= Async STOCK 2 value=StockPrice{dollars=400, cents=2000, name=' Async STOCK 2'})  
meta(partition=0, offset=1) time=58  
Message sent successfully  
sent record(key= Async STOCK 3 value=StockPrice{dollars=600, cents=3000, name=' Async STOCK 3'})  
meta(partition=1, offset=5) time=80  
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 consumer 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 --bootstrap-server kafka0:9092 --replication-factor 1 --partitions 4 --topic mystock-prices
```

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create --zookeeper tos.master.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 topic, mystock-prices with 4 partitions so that message will be distributed across the partition.

Ensure that the following method is not commented.

```
SimpleStockProducer.java StockPrice.java StockPriceSerializer.java
133             final int dollarAmount = i* 200;
134             final int centAmount = i * 1000;
135             final StockPrice stockPrice = new StockPrice(" Async STOCK "+ 
136             return new ProducerRecord<>(TOPIC, stockPrice.getName(), stockPrice);
137         }
138
139     public static void main(String[] args) {
140
141         //invokeSync();
142         invokeAsync();
143
144     }
145
```

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 --bootstrap-server kafka0:9092 --topic mystock-prices
```

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --describe --zookeeper tos.master.com:2181 --topic mystock-prices
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 kafka0:9092 --topic mystock-prices --from-beginning
```

```
[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**

## 4. Java API – Consumer – Json Deserialization - 60 Minutes

Learning outcome:

- JSON deserialization and retry option

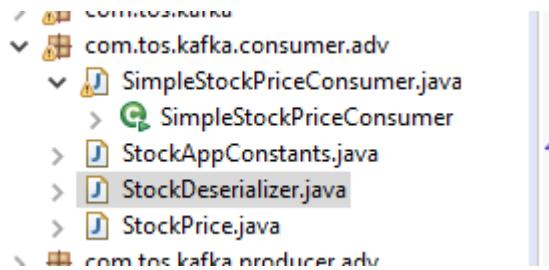
We will use the same Maven project that was created earlier.

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:



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:8082";

}
```

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.time.Duration;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
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;
```

```
public class SimpleStockPriceConsumer
```

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,
"earliest");
    // 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 = 10;
        int noRecordsCount = 0;
        //consumer.seekToBeginning(consumer.assignment());
        while (true) {
            ConsumerRecords<String, StockPrice> consumerRecords
= consumer.poll(Duration.ofMillis(2000));
            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();
    }
}
```

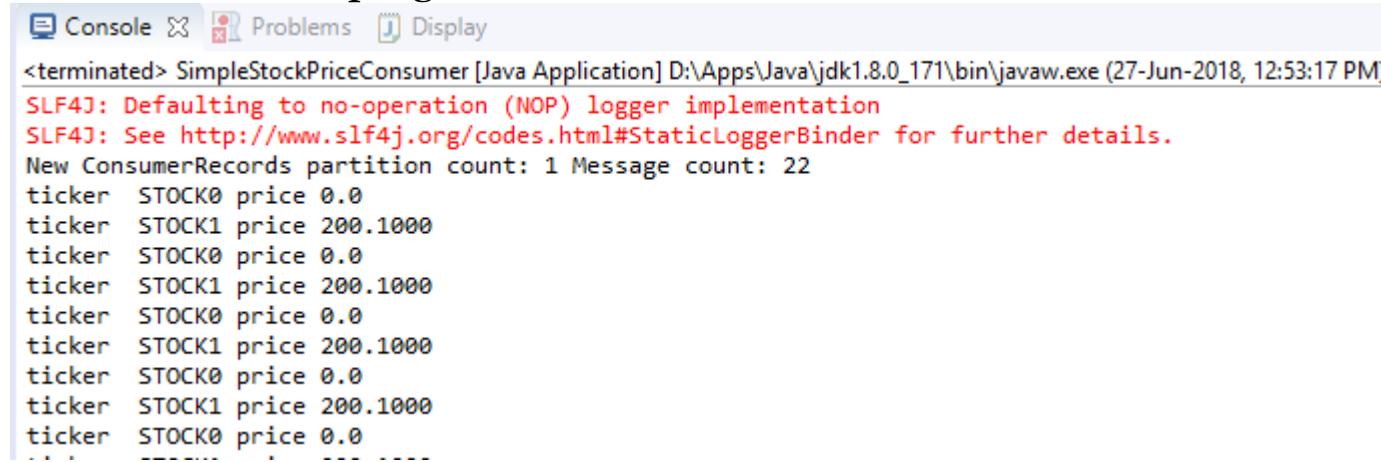
```
}
```

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.



The screenshot shows a Java application running in an IDE's terminal. The title bar says "SimpleStockPriceConsumer [Java Application]". The output window displays log messages from SLF4J and consumer records being processed. The log includes:

```
<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
... -----
```

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 -----

## 5. Streaming – Exactly Once API – 60 Minutes

Case:

It will demonstrate Transactional consume-transform-produce Loop:

For our example, we're going to consume messages from an input topic, **input**.

Then for each sentence, we'll count every word and send the individual word counts to an output topic, **output**.

In the example, we'll assume that there is already transactional data available in the **input** topic.

Create a maven Java Project or use the existing project

Create and copy the following two classes.

TransactionalWordCount.java create in the following package:

```
package com.osteck.transaction;
```

Import the following packages.

```
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_C
```

```
LASS_CONFIG;
import static
org.apache.kafka.clients.consumer.ConsumerConfig.VALUE_DESERIALIZER
_CLASS_CONFIG;
import static
org.apache.kafka.clients.producer.ProducerConfig.ENABLE_IDEMPOTENCE
_CONFIG;
import static
org.apache.kafka.clients.producer.ProducerConfig.KEY_SERIALIZER_CLA
SS_CONFIG;
import static
org.apache.kafka.clients.producer.ProducerConfig.TRANSACTIONAL_ID_C
ONFIG;
import static
org.apache.kafka.clients.producer.ProducerConfig.VALUE_SERIALIZER_C
LASS_CONFIG;

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;
```

Add the following properties for Producer and Consumer Object. It defines the topics for producer and consumer.

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

Define the following method for pushing message to the broker. Changes the boostral server config property with that of your broker ip and port no.

```
private static KafkaProducer<String, String>
createKafkaProducer() {
    Properties props = new Properties();
    props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:8082");
```

```
    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);

}
```

Additionally, though, we need to specify a *transactional.id* and enable *idempotence*:

Because we've enabled idempotence, Kafka will use this transaction id as part of its algorithm to **deduplicate any message this producer sends**, ensuring idempotency.

Simply put, if the producer accidentally sends the same message to Kafka more than once, these settings enable it to notice.

All that we need to do is **make sure the transaction id is distinct for each producer**, though consistent across restarts

The following method define the code for connecting to the topic for transactional read. Its as transactional aware customer. When we consume, we can read all the messages on a topic partition in order. Though, we can indicate with **isolation.level** that we should wait to read transactional messages until the associated transaction has been committed:

```
private static KafkaConsumer<String, String> createKafkaConsumer()
{
    Properties props = new Properties();
    props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:8082");
    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;
}
```

Using a value of **read\_committed** ensures that we don't read any transactional messages before the transaction completes.

The default value of ***isolation.level*** is ***read\_uncommitted***.

The following code enables the producer for transaction.

```
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();

}

}
```

Once we are ready, then we also need to call *initTransaction* to prepare the producer to use transactions:

```
producer.initTransactions();
```

This registers the producer with the broker as one that can use transactions, **identifying it by its *transactional.id* and a sequence number, or epoch**. In turn, the broker will use these to write-ahead any actions to a transaction log.

And consequently, **the broker will remove any actions from that log that belong to a producer with the same transaction id and earlier epoch**, presuming them to be from defunct transactions.

Now that we have the producer and consumer both configured to write and read transactionally, we can consume records from our input topic and count each word in each record:

Note, that there is nothing transactional about the above code. But, since we used `read_committed`, it means that no messages that were written to the input topic in the same transaction will be read by this consumer until they are all written.

Now, we can send the calculated word count to the **output** topic.

```
package com.ostechn.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:

The screenshot shows the Eclipse IDE interface. In the top bar, there are tabs for 'Package Explorer', 'TransactionalWordCount.java', 'join-stream/pom.xml', 'exactly-once/pom.xml', and 'Tuple.java'. The 'TransactionalWordCount.java' tab is active. On the left, the 'Package Explorer' view shows two projects: 'exactly-once' and 'join-stream'. The 'exactly-once' project contains a 'src/main/java' folder with a package 'com.ostech.transaction' containing a class 'TransactionalWordCount'. This class is highlighted with a red rectangle. Below it are files 'Replicator.java', 'src/main/resources', 'src/test/java', 'src/test/resources', 'JRE System Library [J2SE-1.5]', 'Maven Dependencies', 'src', 'target', and 'pom.xml'. The 'join-stream' project has similar structure. In the bottom right corner, the 'Console' tab is active, displaying the following log output:

```
TransactionalWordCount [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/java (20 Aug, 2020 1:1)
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 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 gets started.

Execute the following from a separate terminal:

```
# kafka-console-producer.sh --broker-list kafka0:9092 --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.sh --bootstrap-server kafka0:9092 --topic output --from-beginning
```

```
[root@kafka0 ~]#  
[root@kafka0 ~]# kafka-console-producer.sh --broker-list kafka0:9092 --topic input  
>Let us count the number of count  
>how much the count is  
>
```

```
[root@kafka0 /]# kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic output --from-beginning
the-1
number-1
of-1
count-2
Let-1
us-1
the-1
how-1
count-1
is-1
much-1
[]
```

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();
80
```

Stop the program and start it again

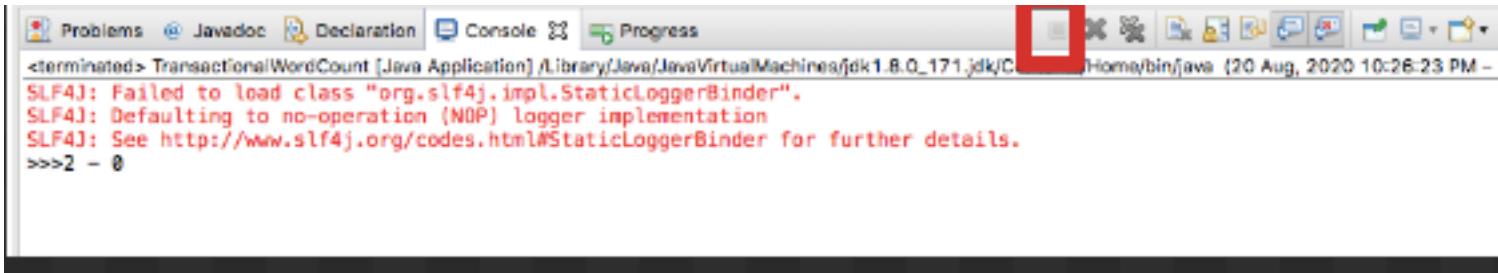
And type a sentence on the producer terminal:

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

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

```
number-1  
of-1  
count-2  
Let-1  
us-1  
the-1  
how-1  
count-1  
is-1  
much-1  
works-1  
transactions-1
```

stop the program execution:



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

Verify the messages:

Let us observe the messages of the Topic: **input**

```
# kafka-dump-log --print-data-log --files '/opt/data/kafka-logs/input-0/oooooooooooooooooooo.log' --deep-iteration
```

```
[root@kafka0 ~]# kafka-dump-log --print-data-log --files '/opt/data/kafka-logs/input-0/oooooooooooooooooooo.log' --deep-iteration
Dumping /opt/data/kafka-logs/input-0/oooooooooooooooooooo.log
Starting offset: 0
baseOffset: 0 lastOffset: 0 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position: 0 CreateTime: 1646544016508 size: 100 magic: 2 compresscodec: none crc: 103376667 isValid: true
I offset: 0 CreateTime: 1646544016508 keySize: -1 valueSize: 32 sequence: -1 headerKeys: 0 payload: Let us count the number of count
baseOffset: 1 lastOffset: 1 count: 1 baseSequence: -1 lastSequence: -1 producerId: -1 producerEpoch: -1 partitionLeaderEpoch: 0 isTransactional: false isControl: false position: 100 CreateTime: 1646544257010 size: 76 magic: 2 compresscodec: none crc: 3262270893 isValid: true
I offset: 1 CreateTime: 1646544257010 keySize: -1 valueSize: 8 sequence: -1 headerKeys: 0 payload: count me
[root@kafka0 ~]#
```

Here, producerID and the sequence are assigned to -1, that is transaction is not enabled for the messages of topic, **input**.

Now, Let us observe the messages of the Topic: **output**

```
# kafka-dump-log --print-data-log --files '/opt/data/kafka-logs/output-0/oooooooooooooooooooo.log' --deep-iteration
```

```
[root@kafka0 /]# kafka-dump-log --print-data-log --files '/opt/data/kafka-logs/output-0/00000000000000000000.log' --deep-iteration
Dumping /opt/data/kafka-logs/output-0/00000000000000000000.log
Starting offset: 0
baseOffset: 0 lastOffset: 5 count: 6 baseSequence: 0 lastSequence: 5 producerId: 1000 producerEpoch: 8 partitionLeaderEpoch: 0 isTransactional: true isControl: false position: 0 CreateTime: 1646544017585 size: 157 magic: 2 compresscodec: none crc: 2637355025 isvalid: true
| offset: 0 CreateTime: 1646544017571 keySize: 3 valueSize: 5 sequence: 0 headerKeys: 0 key: the payload: the-1
| offset: 1 CreateTime: 1646544017582 keySize: 6 valueSize: 8 sequence: 1 headerKeys: 0 key: number payload: number-1
| offset: 2 CreateTime: 1646544017584 keySize: 2 valueSize: 4 sequence: 2 headerKeys: 0 key: of payload: of-1
| offset: 3 CreateTime: 1646544017585 keySize: 5 valueSize: 7 sequence: 3 headerKeys: 0 key: count payload: count-2
| offset: 4 CreateTime: 1646544017585 keySize: 3 valueSize: 5 sequence: 4 headerKeys: 0 key: Let payload: Let-1
| offset: 5 CreateTime: 1646544017585 keySize: 2 valueSize: 4 sequence: 5 headerKeys: 0 key: us payload: us-1
baseOffset: 6 lastOffset: 6 count: 1 baseSequence: -1 lastSequence: -1 producerId: 1000 producerEpoch: 8 partitionLeaderEpoch: 0 isTransactional: true isControl: true position: 157 CreateTime: 1646544017658 size: 78 magic: 2 compresscodec: none crc: 2247913521 isvalid: true
| offset: 6 CreateTime: 1646544017658 keySize: 4 valueSize: 6 sequence: -1 headerKeys: 0 endTxnMarker: COMMIT coordinatorEpoch: 0
```

As you have observed, the commit transaction is done at the end of the batch. There are producerID and sequence no maintain for each message under a transaction.

When you abort:

```
# kafka-dump-log --print-data-log --files '/opt/data/kafka-logs/output-0/oooooooooooooooooooo.log' --deep-iteration
```

```
baseOffset: 13 lastOffset: 14 count: 2 baseSequence: 0 lastSequence: 1 producerId: 1000 producerEpoch: 10 partitionLeaderEpoch: 0 isTransactional: true
isControl: false position: 453 CreateTime: 1646546263077 size: 113 magic: 2 compresscodec: none crc: 1144070539 isValid: true
| offset: 13 CreateTime: 1646546263063 keySize: 5 valueSize: 7 sequence: 0 headerKeys: [] key: works payload: works-1
| offset: 14 CreateTime: 1646546263077 keySize: 12 valueSize: 14 sequence: 1 headerKeys: [] key: transactions payload: transactions-1
baseOffset: 15 lastOffset: 15 count: 1 baseSequence: -1 lastSequence: -1 producerId: 1000 producerEpoch: 10 partitionLeaderEpoch: 0 isTransactional: true
isControl: true position: 566 CreateTime: 1646546263281 size: 78 magic: 2 compresscodec: none crc: 3770326941 isValid: true
| offset: 15 CreateTime: 1646546263281 keySize: 4 valueSize: 6 sequence: -1 headerKeys: [] endTxnMarker: ABORT coordinatorEpoch: 0
[root@kafka0 /]# kafka-dump-log --print-data-log --files '/opt/data/kafka-logs/output-0/00000000000000000000.log' --deep-iteration
```

Let us write a consumer that consume only the committed message.

```
package com.ostechn.transaction;

public class ConsumeOnlyTransactionalMessage {
```

Import the following:

```
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_CONFIG;
import static
```

```
org.apache.kafka.clients.consumer.ConsumerConfig.VALUE_DESERIALIZER  
_CLASS_CONFIG;  
  
import java.time.Duration;  
import java.util.Properties;  
  
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;
```

Define the configuration:

```
private static final String CONSUMER_GROUP_ID = "my-group-id-tr1";  
private static final String INPUT_TOPIC = "output";
```

Define the consumer configuration:

```
props.put(ISOLATION_LEVEL_CONFIG, "read_committed");
```

The above line states that only committed message will be consumed by the consumer.

```
private static KafkaConsumer<String, String> createKafkaConsumer() {
    Properties props = new Properties();
    props.put(BOOTSTRAP_SERVERS_CONFIG, "localhost:8082");
    props.put(GROUP_ID_CONFIG, CONSUMER_GROUP_ID);
    props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
    "earliest");
    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;
}
```

The consumer that polls for the message.

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

    final int giveUp = 100;
    int noRecordsCount = 0;

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

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

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

```
        record.partition(), record.offset());
    });

    consumer.commitAsync();
}
consumer.close();
System.out.println("DONE");
}
```

Invoke the consume method from the main method.

```
public static void main(String[] args) throws InterruptedException
{
    // TODO Auto-generated method stub
    runConsumer();
}
```

Execute the program:

```
-----  
[groupId=my-group-id-tr2] Resetting offset for partition output-0 to offset 0.  
Consumer Record:(the, the-1, 0, 0)  
Consumer Record:(number, number-1, 0, 1)  
Consumer Record:(of, of-1, 0, 2)  
Consumer Record:(count, count-2, 0, 3)  
Consumer Record:(Let, Let-1, 0, 4)  
Consumer Record:(us, us-1, 0, 5)  
Consumer Record:(the, the-1, 0, 7)  
Consumer Record:(how, how-1, 0, 8)  
Consumer Record:(count, count-1, 0, 9)  
Consumer Record:(is, is-1, 0, 10)  
Consumer Record:(much, much-1, 0, 11)
```

As observed above, Only the committed message get consumed.

Next comment the following:

```
// props.put(ISOLATION_LEVEL_CONFIG, "read_committed");
```

Update the consumer group ID

```
private static final String CONSUMER_GROUP_ID = "my-group-id-tr3";
```

Excute the main program:

```
groupId=my-group-id-tr3] Resetting offset for partition output-0 to offset 0.  
Consumer Record:(the, the-1, 0, 0)  
Consumer Record:(number, number-1, 0, 1)  
Consumer Record:(of, of-1, 0, 2)  
Consumer Record:(count, count-2, 0, 3)  
Consumer Record:(Let, Let-1, 0, 4)  
Consumer Record:(us, us-1, 0, 5)  
Consumer Record:(the, the-1, 0, 7)  
Consumer Record:(how, how-1, 0, 8)  
Consumer Record:(count, count-1, 0, 9)  
Consumer Record:(is, is-1, 0, 10)  
Consumer Record:(much, much-1, 0, 11)  
Consumer Record:(works, works-1, 0, 13)  
Consumer Record:(transactions, transactions-1, 0, 14)
```

As observed, the uncommitted records get consumed too.

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

## 6. Schema registry – 90 Minutes

Create a java maven project or include the earlier project.

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.



Topics	stateful_processor_id-stock-transactions-changelog	...	1
	stocks	...	1
	stocks-out	...	1
	stocks-source	...	1
	transaction-summary	...	1
	transactions	...	1

Or using CLI:

```
#kafka-topics.sh --create --bootstrap-server kafka:9092 --partitions 1 --replication-factor 1 --topic transactions
```

## 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](#).

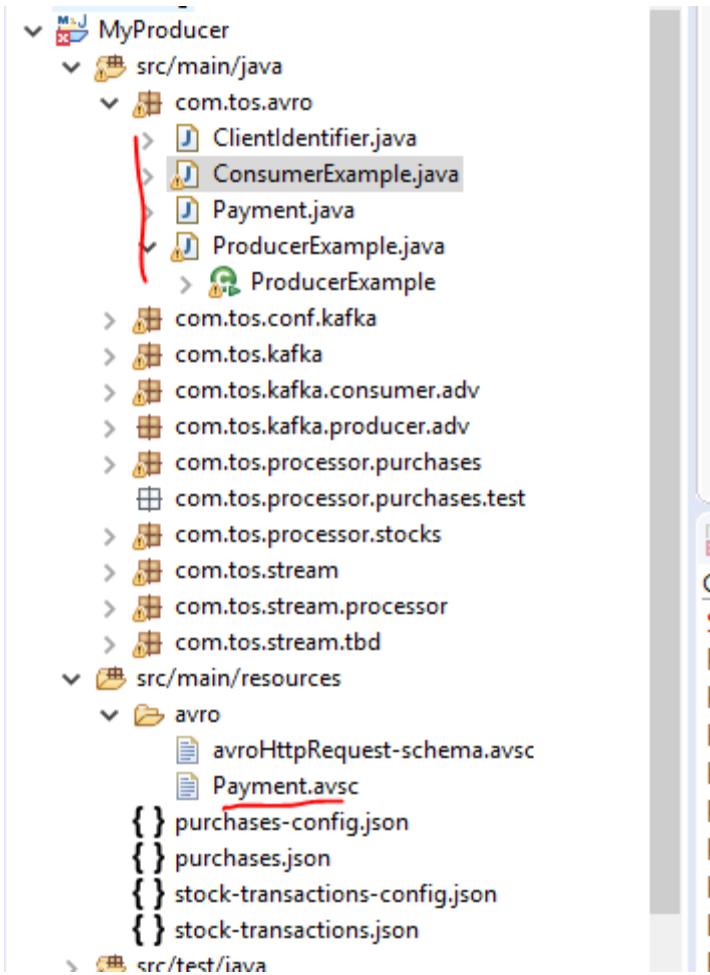
Copy the following content in src/main/resources folder with file Name:Payment.avsc

```
{"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*.

Project structure will be as shown after performing this lab.



## 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

## 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://localhost: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
```

*Execute Maven → Generate Sources.*

It will generate Payment.java file in the same package as the main class. Its specified in the pom.xml

*Pom.xml has the plugins for generating sources for avro. Ensure that avro schema file is in the below directores as specified below.*

```
|5@      <plugin>
|6        <groupId>org.apache.avro</groupId>
|7        <artifactId>avro-maven-plugin</artifactId>
|8        <version>${avro.version}</version>
|9@      <executions>
|10@        <execution>
|11          <phase>generate-sources</phase>
|12          <goals>
|13            <goal>schema</goal>
|14          </goals>
|15@        <configuration>
|16          <sourceDirectory>${project.basedir}/src/main/resources/</sourceDirectory>
|17@        <includes>
|18          <include>Payment.avsc</include>
|19        </includes>
|20        <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
|21      </configuration>
|22      </execution>
|23    </executions>
|24  </plugin>
|25</plugins>
```

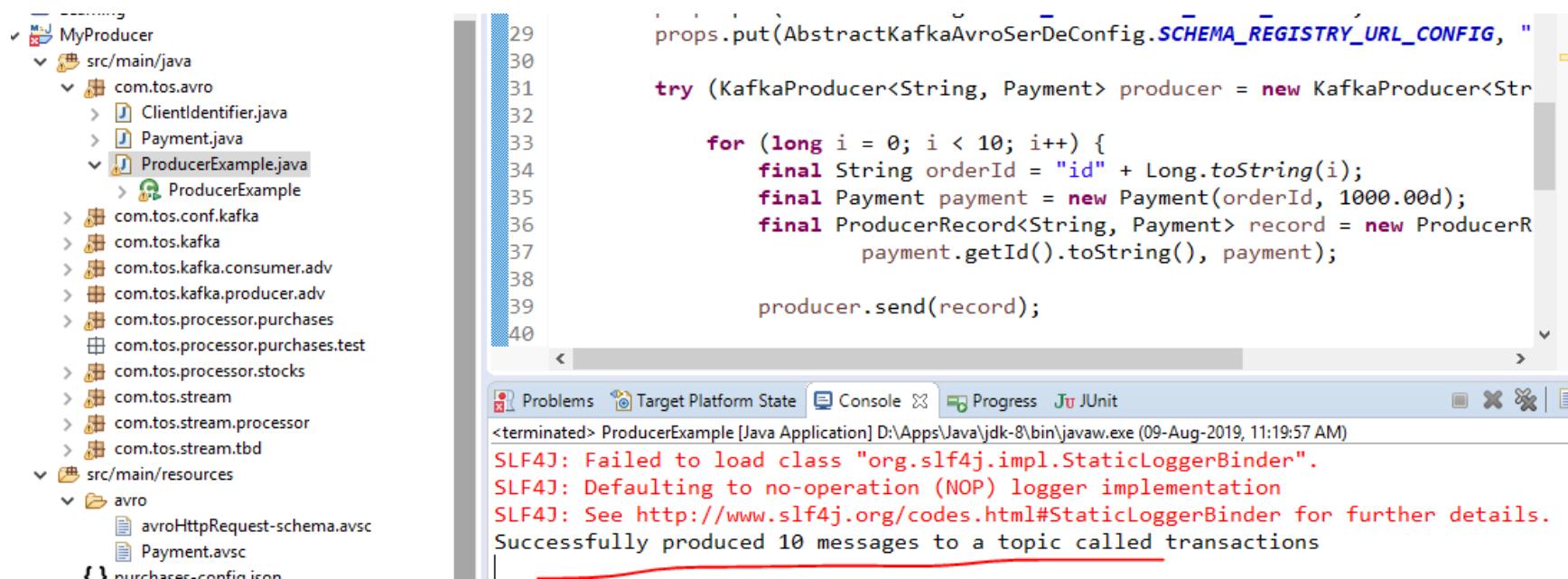
## Run the Producer

- 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.

- Run **ProducerExample**, which produces Avro-formatted messages to the **transactions** topic.



The screenshot shows the Eclipse IDE interface. On the left is the Project Explorer view, displaying the package structure of **MyProducer**. The **src/main/java** folder contains several Java files under the **com.tos.avro** package, including **ClientIdentifier.java**, **Payment.java**, and **ProducerExample.java**. Below these are other utility classes like **com.tos.conf.kafka**, **com.tos.kafka**, etc. The **src/main/resources** folder contains Avro schema files: **avroHttpRequest-schema.avsc** and **Payment.avsc**. A configuration file **purchases-config.json** is also listed.

The right side of the screen shows the code editor for **ProducerExample.java**. The code is as follows:

```

29     props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://localhost:8033");
30
31     try (KafkaProducer<String, Payment> producer = new KafkaProducer<String, Payment>(props)) {
32
33         for (long i = 0; i < 10; i++) {
34             final String orderId = "id" + Long.toString(i);
35             final Payment payment = new Payment(orderId, 1000.00d);
36             final ProducerRecord<String, Payment> record = new ProducerRecord<String, Payment>(topic, payment.getId().toString(), payment);
37
38             producer.send(record);
39
40         }
41     }

```

Below the code editor is the **Console** view, which displays the application's output:

```

<terminated> ProducerExample [Java Application] D:\Apps\Java\jdk-8\bin\javaw.exe (09-Aug-2019, 11:19:57 AM)
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.
Successfully produced 10 messages to a topic called transactions

```

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.

The screenshot shows the Confluent Platform UI for managing topics. The left sidebar has sections for MONITORING, MANAGEMENT, ALERTS, and DEVELOPMENT. Under MANAGEMENT, 'Topics' is selected and highlighted in purple. The main area shows 'MANAGEMENT > TOPICS > transactions'. Below this, tabs for STATUS, SCHEMA, INSPECT, and SETTINGS are present, with 'SCHEMA' being the active tab. Under 'SCHEMA', there are two tabs: 'Value' (selected) and 'Key'. Below these tabs are buttons for 'Edit schema', 'Version history', and 'Download'. A version number 'Version 1' is also visible. The schema itself is displayed as a JSON object with line numbers from 1 to 14:

```
1  {
2    "type": "record",
3    "name": "Payment",
4    "namespace": "com.tos.avro",
5    "fields": [
6      {
7        "name": "id",
8        "type": "string"
9      },
10     {
11       "name": "amount",
12       "type": "double"
13     }
14   ]
```

Or Using CLI:

```
# curl -X GET http://localhost:8081/subjects
# curl -X GET http://localhost:8081/subjects/transactions-value/versions/latest
```

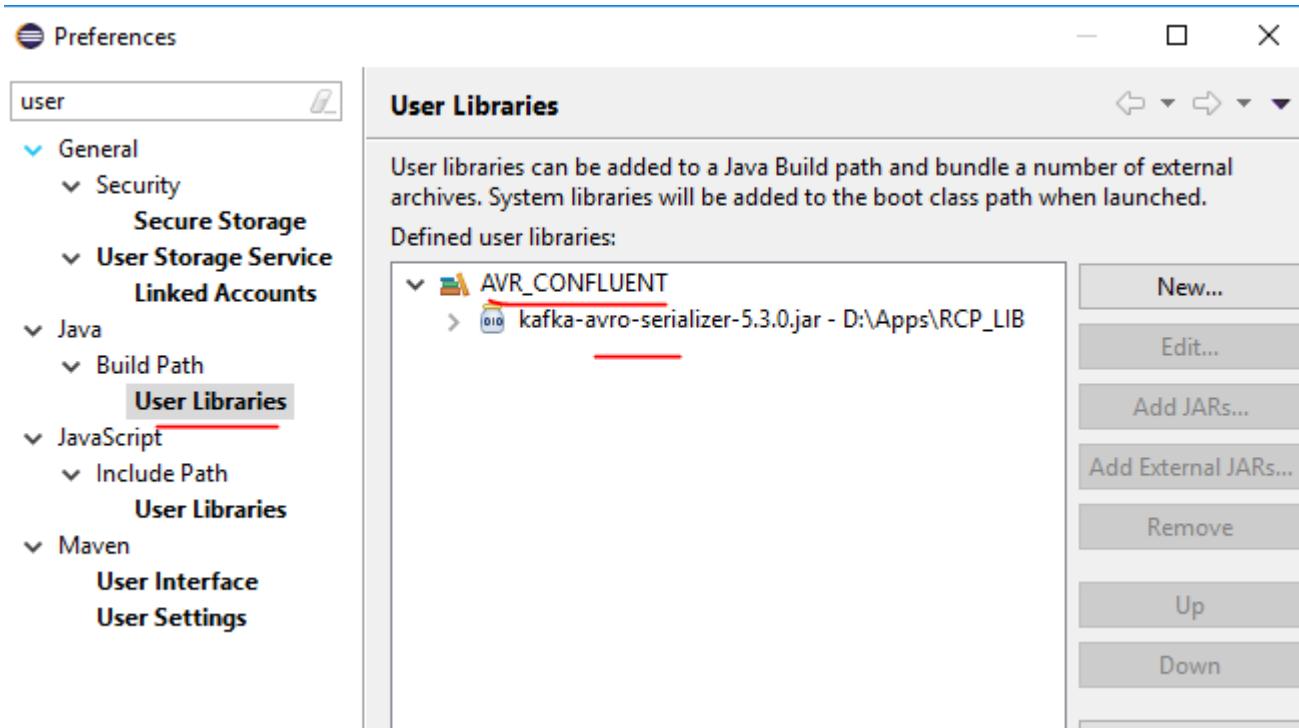
```
[root@kafka0 kafka-logs]# curl -X GET http://localhost:8081/subjects
["transactions-value"]
[root@kafka0 kafka-logs]# curl -X GET http://localhost:8081/subjects/transactions-value/versions/latest
{"subject": "transactions-value", "version": 1, "id": 1, "schema": "{\"type\": \"record\", \"name\": \"Payment\", \"namespace\": \"com.tos.registry\", \"fields\": [{\"name\": \"id\", \"type\": \"string\"}, {"name\": \"amount\", \"type\": \"double\"}]}"}[root@kafka0 kafka-logs]#
```

Consumer CLI:

```
# kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic transactions --from-beginning
```

```
processed a total of 0 messages
[root@kafka0 kafka-logs]# kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic transactions --from-beginning
id0@0
id1@0
id2@0
id3@0
id4@0
id5@0
id6@0
id7@0
id8@0
id9@0
```

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 java.time.Duration;
import java.util.Collections;
import java.util.Properties;

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.common.serialization.StringDeserializer;
```

```
import io.confluent.kafka.serializers.AbstractKafkaAvroSerDeConfig;
import io.confluent.kafka.serializers.KafkaAvroDeserializer;
import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;
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,
        "localhost:8082");
        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://localhost: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(Duration.ofMillis(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. run [ConsumerExample](#) (assuming you already ran the [ProducerExample](#) above).

You should see:

The screenshot shows an IDE interface with a file tree on the left and a code editor on the right. The file tree for 'MyProducer' includes packages like com.tos.avro, com.tos.conf.kafka, com.tos.kafka.consumer.adv, com.tos.kafka.producer.adv, com.tos.processor.purchases, com.tos.processor.stocks, com.tos.stream, com.tos.stream.processor, and com.tos.stream.tbd. The code editor displays Java code for a 'ConsumerExample' class, specifically the main method which configures a consumer with bootstrap servers at '192.168.139.132:9092', group ID 'test-payments', and auto-commit enabled. The output window below shows the consumer reading 10 records from a topic, each with an ID and an amount of 1000.0.

```
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");  
}  
  
key = id0, value = {"id": "id0", "amount": 1000.0}  
key = id1, value = {"id": "id1", "amount": 1000.0}  
key = id2, value = {"id": "id2", "amount": 1000.0}  
key = id3, value = {"id": "id3", "amount": 1000.0}  
key = id4, value = {"id": "id4", "amount": 1000.0}  
key = id5, value = {"id": "id5", "amount": 1000.0}  
key = id6, value = {"id": "id6", "amount": 1000.0}  
key = id7, value = {"id": "id7", "amount": 1000.0}  
key = id8, value = {"id": "id8", "amount": 1000.0}  
key = id9, value = {"id": "id9", "amount": 1000.0}
```

2. Hit **Ctrl-C** to stop.

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

## 7. DSL - Transform a stream of events – 60 Minutes

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.

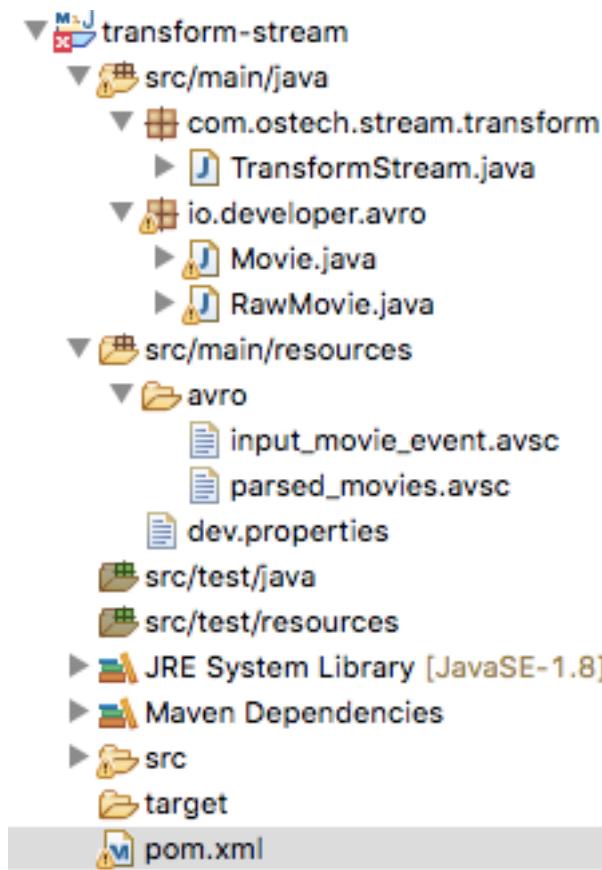
Input : {"id":294,"title":"Die Hard::1988","genre":"action"}

Output: {"id":294,"title":"Die Hard","release\_year":1988,"genre":"action"}

Create a Maven Project:

com.ostechnotes:transform-stream

At the end you should have the project structure as shown below:



Update the pom.xml with the following entry.

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": "genre", "type": "string"},  
    {"name": "year", "type": "int"},  
    {"name": "rating", "type": "float"},  
    {"name": "actors", "type": "array", "items": "string"}  
  ]  
}
```

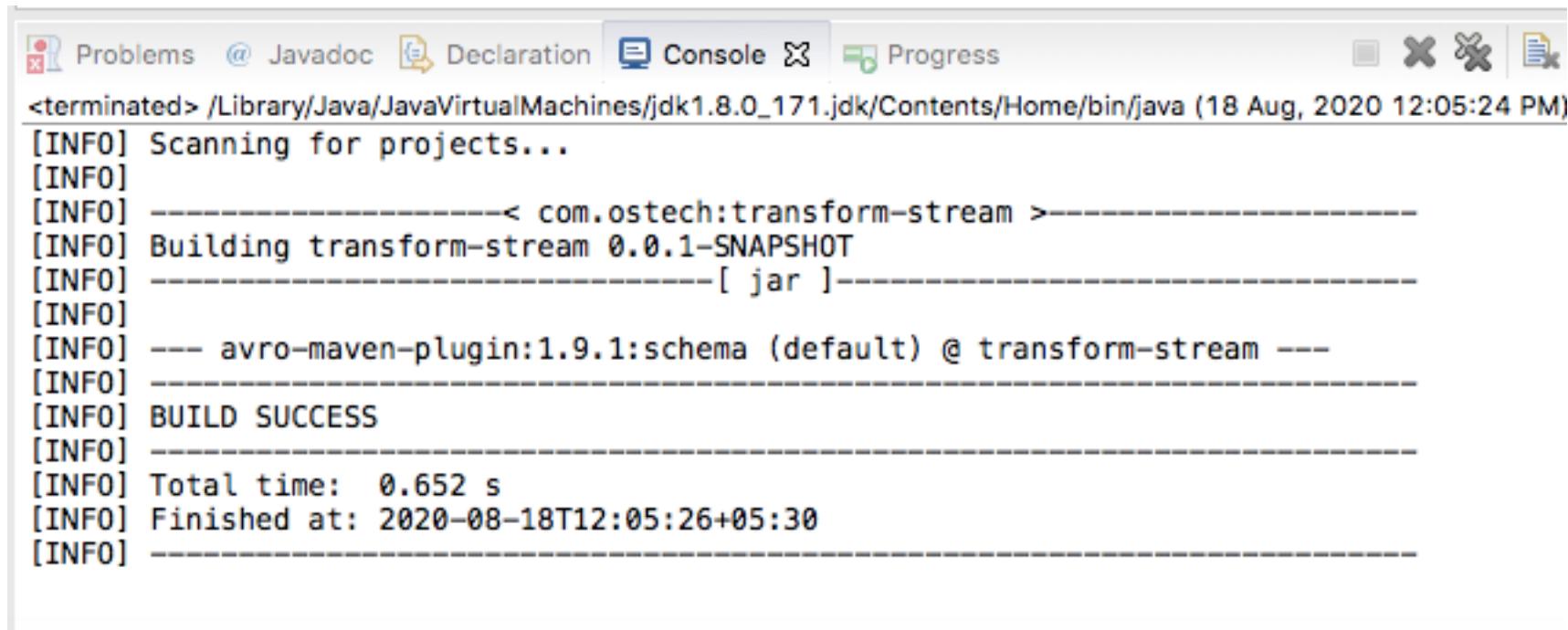
```
"fields": [
    {"name": "id", "type": "long"},
    {"name": "title", "type": "string"},
    {"name": "release_year", "type": "int"},
    {"name": "genre", "type": "string"}
]
```

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:

Include the above Avro Schema in the Avro-maven-plugin source directory to generate the source code.

```
97      <artifactId>avro-maven-plugin</artifactId>
98      <version>${avro.version}</version>
99      <executions>
100     <execution>
101       <phase>generate-sources</phase>
102       <goals>
103         <goal>schema</goal>
104       </goals>
105       <configuration>
106         <sourceDirectory>${project.basedir}/src/main/resources/</sourceDirectory>
107         <sourceDirectory>${project.basedir}/src/main/resources/avro</sourceDirectory>
108       <includes>
109         <include>input_movie_event.avsc</include>
110         <include>parsed_movies.avsc</include>
111         <include>Payment.avsc</include>
112       </includes>
113       <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
114     </configuration>
115   </execution>
```

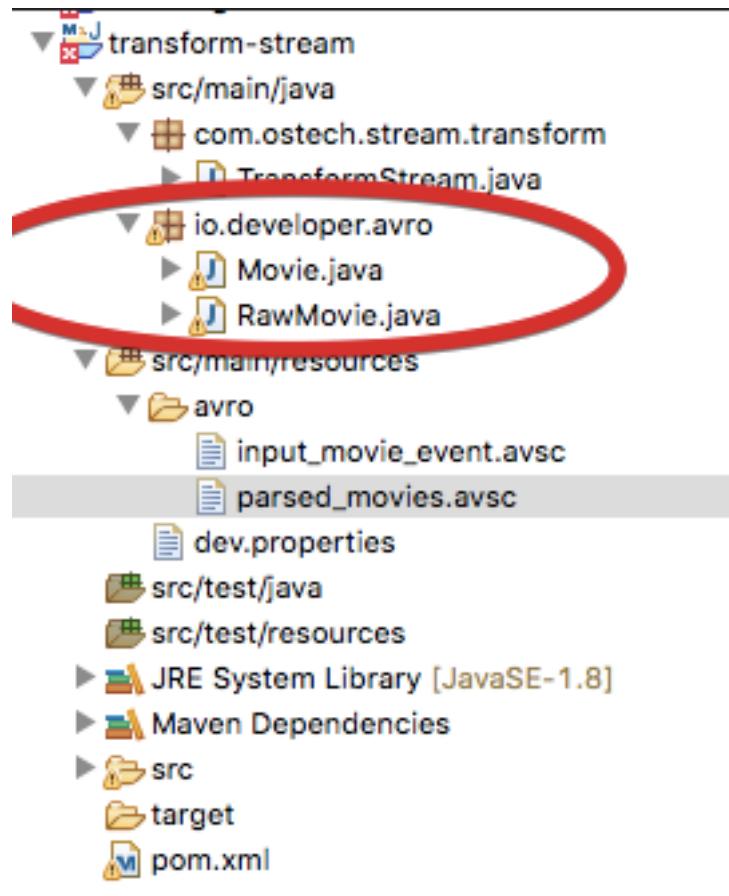
Maven → Generated-Sources



The screenshot shows a software interface with a toolbar at the top containing icons for Problems, Javadoc, Declaration, Console, and Progress. The 'Console' tab is selected. The main area displays the following text:

```
<terminated> /Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/java (18 Aug, 2020 12:05:24 PM)
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.ostechnix:transform-stream >-----
[INFO] Building transform-stream 0.0.1-SNAPSHOT
[INFO]           [ jar ]-
[INFO]
[INFO] --- avro-maven-plugin:1.9.1:schema (default) @ transform-stream ---
[INFO]
[INFO] BUILD SUCCESS
[INFO]
[INFO] Total time:  0.652 s
[INFO] Finished at: 2020-08-18T12:05:26+05:30
[INFO] -----
```

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



Create the Kafka Streams topology

Then create the following file **TransformStream.java** inside package  
com.ostechnote.stream.transform

```
package com.ostechnote.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,
Serdess.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_CO
NFIG,
        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()`:

Ensure the dev.properties is refer with the correct path.

Start the kafka and the registry server

Compile and run the Kafka Streams program

The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays the project structure under 'LearningKafka'. The 'transform-stream' project contains a 'src/main/java' folder with a 'com.ostech.stream.transform' package containing a 'TransformStream.java' file, which is currently selected and shown in the central code editor. The code editor shows Java code for transforming raw movie data into a Movie object. Below the code editor are tabs for 'Problems', 'Javadoc', 'Declaration', 'Console', and 'Progress'. The 'Console' tab shows log output related to SLF4J loading issues:

```
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.
```

*Produce events to the input topic*

In a new terminal, run:

Copy the avsc file in a specific folder and Execute the following.

```
#kafka-console-producer --topic raw-movies --broker-list kafka:9092 --property  
value.schema="$(
```

```
[  
./kafka-console-producer --topic raw-movies --broker-list localhost:9092 --property  
value.schema="$(]
```

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.ContextHandler:825)  
[2020-09-28 12:40:20,805] INFO Started o.e.j.s.ServletContextHandler@27aae97b{/ws,null,AVAILABLE} (org.eclipse.jetty.server.handler.ContextHandler: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.AbstractConnector: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.

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

```
# kafka-console-consumer --topic movies --bootstrap-server kafka0:9092 --from-beginning
```

```
[root@0945477d457b bin]# sh kafka-console-consumer --topic movies --bootstrap-server localhost:9092 --from-beginning
{"id":294,"title":"Die Hard","release_year":1988,"genre":"action"}
{"id":354,"title":"Tree of Life","release_year":2011,"genre":"drama"}
{"id":782,"title":"A Walk in the Clouds","release_year":1995,"genre":"romance"}
{"id":128,"title":"The Big Lebowski","release_year":1998,"genre":"comedy"}
{"id":294,"title":"Die Hard","release_year":1988,"genre":"action"}
{"id":354,"title":"Tree of Life","release_year":2011,"genre":"drama"}
{"id":782,"title":"A Walk in the Clouds","release_year":1995,"genre":"romance"}
{"id":128,"title":"The Big Lebowski","release_year":1998,"genre":"comedy"}
```

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

```
# kafka-console-consumer --topic raw-movies --bootstrap-server kafka0:9092 --from-beginning
```

```
(base) Henrys-MacBook-Air:~ henrypotsangbam$ kafka-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"}
```

### Task:

Add: peek method to get a print after transformation:

```
44     KStream<String, RawMovie> rawMovies = builder.stream(inputTopic);
45     KStream<Long, Movie> movies = rawMovies.map((key, rawMovie) ->
46         new KeyValue<Long, Movie>(rawMovie.getId(), convertRawMovie(rawMovie)))
47         .peek((k,v) -> System.out.println("Key: " + k + " , Movie : " + v.getTitle()));
48
49 //    movies.to("movies", Produced.with(Serdes.Long(), movieAvroSerde(envProps)));
50
```

The screenshot shows an IDE's terminal or console window. At the top, there is some Java code. Below it, the console output shows several SLF4J log messages indicating that the static logger binder failed to load, defaulting to a no-operation (NOP) implementation. At the bottom, the output shows the result of the peek operation, where the key '128' and its corresponding movie title 'The Big Lebowski' are printed.

```
Console ☐
TransformStream (1) [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (03-Mar-2022, 10:35:23 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.
Key: 128 , Movie : The Big Lebowski
```

Your output should be as shown above, when you enter the following in the producer console.

```
{"id": 128, "title": "The Big Lebowski::1998", "genre": "comedy"}
```

```
[root@kafka0 code]# kafka-console-producer --topic raw-movies --broker-list kafka0:9092 --property value.schema="$(  
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"}  
{"id": 128, "title": "The Big Lebowski::1998", "genre": "comedy"}
```

---

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

## 8. Stream Using – FMV Stateless – 45 Minutes

In this lab, following actions will be performed.

- Kstream creation from the Topic
- Perform transformation using flatMapValues 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.

It will be calculate the count for the complete execution.

Create a class file with the following specification.

Package: com.osteck.ktable

Class: WordCountLambdaExample.java

Update the following with the IP address of your broker.

```
final String bootstrapServers = "kafka:8082";
```

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

Import the following packages and classes:

```
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;
```

Define the Input and output topic.

```
public class WordCountLambdaExample {
    final static String inputTopic = "streams-plaintext-input";
    final static String outputTopic = "streams-wordcount-output";

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

Define the config properties: The state of the aggregation will be stored in directory specified by StreamsConfig.**STATE\_DIR\_CONFIG**.

```
private static Properties config() {
    Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-
temperature");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:8082");
    props.put(StreamsConfig.STATE_DIR_CONFIG, "//tmp//WC");
    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);
    return props;
}
```

Transformation logic define below:

- a) Read the stream from input topic.
- b) Convert each word of a sentences into array of words like (“kafka”, “again”).
- c) Group the list by word i.e “kafka”
- d) Finally count the occurrence of the word using count aggregation.

```
public static void countWordByUsingFValues() {  
  
    Properties props = config();  
    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);
```

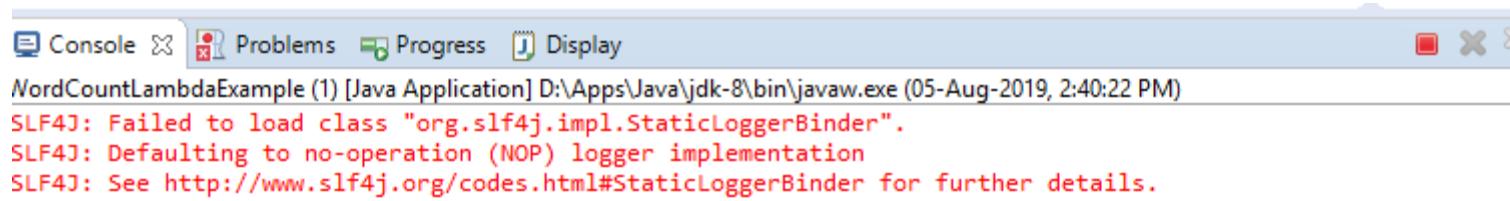
```
}
```

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 \
--bootstrap-server kafka0:9092 --partitions 1 --replication-factor 1
# sh kafka-topics.sh --create --topic streams-wordcount-output \
--bootstrap-server kafka0:9092 --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 kafkao:9092 --topic streams-plaintext-input
```

```
(base) [root@tos ~]# kafka-console-producer --broker-list tos.hp.com:9092 --topi
c streams-plaintext-input
>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 kafkao: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  
mmun.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 -----

**9. DSL: stateful transformations – reduce – 45 Minutes**

Demonstrate the following APIs:

- Ktable
- Filter
- selectKey
- Aggregation: GroupBy & 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 accordingly.

```
#kafka-topics.sh --create --topic numbers-topic --bootstrap-server kafkao:9092 --partitions 1 --replication-factor 1
```

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

```
#kafka-topics.sh --create --topic sum-of-odd-numbers-topic --bootstrap-server kafkao:9092 --partitions 1 --replication-factor 1
```

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

On Confluent Kafka Only:

```
zookeeper is [UP]
(base) [root@tos apps]# kafka-topics --create --topic numbers-topic --zookeeper tos.hp.com:2181 --partitions 1 --replication-factor 1
Created topic numbers-topic.
(base) [root@tos apps]# kafka-topics --create --topic sum-of-odd-numbers-topic --zookeeper tos.hp.com:2181 --partitions 1 --replication-factor 1
Created topic sum-of-odd-numbers-topic.
```

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] :
"kafka0:8082";
    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);
}

// 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] :
"kafka0:8082";
    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);

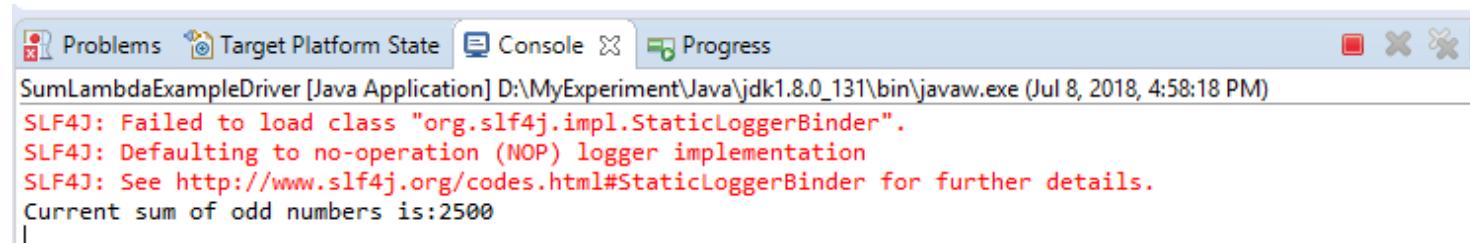
producer.flush();
}

// 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.

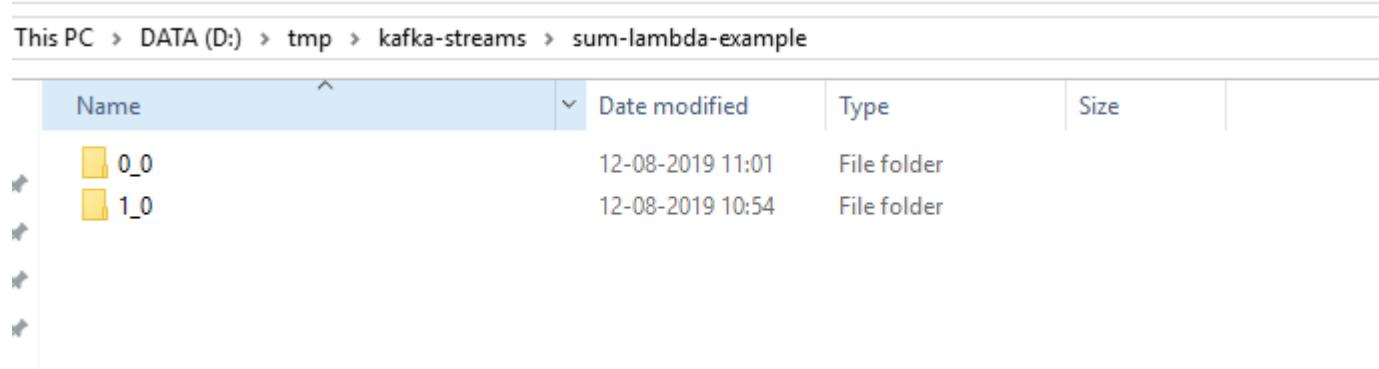
SumLambdaExampleDriver → It will generate message and sent messages to the i/p topic and consume from the o/p topic and display to the console.



The screenshot shows an IDE's Console tab with the following log output:

```
SumLambdaExampleDriver [Java Application] D:\MyExperiment\Java\jdk1.8.0_131\bin\javaw.exe (Jul 8, 2018, 4:58:18 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.
Current sum of odd numbers is:2500
```

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



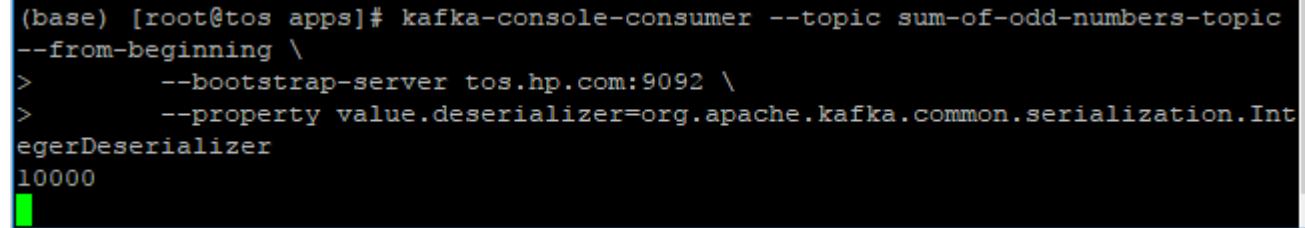
This PC > DATA (D:) > tmp > kafka-streams > sum-lambda-example

Name	Date modified	Type	Size
0_0	12-08-2019 11:01	File folder	
1_0	12-08-2019 10:54	File folder	

This is where the message is stored 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 localhost:8082 \
--property
value.deserializer=org.apache.kafka.common.serialization.IntegerDeserializer
```



A terminal window showing the execution of a Kafka consumer command. The command is:

```
(base) [root@tos apps]# 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
10000
```

The output shows the consumer reading from the topic, with the number 10000 being printed at the end of the line.

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

## 10. Running your first Kafka Streams Application: WordCount – 60 minutes

This lab will demonstrate how to run a streaming application coded using stream api library.

Create a java class – WordCountApplication.java

Import the following packages and classes.

```
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;

import java.io.FileInputStream;
import java.io.IOException;
```

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

It configures the parameter required to connect to the kafka broker.

```
public static final String INPUT_TOPIC = "streams-plaintext-input";
public static final String OUTPUT_TOPIC = "streams-wordcount-
output";

static Properties getStreamsConfig(final String[] args) throws
IOException {
    final Properties props = new Properties();
    if (args != null && args.length > 0) {
        try (final FileInputStream fis = new
FileInputStream(args[0])) {
            props.load(fis);
        }
        if (args.length > 1) {
            System.out.println("Warning: Some command line
arguments were ignored. This demo only accepts an optional
```

```
configuration file.");
        }
    }
    props.putIfAbsent(StreamsConfig.APPLICATION_ID_CONFIG,
"streams-wordcount");
    props.putIfAbsent(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
"localhost:8082");

props.putIfAbsent(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG,
0);

props.putIfAbsent(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

props.putIfAbsent(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());

props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest");
    return props;
}
```

Next code creates an instance of stream and perform the transformation.

```
static void createWordCountStream(final StreamsBuilder builder) {  
    final KStream<String, String> source =  
    builder.stream(INPUT_TOPIC);  
  
    final KTable<String, Long> counts = source.peek( (k,v) ->  
    System.out.println("Value" + v))  
        .flatMapValues(value ->  
    Arrays.asList(value.toLowerCase(Locale.getDefault()).split("\\W+"))  
        .groupBy((key, value) -> value)  
        .count();  
  
    // need to override value serde to Long type  
    counts.toStream().to(OUTPUT_TOPIC,  
Produced.with(Serdes.String(), Serdes.Long()));  
}
```

It implements the WordCount algorithm, which computes a word occurrence histogram from the input text. However, unlike other WordCount examples you might have seen before that operate on bounded data, the WordCount demo application behaves slightly differently because it is designed to operate on an **infinite, unbounded stream** of data. Similar to the bounded variant, it is a stateful algorithm that tracks and updates the counts of words. However, since it must assume potentially unbounded input data, it will

periodically output its current state and results while continuing to process more data because it cannot know when it has processed "all" the input data.

Update the following code in the main method().

```
final Properties props = getStreamsConfig(args);

final StreamsBuilder builder = new StreamsBuilder();
createWordCountStream(builder);
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-
wordcount-shutdown-hook") {
    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});

try {
```

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

At the end, you should have the following code structure:

```
1 public class WordCountDemo {  
2  
3     public static final String INPUT_TOPIC = "streams-plaintext-input";  
4     public static final String OUTPUT_TOPIC = "streams-wordcount-output";  
5  
6     static Properties getStreamsConfig(final String[] args) throws IOException {  
7         final Properties props = new Properties();  
8         if (args != null && args.length > 0) {  
9             try (final FileInputStream fis = new FileInputStream(args[0])) {  
10                 props.load(fis);  
11             }  
12             if (args.length > 1) {  
13                 System.out.println("Warning: Some command line arguments were ignored. This demo only accep  
14             }  
15         }  
16         props.putIfAbsent(StreamsConfig.APPLICATION_ID_CONFIG, "streams-wordcount");  
17         props.putIfAbsent(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:8082");  
18         props.putIfAbsent(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);  
19         props.putIfAbsent(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
20                           Serdes.String().getClass().getName());  
21         props.putIfAbsent(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
22                           Serdes.String().getClass().getName());  
23  
24         // setting offset reset to earliest so that we can re-run the demo code with the same pre-loaded da  
25         // Note: To re-run the demo, you need to use the offset reset tool:  
26         // https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool  
27         props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");  
28     }  
29     return props;  
30 }
```

```
static void createWordCountStream(final StreamsBuilder builder) {
    final KStream<String, String> source = builder.stream(INPUT_TOPIC);

    final KTable<String, Long> counts = source.peek( (k,v) -> System.out.println("Value" + v))
        .flatMapValues(value -> Arrays.asList(value.toLowerCase(Locale.getDefault()).split("\\W+")))
        .groupBy((key, value) -> value)
        .count();

    // need to override value serde to Long type
    counts.toStream().to(OUTPUT_TOPIC, Produced.with(Serdes.String(), Serdes.Long()));
}

public static void main(final String[] args) throws IOException {
    final Properties props = getStreamsConfig(args);

    final StreamsBuilder builder = new StreamsBuilder();
    createWordCountStream(builder);
    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-wordcount-shutdown-hook") {
        @Override
        public void run() {
            streams.close();
            latch.countDown();
        }
    });
}
```

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

|

Prepare input topic and start Kafka producer

Next, we create the input topic named streams-plaintext-input and the output topic named streams-wordcount-output:

```
# /opt/kafka/bin/kafka-topics.sh --create \
--bootstrap-server localhost:9092 \
--replication-factor 1 \
--partitions 1 \
--topic streams-plaintext-input
```

Note: we create the output topic with compaction enabled because the output stream is a changelog stream.

```
# /opt/kafka/bin/kafka-topics.sh --create \
--bootstrap-server localhost:9092 \
--replication-factor 1 \
--partitions 1 \
--topic streams-wordcount-output \
--config cleanup.policy=compact
```

The created topic can be described with the same kafka-topics tool:

```
#/opt/kafka/bin/kafka-topics.sh --bootstrap-server localhost:9092 --describe
```

```
Topic: my-failsafe-topic      Partition: 12    Leader: 0      Replicas: 2,0,1 Isr: 0
Topic: streams-plaintext-input TopicId: hG4VK81QRROfD7I4Jp9TdQ PartitionCount: 1      ReplicationFactor: 1      Configs: segment.bytes=1073741824
    Topic: streams-plaintext-input Partition: 0    Leader: 0      Replicas: 0      Isr: 0
Topic: test      TopicId: Kp4hHo8fTR-YAlyg-MgPuQ PartitionCount: 1      ReplicationFactor: 1      Configs: segment.bytes=1073741824
    Topic: test      Partition: 0    Leader: 0      Replicas: 0      Isr: 0
Topic: my-kafka-topic  TopicId: sgEalbYqSkq1wC-aTNtNpQ PartitionCount: 13     ReplicationFactor: 3      Configs: segment.bytes=1073741824
    Topic: my-kafka-topic  Partition: 0    Leader: 0      Replicas: 0,1,2 Isr: 0
    Topic: my-kafka-topic  Partition: 1    Leader: 0      Replicas: 1,2,0 Isr: 0
```

Start the Wordcount Application.

The demo application will read from the input topic **streams-plaintext-input**, perform the computations of the WordCount algorithm on each of the read messages, and continuously write its current results to the output topic **streams-wordcount-output**. Hence there won't be any STDOUT output except log entries as the results are written back into in Kafka.

Now we can start the console producer in a separate terminal to write some input data to this topic:

```
#/opt/kafka/bin/kafka-console-producer.sh --bootstrap-server  
localhost:9092 --topic streams-plaintext-input
```

and inspect the output of the WordCount demo application by reading from its output topic with the console consumer in a separate terminal:

```
> /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server  
localhost:9092 \  
  --topic streams-wordcount-output \  
  --from-beginning \  
  --formatter kafka.tools.DefaultMessageFormatter \  
  --property print.key=true \  
  --property print.value=true \  
  --property  
key.deserializer=org.apache.kafka.common.serialization.StringDeserializer  
\  
  --property  
value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

## Process some data

Now let's write some message with the console producer into the input topic **streams-plaintext-input** by entering a single line of text and then hit <RETURN>. This will send a new message to the input topic, where the message key is null and the message value is the

string encoded text line that you just entered (in practice, input data for applications will typically be streaming continuously into Kafka, rather than being manually entered):

```
all streams lead to kafka

[root@kafka0 ~]#
[root@kafka0 ~]# /opt/kafka/bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic streams-plaintext-input
>all streams lead to kafka
>
```

This message will be processed by the Wordcount application and the following output data will be written to the **streams-wordcount-output** topic and printed by the console consumer:

```
[root@kafka0 /]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
>   --topic streams-wordcount-output \
>   --from-beginning \
>   --formatter kafka.tools.DefaultMessageFormatter \
>   --property print.key=true \
>   --property print.value=true \
>   --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
>   --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer

all      1
streams  1
lead     1
to       1
kafka    1
|
```

Here, the first column is the Kafka message key in java.lang.String format and represents a word that is being counted, and the second column is the message value in java.lang.Longformat, representing the word's latest count.

Now let's continue writing one more message with the console producer into the input topic **streams-plaintext-input**. Enter the text line "hello kafka streams" and hit <RETURN>. Your terminal should look as follows:

```
[root@kafka0 /]# /opt/kafka/bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic streams-plaintext-input
>all streams lead to kafka
>hello kafka streams
>|
```

In your other terminal in which the console consumer is running, you will observe that the WordCount application wrote new output data:

```
[ Name: (null) # /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
> Profile: (null) streams-wordcount-output \
> Command: None ginning \
>   --formatter kafka.tools.DefaultMessageFormatter \
>   --property print.key=true \
>   --property print.value=true \
>   --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
>   --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer

all      1
streams  1
lead     1
to       1
kafka    1
hello    1
kafka    2
streams  2
```

Here the last printed lines kafka 2 and streams 2 indicate updates to the keys kafka and streams whose counts have been incremented from 1 to 2. Whenever you write further input messages to the input topic, you will observe new messages being added to the streams-wordcount-output topic, representing the most recent word counts as computed by the WordCount application. Let's enter one final input text line "join kafka

training" and hit <RETURN> in the console producer to the input topic streams-plaintext-input before we wrap up this quickstart:

```
[root@kafka0 /]# /opt/kafka/bin/kafka-console-producer.sh --bootstrap-server localhost:9092 --topic streams-plaintext-input
>all streams lead to kafka
>hello kafka streams
>join kafka training
>
```

The streams-wordcount-output topic will subsequently show the corresponding updated word counts (see last three lines):

```
[root@kafka0 /]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 \
>   --topic streams-wordcount-output \
>   --from-beginning \
>   --formatter kafka.tools.DefaultMessageFormatter \
>   --property print.key=true \
>   --property print.value=true \
>   --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
>   --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer

all      1
streams  1
lead     1
to       1
kafka    1
hello    1
kafka    2
streams  2
join     1
kafka    3
training      1
```

As one can see, outputs of the Wordcount application is actually a continuous stream of updates, where each output record (i.e. each line in the original output above) is an updated count of a single word, aka record key such as "kafka". For multiple records with the same key, each later record is an update of the previous one.

You can now stop the console consumer, the console producer, the Wordcount application, the Kafka broker and the ZooKeeper server in order via **Ctrl-C**

<https://kafka.apache.org/31/documentationstreams/quickstart>

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

## 11. Stream – DSL and Windows – 45 Minutes

The following features will be demonstrated:

- Window - Tumbling
- 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.ostechnix.windows.

Type the following logic in the class file.

```
package com.ostechnwindows;

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:8082");
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");
// Define the Tumbling Windows:
TimeWindows tumblingWindow =
TimeWindows.ofSizeWithNoGrace(Duration.ofSeconds(TEMPERATURE_WI
```

```
NDOW_SIZE));
    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(tumblingWindow)
        .reduce((value1, value2) -> {
            if (Integer.parseInt(String.valueOf(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:

```
#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:

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

```
(base) [root@tos logs]# kafka-topics --create --bootstrap-server tos.hp.com:9092  
--replication-factor 1 --partitions 1 --topic iot-temperature  
(base) [root@tos logs]# kafka-topics --create --bootstrap-server tos.hp.com:9092  
--replication-factor 1 --partitions 1 --topic iot-temperature-max  
(base) [root@tos logs]#
```

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

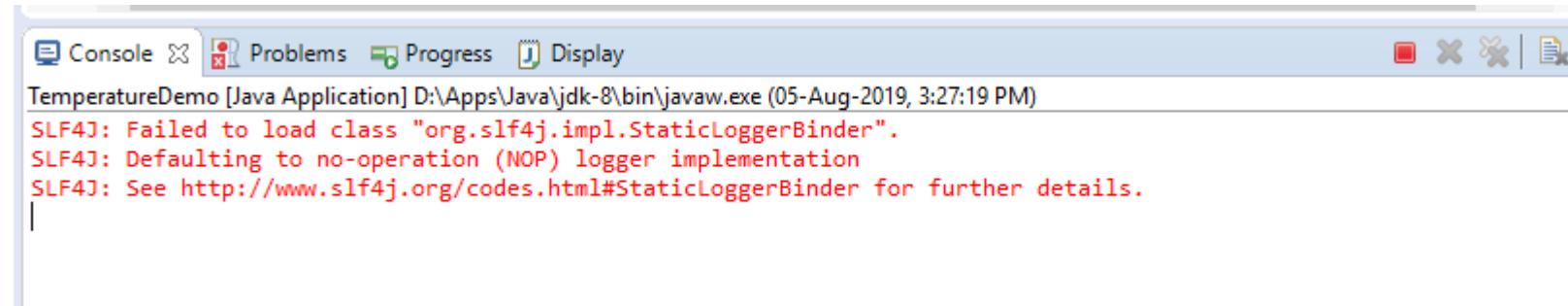
```
#kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic iot-temperature-  
max --from-beginning
```

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 :

```
#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 --topic 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:9092 --topic iot-temperature-max --from-beginning
22
35
```

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

**12. DSL - join a stream and a table together – 90 Minutes**

The following features will be demonstrated:

- Joining two streams.
- The [ValueJoiner](#) interface in the Streams API to combine two stream.

Scenario:

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.

Input stream : Movies and Ratings

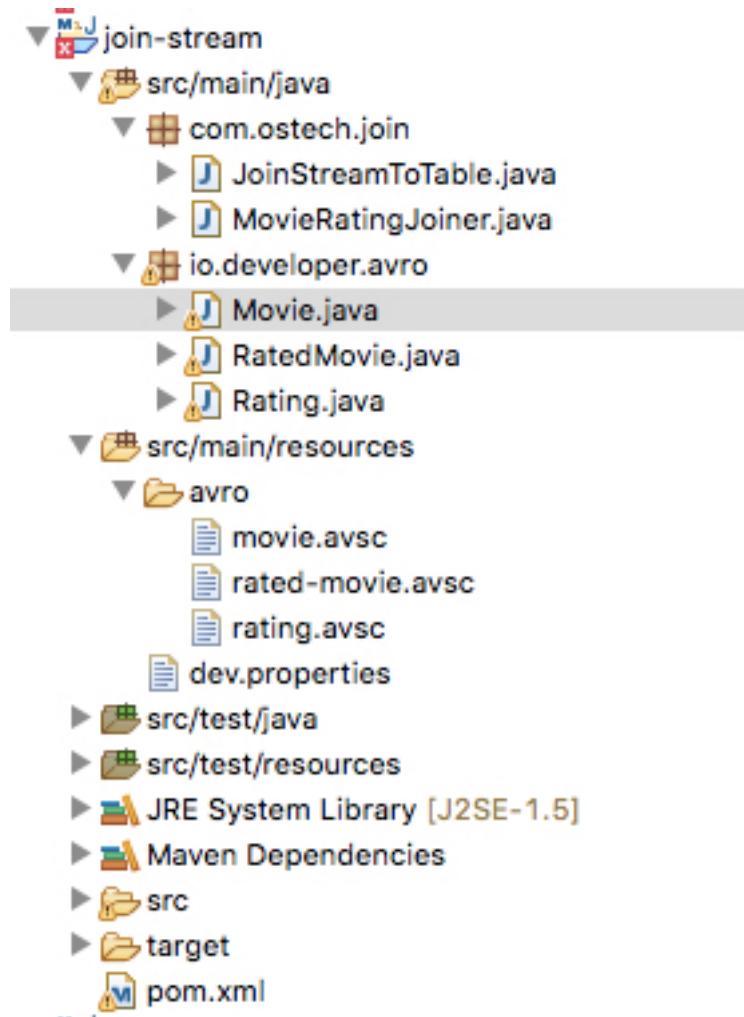
Join stream : Rating against each movie.

Initialize a maven project

com.ostechnix:join-stream

Update the pom.xml.

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



Create a development file at src/main/resources/devj.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.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:

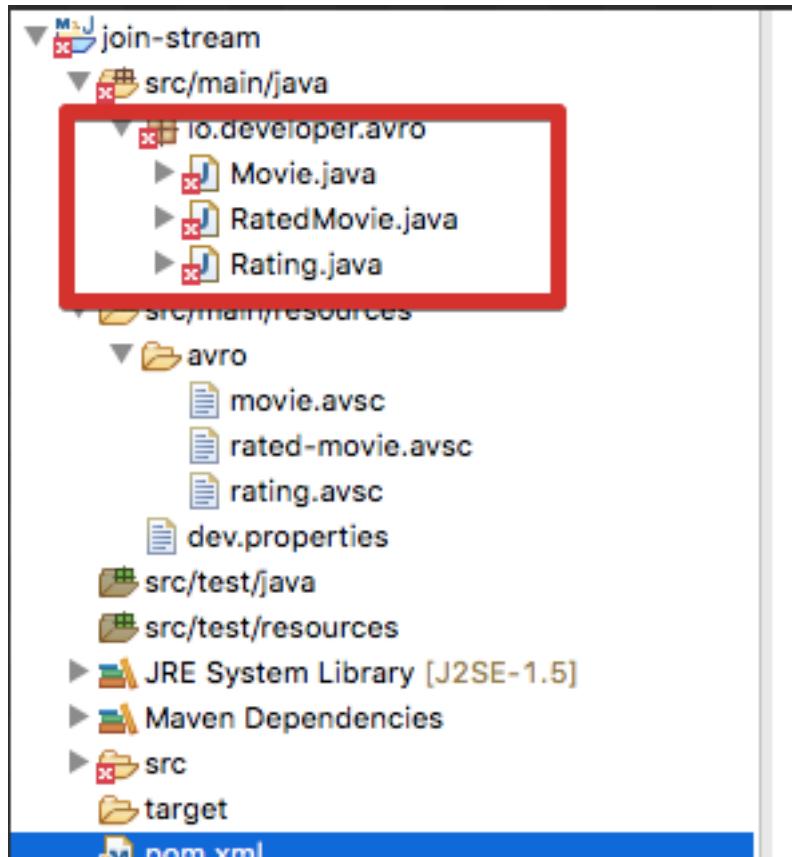
```
{  
    "namespace": "io.developer.avro",  
    "type": "record",  
    "name": "RatedMovie",  
    "fields": [  
        {"name": "id", "type": "long"},  
        {"name": "title", "type": "string"},  
        {"name": "release_year", "type": "int"},  
        {"name": "rating", "type": "double"}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

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:

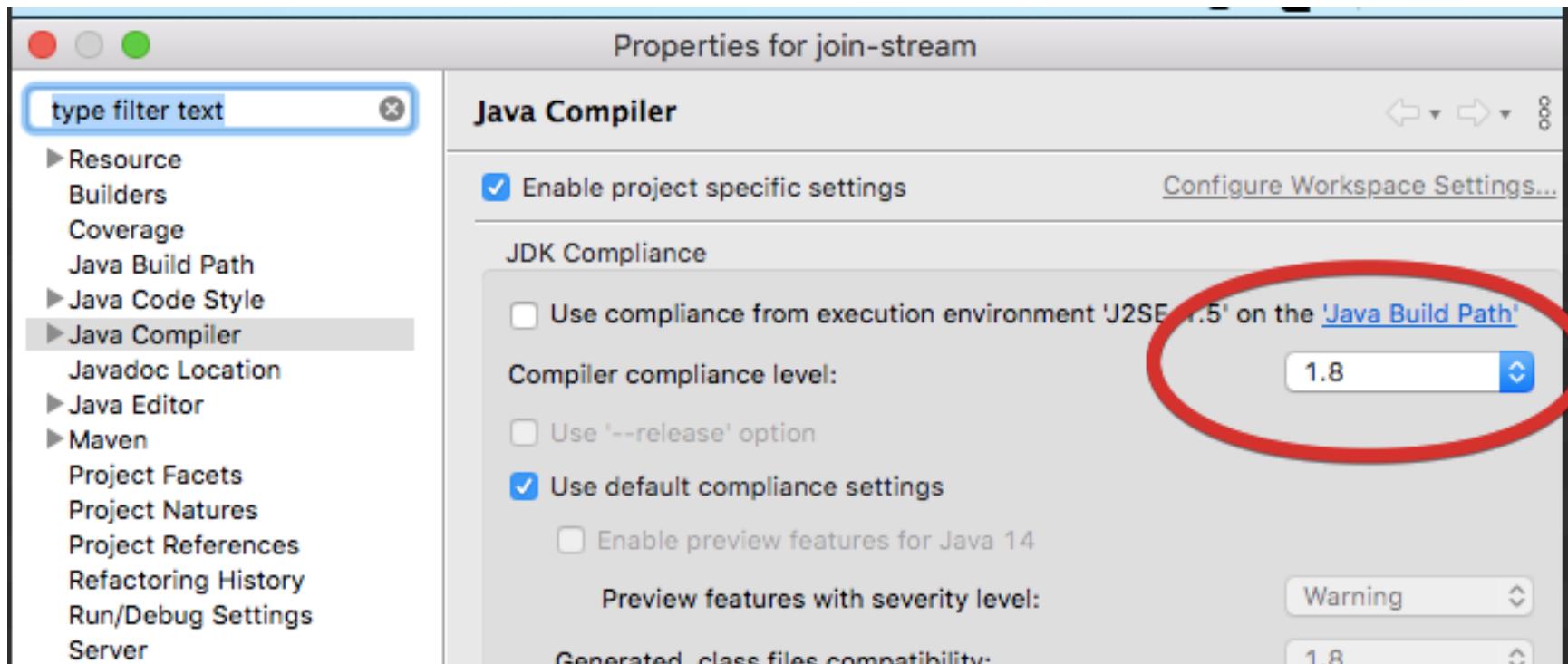
Ensure to include in the pom.xml (avsc paths)

```
105      <configuration>
106          <sourceDirectory>${project.basedir}/src/main/resources/</sourceDirectory>
107          <sourceDirectory>${project.basedir}/src/main/resources/avro</sourceDirectory>
108      <includes>
109          <include>input_movie_event.avsc</include>
110          <include>parsed_movies.avsc</include>
111          <include>Payment.avsc</include>
112          <include>*.avsc</include>
113      </includes>
114      <outputDirectory>${project.basedir}/src/main/java</outputDirectory>
115  </configuration>
116  </execution>
117 </executions>
```

```
# 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.ostechnet.join.stream/JoinStreamToTable.java

```
package com.ostechnet.join.stream;
```

Import the following packages and classes

```
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 {

}
```

Define the method that initialize the properties for the Stream.

```
/*
 * Initialize the properties
 */
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_CONF
IG, envProps.getProperty("schema.registry.url"));

        return props;
}
```

Define the Stream Topology.

```
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();
```

```
}
```

Initialize the AVRO schema for Rated Movie that Join Stream.

```
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;
}
```

Method to create the required topics:

```
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.part  
itions"))),
```

```
    Short.parseShort(envProps.getProperty("rekeyed.movie.topic.replication.factor"))));  
  
    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.factor"))));  
  
    client.createTopics(topics);  
    client.close();  
}
```

Method to load the properties file.

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

Main Method, that invoke the Stream Thread.

```
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.ostechnote.join.stream;

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 associated 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

The screenshot shows the Eclipse IDE interface with the title bar "LearningKafka - join-stream/src/main/java/com/ostech/join/JoinStreamToTable.java - Eclipse IDE". The left side features the "Package Explorer" view, which displays the project structure:

- join-stream (selected)
- src/main/java:
  - JoinStreamToTable.java (highlighted with a red box)
  - Movie.java
  - RatedMovie.java
  - Rating.java
- src/main/resources:
  - avro:
    - movie.avsc
    - rated-movie.avsc
    - rating.avsc
  - dev.properties
- JRE System Library [J2SE-1.5]
- Maven Dependencies

The main editor area shows the Java code for `JoinStreamToTable.java`:

```
131     topology.createTopics(envProps);
132
133     final KafkaStreams streams = new KafkaStreams(topology, streamProps);
134     final CountDownLatch latch = new CountDownLatch(1);
135
136     // Attach shutdown handler to catch Control-C.
137     Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
138         @Override
139         public void run() {
140             streams.close();
141         }
142     });
143 }
```

The "Console" tab in the bottom right shows SLF4J error messages:

```
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.
```

*Load in some movie reference data*

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

Create a movie.avsc schema file in /opt/code 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 kafkao:9092 --property  
value.schema="$(< /opt/code/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}
```

```
[root@kafka0 code]# kafka-console-producer --topic movies --broker-list kafka0:9092 --property value.schema="$(  
< /opt/code/movie.avsc)"
{"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 /opt/code folder.

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

```
#sh kafka-console-producer --topic ratings --broker-list kafka0:9092 --property value.schema=$(  
/opt/code/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 -----

## 14. Processor API integration – 60 Minutes

The following example shows how to leverage Processor API, via the `KStream#process()` method, a custom `Processor` that sends an email notification whenever a Rating count reaches a predefined threshold.

First, we need to implement a custom stream processor, `PopularPersonEmailAlert`, that implements the `Processor` interface:

Add the processor class. It sends email when, the Rating is more than 6.

```
package com.ostechnote.custom.processor;

import org.apache.kafka.streams.processor.api.Processor;
import org.apache.kafka.streams.processor.api.Record;

//A processor that sends an alert message about a popular person to
//a configurable email address
public class PopularPersonEmailAlert implements Processor<String,
Long, Void, Void> {
    private final String emailAddress;

    public PopularPersonEmailAlert(String emailAddress) {
        this.emailAddress = emailAddress;
    }
```

```
@Override  
public void process(Record<String, Long> record) {  
  
    System.out.println(" Sending email to : " + emailAddress  
                      + "\n : Name : " + record.key() + " Total  
Rating is : " + record.value());  
  
}  
  
}
```

Then we can leverage the `PopularPageEmailAlert` processor in the DSL via `KStream#process`.

Define a class.

```
package com.ostechnote.custom.processor;  
  
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.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.Grouped;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Produced;

public class AnalyzeRating {

    public static void main(String[] args) throws Exception {
        Properties envProps = buildStreamsProperties();
        Topology topology = buildTopology(envProps);

        final KafkaStreams streams = new KafkaStreams(topology,
envProps);
        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);
}

/*
 * Initialize the properites required to connect to the Topic
 */
public static Properties buildStreamsProperties() {
    Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG,
"AnalzeRatingApp23");

    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:8082");

    props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
"earliest");
```

```
        props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass());

props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
Serdes.String().getClass());
        props.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG, 10 *
1000);
        // For illustrative purposes we disable record caches.

props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
        // Use a temporary directory for storing state, which
will be automatically removed after the test.
        props.put(StreamsConfig.STATE_DIR_CONFIG, "/tmp/art");
        return props;
    }

/*
 * Read the rating stream , sum all the rating for each
person and save into another topic
*/
public static Topology buildTopology(Properties envProps) {
    final StreamsBuilder builder = new StreamsBuilder();
    final String inputTopic = "myratings";

    KStream<String, String> ratings =
```

```
builder.stream(inputTopic);
        // Input - ("Henry",1) -> Name , Ratings
        KTable<String,Long> myRatings = ratings.filter( (k,v) ->
v !=null && v.trim().length() > 1)
        .map((key, rate) ->{
            String rats[] = rate.trim().split(",");
            return new KeyValue<String,
Long>(rats[0].trim(),Long.valueOf(rats[1].trim()));
        })
        //.peek((k,v) -> System.out.println( k + " Ratings : "
+ v))

    .groupByKey(Grouped.with(Serdes.String(),Serdes.Long()))
        .reduce((v1,v2) -> v1+ v2);

    // myRatings.toStream().peek((k,v) -> System.out.println(
k + " Ratings1 : " + v));
    myRatings.toStream().to("mycompratings",
Produced.with(Serdes.String(), Serdes.Long() ));
    return builder.build();
}
```

}

The program will consume messages from the topic: myratings and will transform the message into Person and Rating then finally aggregate Rating for each person.

Create topics - Input & OutPut :

```
#kafka-topics.sh --create --topic myratings --partitions 2 --bootstrap-server kafka:9092  
#kafka-topics.sh --create --topic mycompratings --partitions 2 --bootstrap-server  
kafka:9092
```

Send input using producer console.

```
#kafka-console-producer.sh --bootstrap-server kafka:9092 --topic myratings
```

Enter some messages in the form (Name, Rating)

```
#Ram,2  
#Shyam,3  
#John,3  
#Ram,2  
#Ram,1
```

Execute:

The main program, `AnalyzeRating.java`

Consume message using console.

```
# kafka-console-consumer.sh --bootstrap-server kafka:9092 \  
--topic mycompratings \  
--from-beginning \  
--property print.key=true --property print.value=true \  
--property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \  
--property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
```

Example output:

```
Created topic mycompratings.
[root@kafka0 code]# kafka-console-producer.sh --bootstrap-server kafka0:9092 --topic myratings
>Ram,2
>Shyam,3
>John,3
>Ram,2
>Ram,1
>
```

Consumer console:

```
[root@kafka0 /]# kafka-console-consumer.sh --bootstrap-server kafka0:9092 \
>    --topic mycompratings \
>    --from-beginning \
>    --property print.key=true --property print.value=true \
>    --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
>    --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
Ram      2
Ram      4
Ram      5
Shyam    3
John    3
|
```

Now let us configure the Processor API. First stop the main program execution.

Add the following line in the Main Program.

```
myRatings.toStream()
    .filter((k,v)-> v.intValue() > 6)
    .process(()-> new
PopularPersonEmailAlert("kstream@apache.com"));;
```

```
76
77     })
78     //.peek((k,v) -> System.out.println( k + " Ratings : " + v))
79     .groupByKey(Grouped.with(Serdes.String(),Serdes.Long()))
80     .reduce((v1,v2) -> v1+ v2);
81
82     myRatings.toStream()
83         .filter((k,v)-> v.intValue() > 6)
84         .process(()-> new PopularPersonEmailAlert("kstream@apache.com"));
85
86     // myRatings.toStream().peek((k,v) -> System.out.println( k + " Ratings1 : " +
87     myRatings.toStream().to("mycompratings", Produced.with(Serdes.String(), Serde
88     return builder.build();
89 }
90
91
```

Execute the Main program after enabling the Processor in the stream.

On the producer console – CLI: Send more messages related to Ram. So, that total rating for Ram is more than 6.

```
Created topic mycompratings.  
[root@kafka0 code]# kafka-console-producer.sh --bootstrap-server kafka0:9092 --topic myratings  
>Ram,2  
>Shyam,3  
>John,3  
>Ram,2  
>Ram,1  
>Ram,2  
->Shyam,1  
>
```

On the consumer console:

```
[root@kafka0 ~]# kafka-console-consumer.sh --bootstrap-server kafka0:9092 \
>     --topic mycompratings \
>     --from-beginning \
>     --property print.key=true --property print.value=true \
>     --property key.deserializer=org.apache.kafka.common.serialization.StringDeserializer \
>     --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer
Ram      2
Ram      4
Ram      5
Shyam    3
John     3
Ram      7
Shyam    4
```

You will observe the following in the main program console.



The screenshot shows a Java application window titled "Console". The title bar includes standard window controls and a tab labeled "Console". The main area displays the following text:

```
AnalyzeRating [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_271.jdk/Contents/Home/bin/java (08-Mar-2022, 8:53:12 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 de
 Sending email to : kstream@apache.com
 : Name : Ram Total Rating is : 7
```

Congrats, you have successfully integrated the Processor API in the Kafka Stream DSL.

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

## 15. Kafka - UDAF

In this lab we will learn how 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>json</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>
        </execution>
    </executions>
</plugin>
```

```
        <configuration>

            <sourceDirectory>${project.basedir}/src/main/resources/avro/</sourceD
irectory>

            <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;
    }
};
```

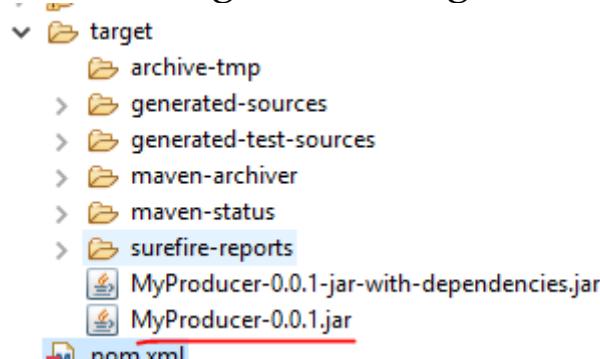
```
}
```

```
<!-- 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": "impressionsontime", "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": "impressionscore", "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 impressionscore 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.

The screenshot shows the Apache Kafka Control Center interface. On the left, there's a sidebar with 'MONITORING' and 'MANAGEMENT' sections. Under 'MONITORING', there are links for 'System health', 'Data streams', and 'Consumer lag'. Under 'MANAGEMENT', there are links for 'Kafka Connect', 'Clusters', and 'Topics', with 'Topics' being the active tab, indicated by a purple background.

The main area is titled 'MANAGEMENT > Topics'. It features a search bar with the text 'imp' and a checkbox labeled 'Show internal topics'. Below this is a table with the following data:

| Name        | Pa | Tot |
|-------------|----|-----|
| impressions | 1  | 1   |

## 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 , impressionscore 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 ( impressionscore )
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 -----

## 16. 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_alice 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"
  }
}

```

```
}
```

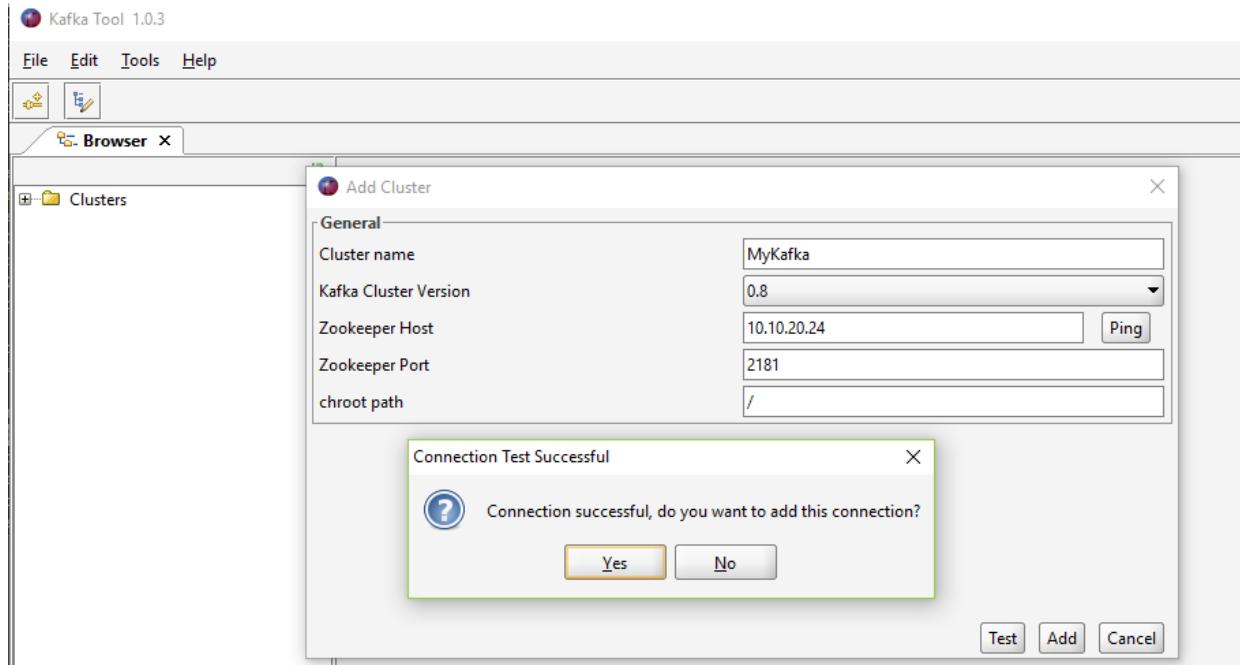
Copy

#### 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}
...
```

## 17. Kafkatools



## 18. 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
# Many listeners map to security protocols, the default is for them to be the s
ame
```

### 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(FutureRe
cordMetadata.java:94)

```
at  
org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMeta  
data.java:64)  
at  
org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMeta  
data.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 start, update with ip address and restart the broker.

## 19. 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 00000000000000000000000000000000.log
[root@tos test-topic-0]# cd ..
[root@tos kafka-logs]# cd my-kafka-connect-0/
[root@tos my-kafka-connect-0]# ls
00000000000000000000000000000000.index      00000000000000000000000000000000.snapshot
00000000000000000000000000000000.log        leader-epoch-checkpoint
00000000000000000000000000000000.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.DumpLogSegments --deep-iteration --print-data-log --files \
> /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000000000000000.log | head -n 4
Dumping /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000000000000000.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.
[root@tos my-kafka-connect-0]#
```

### 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 raw example](#) **JackieChanCommand.json** hosted with [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 [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 raw](#)**jackieChanWorkflow.json** hosted with [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":"HEAD","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_SWOR D","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}
```

[view rawjackieChanCommands.json](#) hosted with [GitHub](#)

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.

## 20. Pom.xml (Standalone)

```
<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.hp.tos</groupId>
  <artifactId>LearningKafka</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <description>All About Kafka</description>
  <properties>
    <algebird.version>0.13.4</algebird.version>
    <avro.version>1.8.2</avro.version>
    <avro.version>1.8.2</avro.version>
    <confluent.version>5.3.0</confluent.version>
    <kafka.version>3.0.0</kafka.version>
    <kafka.scala.version>2.11</kafka.scala.version>
  </properties>
  <repositories>
    <repository>
      <id>confluent</id>
      <url>https://packages.confluent.io/maven/</url>
    </repository>
  </repositories>
```

```
<pluginRepositories>
    <pluginRepository>
        <id>confluent</id>
        <url>https://packages.confluent.io/maven/</url>
    </pluginRepository>
</pluginRepositories>
<dependencies>
    <dependency>
        <groupId>io.confluent</groupId>
        <artifactId>kafka-streams-avro-serde</artifactId>
        <version>${confluent.version}</version>
    </dependency>
    <dependency>
        <groupId>io.confluent</groupId>
        <artifactId>kafka-avro-serializer</artifactId>
        <version>${confluent.version}</version>
    </dependency>
    <dependency>
        <groupId>io.confluent</groupId>
        <artifactId>kafka-schema-registry-client</artifactId>
        <version>${confluent.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
```

```
        <version>${kafka.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
        <version>${kafka.version}</version>
    </dependency>

    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams-test-utils</artifactId>
        <version>${kafka.version}</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro</artifactId>
        <version>${avro.version}</version>
    </dependency>
    <dependency>
        <groupId>org.apache.avro</groupId>
        <artifactId>avro-maven-plugin</artifactId>
        <version>${avro.version}</version>
    </dependency>
    <dependency>
```

```
<groupId>org.apache.avro</groupId>
<artifactId>avro-compiler</artifactId>
<version>${avro.version}</version>
</dependency>
<dependency>
<groupId>com.google.code.gson</groupId>
<artifactId>gson</artifactId>
<version>2.6.2</version>
</dependency>

</dependencies>

<build>
<plugins>
<plugin>
<artifactId>maven-compiler-plugin</artifactId>
<configuration>
    <source>1.8</source>
    <target>1.8</target>
</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/</sourceDirectory>

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

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

```
</plugins>
</build>

</project>
```

## 21. pom.xml (Spring boot)

```
<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</groupId>
  <artifactId>SpringKafka</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>2.9.0</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>
```

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
    <relativePath />
</parent>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
    <!--
https://mvnrepository.com/artifact/com.google.code.gson/gson -->
    <dependency>
        <groupId>com.google.code.gson</groupId>
        <artifactId>gson</artifactId>
    </dependency>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-streams</artifactId>
```

```
</dependency>
<dependency>
    <groupId>org.apache.kafka</groupId>
    <artifactId>kafka-streams-scala_2.13</artifactId>
</dependency>
<dependency>
    <groupId>org.javatuples</groupId>
    <artifactId>javatuples</artifactId>
    <version>1.2</version>
</dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
</project>
```

## 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>

<https://docs.confluent.io/current/ksql/docs/tutorials/examples.html#ksql-examples>