

1.	Producer in Java – 60 Minutes	2
2.	Consumer – Java API – 45 Minutes	19
3.	Understand Kafka Stream – Topology – 30 Minutes	28
4.	Running your first Kafka Streams Application: WordCount – 60 minutes	38
5.	Schema registry	51
6.	Kafkatools	64
7.	Errors	65
1.	LEADER_NOT_AVAILABLE	65
	java.util.concurrent.ExecutionException:	65
8.	Annexure Code:	67
2.	DumplogSegment	67
3.	Data Generator – JSON	68
4.	Resources	75

Last Updated: 15 Nov 2020

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

1. Producer in Java – 60 Minutes

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.

Create Replicated Kafka Topic

Create topic

```
#!/opt/kafka/bin/kafka-topics.sh --create --replication-factor 3 --partitions 13 --topic my-kafka-topic --zookeeper localhost:2181
```

Above we created a topic named my-kafka-topic with 13 partitions and a replication factor of 3.

Then we list the Kafka topics.

```
[root@tos scripts]# /opt/kafka/bin/kafka-topics.sh --create --replication-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 --zookeeper localhost:2181
```

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

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

Maven java project Details:

Group ID: com.tos.kafka

Artifact ID: my-kafka-producer.

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

```
<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>

</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>
          <goal>protocol</goal>
          <goal>idl-protocol</goal>
        </goals>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
```

Construct a Kafka Producer

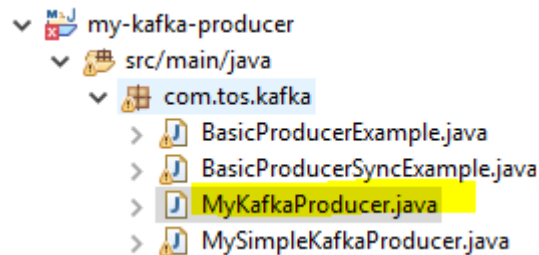
To create a Kafka producer, you will need to pass it a list of bootstrap servers (a list of Kafka brokers). You will also specify a `client.id` that uniquely identifies this Producer client. In this example, we are going to send messages with ids. The message body is a string, so we need a record value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

Create a java class with the following package name:

Class : `MyKafkaProducer`

Package Name : `com.tos.kafka`

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



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

Add the following imports in your code.

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

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

Then, let us define some constant variables as stated below,

BOOTSTRAP_SERVERS is set to [tos.master.com:9092](#), [tos.master.com:9093](#), [tos.master.com:9094](#) which is the three Kafka servers that we started up in the last lesson. Go ahead and make sure all three Kafka servers are running.

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.

Add the following variables in your code.

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

Create Kafka Producer to send records

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

```
private static Producer<Long, String> createProducer() {
    Properties props = new Properties();
    props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,BOOTSTRAP_SERVERS);
    props.put(ProducerConfig.CLIENT_ID_CONFIG, "KafkaExampleProducer");

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

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

Above **createProducer** sets the **BOOTSTRAP_SERVERS_CONFIG** (“bootstrap.servers) property to the list of broker addresses we defined earlier. **BOOTSTRAP_SERVERS_CONFIG** value is a comma separated list of host/port pairs that the Producer uses to establish an initial connection to the Kafka cluster. The producer uses of all servers in the cluster no matter which ones we list here. This list only specifies the initial Kafka brokers used to discover the full set of servers

of the Kafka cluster. If a server in this list is down, the producer will just go to the next broker in the list to discover the full topology of the Kafka cluster.

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

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

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

Send records synchronously with Kafka Producer

Kafka provides a synchronous send method to send a record to a topic. Let’s use this method to send some message ids and messages to the Kafka topic we created earlier.

Add the following in your code.

```

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

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

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

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

        }
    } catch (Exception e) {

        e.printStackTrace();
    }
}

```

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

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

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

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

Running the Kafka Producer.

Next you define the main method.

Add the following imports in your code.

```

public static void main(String[] args) {

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

}

```

The **main** method just calls **runProducer**.

Start a consumer console so that you can consume the message sent by this producer.

```

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

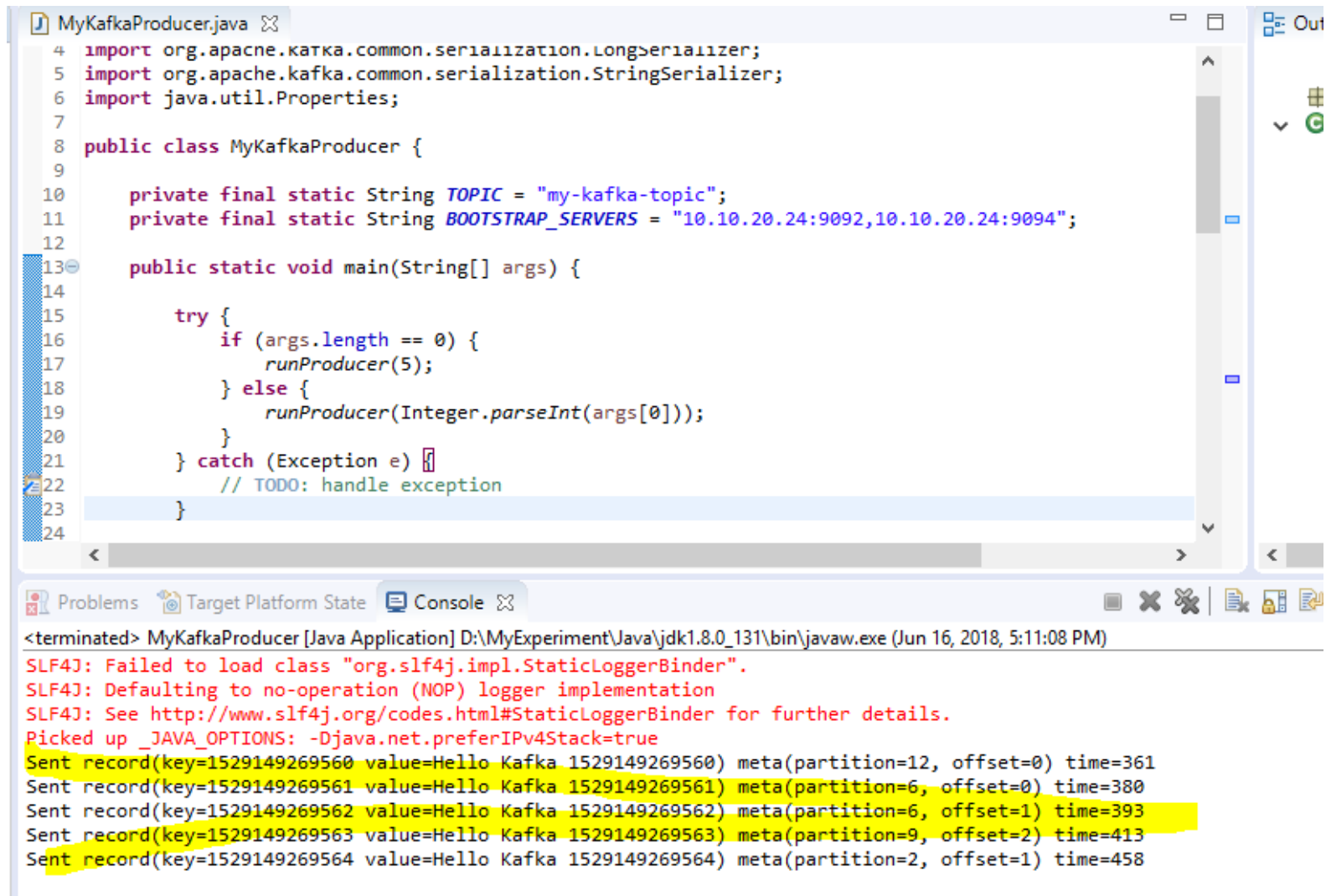
```

```

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-beginnin
g

```

Execute the main program.



The screenshot shows an IDE window with the file `MyKafkaProducer.java` open. The code defines a `MyKafkaProducer` class with a `main` method that sends records to a Kafka topic. The console output shows the program running successfully and sending five records.

```

MyKafkaProducer.java
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     }
25 }

```

Console Output:

```

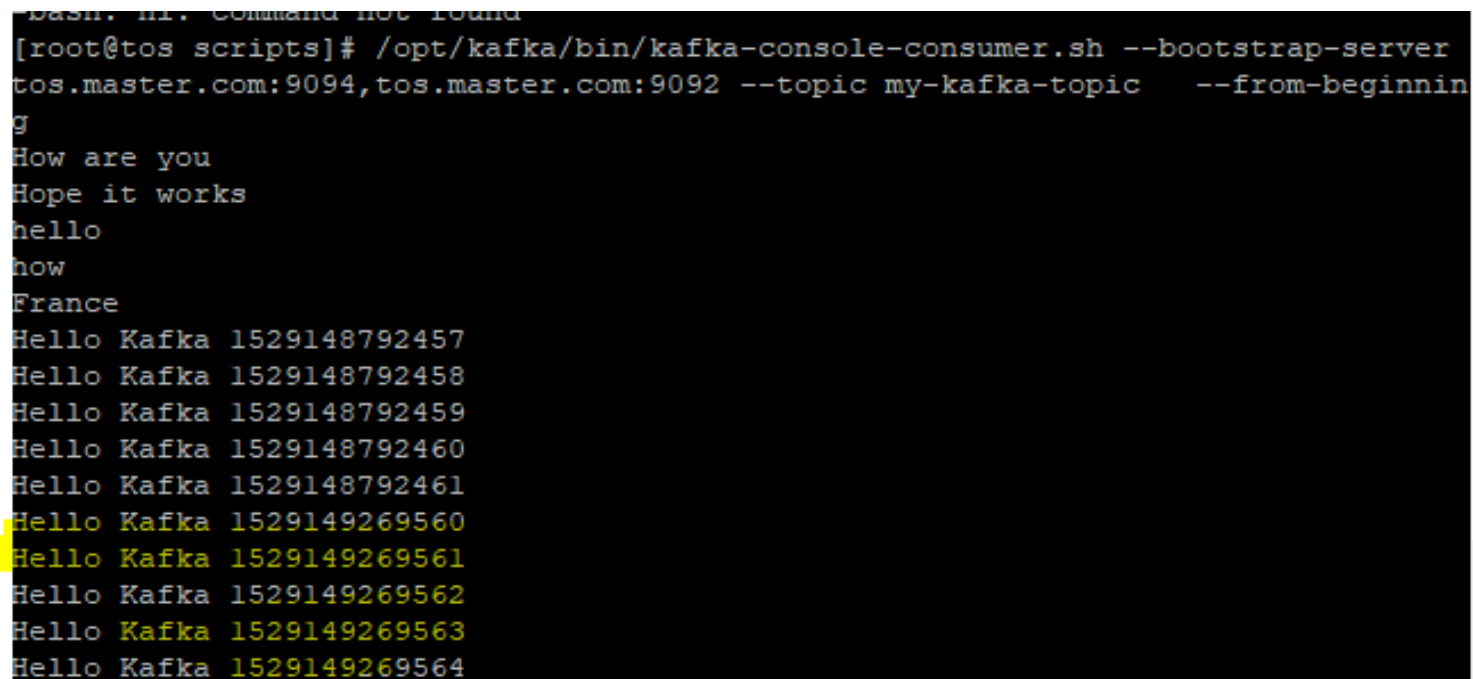
<terminated> MyKafkaProducer [Java Application] D:\MyExperiment\Java\jdk1.8.0_131\bin\javaw.exe (Jun 16, 2018, 5:11:08 PM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder 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 message as shown above.

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

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

A terminal window with a black background and white text. The prompt is [root@tos scripts]#. The command entered is /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 a series of messages: 'How are you', 'Hope it works', 'hello', 'now', 'France', and then a series of 'Hello Kafka' messages with increasing numeric IDs. The message 'Hello Kafka 1529149269561' is highlighted with a yellow background. The terminal window has a scrollbar on the right side.

```
Dash. Hi. 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-beginnin
g
How are you
Hope it works
hello
now
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
```

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.

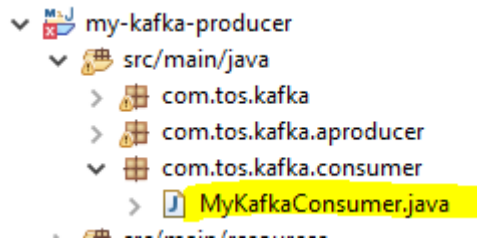
KafkaConsumerExample.java - imports and constants

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.util.Collections;
import java.util.Properties;

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

}
```

Notice that `KafkaConsumerExample` imports `LongDeserializer` which gets configured as the Kafka record key deserializer, and imports `StringDeserializer` which gets set up as the record value deserializer. The constant `BOOTSTRAP_SERVERS` gets set to `localhost:9092,localhost:9093,localhost:9094` which is the three Kafka servers that we started up in the last lesson. Go ahead and make sure all three Kafka servers are running. The constant `TOPIC` gets set to the replicated Kafka topic that you created in the last tutorial.

Create Kafka Consumer consuming Topic to Receive Records

Now, that you imported the Kafka classes and defined some constants, let's create the Kafka consumer.

`KafkaConsumerExample.java` - Create Consumer to process Records

Add the following method that will initialize the consumer parameters.

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

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

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

To create a Kafka consumer, you use `java.util.Properties` and define certain properties that we pass to the constructor of a `KafkaConsumer`.

Above `KafkaConsumerExample.createConsumer` sets

the `BOOTSTRAP_SERVERS_CONFIG` ("bootstrap.servers") property to the list of broker addresses we defined

earlier. **BOOTSTRAP_SERVERS_CONFIG** value is a comma separated list of host/port pairs that the **Consumer** uses to establish an initial connection to the Kafka cluster. Just like the producer, the consumer uses 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 Deserializer class for Kafka record values that implements the Kafka Deserializer interface. Notice that we set this to **StringDeserializer** as the message body in our example are strings.

Important notice that you need to subscribe the consumer to the topic **consumer.subscribe(Collections.singletonList(TOPIC));**. The subscribe method takes a list of topics to subscribe to, and this list will replace the current subscriptions if any.

Process messages from Kafka with Consumer

Now, let's process some records with our Kafka Producer.

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

```

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

    final int giveUp = 100;
    int noRecordsCount = 0;

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

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

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

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

```

Notice you use `ConsumerRecords` which is a group of records from a Kafka topic partition. The `ConsumerRecords` class is a container that holds a list of `ConsumerRecord(s)` per partition for a particular topic. There is one `ConsumerRecord` list for every topic partition returned by a the `consumer.poll()`.

Notice if you receive records (`consumerRecords.count() != 0`), then `runConsumer` method calls `consumer.commitAsync()` which commit offsets returned on the last call to `consumer.poll(...)` for all the subscribed list of topic partitions.

Kafka Consumer Poll method

The poll method returns fetched records based on current partition offset. The poll method is a blocking method waiting for specified time in seconds. If no records are available after the time period specified, the poll method returns an empty `ConsumerRecords`.

When new records become available, the poll method returns straight away.

You can control the maximum records returned by the `poll()`

with `props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);`. The poll method is not thread safe and is not meant to get called from multiple threads.

Running the Kafka Consumer

Next you define the `main` method.

```
public class KafkaConsumerExample {

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

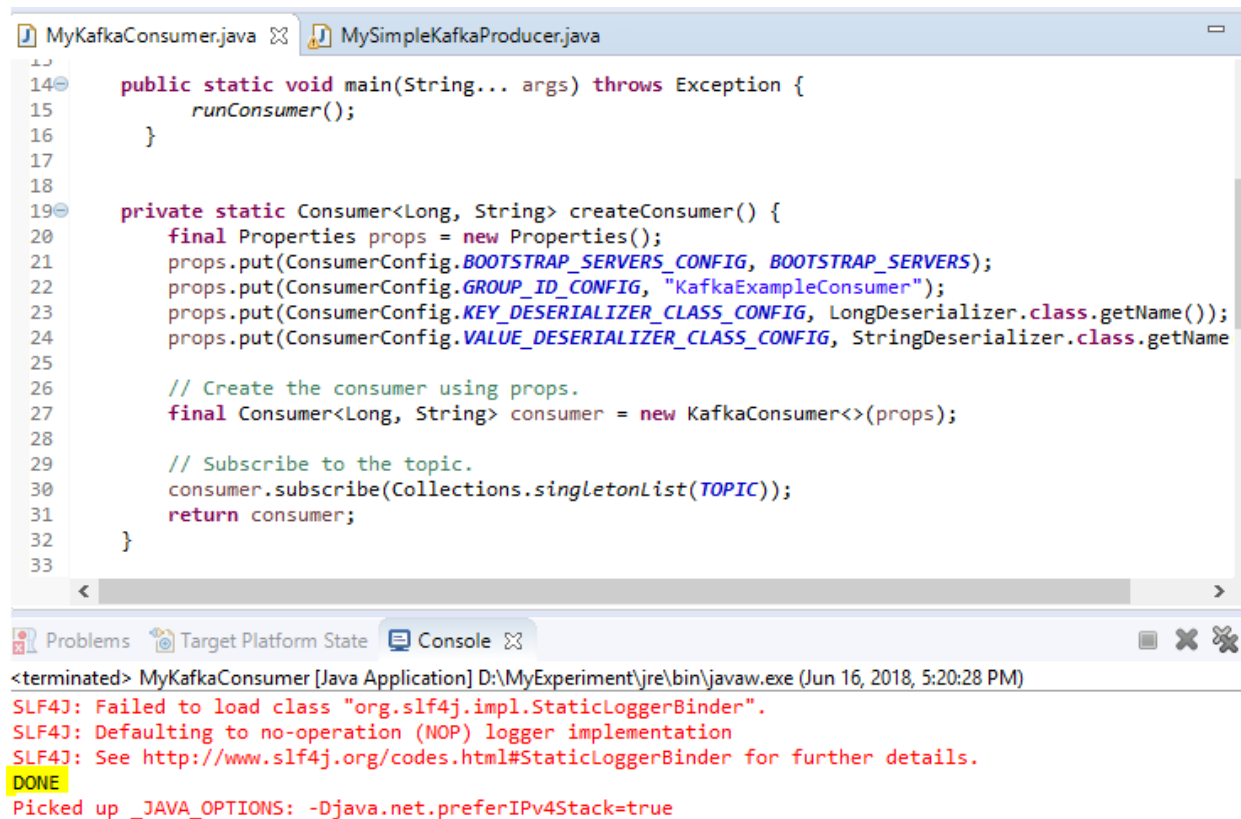


```
}
```

The `main` method just calls `runConsumer`.

Try running the consumer and producer.

Run the consumer from your IDE. Then run the producer from the last tutorial from your IDE. You should see the consumer get the records that the producer sent.



```

MyKafkaConsumer.java  MySimpleKafkaProducer.java
14 public static void main(String... args) throws Exception {
15     runConsumer();
16 }
17
18
19 private static Consumer<Long, String> createConsumer() {
20     final Properties props = new Properties();
21     props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
22     props.put(ConsumerConfig.GROUP_ID_CONFIG, "KafkaExampleConsumer");
23     props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, LongDeserializer.class.getName());
24     props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
25
26     // Create the consumer using props.
27     final Consumer<Long, String> consumer = new KafkaConsumer<>(props);
28
29     // Subscribe to the topic.
30     consumer.subscribe(Collections.singletonList(TOPIC));
31     return consumer;
32 }
33

```

Problems Target Platform State Console

```

<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 see there are no records. Why?? It was already consumed by the terminal windows which offset is committed. Stop the consumer console.

Execute the Producer and then execute the consumer

```
<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.
Caused by: org.apache.kafka.common.errors.SerializationException: Size of data received by LongDeserializer :
Picked up _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="STDOUT"
    class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
    </pattern>
    </encoder>
  </appender>

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

```
        </encoder>
    </appender>

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

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

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

Leave **org.apache.kafka.common.metrics** or what Kafka is doing under the covers is drowned by metrics logging.

Lab End Here

3. Understand Kafka Stream – Topology – 30 Minutes

Kafka Streams is a client library for building mission-critical real-time applications and microservices, where the input and/or output data is stored in Kafka clusters. Kafka Streams combines the simplicity of writing and deploying standard Java and Scala applications on the client side with the benefits of Kafka's server-side cluster technology to make these applications highly scalable, elastic, fault-tolerant, distributed, and much more.

This tutorial will demonstrate how to run a streaming application coded in this library. In this guide we will start from scratch on setting up your own project to write a stream processing application using Kafka Streams.

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

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

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

```
<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>
</dependencies>
```

```
<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>
</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>
      <goal>protocol</goal>
      <goal>idl-protocol</goal>
    </goals>
  </execution>
</executions>
</plugin>
</plugins>
</build>
```


Writing a first Streams application: Pipe

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

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

public class Pipe {

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

    }

}
```

We are going to write all the code in the `main` function of this pipe program.

Import the following class.

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

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

```
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.kstream.Produced;
import org.apache.kafka.streams.state.KeyValueStore;
```

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

Add the following properties code.

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

Next we will define the computational logic of our Streams application. In Kafka Streams this computational logic is defined as a `topology` of connected processor nodes. We can use a topology builder to construct such a topology,

And then create a source stream from a Kafka topic named `streams-plaintext-input` using this topology builder:

Now we get a `KStream` that is continuously generating records from its source Kafka topic `streams-plaintext-input`.

The records are organized as `String` typed key-value pairs. The simplest thing we can do with this stream is to write it into another Kafka topic, say it's named `streams-pipe-output`. Note that we can also concatenate the above two lines into a single line.

We can then inspect what kind of `topology` is created from this builder and print its description to standard output.

Copy the following code inside the main class.

```

final StreamsBuilder builder = new StreamsBuilder();

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

final Topology topology = builder.build();

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

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

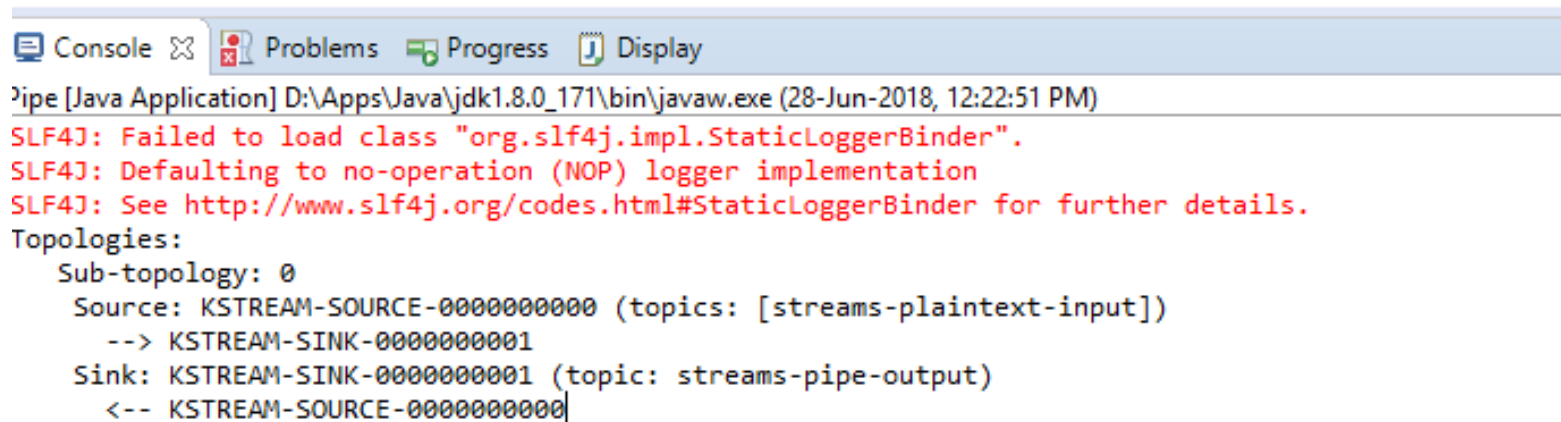
// attach shutdown handler to catch control-c
Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
    @Override
    public void run() {
        streams.close();
        latch.countDown();
    }
});

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

```

If we just stop here, compile and run the program, it will output the following information:

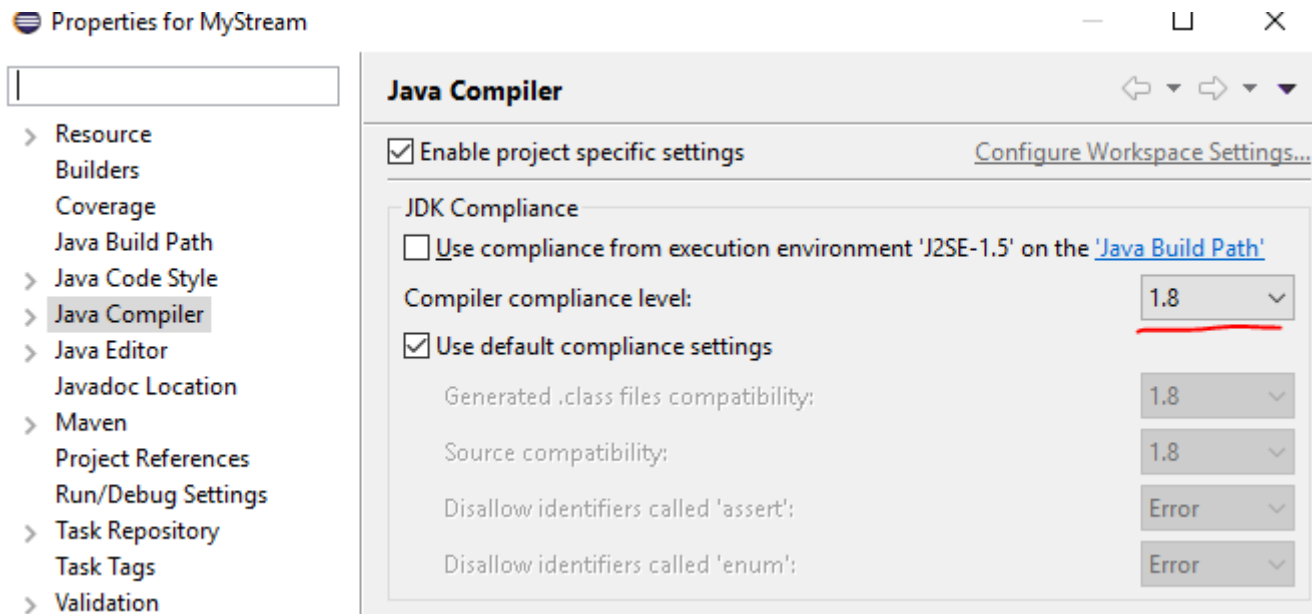
Run Maven install, then execute the main program. Ensure that you have started the Broker.



```

Pipe [Java Application] D:\Apps\Java\jdk1.8.0_171\bin\javaw.exe (28-Jun-2018, 12:22:51 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.
Topologies:
  Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [streams-plaintext-input])
      --> KSTREAM-SINK-0000000001
    Sink: KSTREAM-SINK-0000000001 (topic: streams-pipe-output)
      <-- KSTREAM-SOURCE-0000000000
  
```

As shown above, it illustrates that the constructed topology has two processor nodes, a source node **KSTREAM-SOURCE-0000000000** and a sink node **KSTREAM-SINK-0000000001**. **KSTREAM-SOURCE-0000000000** continuously read records from Kafka topic **streams-plaintext-input** and pipe them to its downstream node **KSTREAM-SINK-0000000001**; **KSTREAM-SINK-0000000001** will write each of its received record in order to another Kafka topic **streams-pipe-output** (the **-->** and **<--** arrows dictates the downstream and upstream processor nodes of this node, i.e. "children" and "parents" within the topology graph). It also illustrates that this simple topology has no global state stores associated with it. You may need to configure project to support 1.8 java in case you have not done it before.



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

4. 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 java.util.Arrays;
import java.util.Properties;

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.Consumed;
import org.apache.kafka.streams.kstream.KStream;
```

```
import org.apache.kafka.streams.kstream.KTable;  
import org.apache.kafka.streams.kstream.Produced;
```

Update the following code in the main method().

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

```
Properties props = new Properties();  
    props.put(StreamsConfig.APPLICATION_ID_CONFIG,  
"wordcount-application");  
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,  
"localhost:8081");  
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,  
Serdes.String().getClass());  
  
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,  
Serdes.String().getClass());
```

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

```
StreamsBuilder builder = new StreamsBuilder();
    // Serializers/deserializers (serde) for String and Long
types
    final Serde<String> stringSerde = Serdes.String();
    final Serde<Long> longSerde = Serdes.Long();

    // Construct a `KStream` from the input topic "streams-
    plaintext-input", where message values
    // represent lines of text (for the sake of this
example, we ignore whatever may be stored
    // in the message keys).
    KStream<String, String> textLines = builder.stream(
        "streams-plaintext-input",
        Consumed.with(stringSerde, stringSerde)
    );

    KTable<String, Long> wordCounts = textLines
        // Split each text line, by whitespace, into words.
        .flatMapValues(value ->
Arrays.asList(value.toLowerCase().split("\\W+")))

    // Group the text words as message keys
```



```
        .groupBy((key, value) -> value)

        // Count the occurrences of each word (message key).
        .count();

        // Store the running counts as a changelog stream to the
output topic.
        wordCounts.toStream().to("streams-wordcount-output",
Produced.with(Serdes.String(), Serdes.Long()));

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

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.

```

17 public static void main(final String[] args) throws Exception {
18     Properties props = new Properties();
19     props.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-application");
20     props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:8081");
21     props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass());
22     props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass());
23
24     StreamsBuilder builder = new StreamsBuilder();
25     // Serializers/deserializers (serde) for String and Long types
26     final Serde<String> stringSerde = Serdes.String();
27     final Serde<Long> longSerde = Serdes.Long();
28
29     // Construct a `KStream` from the input topic "streams-plaintext-input", where message values
30     // represent lines of text (for the sake of this example, we ignore whatever may be stored
31     // in the message keys).
32     KStream<String, String> textLines = builder.stream(
33         "streams-plaintext-input",
34         Consumed.with(stringSerde, stringSerde)
35     );
36
37     KTable<String, Long> wordCounts = textLines
38         // Split each text line, by whitespace, into words.
39         .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
40
41         // Group the text words as message keys
42         .groupBy((key, value) -> value)
43

```

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: hG4VK81QRR0fD7I4Jp9TdQ 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.Long` format, 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/30/documentation/streams/quickstart>

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

5. Schema registry

In the theory – Explain Avro ah bit.

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

Create the transactions topic

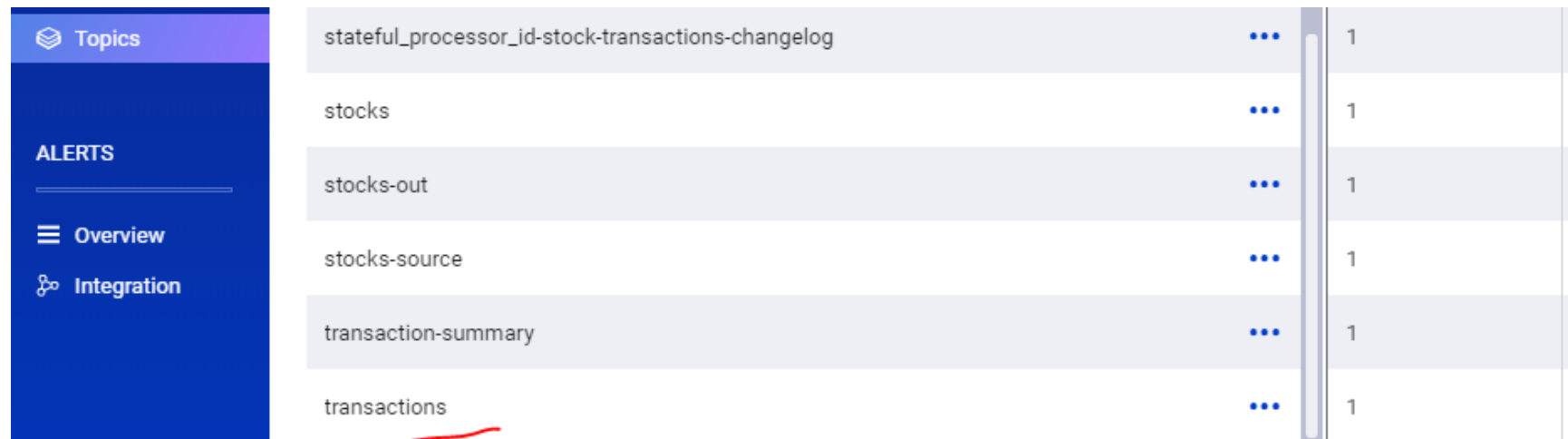
For the exercises in this tutorial, you will be producing to and consuming from a topic called **transactions**. Create this topic in Control Center.

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

Click into the cluster, select **Topics** and click **Add a topic**.

Name the topic **transactions** and click **Create with defaults**.

The new topic is displayed.



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

Schema Definition

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

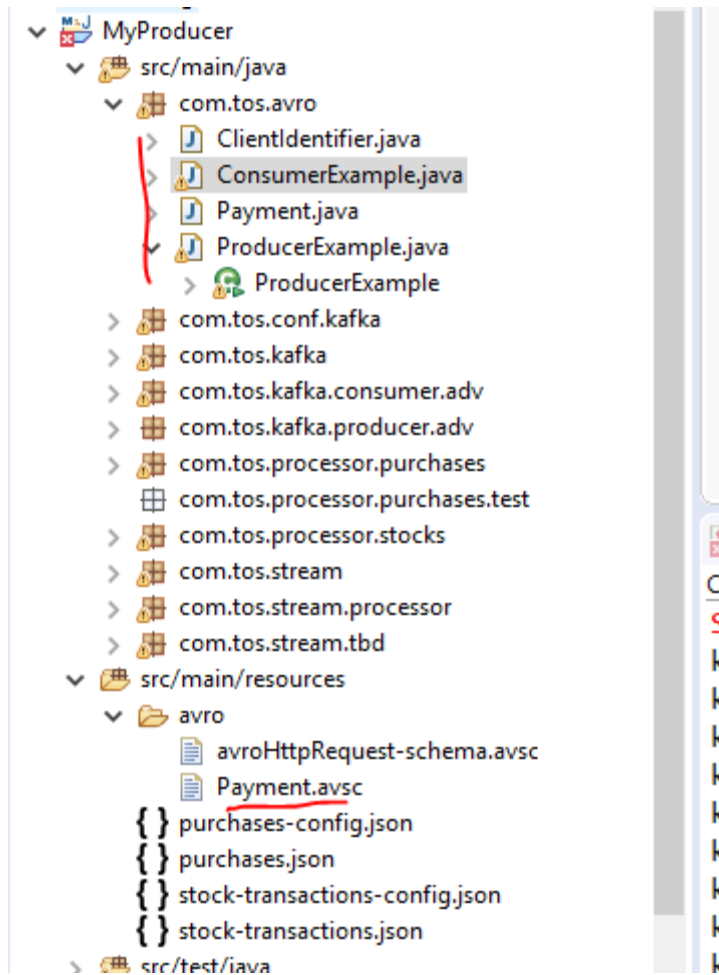
- producers will write data in a schema
- consumers will be able to read that data

Consider the [original Payment schema](#).

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

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

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



Update pom.xml

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

The `pom.xml` file may also include:

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

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

Configuring Avro

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

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

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

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

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

Java Producers

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

Example Producer Code

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

// Code Begins Here

```
package com.tos.avro;
```

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

```
    private static final String TOPIC = "transactions";
```

```
    @SuppressWarnings(""InfiniteLoopStatement")
```

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

```
        final Properties props = new Properties();
        props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.139.132:9092");
        props.put(ProducerConfig.ACKS_CONFIG, "all");
        props.put(ProducerConfig.RETRIES_CONFIG, 0);
        props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, KafkaAvroSerializer.class);
        props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://192.168.139.132:8081");
```

```

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

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

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

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

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

}

}
// Code Ends Here

```

Run the Producer

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

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

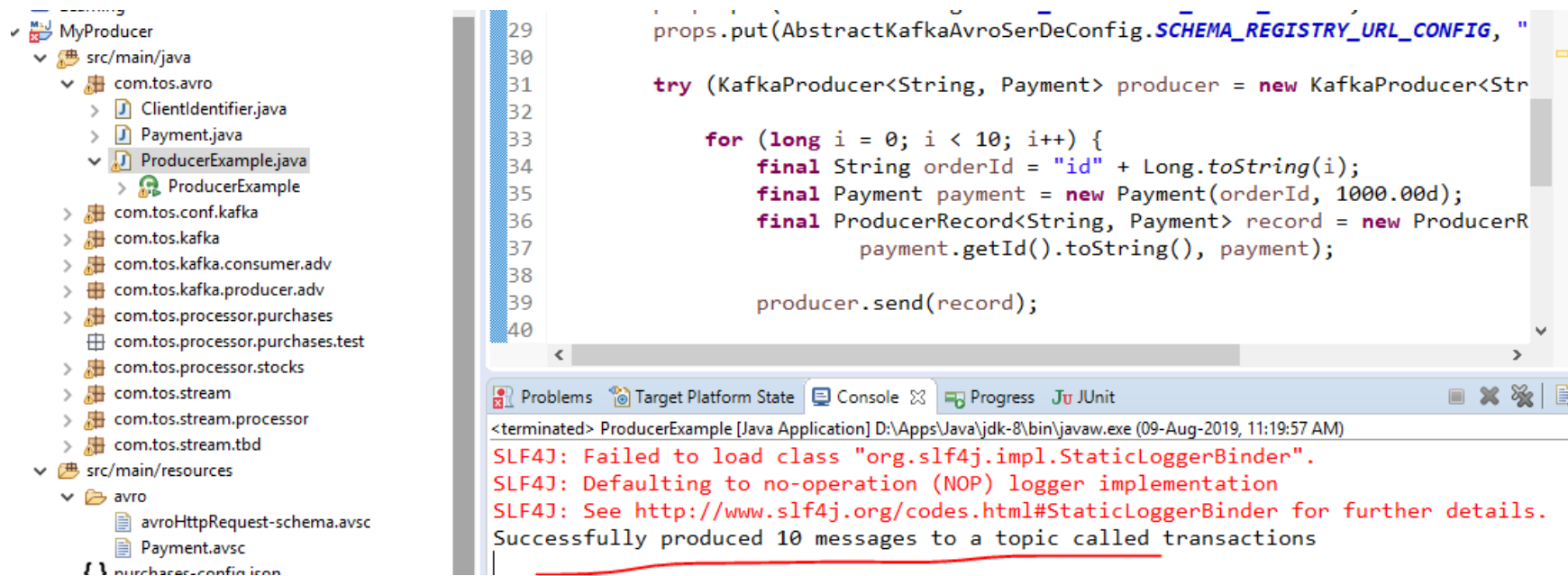
2. mvn clean **compile** package

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

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

You should see no messages because no messages have been produced to this topic yet.

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



The screenshot shows an IDE with a project named 'MyProducer'. The file explorer on the left shows the project structure, including 'src/main/java/com/tos/avro/ProducerExample.java'. The main editor displays the code for 'ProducerExample.java', which is a Java class that produces Avro-formatted messages to a Kafka topic named 'transactions'. The code includes imports for Kafka and Avro, and a main method that creates a KafkaProducer and sends 10 messages.

```

29 props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "
30
31 try (KafkaProducer<String, Payment> producer = new KafkaProducer<Str
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 ProducerR
37             payment.getId().toString(), payment);
38
39         producer.send(record);
40

```

The console output at the bottom shows the following messages:

```

<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.

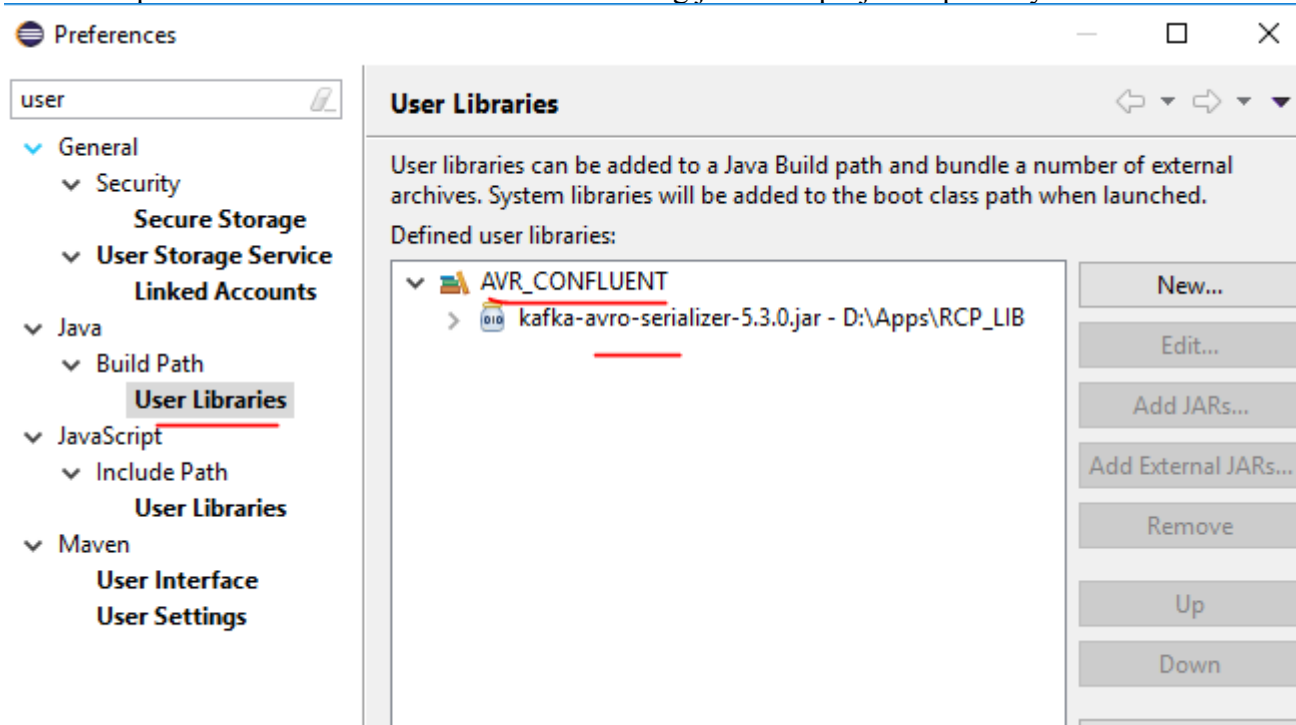
At <http://localhost:9021/>, click into the cluster on the left, then go to **Topics** -> **transactions** -> **Messages**.

Verify the schema too from the control Center.

The screenshot displays the Apache Kafka Control Center interface. On the left is a dark blue sidebar with navigation links: MONITORING (System health, Data streams, Consumer lag), MANAGEMENT (Kafka Connect, Clusters, Topics), ALERTS (Overview, Integration), and DEVELOPMENT (KSQL). The main panel is titled 'MANAGEMENT > TOPICS > transactions'. Below the title are tabs for STATUS, SCHEMA (selected), INSPECT, and SETTINGS. The SCHEMA tab shows a 'Value' key and a JSON schema for 'Version 1'. The schema is a record type named 'Payment' in the namespace 'com.tos.avro', containing two fields: 'id' of type 'string' and 'amount' of type 'double'.

```
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   ]
}
```

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



Java Consumers

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

Example Consumer Code

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

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

// Code Begins Here

```
package com.tos.avro;

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

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

import io.confluent.kafka.serializers.KafkaAvroDeserializer;

import io.confluent.kafka.serializers.KafkaAvroDeserializerConfig;

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

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

    @SuppressWarnings("InfiniteLoopStatement")
    public static void main(final String[] args) {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "192.168.139.132:9092");
```

```

props.put(ConsumerConfig.GROUP_ID_CONFIG, "test-payments");
props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
props.put(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
props.put(AbstractKafkaAvroSerDeConfig.SCHEMA_REGISTRY_URL_CONFIG, "http://192.168.139.132:8081");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, KafkaAvroDeserializer.class);
props.put(KafkaAvroDeserializerConfig.SPECIFIC_AVRO_READER_CONFIG, true);

```

```

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

```

```

    consumer.subscribe(Collections.singletonList(TOPIC));

```

```

    while (true) {

```

```

        final ConsumerRecords<String, Payment> records = consumer.poll(100);

```

```

        for (final ConsumerRecord<String, Payment> record : records) {

```

```

            final String key = record.key();

```

```

            final Payment value = record.value();

```

```

            System.out.printf("key = %s, value = %s%n", key, value);

```

```

        }

```

```

    }

```

```

}

```

```

}

```

```

}

```

```

// Code Ends Here

```

Run the Consumer

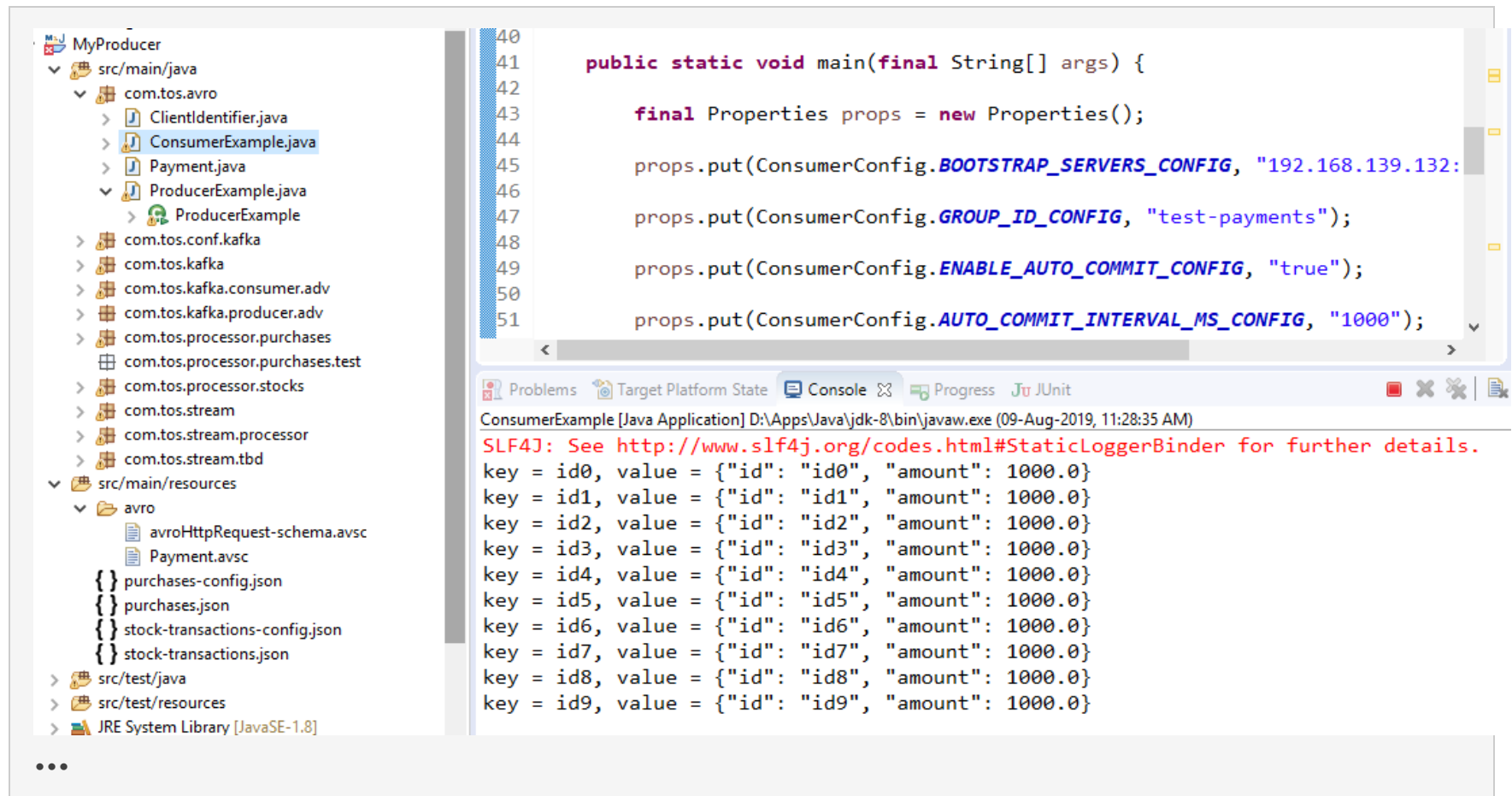
1. To run this consumer, first compile the project.

2. mvn clean `compile` package

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

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

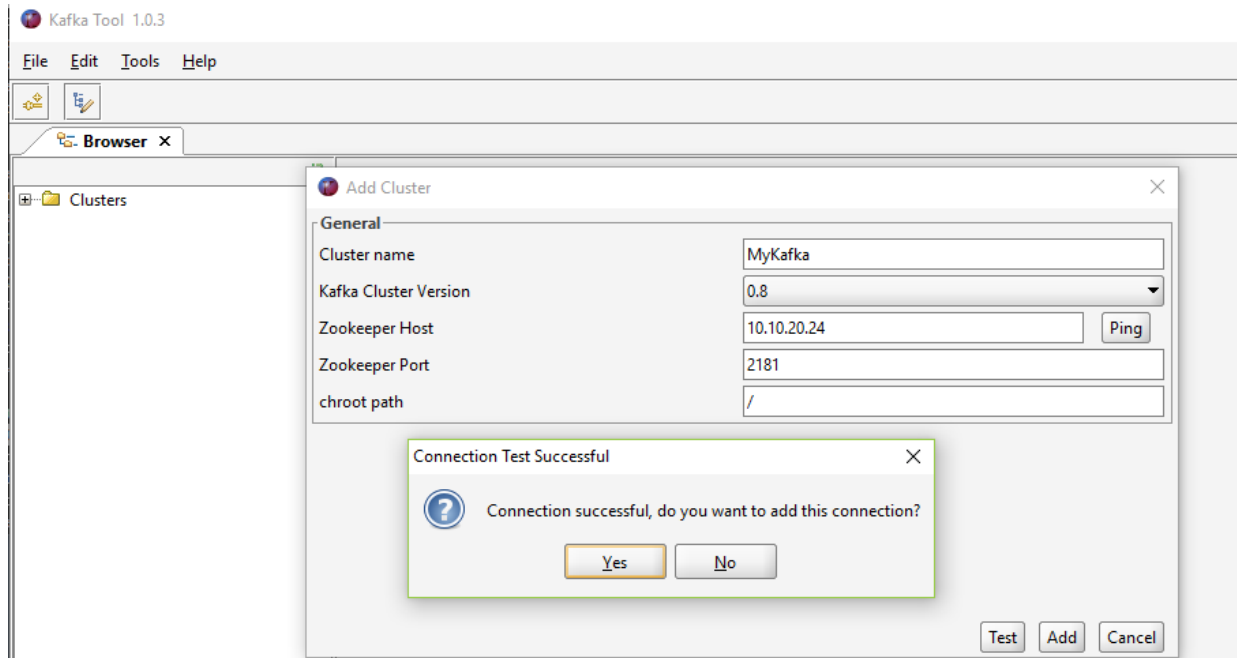
You should see:



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

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

6. Kafkatools



7. 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 listener names to security protocols - the default is for them to be the s
```

java.util.concurrent.ExecutionException: org.apache.kafka.common.errors.TimeoutException: Expiring

1 record(s) for my-kafka-topic-6: 30037 ms has passed since batch creation plus linger time

at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.valueOrError(FutureRecordMetadata.java:94)

at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:64)

at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:29)

at com.tos.kafka.MyKafkaProducer.runProducer(MyKafkaProducer.java:97)

at com.tos.kafka.MyKafkaProducer.main(MyKafkaProducer.java:18)

Caused by: org.apache.kafka.common.errors.TimeoutException: Expiring 1 record(s) for my-kafka-topic-6: 30037 ms has passed since batch creation plus linger time.

Solution:

Update the following in all the server properties: /opt/kafka/config/server.properties

```
# listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://tos.master.com:9093

# Hostname and port the broker will advertise to producers and consumers. If not
# set,
# it uses the value for "listeners" if configured. Otherwise, it will use the v
# value
# 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.8A2li7O4/connect/connect.properties

Update localhost to resolve to the ip in /etc/hosts.

In case the hostname doesn't started, updated with ip address and restart the broker.

8. 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-0/00000000000000000000.log | head -n 4
```

```
[root@tos test-topic-0]# more 00000000000000000000.log
[root@tos test-topic-0]# cd ../
[root@tos kafka-logs]# cd my-kafka-connect-0/
[root@tos my-kafka-connect-0]# ls
00000000000000000000.index      0000000000000000000011.snapshot
00000000000000000000.log       leader-epoch-checkpoint
00000000000000000000.timeindex
[root@tos my-kafka-connect-0]# more *log
\██████████afka Connector.--More--(53%)

[root@tos my-kafka-connect-0]# pwd
/tmp/kafka-logs/my-kafka-connect-0
[root@tos my-kafka-connect-0]# /opt/kafka/bin/kafka-run-class.sh kafka.tools.Dum
pLogSegments --deep-iteration --print-data-log --files \
> /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log | head -n 4
Dumping /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log
Starting offset: 0
offset: 0 position: 0 CreateTime: 1530552634675 invalid: 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 invalid: 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 through a programmable interface that happens to take json

as an input via a Kafka queue and you can command him to perform different fighting moves in different martial arts styles. A Jackie Chan command might look like this:

```
{
  "timestamp": "2015-05-20T22:05:44.789Z",
  "style": "DRUNKEN_BOXING",
  "action": "PUNCH",
  "weapon": "CHAIR",
  "target": "ARMS",
  "strength": 8.3433
}
```

[view rawexampleJackieChanCommand.json](#) hosted with [by GitHub](#)

Now, we want to have some fun with our awesome Jackie Chan robot, so we are going to make him do random moves using our json-data-generator! First we need to define a Simulation Config and then a Workflow that Jackie will use.

SIMULATION CONFIG

Let's take a look at our example Simulation Config:

```
{
  "workflows": [{
    "workflowName": "jackieChan",
    "workflowFilename": "jackieChanWorkflow.json"
  }],
  "producers": [{
    "type": "kafka",
    "broker.server": "192.168.59.103",
```

```

    "broker.port": 9092,
    "topic": "jackieChanCommand",
    "flatten": false,
    "sync": false
  }
}

```

[view rawjackieChanSimConfig.json](#) hosted with [by GitHub](#)

As you can see, there are two main parts to the Simulation Config. The Workflows name and list the workflow configurations you want to use. The Producers are where the Generator will send the events to. At the time of writing this, we have three supported Producers:

- A Logger that sends events to log files
- A [Kafka](#) Producer that will send events to your specified Kafka Broker
- A [Tranquility](#) Producer that will send events to a [Druid](#) cluster.

You can find the full configuration options for each on the [github](#) page. We used a Kafka producer because that is how you command our Jackie Chan robot.

WORKFLOW CONFIG

The Simulation Config above specifies that it will use a Workflow called `jackieChanWorkflow.json`. This is where the meat of your configuration would live. Let's take a look at the example Workflow config and see how we are going to control Jackie Chan:

```

{
  "eventFrequency": 400,
  "varyEventFrequency": true,
  "repeatWorkflow": true,
  "timeBetweenRepeat": 1500,
  "varyRepeatFrequency": true,
  "steps": [{
    "config": [{
      "timestamp": "now()",
      "style": "random('KUNG_FU','WUSHU','DRUNKEN_BOXING')",
      "action": "random('KICK','PUNCH','BLOCK','JUMP')",
      "weapon": "random('BROAD_SWORD','STAFF','CHAIR','ROPE')",
      "target": "random('HEAD','BODY','LEGS','ARMS')",
      "strength": "double(1.0,10.0)"
    }
  ],
  "duration": 0
}]
}

```

[view rawjackieChanWorkflow.json](#) hosted with [by GitHub](#)

The Workflow defines many things that are all defined on the github page, but here is a summary:

- At the top are the properties that define how often events should be generated and if / when this workflow should be repeated. So this is like saying we want Jackie Chan to do a martial arts move every 400 milliseconds (he's FAST!), then take a break for 1.5 seconds, and do another one.
- Next, are the Steps that this Workflow defines. Each Step has a config and a duration. The duration specifies how long to run this step. The config is where it gets interesting!

WORKFLOW STEP CONFIG

The Step Config is your specific definition of a json event. This can be any kind of json object you want. In our example, we want to generate a Jackie Chan command message that will be sent to his control unit via Kafka. So we define the command message in our config, and since we want this to be fun, we are going to randomly generate what kind of style, move, weapon, and target he will use.

You'll notice that the values for each of the object properties look a bit funny. These are special Functions that we have created that allow us to generate values for each of the properties. For instance, the “random('KICK','PUNCH','BLOCK','JUMP')” function will randomly choose one of the values and output it as the value of the “action” property in the command message. The “now()” function will output the current date in an ISO8601 date formatted string. The “double(1.0,10.0)” will generate a random double between 1 and 10 to determine the strength of the action that Jackie Chan will perform. If we wanted to, we could make Jackie Chan perform combo moves by defining a number of Steps that will be executed in order.

There are many more Functions available in the generator with everything from random string generation, counters, random number generation, dates, and even support for randomly generating arrays of data. We also support the ability to reference other randomly generated values. For more info, please check out the [full documentation](#) on the github page.

Once we have defined the Workflow, we can run it using the json-data-generator. To do this, do the following:

1. If you have not already, go ahead and [download the most recent release](#) of the json-data-generator.
2. Unpack the file you downloaded to a directory.

```
(tar -xvf json-data-generator-1.4.0-bin.tar -C /apps )
```

3. Copy your custom configs into the conf directory
4. Then run the generator like so:

```
1. java -jar json-data-generator-1.4.0.jar jackieChanSimConfig.json
```

You will see logging in your console showing the events as they are being generated. The jackieChanSimConfig.json generates events like these:

```
{ "timestamp": "2015-05-20T22:21:18.036Z", "style": "WUSHU", "action": "BLOCK", "weapon": "CHAIR", "target": "BODY", "strength": 4.7912 }
{ "timestamp": "2015-05-20T22:21:19.247Z", "style": "DRUNKEN_BOXING", "action": "PUNCH", "weapon": "BROAD_SWORD", "target": "ARMS", "strength": 3.0248 }
{ "timestamp": "2015-05-20T22:21:20.947Z", "style": "DRUNKEN_BOXING", "action": "BLOCK", "weapon": "ROPE", "target": "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_SWORD", "target": "HEAD", "strength": 4.6202 }
{ "timestamp": "2015-05-20T22:21:25.195Z", "style": "KUNG_FU", "action": "JUMP", "weapon": "ROPE", "target": "ARMS", "strength": 7.5303 }
{ "timestamp": "2015-05-20T22:21:26.492Z", "style": "DRUNKEN_BOXING", "action": "PUNCH", "weapon": "STAFF", "target": "HEAD", "strength": 1.124 }
```

```

7}
{"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":6.268
6}
{"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 [by 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.

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>