Before starting with an example, let's get familiar first with the common terms and some commands used in Kafka.

**Record:** Producer sends messages to Kafka in the form of records. A record is a key-value pair. It contains the topic name and partition number to be sent. Kafka broker keeps records inside topic partitions. Records sequence is maintained at the partition level. You can define the logic on which basis partition will be determined.
**Topic:** Producer writes a record on a topic and the consumer listens to it. A topic can have many partitions but must have at least one.
**Partition:** A topic partition is a unit of parallelism in Kafka, i.e. two consumers cannot consume messages from the same partition at the same time. A consumer can consume from multiple partitions at the same time.
**Offset:** A record in a partition has an offset associated with it. Think of it like this: partition is like an array; offsets are like indexs.
**Producer:** Creates a record and publishes it to the broker.
**Consumer:** Consumes records from the broker.
**Commands:** In Kafka, a setup directory inside the bin folder is a script (kafka-topics.sh), using which, we can create and delete topics and check the list of topics. Go to the Kafka home directory.

- **Start Zookeeper** - bin/./zookeeper-server-start.sh config/zookeeper.properties
- **Start Kafka** - bin/./kafka-server-start.sh config/server.properties
- Execute this command to see the list of all topics.
  - **bin/windows/kafka-topics.bat --list --zookeeper localhost:2181 .**
  - *localhost:2181* is the Zookeeper address that we defined in the server.properties file in the previous article.
- Execute this command to create a topic.
  - **./bin/./kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 100 --topic demo**

```
package com.demo.kafka.constants;

public interface IKafkaConstants {
     public static String KAFKA_BROKERS = "localhost:9092";

     public static Integer MESSAGE_COUNT=1000;

     public static String CLIENT_ID="client1";

     public static String TOPIC_NAME="demo";

     public static String GROUP_ID_CONFIG="consumerGroup10";

     public static Integer MAX_NO_MESSAGE_FOUND_COUNT=100;

     public static String OFFSET_RESET_LATEST="latest";

     public static String OFFSET_RESET_EARLIER="earliest";

     public static Integer MAX_POLL_RECORDS=1;
}
```

    - replication-factor: if Kafka is running in a cluster, this determines on how many brokers a partition will be replicated. The partitions argument defines how many partitions are in a topic.

    - After a topic is created you can increase the partition count but it cannot be decreased. demo, here, is the topic name.

- Execute this command to see the information about a topic.
    - **./bin/./kafka-topics.sh --describe --topic demo --zookeeper localhost:2181 .**

Now that we know the common terms used in Kafka and the basic commands to see information about a topic ,let's start with a working example.

The above snippet contains some constants that we will be using further.

```
package com.demo.kafka.producer;

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.common.serialization.LongSerializer;
import org.apache.kafka.common.serialization.StringSerializer;

import com.demo.kafka.constants.IKafkaConstants;

public class ProducerCreator {

     public static Producer<Long, String> createProducer() {
          Properties props = new Properties();
          props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
IKafkaConstants.KAFKA_BROKERS);
          props.put(ProducerConfig.CLIENT_ID_CONFIG,
IKafkaConstants.CLIENT_ID);
          props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
LongSerializer.class.getName());

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

The above snippet creates a Kafka producer with some properties.

- BOOTSTRAP_SERVERS_CONFIG: The Kafka broker's address. If Kafka is running in a cluster then you can provide comma (,) seperated addresses. For example:localhost: 9091,localhost:9092

- CLIENT_ID_CONFIG: Id of the producer so that the broker can determine the source of the request.

- KEY_SERIALIZER_CLASS_CONFIG: The class that will be used to serialize the key object. In our example, our key is Long, so we can use the LongSerializer class to serialize the key. If in your use case you are using some other object as the key then you can create your custom serializer class by implementing the **Serializer** interface of Kafka and overriding the serialize method.

- VALUE_SERIALIZER_CLASS_CONFIG: The class that will be used to serialize the value object. In our example, our value is String, so we can use the StringSerializer class to serialize the key. If your value is some other object then you create your custom serializer class. For example:

The above snippet creates a Kafka consumer with some properties.

- BOOTSTRAP_SERVERS_CONFIG: The Kafka broker's address. If Kafka is running in a cluster then you can provide comma (,) seperated addresses. For example: localhost: 9091,localhost:9092.

- GROUP_ID_CONFIG: The consumer group id used to identify to which group this consumer belongs.

- KEY_DESERIALIZER_CLASS_CONFIG: The class name to deserialize the key object. We have used Long as the key so we will be using **LongDeserializer** as the deserializer class. You can create your custom deserializer by implementing the **Deserializer** interface provided by Kafka.

- VALUE_DESERIALIZER_CLASS_CONFIG: The class name to deserialize the value object. We have used String as the value so we will be using **StringDeserializer** as the

```
package com.demo.kafka.serializer;

import java.util.Map;

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

import com.demo.kafka.pojo.CustomObject;
import com.fasterxml.jackson.databind.ObjectMapper;

public class CustomSerializer implements Serializer<CustomObject> {

	@Override
	public void configure(Map<String, ?> configs, boolean isKey) {

	}

	@Override
	public byte[] serialize(String topic, CustomObject data) {
		byte[] retVal = null;
		ObjectMapper objectMapper = new ObjectMapper();
		try {
			retVal =
objectMapper.writeValueAsString(data).getBytes();
		} catch (Exception exception) {
			System.out.println("Error in serializing object" +
data);
		}
		return retVal;
	}

	@Override
	public void close() {

	}

}
```

deserializer class. You can create your custom deserializer. For example:

- MAX_POLL_RECORDS_CONFIG: The max count of records that the consumer will fetch in one iteration.
- ENABLE_AUTO_COMMIT_CONFIG: When the consumer from a group receives a message it must commit the offset of that

record. If this configuration is set to be true then, periodically, offsets will be committed, but, for the production level, this should be false and an offset should be committed manually.

- AUTO_OFFSET_RESET_CONFIG: For each consumer group, the last committed offset value is stored. This configuration comes handy if no offset is committed for that group, i.e. it is the new group created.
  - Setting this value to **earliest** will cause the consumer to fetch records from the beginning of offset i.e from

```
package com.demo.kafka.consumer;

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

import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.LongDeserializer;
import org.apache.kafka.common.serialization.StringDeserializer;

import com.demo.kafka.constants.IKafkaConstants;

public class ConsumerCreator {

	public static Consumer<Long, String> createConsumer() {
		final Properties props = new Properties();
		props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
IKafkaConstants.KAFKA_BROKERS);
		props.put(ConsumerConfig.GROUP_ID_CONFIG,
IKafkaConstants.GROUP_ID_CONFIG);
		props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
LongDeserializer.class.getName());

props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
		props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG,
IKafkaConstants.MAX_POLL_RECORDS);
		props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG,
"false");
		props.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG,
IKafkaConstants.OFFSET_RESET_EARLIER);

		final Consumer<Long, String> consumer = new
KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList(IKafkaConstants.TOPIC
_NAME));
		return consumer;
	}

}
```

zero.

- Setting this value to **latest** will cause the consumer to fetch records from the new records. By new records mean those created after the consumer group

```
package com.demo.kafka;

import java.util.concurrent.ExecutionException;

import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.producer.RecordMetadata;

import com.demo.kafka.constants.IKafkaConstants;
import com.demo.kafka.consumer.ConsumerCreator;
import com.demo.kafka.producer.ProducerCreator;

public class App {
	public static void main(String[] args) {
		runProducer();
		runConsumer();
	}

	static void runConsumer() {
		Consumer<Long, String> consumer =
ConsumerCreator.createConsumer();

		int noMessageToFetch = 0;

		while (true) {
			final ConsumerRecords<Long, String> consumerRecords =
consumer.poll(1000);
			if (consumerRecords.count() == 0) {
				noMessageToFetch++;
				if (noMessageToFetch >
IKafkaConstants.MAX_NO_MESSAGE_FOUND_COUNT)
					break;
				else
					continue;
			}

			consumerRecords.forEach(record -> {
				System.out.println("Record value " +
record.value());
				System.out.println("Record partition " +
record.partition());
				System.out.println("Record offset " +
record.offset());
			});
			consumer.commitAsync();
		}
		consumer.close();
	}
```

```
      static void runProducer() {
            Producer<Long, String> producer =
ProducerCreator.createProducer();

            for (int index = 0; index < IKafkaConstants.MESSAGE_COUNT;
index++) {
                  final ProducerRecord<Long, String> record = new
ProducerRecord<Long, String>(IKafkaConstants.TOPIC_NAME,
                              "This is record " + index);
                  try {
                        RecordMetadata metadata =
producer.send(record).get();
                        System.out.println("Record sent with key " + index
+ " to partition " + metadata.partition()
                                    + " with offset " +
metadata.offset());
                  } catch (ExecutionException e) {
                        System.out.println("Error in sending record");
                        System.out.println(e);
                  } catch (InterruptedException e) {
                        System.out.println("Error in sending record");
                        System.out.println(e);
                  }
            }
      }
}
```

became active.

```
package com.demo.kafka.deserializer;

import java.util.Map;

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

import com.demo.kafka.pojo.CustomObject;
import com.fasterxml.jackson.databind.ObjectMapper;

public class CustomDeserializer implements
Deserializer<CustomObject> {
	@Override
	public void configure(Map<String, ?> configs, boolean isKey)
{
	}

	@Override
	public CustomObject deserialize(String topic, byte[] data) {
		ObjectMapper mapper = new ObjectMapper();
		CustomObject object = null;
		try {
			object = mapper.readValue(data,
CustomObject.class);
		} catch (Exception exception) {
			System.out.println("Error in deserializing bytes "
+ exception);
		}
		return object;
	}

	@Override
	public void close() {
	}
}
```

The above snippet explains how to produce and consume messages from a Kafka broker. If you want to run a producer then call the **runProducer** function from the main function. If you want to run a consumeer, then call the **runConsumer** function from the main function.

- The offset of records can be committed to the broker in both asynchronous and synchronous ways. Using the synchronous way, the thread will be blocked until an offset has not been written to the broker.

RUN the App.java