Kafka Simple Producer

Let us create an application for publishing and consuming messages using a Java client. Kafka producer client consists of the following API's.

KafkaProducer API

Let us understand the most important set of Kafka producer API in this section. The central part of the KafkaProducer API is `KafkaProducer` class. The KafkaProducer class provides an option to connect a Kafka broker in its constructor with the following methods:

KafkaProducer class provides send method to send messages asynchronously to a topic. The signature of send() is as follows:

```
producer.send(new ProducerRecord<byte[],byte[]>(topic, partit
ion, key1, value1) , callback);
```

- ProducerRecord - The producer manages a buffer of records waiting to be sent
- Callback - A user-supplied callback to execute when the record has been acknowledged by the server (null indicates no callback)

KafkaProducer class provides a flush method to ensure all previously sent messages have been actually completed. Syntax of the flush method is as follows:

```
public void flush()
```

KafkaProducer class provides partitionFor method, which helps in getting the partition metadata for a given topic. This can be used for custom partitioning. The signature of this method is as follows:

```
public partitionsFor(string topic)
```

It returns the metadata of the topic.

KafkaProducer class provides metrics method that is used to return a map of metrics maintained by the producer. The signature of this method is as follows:

```
public  Map  metrics()
```

It returns the map of internal metrics maintained by the producer.

```
public void close() – KafkaProducer class provides close meth
od blocks until all previously sent requests are completed.
```

Producer API

The central part of the Producer API is `Producer` class. Producer class provides an option to connect Kafka broker in its constructor by the following methods.

The Producer Class

The Producer class provides send method to send messages to either single or multiple topics using the following signatures:

```
public void send(KeyedMessage<k,v> message) // sends the
 data to a single topic,par- titioned by key using either
 sync or async producer
public void send(List<KeyedMessage<k,v>> messages) // sen
ds data to multiple topics. Properties prop = new Propert
ies();
prop.put(producer.type,”async”)
ProducerConfig config = new ProducerConfig(prop);
```

There are two types of producers – **Sync** and **Async**.

The same API configuration applies to “Sync” producer as well. The difference between them is a sync producer sends messages directly, but sends messages in background. Async producer is preferred when you want a higher throughput. In the previous releases like 0.8, an async producer does not have a callback for send() to register error handlers. This is available only in the current release of 0.9.

```
public void close()
```

Producer class provides close method to close the producer pool

connections to all Kafka brokers.

Configuration Settings

The Producer API’s main configuration settings are listed in the following table for better understanding:

Setting	What it Does
client.id	identifies producer application
producer.type	either sync or async
acks	The acks config controls the criteria under producer requests are con- sidered complete
retries	If producer request fails, then automatically retry with specific value
bootstrap.servers	bootstrapping list of brokers
linger.ms	if you want to reduce the number of requests you can set linger.ms to something greater than some value
key.serializer	Key for the serializer interface
value.serializer	value for the serializer interface
batch.size	Buffer size
buffer.memory	controls the total amount of memory available to the producer for buffering

ProducerRecord API

ProducerRecord is a key/value pair that is sent to Kafka cluster.ProducerRecord class constructor for creating a record with partition, key and value pairs using the following signature:

```
public ProducerRecord (string topic, int partition, k key, v value)
```

- Topic - user defined topic name that will appended to record.
- Partition - partition count.
- Key - The key that will be included in the record.
- Value - Record contents.

```
public ProducerRecord (string topic, k key, v value)
```

ProducerRecord class constructor is used to create a record with key, value pairs and without partition.

- Topic - Create a topic to assign record.
- Key - key for the record.
- Value - record contents.

```
public ProducerRecord (string topic, v value)
```

ProducerRecord class creates a record without partition and key.

- Topic - create a topic.
- Value - record contents.

The ProducerRecord class methods are listed in the following table:

Method	What it Does
`public string topic()`	Topic will append to the record
`public K key()`	Key that will be included in the record. If no such key, null will be returned here
`public V value()`	Record contents
`partition()`	Partition count for the record

The SimpleProducer Application

> *Note: don't do this if Kafka is running*

Before creating the application, first start ZooKeeper and Kafka broker then create your own topic in Kafka broker using create topic command.

Now create a java class named `SimpleProducer.java` and type in the following code (probably want to cut-n-paste):

```
import java.util.Properties; //import util.properties package
s
import org.apache.kafka.clients.producer.Producer; //import s
imple producer packages
import org.apache.kafka.clients.producer.KafkaProducer; //imp
ort KafkaProducer packages
```

```
import org.apache.kafka.clients.producer.ProducerRecord; //im
port ProducerRecord packages public class SimpleProducer {

//Create  java  class  named  “SimpleProducer”
public  static  void  main(String[]  args)  throws  Exception
  {

  //  Check  arguments  length  value
  if(args.length  ==  0)
  {
    System.out.println("Enter  topic  name”);
    return;
  }

  String  topicName  =  args[0].toString(); //Assign  topicNa
me  to  string  variable

  Properties  props  =  new  Properties(); //  create  instan
ce  for  properties  to  access  producer  configs
  props.put("bootstrap.servers",  “localhost:9092"); //Assign
  localhost  id
  props.put("acks",  “all"); //Set  acknowledgements  for  pr
oducer  requests.
  props.put("retries",  0); //If  the  request  fails,  the
producer  can  automatically  retry,
  props.put("batch.size",  16384); //Specify  buffer  size  i
n  config
  props.put("linger.ms",  1); //Reduce  the  no  of  requests
```

```
  less  than  0
  props.put("buffer.memory",  33554432); //The  buffer.memory
  controls  the  total  amount  of  memory  available  to the
  producer  for  buffering
  props.put("key.serializer",  "org.apache.kafka.common.seria
lization.StringSerializer");
  props.put("value.serializer",  "org.apache.kafka.common.ser
ialization.StringSerializer");

  Producer<String,  String>  producer  =  new  KafkaProducer<
String, String>(props);

  for(int  i  =  0;  i  <  10;  i++)
   producer.send(new  ProducerRecord<String,  String>(topicNa
me, Integer.toString(i), Integer.toString(i)));
   System.out.println("Message  sent  successfully");
   producer.close();
  }
}
```

Compilation – The application can be compiled using the following command.

> *Note: the release of Kafka may vary. Code accordingly*

```
javac  -cp  "/path/to/kafka/kafka_2.11-0.9.0.0/lib/*"  *.java
```

Execution – The application can be executed using the following command.

```
java -cp "/path/to/kafka/kafka_2.11-0.9.0.0/lib/*":. SimpleProducer <topic-name>
```

Output

```
Message sent successfully
To check the above output open new terminal and type Consumer CLI command to receive messages.
>> bin/kafka-console-consumer.sh --zookeeper localhost:2181 –topic <topic-name> – from-beginning

1
2
3
4
5
6
7
8
9
10
```

Simple Consumer Example

As of now we have created a producer to send messages to Kafka cluster.

Now let us create a consumer to consume messages form the Kafka cluster. KafkaConsumer API is used to consume messages from the Kafka cluster. KafkaConsumer class constructor is defined below.

```
public  KafkaConsumer(java.util.Map<java.lang.String,java.lang.Object> configs)
```

configs - Return a map of consumer configs.

KafkaConsumer class has the following significant methods that are listed in the table below:

Method	What it Does
`public java.util.Set<TopicPartition> assignment()`	Get the set of partitions currently assigned by the consumer.
`public string subscription()`	Subscribe to the given list of topics to get dynamically assigned partitions
`public void subscribe(java.util.List<java.lang.String> topics, ConsumerRebalanceListener listener)`	First argument topics refers to subscribing topics list and second argument listener refers to get notifications on partition assignment/revocation for the subscribed topics

`public void unsubscribe()`	Unsubscribe the topics from the given list of partitions
`public void subscribe(java.util.List<java.lang.String> topics)`	Subscribe to the given list of topics to get dynamically assigned partitions. If the given list of topics is empty, it is treated the same as unsubscribe()
`public void subscribe(java.util.regex.Pattern pattern, ConsumerRebalanceListener listener)`	The argument pattern refers to the subscribing pattern in the format of regular expression and the listener argument gets notifications from the subscribing pattern
`public void assign(java.util.List<TopicPartition> partitions)`	Manually assign a list of partitions to the customer
`poll()`	Fetch data for the topics or partitions specified using one of the subscribe/assign APIs. This will return error, if the topics are not subscribed before the polling for data
`public void commitSync()`	Commit offsets returned on the last

	poll() for all the subscribed list of topics and partitions. The same operation is applied to `commitAsyn()`
`public void seek(TopicPartition partition, long offset)`	Fetch the current offset value that consumer will use on the next poll() method
`public void resume()`	Resume the paused partitions
`public void wakeup()`	Wakeup the consumer

ConsumerRecordAPI

The ConsumerRecord API is used to receive records from the Kafka cluster. This API consists of a topic name, partition number, from which the record is being received and an offset that points to the record in a Kafka partition. ConsumerRecord class is used to create a consumer record with specific topic name, partition count and <key, value> pairs. It has the following signature:

```
public  ConsumerRecord(string topic, int partition,  long  of
fset, K key,  V value)
```

- Topic - The topic name for consumer record received from the Kafka cluster
- Partition - Partition for the topic

- Key - The key of the record, if no key exists null will be returned
- Value - Record contents

ConsumerRecords API

ConsumerRecords API acts as a container for ConsumerRecord. This API is used to keep the list of ConsumerRecord per partition for a particular topic. Its Constructor is defined below:

```
public  ConsumerRecords(java.util.Map<TopicPartition,java.uti
l.List<ConsumerRecord<K,V>>> records)
```

- TopicPartition - Return a map of partition for a particular topic
- Records - Return list of ConsumerRecord. ConsumerRecords class has the following methods defined

Method	What it Does
`public int count()`	The number of records for all the topics
`public Set partitions()`	The set of partitions with data in this record set (if no data was returned then the set is empty)
`public Iterator iterator()`	Iterator enables you to cycle through a collection, obtaining or re- moving elements
`public List records()`	Get list of records for the given partition

Configuration Settings

The configuration settings for the Consumer client API main configuration settings are listed below:

Setting	What it Does
bootstrap.servers	Bootstrapping list of brokers
group.id	Assigns an individual consumer to a group
enable.auto.commit	Enable auto commit for offsets if the value is true, otherwise not committed
auto.commit.interval.ms	Return how often updated consumed offsets are written to ZooKeeper
session.timeout.ms	Indicates how many milliseconds Kafka will wait for the ZooKeeper to respond to a request (read or write) before giving up and continuing to consume messages

SimpleConsumerApplication

The producer application steps remain the same here. First, start your ZooKeeper and Kafka broker. Then create a `SimpleConsumer` application with the java class named `SimpleConsumer.java` and type the following code:

```
import  java.util.Properties;
import  java.util.Arrays;
import  org.apache.kafka.clients.consumer.KafkaConsumer;
```

```
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

public class SimpleConsumer {

  public  static  void  main(String[]  args)  throws  Excepti
on  {

   if(args.length  ==  0)
   {
     System.out.println("Enter  topic  name");
     return;
   }

   String  topicName  =  args[0].toString(); Properties  prop
s  =  new  Properties();

   //Kafka  consumer  configuration
   settings props.put("bootstrap.servers",  "localhost:9092")
;
   props.put("group.id",  "test");
   props.put("enable.auto.commit",  "true");
   props.put("auto.commit.interval.ms",  "1000");
   props.put("session.timeout.ms",  "30000");
   props.put("key.deserializer",  "org.apache.kafka.common.se
rializa- tion.StringDeserializer");
   props.put("value.deserializer",  "org.apache.kafka.common.
serializa- tion.StringDeserializer");
```

```
   KafkaConsumer<String,  String>  consumer  =  new  KafkaCon
sumer<String, String>(props);
   consumer.subscribe(Arrays.asList(topicName));

   //KafkaConsumer  subscribes  list  of  topics  here.
   System.out.println("Subscribed  to  topic  "  +  topicName
);  //print  the topic name

   //  print  the  offset,key  and  value  for  the  consumer
  records.
   int  i  =  0; while  (true)  {
    ConsumerRecords<String,  String>  records  =  consumer.po
ll(100);
    for  (ConsumerRecord<String,  String>  record  :  records
)
     System.out.printf("offset  =  %d,  key  =  %s,  value =
 %s\n",  record.offset(),  record.key(),  record.value());
   }
}
}
```

> *Note: the release of Kafka may vary. Code accordingly*

Compilation – The application can be compiled using the following command.

```
javac  -cp  “/path/to/kafka/kafka_2.11-0.9.0.0/lib/*”  *.java
```

Execution – Now execute:

```
java  -cp  “/path/to/kafka/kafka_2.11-0.9.0.0/lib/*”:.  SimpleConsumer  <topic-name>
```

Input – Open the producer CLI and send some messages to the topic. You can put the smple input as `Hello Consumer`.

Output – Following will be the output:

```
Subscribed  to  topic  Hello-Kafka
offset  =  3,  key  =  null,  value  =  Hello  Consumer
```