Last Updated: 30 March 2022

2 Kafka – Dev Ops

1. Producer in Java – 90 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.

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-fa
ctor 3    --partitions 13    --topic my-kafka-topic     --zookeeper  localhost:2
181
Created topic "my-kafka-topic".
[root@tos scripts]#
```

List created topics, you can verify the topic now
/opt/kafka/bin/kafka-topics.sh --list --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]#
```

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 dependency in pom.xml.

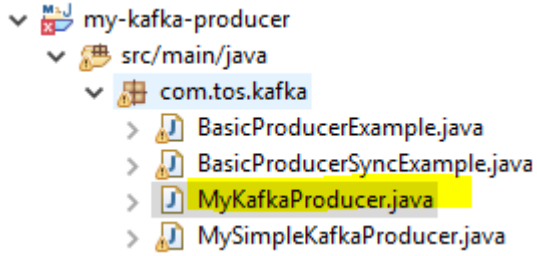
Construct a Kafka Producer

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.

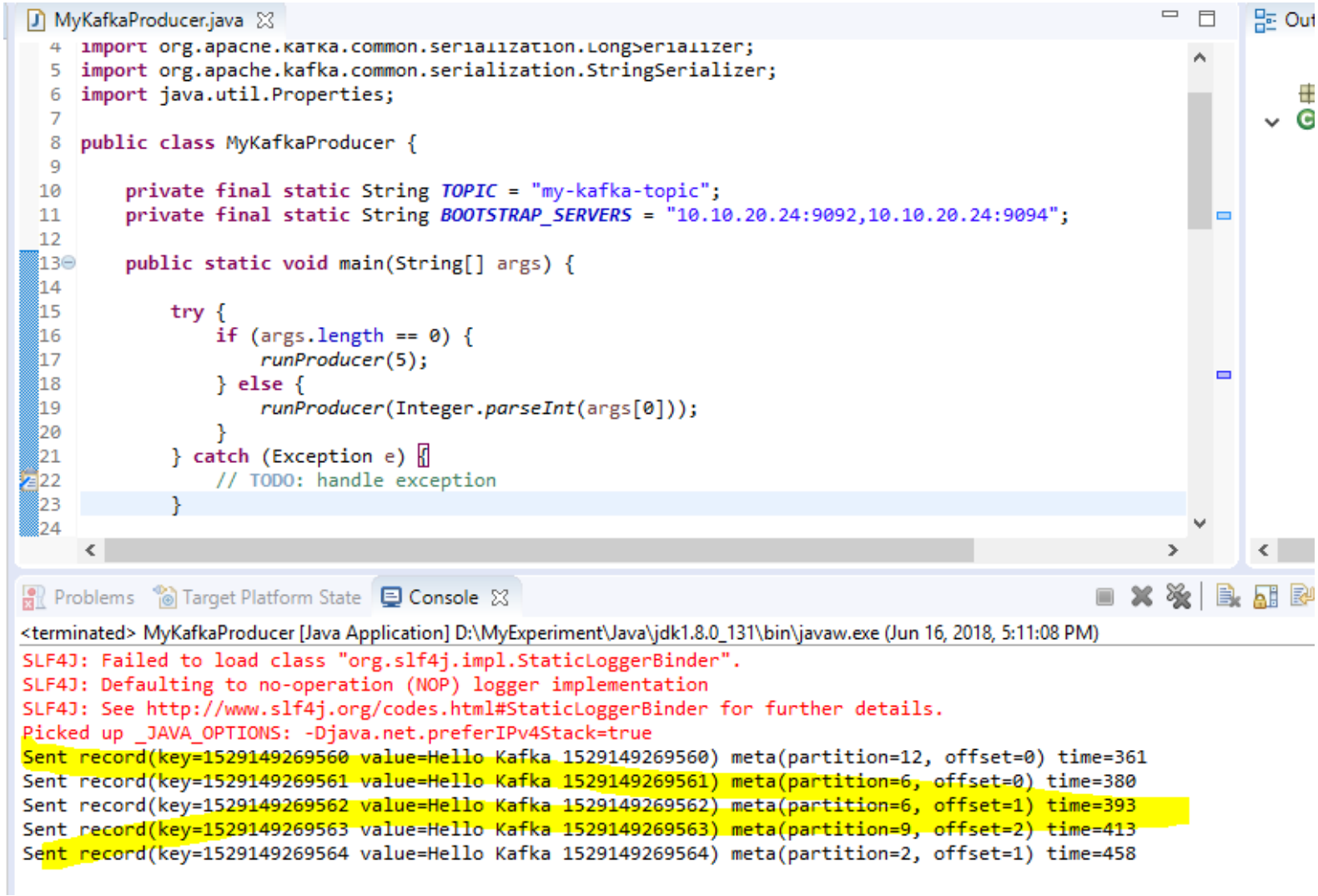
```
public static void main(String[] args) {

        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

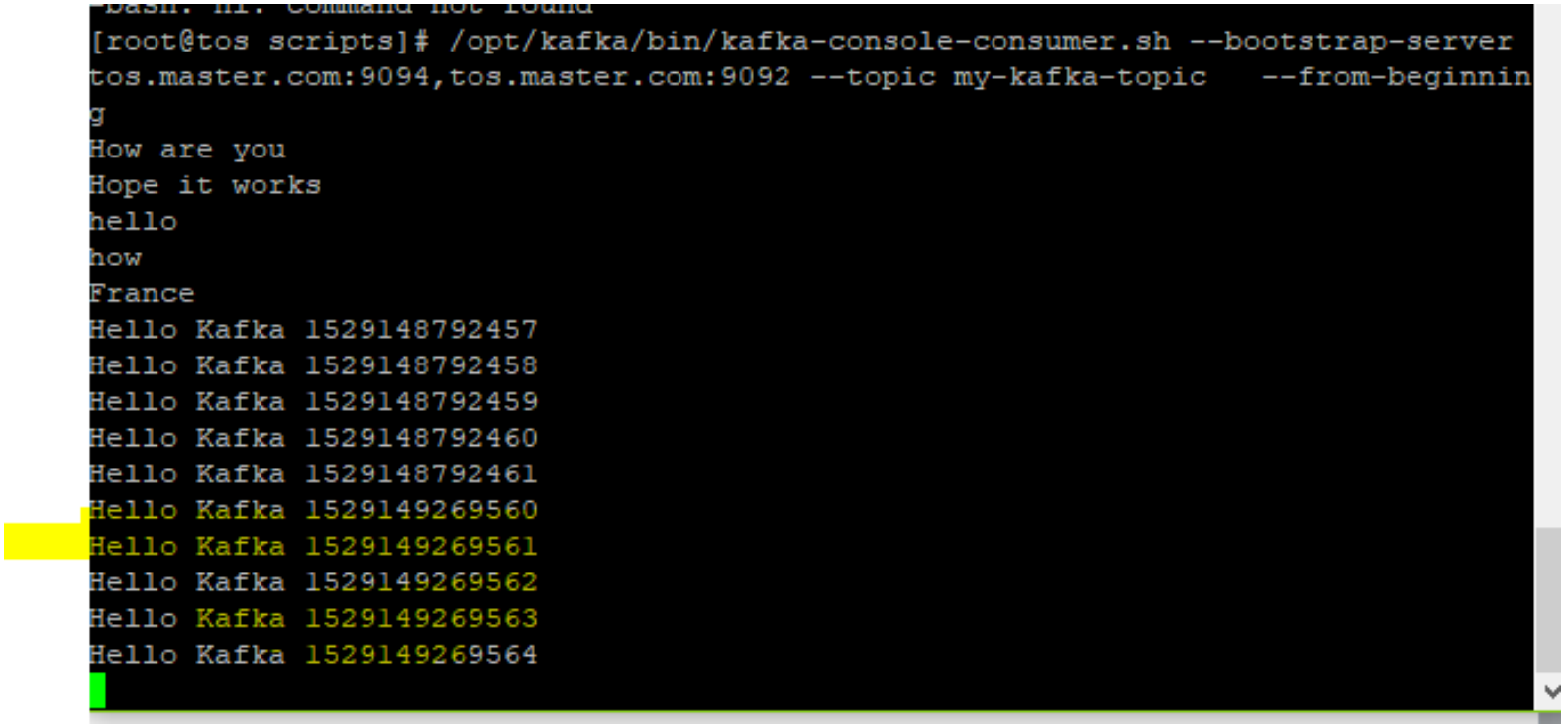
```
[root@tos scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server
tos.master.com:9094,tos.master.com:9092 --topic my-kafka-topic   --from-beginnin
g
```

Execute the main program.



You should be able to view the 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



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 – 60 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.

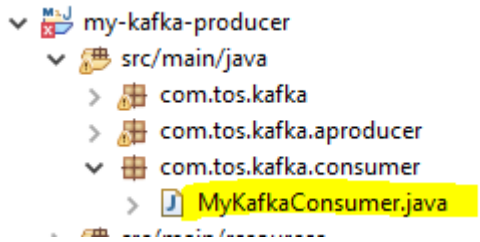
Kafka Consumer imports and constants

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;
                    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

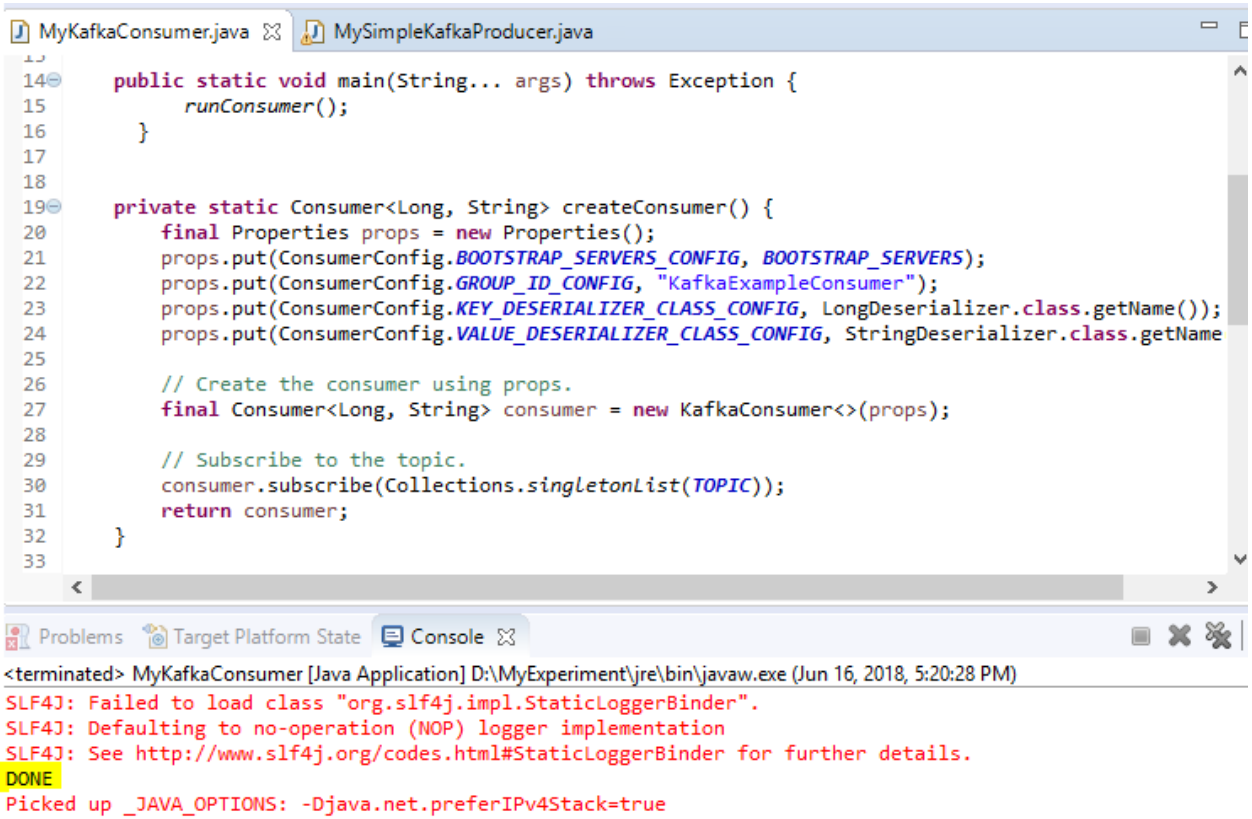
Next you define the main method.

```
public class KafkaConsumer {

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

The main method just invokes runConsumer.

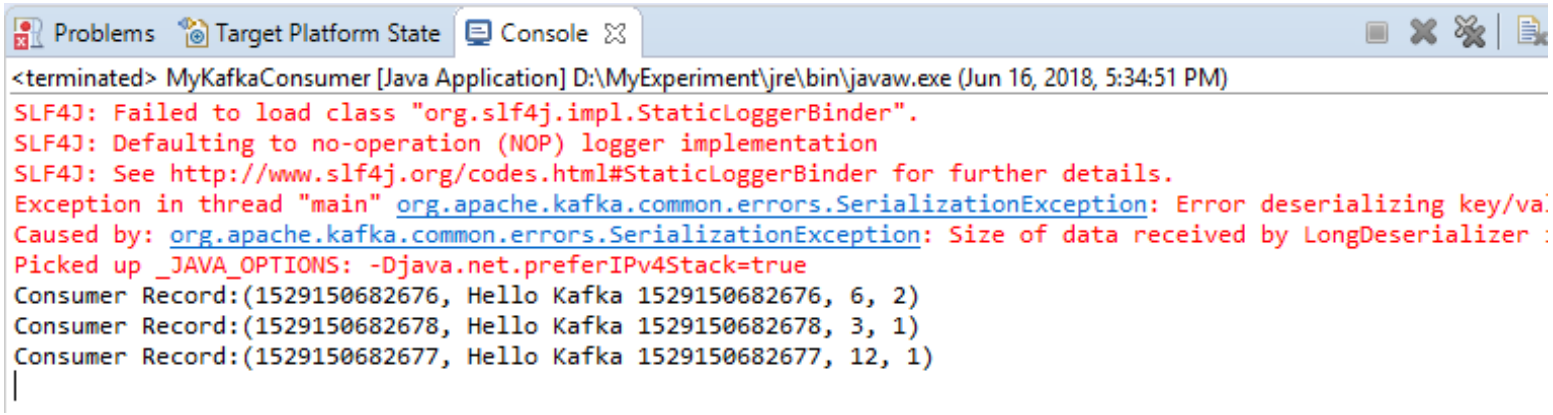
Try running the consumer program.

Run the consumer from your IDE.



As you can observer 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.



Logging set up for Kafka

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

src/main/resources/logback.xml

<configuration>

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


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

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

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

Lab End Here ----------

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

Learning Outcomes:

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

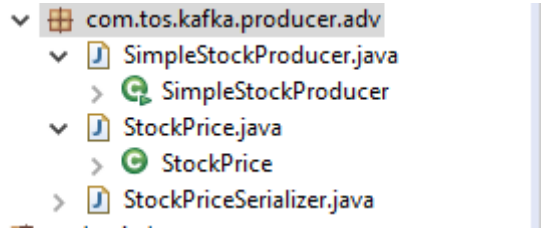
In this lab, we will be sending a JSON object using a custom serialize and in async mode. The Stock Price Producer example has the following classes:

- 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.


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 0
        props.put("linger.ms", 100);
```

```
            // The buffer.memory controls the total amount of memory available to the
            // producer for buffering.
            props.put("buffer.memory", 302);

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

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

    }
```

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

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

            // 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 ()

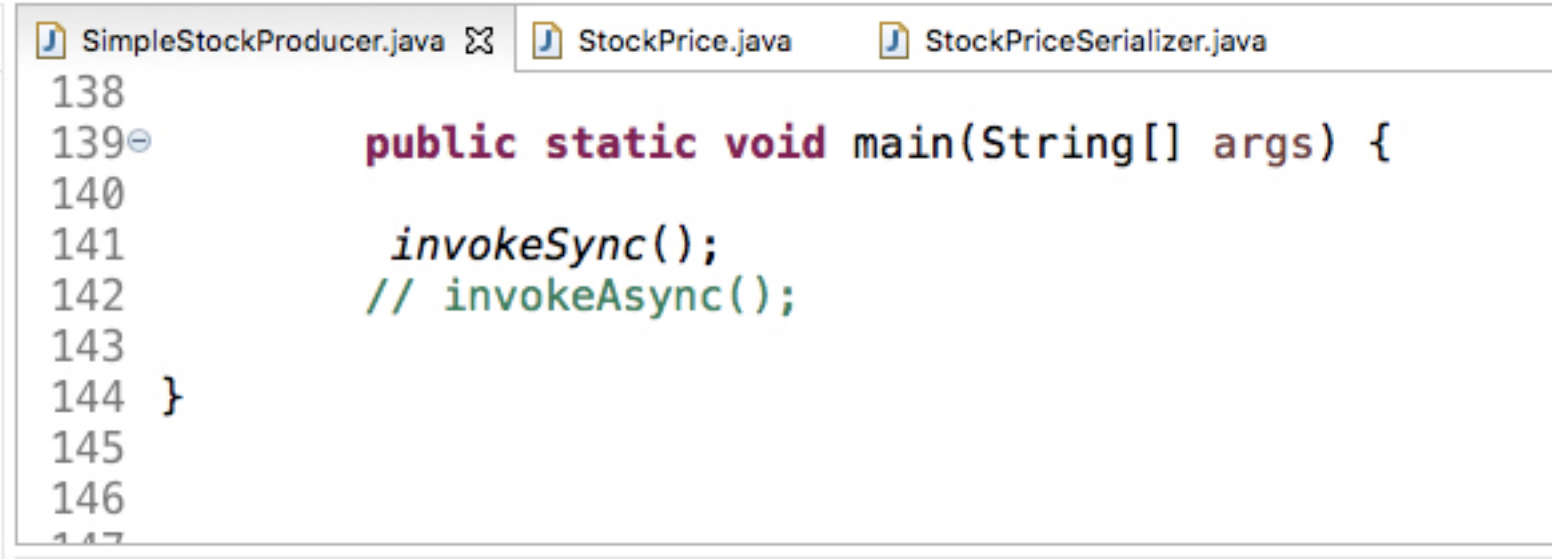
#sh start3Brokers.sh

```
[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 kafka0:9092 --replication-factor 1 --partitions 2 --topic stock-prices

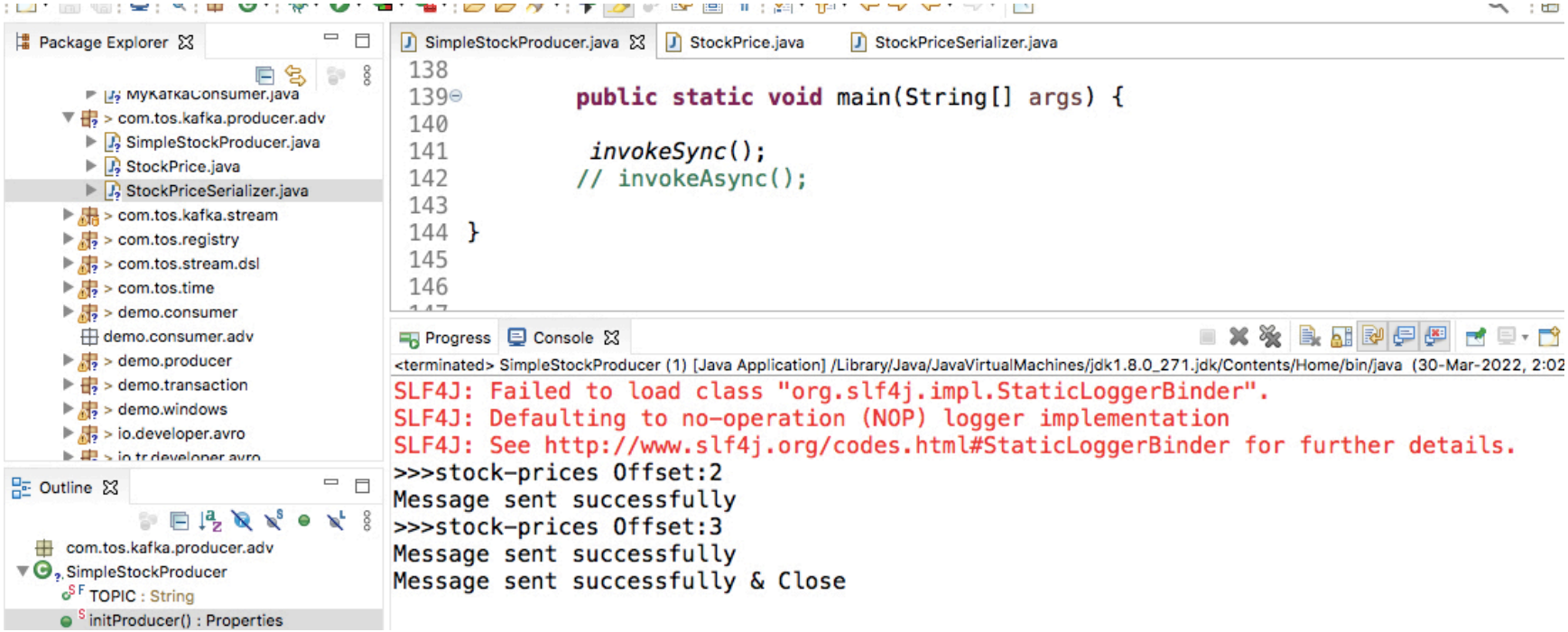
We will send the JSON message synchronously.

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



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

Execute the main program.



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

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

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

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.mast
er.com:2181 --replication-factor 2 --partitions 4 --topic mystock-prices
Created topic "mystock-prices".
```

Update the topic name as shown below.

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

We have created the 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(), stoc
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-pric
es
Topic:mystock-prices    PartitionCount:4        ReplicationFactor:2     Configs:
        Topic: mystock-prices   Partition: 0    Leader: 0       Replicas: 0,1   Isr: 0,1
        Topic: mystock-prices   Partition: 1    Leader: 1       Replicas: 1,2   Isr: 1,2
        Topic: mystock-prices   Partition: 2    Leader: 2       Replicas: 2,0   Isr: 2,0
        Topic: mystock-prices   Partition: 3    Leader: 0       Replicas: 0,2   Isr: 0,2
[root@tos scripts]#
```

Consumer:
#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server 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 Deserialzation - 60 Minutes

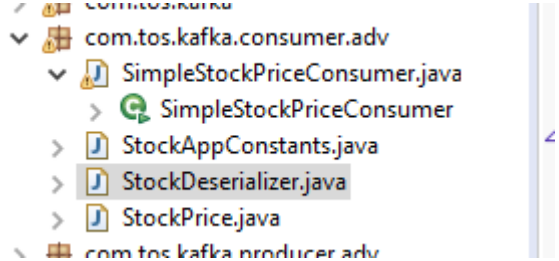
Learning outcome:

- 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.

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

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

5. 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:

```
public class WordCountDemo {

    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 accep
            }
        }
        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());

        // setting offset reset to earliest so that we can re-run the demo code with the same pre-loaded da
        // Note: To re-run the demo, you need to use the offset reset tool:
        // https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Streams+Application+Reset+Tool
        props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
        return props;
```

```
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:

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
```

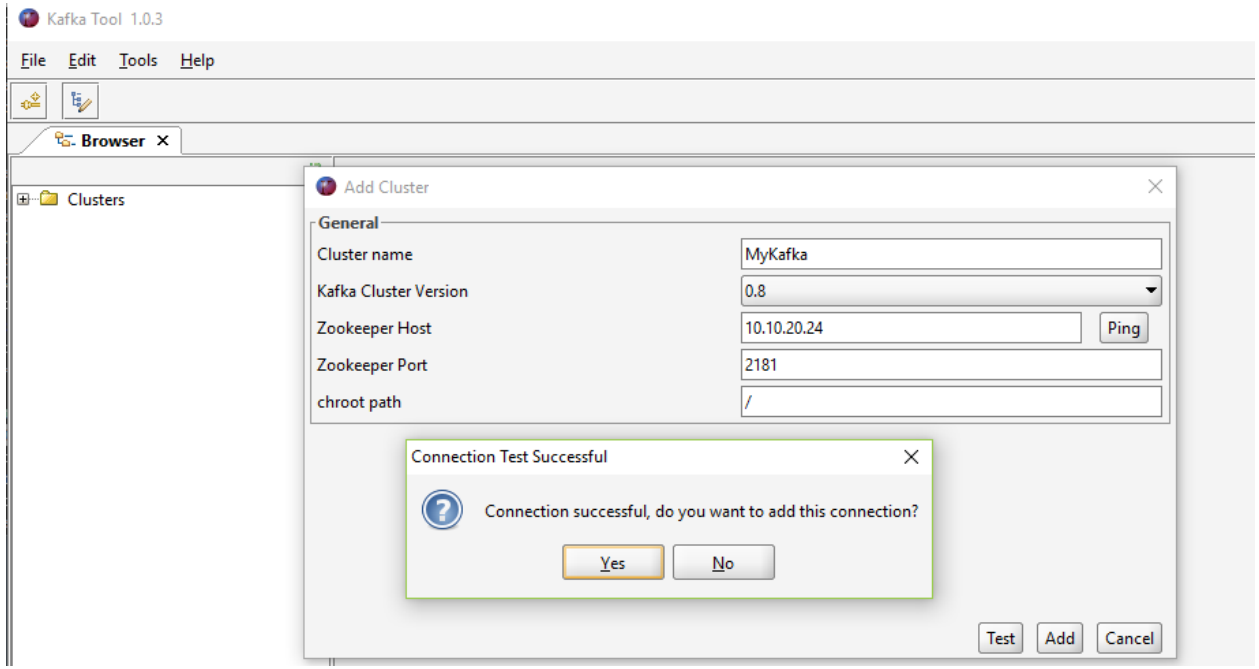
As one can see, outputs of the Wordcount application is actually a continuous stream of updates, where each output record (i.e. each line in the original output above) is an updated count of a single word, aka record key such as "kafka". For multiple records with the same key, each later record is an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

----------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
```

java.util.concurrent.ExecutionException:

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

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

at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:64)
at org.apache.kafka.clients.producer.internals.FutureRecordMetadata.get(FutureRecordMetadata.java:29)
at com.tos.kafka.MyKafkaProducer.runProducer(MyKafkaProducer.java:97)
at com.tos.kafka.MyKafkaProducer.main(MyKafkaProducer.java:18)
Caused by: org.apache.kafka.common.errors.TimeoutException: Expiring 1 record(s) for my-kafka-topic-6: 30037 ms has passed since batch creation plus linger time.

Solution:

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

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

# Hostname and port the broker will advertise to producers and consumers. If not
 set,
# it uses the value for "listeners" if configured.  Otherwise, it will use the v
alue
# returned from java.net.InetAddress.getCanonicalHostName().
advertised.listeners=PLAINTEXT://tos.master.com:9093

# Maps listener names to security protocols, the default is for them to be the s
ame. See the config documentation for more details
#listener.security.protocol.map=PLAINTEXT:PLAINTEXT,SSL:SSL,SASL_PLAINTEXT:SASL_
PLAINTEXT,SASL_SSL:SASL_SSL
```

Its should be updated with your hostname and restart the broker

Changes in the following file, if the hostname is to be changed.

//kafka/ Server.properties and control center

/apps/confluent/etc/confluent-control-center/control-center-dev.properties

/apps/confluent/etc/ksql/ksql-server.properties
/tmp/confluent.8A2Ii7O4/connect/connect.properties

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

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

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/oooooooooooooooooo.log | head -n 4

```
[root@tos test-topic-0]# more 00000000000000000000.log
[root@tos test-topic-0]# cd ../
[root@tos kafka-logs]# cd my-kafka-connect-0/
[root@tos my-kafka-connect-0]# ls
00000000000000000000.index      00000000000000000011.snapshot
00000000000000000000.log        leader-epoch-checkpoint
00000000000000000000.timeindex
[root@tos my-kafka-connect-0]# more *log
\▒▒▒▒▒▒▒▒afka Connector.--More--(53%)

[root@tos my-kafka-connect-0]# pwd
/tmp/kafka-logs/my-kafka-connect-0
[root@tos my-kafka-connect-0]# /opt/kafka/bin/kafka-run-class.sh kafka.tools.Dum
pLogSegments --deep-iteration --print-data-log --files \
> /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log | head -n 4
Dumping /tmp/kafka-logs/my-kafka-connect-0/00000000000000000000.log
Starting offset: 0
offset: 0 position: 0 CreateTime: 1530552634675 isvalid: true keysize: -1 values
ize: 31 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -1 sequence:
-1 isTransactional: false headerKeys: [] payload: This Message is from Test File
.
offset: 1 position: 0 CreateTime: 1530552634677 isvalid: true keysize: -1 values
ize: 43 magic: 2 compresscodec: NONE producerId: -1 producerEpoch: -1 sequence:
-1 isTransactional: false headerKeys: [] payload: It will be consumed by the Kaf
ka Connector.
```

3. Data Generator – JSON

Streaming Json Data Generator

Downloading the generator

You can always find the most recent release over on github where you can download the bundle file that contains the runnable application and example configurations. Head there now and download a release to get started!

Configuration

The generator runs a Simulation which you get to define. The Simulation can specify one or many Workflows that will be run as part of your Simulation. The Workflows then generates Events and these Events are then sent somewhere. You will also need to define Producers that are used to send the Events generated by your Workflows to some destination. These destinations could be a log file, or something more complicated like a Kafka Queue.

You define the configuration for the json-data-generator using two configuration files. The first is a Simulation Config. The Simulation Config defines the Workflows that should be run and different Producers that events should be sent to. The second is a Workflow configuration (of which you can have multiple). The Workflow defines the frequency of Events and Steps that the Workflow uses to generate the Events. It is the Workflow that defines the format and content of your Events as well.

For our example, we are going to pretend that we have a programmable Jackie Chan robot. We can command Jackie Chan though a programmable interface that happens to take json

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

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

view rawexampleJackieChanCommand.json hosted with by GitHub

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

SIMULATION CONFIG

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

```
{
   "workflows": [{
        "workflowName": "jackieChan",
        "workflowFilename": "jackieChanWorkflow.json"
     }],
```

```
    "producers": [{
      "type": "kafka",
      "broker.server": "192.168.59.103",
      "broker.port": 9092,
      "topic": "jackieChanCommand",
      "flatten": false,
      "sync": false


    }]
  }
```

view rawjackieChanSimConfig.json hosted with by **GitHub**

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

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

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

WORKFLOW CONFIG

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

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

view rawjackieChanWorkflow.json hosted with by **GitHub**

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

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

WORKFLOW STEP CONFIG

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

You'll notice that the values for each of the object properties look a bit funny. These are special Functions that we have created that allow us to generate values for each of the properties. For instance, the "random('KICK','PUNCH','BLOCK','JUMP')" function will randomly choose one of the values and output it as the value of the "action" property in the command message. The "now()" function will output the current date in an ISO8601 date formatted string. The "double(1.0,10.0)" will generate a random double between 1 and 10

to determine the strength of the action that Jackie Chan will perform. If we wanted to, we could make Jackie Chan perform combo moves by defining a number of Steps that will be executed in order.

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

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

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

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

3. Copy your custom configs into the conf directory
4. Then run the generator like so:
    1. java -jar json-data-generator-1.4.0.jar jackieChanSimConfig.json

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

{"timestamp":"2015-05-20T22:21:18.036Z","style":"WUSHU","action":"BLOCK","weapon":"CHAIR","target":"BODY","strength":4.7912}
{"timestamp":"2015-05-20T22:21:19.247Z","style":"DRUNKEN_BOXING","action":"PUNCH","weapon":"BROAD_SWORD","target":"ARMS","strength":3.0248}
{"timestamp":"2015-05-20T22:21:20.947Z","style":"DRUNKEN_BOXING","action":"BLOCK","weapon":"ROPE","target":"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":1.1247}
{"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 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.

9. 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/</sourceDire
ctory>

<sourceDirectory>${project.basedir}/src/main/resources/avro</source
Directory>
              <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>
```

10. 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