Lab 3: Writing a Kafka Consumer in Java

Welcome to the session 3 lab. The work for this lab is done in `~/kafka-training/lab3`. In this lab, you are going to create simple Java Kafka consumer. Please refer to the Kafka course notes for any updates or changes to this lab. The latest version of this lab lives here.

In this lab, you are going to create a simple *Kafka Consumer*. This consumer consumes messages from the Kafka Producer you wrote in the last lab. This lab demonstrates how to process records from a *Kafka topic* with a *Kafka Consumer*.

This lab 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.

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 lab.

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

~/kafka-training/lab3/src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java

```java
package com.cloudurable.kafka;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.common.serialization.LongDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;


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

public class KafkaConsumerExample {

    private final static String TOPIC = "my-example-topic";
    private final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";
    ...
}
```

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

ACTION **- EDIT src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java and add the constants above.**

Create Kafka Consumer using 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

~/kafka-training/lab3/src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java

```
public class KafkaConsumerExample {
  ...

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

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

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

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

Above `KafkaConsumerExample.createConsumer` sets the `BOOTSTRAP_SERVERS_CONFIG` ("bootstrap.servers") property to the list of broker addresses we defined earlier. `BOOTSTRAP_SERVERS_CONFIG` value is a comma separated list of host/port pairs that the `Consumer` uses to establish an initial connection to the Kafka cluster. Just like the producer, the consumer uses of all servers in the cluster no matter which ones we list here.

The `GROUP_ID_CONFIG` identifies the consumer group of this consumer.

The `KEY_DESERIALIZER_CLASS_CONFIG` ("key.deserializer") is a Kafka Deserializer class for Kafka record keys that implements the Kafka Deserializer interface. Notice that we set this to `LongDeserializer` as the message ids in our example are longs.

The `VALUE_DESERIALIZER_CLASS_CONFIG` ("value.deserializer") is a Kafka Serializer class for Kafka record values that implements the Kafka Deserializer interface. Notice that we set this to `StringDeserializer` as the message body in our example are strings.

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

ACTION - EDIT src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java and finish the createConsumer method.

Process messages from Kafka with Consumer

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

KafkaConsumerExample.java - Process records from Consumer

~/kafka-training/lab3/src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java

```
public class KafkaConsumerExample {
  ...


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

        final int giveUp = 100;   int noRecordsCount = 0;

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

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

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

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

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

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

ACTION - EDIT src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java and finish the runConsumer method.

ACTION - RUN src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java from the IDE.

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 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.

ACTION - EDIT src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java and edit the createConsumer method to add `props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100)`.

ACTION - RE RUN Producer from last lab to create some more records.

ACTION - RUN src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java from the IDE.

Running the Kafka Consumer

Next you define the `main` method.

KafkaConsumerExample.java - Running the Consumer

~/kafka-training/lab3/src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java

```java
public class KafkaConsumerExample {

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

The `main` method just calls `runConsumer`.

Beyond this point we don't call out actions as most of the code is running. You will make small changes to the code and rerun the producer or consumer (or both). Follow along.

Try running the consumer and producer

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

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 gradle build (`compile 'ch.qos.logback:logback-classic:1.2.2'`).

~/kafka-training/lab3/solution/src/main/resources/logback.xml

```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.

Try this: Three Consumers in same group and one Producer sending 25 messages

Run the consumer example three times from your IDE. Then change Producer to send 25 records instead of 5. Then run the producer once from your IDE. What happens? The consumers should share the messages.

Producer Output

```
sent record(key=1495048417121 value=..) meta(partition=6, offset=16) time=118
sent record(key=1495048417131 value=..) meta(partition=6, offset=17) time=120
sent record(key=1495048417133 value=..) meta(partition=12, offset=17) time=120
sent record(key=1495048417140 value=..) meta(partition=12, offset=18) time=121
sent record(key=1495048417143 value=..) meta(partition=12, offset=19) time=121
sent record(key=1495048417123 value=..) meta(partition=0, offset=19) time=121
sent record(key=1495048417126 value=..) meta(partition=0, offset=20) time=121
sent record(key=1495048417134 value=..) meta(partition=0, offset=21) time=122
sent record(key=1495048417122 value=..) meta(partition=3, offset=19) time=122
sent record(key=1495048417127 value=..) meta(partition=3, offset=20) time=122
sent record(key=1495048417139 value=..) meta(partition=3, offset=21) time=123
sent record(key=1495048417142 value=..) meta(partition=3, offset=22) time=123
sent record(key=1495048417136 value=..) meta(partition=10, offset=19) time=127
sent record(key=1495048417144 value=..) meta(partition=1, offset=26) time=128
sent record(key=1495048417125 value=..) meta(partition=5, offset=22) time=128
sent record(key=1495048417138 value=..) meta(partition=5, offset=23) time=128
sent record(key=1495048417128 value=..) meta(partition=8, offset=21) time=129
sent record(key=1495048417124 value=..) meta(partition=11, offset=18) time=129
sent record(key=1495048417130 value=..) meta(partition=11, offset=19) time=129
sent record(key=1495048417132 value=..) meta(partition=11, offset=20) time=130
sent record(key=1495048417141 value=..) meta(partition=11, offset=21) time=130
sent record(key=1495048417145 value=..) meta(partition=11, offset=22) time=131
sent record(key=1495048417129 value=..) meta(partition=2, offset=24) time=132
sent record(key=1495048417135 value=..) meta(partition=2, offset=25) time=132
sent record(key=1495048417137 value=..) meta(partition=2, offset=26) time=132
```

Notice the producer sends 25 messages.

Consumer 0 in same group

```
Consumer Record:(1495048417121, Hello Mom 1495048417121, 6, 16)
Consumer Record:(1495048417131, Hello Mom 1495048417131, 6, 17)
Consumer Record:(1495048417125, Hello Mom 1495048417125, 5, 22)
Consumer Record:(1495048417138, Hello Mom 1495048417138, 5, 23)
Consumer Record:(1495048417128, Hello Mom 1495048417128, 8, 21)
```

Consumer 1 in same group

```
Consumer Record:(1495048417123, Hello Mom 1495048417123, 0, 19)
Consumer Record:(1495048417126, Hello Mom 1495048417126, 0, 20)
Consumer Record:(1495048417134, Hello Mom 1495048417134, 0, 21)
Consumer Record:(1495048417144, Hello Mom 1495048417144, 1, 26)
Consumer Record:(1495048417122, Hello Mom 1495048417122, 3, 19)
Consumer Record:(1495048417127, Hello Mom 1495048417127, 3, 20)
Consumer Record:(1495048417139, Hello Mom 1495048417139, 3, 21)
Consumer Record:(1495048417142, Hello Mom 1495048417142, 3, 22)
Consumer Record:(1495048417129, Hello Mom 1495048417129, 2, 24)
Consumer Record:(1495048417135, Hello Mom 1495048417135, 2, 25)
Consumer Record:(1495048417137, Hello Mom 1495048417137, 2, 26)
```

Consumer 2 in same group

```
Consumer Record:(1495048417136, Hello Mom 1495048417136, 10, 19)
Consumer Record:(1495048417133, Hello Mom 1495048417133, 12, 17)
Consumer Record:(1495048417140, Hello Mom 1495048417140, 12, 18)
Consumer Record:(1495048417143, Hello Mom 1495048417143, 12, 19)
Consumer Record:(1495048417124, Hello Mom 1495048417124, 11, 18)
Consumer Record:(1495048417130, Hello Mom 1495048417130, 11, 19)
Consumer Record:(1495048417132, Hello Mom 1495048417132, 11, 20)
Consumer Record:(1495048417141, Hello Mom 1495048417141, 11, 21)
Consumer Record:(1495048417145, Hello Mom 1495048417145, 11, 22)
```

Can you answer these questions?

Which consumer owns partition 10?

How many ConsumerRecords objects did Consumer 0 get?

What is the next offset from Partition 11 that Consumer 2 should get?

Why does each consumer get unique messages?

Which consumer owns partition 10?

Consumer 2 owns partition 10.

How many ConsumerRecords objects did Consumer 0 get?

3

What is the next offset from Partition 11 that Consumer 2 should get?

22

Why does each consumer get unique messages?

Each gets its share of partitions for the topic.

Try this: Three Consumers in different Consumer group and one Producer sending 5 messages

Modify the consumer, so each consumer processes will have a unique group id.

Stop all consumers and producers processes from the last run.

Then execute the consumer example three times from your IDE. Then change producer to send five records instead of 25. Then run the producer once from your IDE. What happens? The consumers should each get a copy of the messages.

First, let's modify the Consumer to make their group id unique as follows:

KafkaConsumerExample - Make the Consumer group id unique

~/kafka-training/lab3/src/main/java/com/cloudurable/kafka/KafkaConsumerExample.java

```
public class KafkaConsumerExample {

    private final static String TOPIC = "my-example-topic";
    private final static String BOOTSTRAP_SERVERS =
            "localhost:9092,localhost:9093,localhost:9094";


    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" +
                                            System.currentTimeMillis());

                                            ...
  }
...
}
```

Notice, to make the group id unique you just add `System.currentTimeMillis()` to it.

Producer Output

```
sent record(key=1495049585396 value=..) meta(partition=7, offset=30) time=134
sent record(key=1495049585392 value=..) meta(partition=4, offset=24) time=138
sent record(key=1495049585393 value=..) meta(partition=4, offset=25) time=139
sent record(key=1495049585395 value=..) meta(partition=4, offset=26) time=139
sent record(key=1495049585394 value=..) meta(partition=11, offset=25) time=140
```

Notice the producer sends 25 messages.

Consumer 0 in own group

```
Consumer Record:(1495049585396, Hello Mom 1495049585396, 7, 30)
Consumer Record:(1495049585394, Hello Mom 1495049585394, 11, 25)
Consumer Record:(1495049585392, Hello Mom 1495049585392, 4, 24)
Consumer Record:(1495049585393, Hello Mom 1495049585393, 4, 25)
Consumer Record:(1495049585395, Hello Mom 1495049585395, 4, 26)
```

Consumer 1 in unique consumer group

```
Consumer Record:(1495049585396, Hello Mom 1495049585396, 7, 30)
Consumer Record:(1495049585394, Hello Mom 1495049585394, 11, 25)
Consumer Record:(1495049585392, Hello Mom 1495049585392, 4, 24)
Consumer Record:(1495049585393, Hello Mom 1495049585393, 4, 25)
Consumer Record:(1495049585395, Hello Mom 1495049585395, 4, 26)
```

Consumer 2 in its own consumer group

```
Consumer Record:(1495049585396, Hello Mom 1495049585396, 7, 30)
Consumer Record:(1495049585394, Hello Mom 1495049585394, 11, 25)
Consumer Record:(1495049585392, Hello Mom 1495049585392, 4, 24)
Consumer Record:(1495049585393, Hello Mom 1495049585393, 4, 25)
Consumer Record:(1495049585395, Hello Mom 1495049585395, 4, 26)
```

Can you answer these questions?

Which consumer owns partition 10?

How many ConsumerRecords objects did Consumer 0 get?

What is the next offset from Partition 11 that Consumer 2 should get?

Why does each consumer get unique messages?

Which consumer owns partition 10?

They all do! Since they are all in a unique consumer group, and there is only one consumer in each group, then each consumer we ran owns all of the partitions.

How many ConsumerRecords objects did Consumer 0 get?

3

What is the next offset from Partition 11 that Consumer 2 should get?

26

Why does each consumer get the same messages?

They do because they are each in their own consumer group, and each consumer group is a subscription to the topic.

Conclusion Kafka Consumer example

You created a simple example that creates a *Kafka consumer* to consume messages from the Kafka Producer you created in the last lab. We used the replicated Kafka topic from producer lab. You created a *Kafka Consumer* that uses the topic to receive messages. The *Kafka consumer* uses the `poll` method to get N number of records.

Consumers in the same group divide up and share partitions as we demonstrated by running three consumers in the same group and one producer. Each consumer groups gets a copy of the same data. More precise, each consumer group really has a unique set of offset/partition pairs per.

Review Kafka Consumer

How did we demonstrate Consumers in a Consumer Group dividing up topic partitions and sharing them?

We ran three consumers in the same consumer group, and then sent 25 messages from the producer. We saw that each consumer owned a set of partitions.

How did we demonstrate Consumers in different Consumer Groups each getting their own offsets?

We ran three consumers each in its own unique consumer group, and then sent 5 messages from the producer. We saw that each consumer owned every partition.

How many records does poll get?

However many you set in with `props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 100);` in the properties that you pass to `KafkaConsumer`.

Does a call to poll ever get records from two different partitions?

Yes. Records come from one server but a server can be a leader for many partitions. So a server can serve up records from many partitions on poll.

Related content

- Kafka Tutorial: Kafka Producer
- Kafka Architecture
- What is Kafka?
- Kafka Topic Architecture
- Kafka Consumer Architecture
- Kafka Producer Architecture
- Kafka and Schema Registry
- Kafka and Avro
- Kafka Tutorial Slides
- Kafka from the command line
- Kafka clustering and failover basics