

Pre-requisite	2
1. DataGenerator	3
2. Producer in Java – 90 Minutes	6
3. Consumer – Java API – 60 Minutes	21
4. Kafkatools.....	31
5. Logging.....	32
6. Errors	33
1. LEADER_NOT_AVAILABLE.....	33
java.util.concurrent.ExecutionException:	33
7. Annexure Code:.....	36
2. DumplogSegment.....	36
3. Data Generator – JSON	37
8. Pom.xml (Standalone)	45

Last Updated: 20 March 2023

JDK : Jdk : 1.11

Apache kafka : kafka_2.13-3.2.1.tar

Pre-requisite

Create a Java Maven Project and replace the pom.xml content from the Annexure.

Maven java project Details:

Group ID: com.tos.kafka

Artifact ID: my-kafka-producer

You need to start zookeeper and kafka broker node/s 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. You can have one Kafka node also.

Install confluent Kafka

Registry and confluent CLI

1. DataGenerator

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.

2. Producer in Java – 90 Minutes

Learning outcome:

- Sending Message Synchronously
- Understand 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.

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; my-kafka-topic with 13 partitions and a replication factor of 3/1. Then we list the Kafka topics.

7 Kafka – Development Java API

```
[root@kafka0 scripts]# /opt/kafka/bin/kafka-topics.sh --create --replication-factor 1 --partitions 13 --topic my-kafka-topic --bootstrap-server kafka0:9092
Created topic my-kafka-topic.
[root@kafka0 scripts]#
```

List created topics, you can verify the topic using the following command

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

```
[root@kafka0 scripts]# /opt/kafka/bin/kafka-topics.sh --list --bootstrap-server kafka0:9092
my-kafka-topic
[root@kafka0 scripts]#
```

We will use maven to create the java project.

Construct a Kafka Producer

To create a Kafka producer, you will need to pass 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

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 java.util.Properties;  
  
import org.apache.kafka.clients.producer.KafkaProducer;  
import org.apache.kafka.clients.producer.Producer;
```



```
import org.apache.kafka.clients.producer.ProducerConfig;  
import org.apache.kafka.clients.producer.ProducerRecord;  
import org.apache.kafka.clients.producer.RecordMetadata;  
import org.apache.kafka.common.serialization.LongSerializer;  
import org.apache.kafka.common.serialization.StringSerializer;
```

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 `kafkao:9092`, `kafkao:9093`, `kafkao: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 =
    "kafka0:9092,kafka0:9093,kafka0:9094";

```

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 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) {  
        e.printStackTrace();  
    }  
}
```

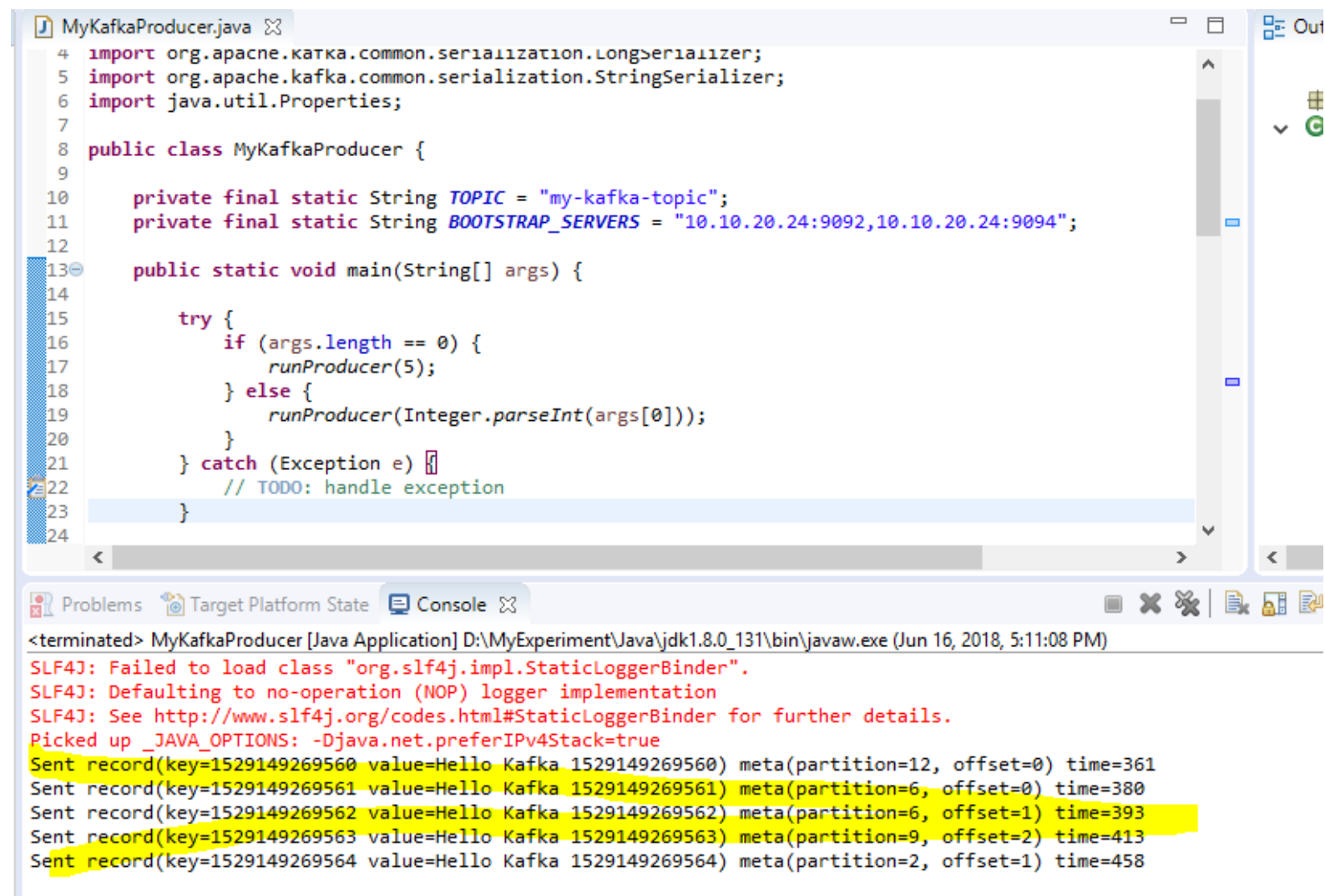
The **main** method just calls **runProducer**.

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

```
#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic my-kafka-topic --from-beginning
```

```
[root@kafka0 scripts]#  
[root@kafka0 scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic my-kafka-topic --from-beginning
```

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's execution, including SLF4J warnings and five successful record sends.

```

1  MyKafkaProducer.java
2
3  4 import org.apache.kafka.common.serialization.LongSerializer;
4  5 import org.apache.kafka.common.serialization.StringSerializer;
5  6 import java.util.Properties;
6  7
7  8 public class MyKafkaProducer {
8  9
9  10     private final static String TOPIC = "my-kafka-topic";
10 11     private final static String BOOTSTRAP_SERVERS = "10.10.20.24:9092,10.10.20.24:9094";
11 12
12 13     public static void main(String[] args) {
13 14
14 15         try {
15 16             if (args.length == 0) {
16 17                 runProducer(5);
17 18             } else {
18 19                 runProducer(Integer.parseInt(args[0]));
19 20             }
20 21         } catch (Exception e) {
21 22             // TODO: handle exception
22 23         }
23 24
24

```

Problems Target Platform State Console

<terminated> MyKafkaProducer [Java Application] D:\MyExperiment\Java\jdk1.8.0_131\bin\javaw.exe (Jun 16, 2018, 5:11:08 PM)

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
 SLF4J: Defaulting to no-operation (NOP) logger implementation
 SLF4J: See <http://www.slf4j.org/codes.html#StaticLoggerBinder> for further details.
 Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true
 Sent record(key=1529149269560 value=Hello Kafka 1529149269560) meta(partition=12, offset=0) time=361
 Sent record(key=1529149269561 value=Hello Kafka 1529149269561) meta(partition=6, offset=0) time=380
 Sent record(key=1529149269562 value=Hello Kafka 1529149269562) meta(partition=6, offset=1) time=393
 Sent record(key=1529149269563 value=Hello Kafka 1529149269563) meta(partition=9, offset=2) time=413
 Sent record(key=1529149269564 value=Hello Kafka 1529149269564) meta(partition=2, offset=1) time=458

You should be able to view the messages as shown above.

You can also verify from the consumer console that whatever messages sent from the producer gets displayed in the consumer console as shown below.

If not open, the consumer console before.

```
#/opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic my-kafka-topic --from-beginning
```

```
[root@kafka0 scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic my-kafka-topic --from-beginning
Hello Kafka 1679201928593
Hello Kafka 1679201928594
Hello Kafka 1679201928595
Hello Kafka 1679201928596
Hello Kafka 1679201928597
```

Let us add a method that send messages in an Async with a callback. The following send message and print the offset along with the partition ID in the console.

```
public static void sendAsync() {
    final Producer<Long, String> producer =
createProducer();
    for (int i = 0; i < 2; i++) {
        long time = System.currentTimeMillis();
        /*
```

```

        * Print the Offset of the message in async mode.
        */
        producer.send(new ProducerRecord<Long,
String>(TOPIC,time , "Seding Async : " + Long.toString(time)),
(rmd, ex) -> {
            System.out.println(" Return Async
Offset: " + rmd.offset() + " partition : " + rmd.partition());
        } );
        System.out.print("Sent : " + time);
        producer.flush();
    }
    producer.close();
}

```

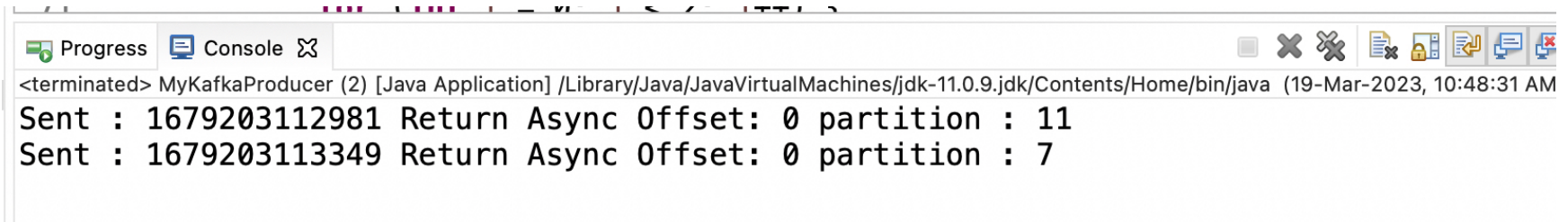
Invoke and Execute the above method from the main.

```

    public static void main(String[] args) {
        try {
            if (args.length == 0) {
                //runProducer(5);
                sendAsync();
            } else {
                runProducer(Integer.parseInt(args[0]));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

Comment the runProducer Method.



```

Progress Console
<terminated> MyKafkaProducer (2) [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.9.jdk/Contents/Home/bin/java (19-Mar-2023, 10:48:31 AM)
Sent : 1679203112981 Return Async Offset: 0 partition : 11
Sent : 1679203113349 Return Async Offset: 0 partition : 7

```

As shown above, the first message get inserted in the offset 0 of partition 11. Check what partition number gets displayed in case of your execution.

You should be able to consume in the consumer console too.

```
[root@kafka0 scripts]# /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server kafka0:9092 --topic my-kafka-topic --from-beginning
Hello Kafka 1679201928593
Hello Kafka 1679201928594
Hello Kafka 1679201928595
Hello Kafka 1679201928596
Hello Kafka 1679201928597
Seding Async : 1679203054368
Seding Async : 1679203054829
```

In the next lab we will consume these messages from a Java client.

Lab End here

3. 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.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 =
        "kafka0:9092";
```

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 `kafka0:9092,kafka0:9093,kafka0:9094` 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

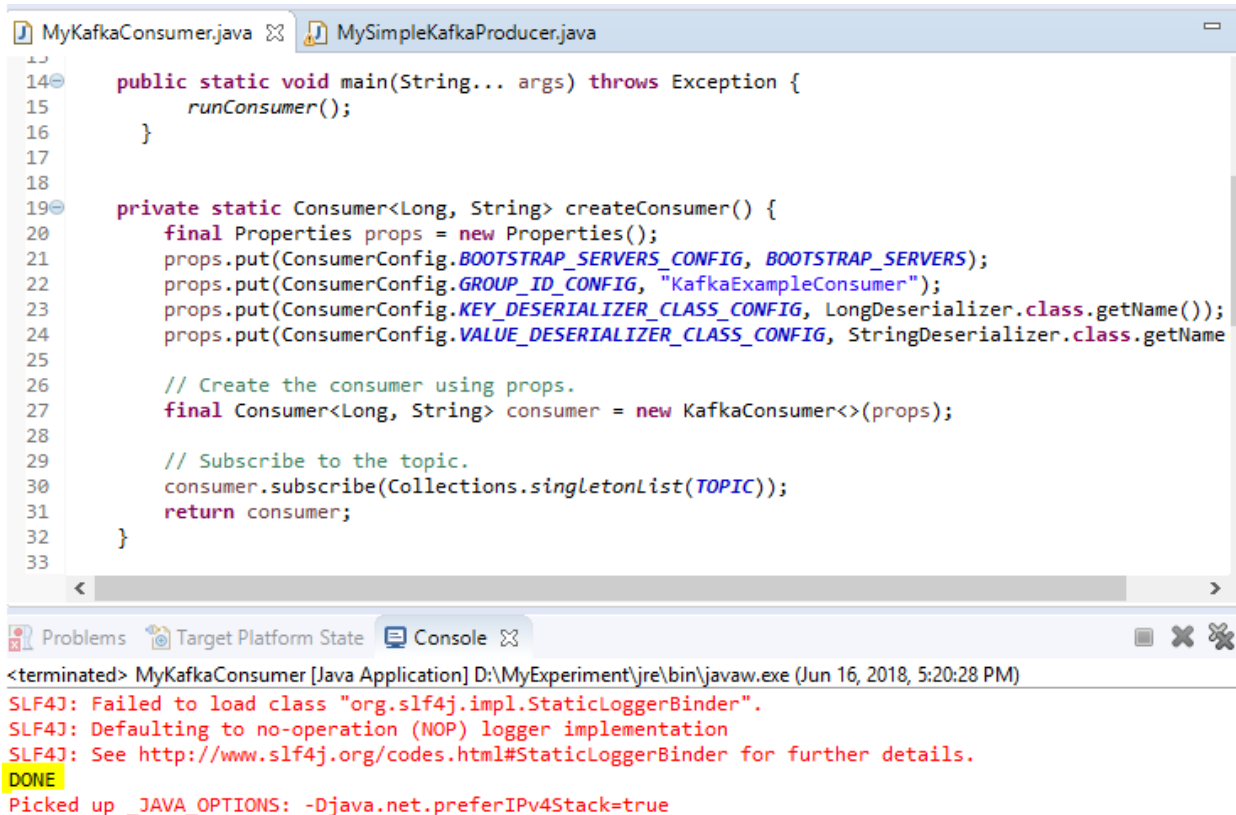
Next you define the `main` method.

```
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.



The screenshot shows an IDE with two tabs: `MyKafkaConsumer.java` and `MySimpleKafkaProducer.java`. The `MyKafkaConsumer.java` tab is active, displaying the following code:

```

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

```

Below the code editor, the `Console` tab is active, showing the following output:

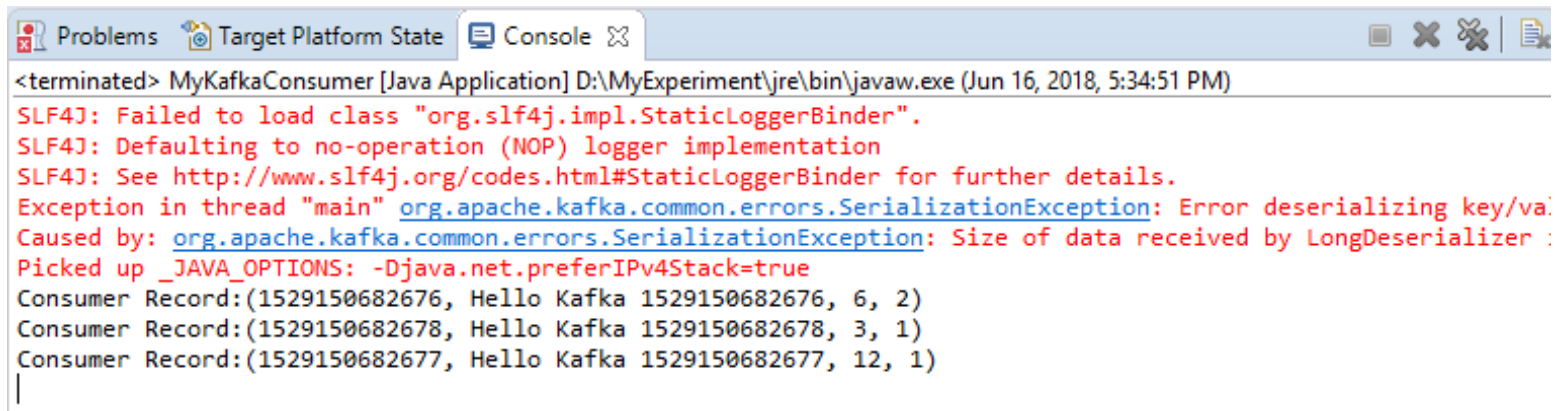
```

<terminated> MyKafkaConsumer [Java Application] D:\MyExperiment\jre\bin\javaw.exe (Jun 16, 2018, 5:20:28 PM)
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
DONE
Picked up _JAVA_OPTIONS: -Djava.net.preferIPv4Stack=true

```

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

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



```

Problems Target Platform State Console
<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 `slf4j`. 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

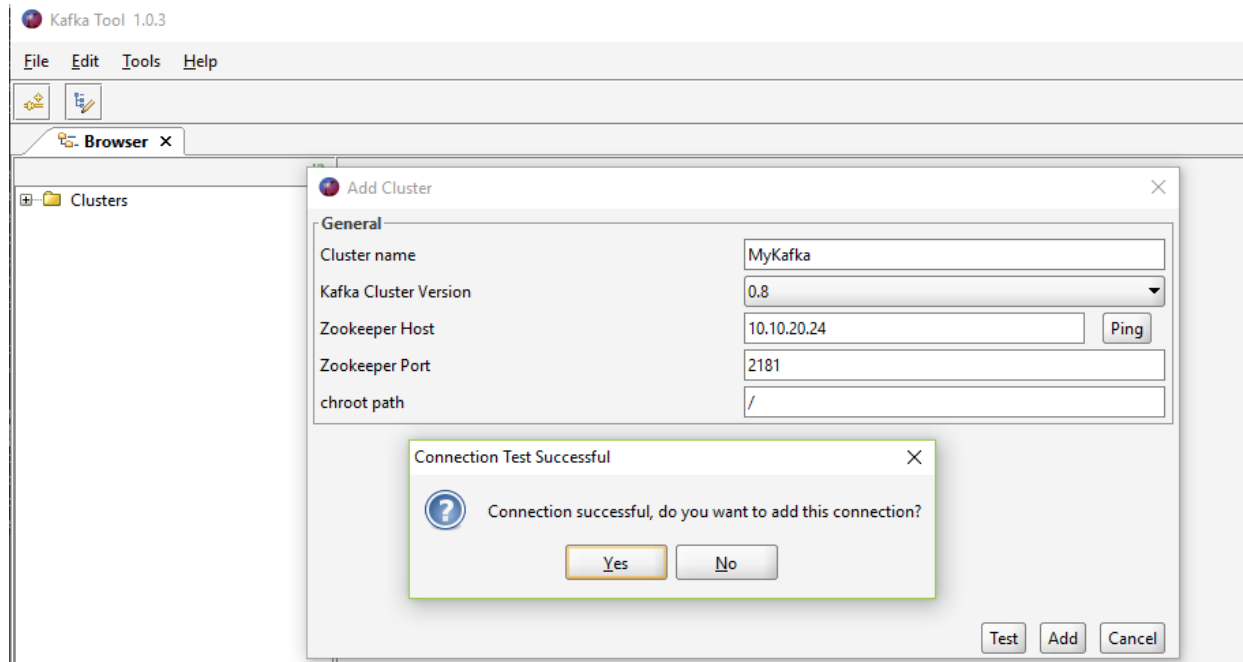
```
<configuration>
  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
    </encoder>
  </appender>

  <root level="ERROR">
    <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 -----

4. Kafkatools



5. Logging

Update pom.xml

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-core</artifactId>
  <version>1.2.6</version>
</dependency>
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.2.6</version>
</dependency>
```

Include a resource setting

6. 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.8A2Ii7O4/connect/connect.properties
```

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

In case the hostname doesn't start, update with ip address and restart the broker.

7. 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:

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

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

7. Copy your custom configs into the conf directory
8. 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.

8. 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>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-core</artifactId>
        <version>1.2.6</version>
    </dependency>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.6</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/avro</so
sourceDirectory>

                    <includes>
                        <include>*.avsc</include>
                    </includes>

                    <outputDirectory>${project.basedir}/src/main/java</outputDirect
ory>

```

```
        </configuration>  
    </execution>  
</executions>  
</plugin>  
</plugins>  
</build>  
</project>
```

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>