# JavaCro 2016 Kafka demo

Few days ago JavaCro 2016 conference was held in Rovinj, Croatia, where I presented basic Kafka concepts to (mostly) people of croatian Java community and demonstrated simple spring Boot application using Kafka 0.9. This presentation focused on new consumer API and how to use it.

## Starting Kafka

So lets start with setup.

Download Kafka platform 2.0.1 from http://packages.confluent.io/archive/2.0/confluent-2.0.1-2.11.7.zip and unzip it to location of your choice. Let's call this location confluent home folder.

Now we can start Kafka server, but as explained in my previous post, Kafka requires Apache ZooKeeper which is used for managing cluster configuration information.

Navigate to confluent home folder and run :

```
bin/zookeeper-server-start etc/kafka/zookeeper.properties
```

And after it's up Kafka  broker can be started (in new terminal):

```
bin/kafka-server-start etc/kafka/server.properties
```

Kafka is now started and ready to use. We started only one broker for testing purposes, but starting more brokers is trivial, only thing you need to prepare is another server.properties file since you want to configure different port then the one first broker is using.

## Demo App

This demo app is spring boot web application which allows us to:

- create Kafka topics,
- start producers,
- control producers speed
- start consumers
- control consumer message processing speed

You can find source code at github repository.
Core package contains most important classes: JavaCroProducer and JavaCroConsumer. These classes represent our kafka clients and use producer and consumer api to talk to kafka cluster.

Video of the demo presented.

In this video we can see how to add topics, producers and consumers, and how Kafka automatically re-balances consumer while they connect or disconnect. Gears are used for showing current speed of consumers and producers, and progress like bars are used to display topic partition and their current message count.

## Producer

First, lets take a look at JavaCroProducer and see what we need to start it.
In order to communicate with Kafka (to send some messages to it) we need to create and configure KafkaProducer instance. I takes Properties object as argument so we must instantiate one and fill it with basic properties necessary for it to work.

```
Properties config = new Properties();
config.put("client.id", id);
config.put("bootstrap.servers", "localhost:9092");
config.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
config.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
config.put("acks", "1");

KafkaProducer producer = new KafkaProducer(config);
```

- client.id property is used to uniquely identify our producer to Kafka cluster. In this case it is passed as constructor parameter of JavaCroProducer class. It's actually simple long incremented every time we create new JavaCroProducer instance.

- bootstrap.servers property is list of kafka broker identifiers consisting of host:port pairs, coma separated. We must provide at least one broker identifier, and if we have more then one broker in cluster it is recommended to provide two or more broker identifiers for redundancy. We don't have to provide all brokers identifiers, actually one is enough for producer to get all the info it needs, other identifiers are used just in case first broker fails or some other reason prevents producer to communicate with it.

- ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG is telling KafkaProducer which serializer to use for message keys  when sending messages to Kafka. In this example built in String serializer is used.

- ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG is telling KafkaProducer which serializer to use for message values  when sending messages to Kafka. In this example built in String serializer is used.

- acks property tells KafkaProducer when to consider a message successfully sent to Kafka broker. 0 value means that producer will consider message sent as soon as it puts it to socket buffer. Value 1 means thas message is considered sent when partition leader broker writes it to partition log, and finally value all means that producer will consider message sent only after partition leader and all it's replicas write message to partition log.
Now we have our producer ready and we can us it to  send messages. Something like this:

```
ProducerRecord record = new ProducerRecord("topicName", msgKey"", "msgValue");
producer.send(record);
```

We need ProducerRecord instance  to send the message and it's constructor takes three arguments:
- topic name : string, name of kafka topic to which you want to send the message.
- message key  : object,  it is used to decide on which partition will message go if the topic has more then one partition. If message key is not null, every message with same key will go to same partition, so if we want all messages  with same business property (for example sender id) to end up on same partition, we could use that property as message key.
- message value : string,  this is actual message payload - the content of the message.
Note that consumers will receive both message key and value so you can use key as some kind of extra message information holder.

There is overloaded constructor which takes partition number instead of message key argument, and it is used to explicitly set target partition of the topic. KafkaProducer provides methods for discovering how many partition specific topic has.

KafkaProducer.send() method is asynchronous but if we want to send the messages in synchronous way (not really recommended) we can do this, and here is how: send method returns Future so we wait until we get sending result by calling Future.get() method on it.

If we need to examine the result of send method but without blocking sending process, there is a way:

overloaded version of send method receives second argument, implementation of org.apache.kafka.clients.producer.Callback interface.

It's method onCompletion(RecordMetadata metadata, Exception exception) will be called when message sending finishes, whether successfully or not. If message is sent successfully we can get meta data from RecordMetadata object (such as topic message was sent to, position of the message written to topic partition, number of partition which message was written to...) Second argument - Exception will be null if no error occurred, and not null otherwise.

## Consumer

JavaCroConsumer represents our consumer process and it uses KafkaConsumer to communicate with Kafka cluster.
First thing to do is to create and configure KafkaConsumer object:

```
Properties config = new Properties();
config.put("client.id", id);
config.put("group.id", group);

config.put("bootstrap.servers", "localhost:9092");
config.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
config.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
config.put("auto.offset.reset", "earliest");

consumer = new KafkaConsumer(config);
consumer.subscribe( Arrays.asList(new String[]{topic}));
```

Let's go through all KafkaConsumer config properties :
- client.id : unique identifier of this consumer
- group.id : group we want consumer to belong to. Group is set of consumers for handling specific business task (logical consumer).
- bootstrap.servers : list of kafka broker identifiers consisting of host:port pairs, coma separated. Like with producers, here we also must provide at least one broker identifier, and if we have more then one broker in cluster it is recommended to provide two or more broker identifiers for redundancy.
- ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG - deserializer implementation class for deserializing message keys
- ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG - deserializer implementation class for deserializing message values
- auto.offset.reset : this property tells kafka from which position on partition we want new consumer to start reading messages in case no other consumer from the group already read from it.
Default behaviour is to start reading from latest message existing on partitio at the time consumer connected Kafka cluster. In our case we set it to earliest, because we want to start processing messages from the beginning of the partition.

After instantiating KafkaConsumer object we must tell it from which topics we want to read the messages.
And now we can start reading messages:

```
while (!shutdown.get()) {
            ConsumerRecords records = consumer.poll(1000);
            records.forEach(record -&gt; processMessage(record));
      }
```

Here we have loop running until shutdown and in each iteration it reads messages using consumer.poll() method. It receives timeout argument, and it will block current thread until messages are available or timeout expires, whatever comes first. In both cases this method will return Consu merRecords object which is collection of ConsumerRecord instances. Consumer record is key-value pair, representing message key and value and also containing some metadata of the message such as topic from which message was read, number of partition which hold this message and message offset (position) on it's partition.

Very important concept of consumer API are consumer groups. Consumer groups concept is Kafka's abstraction of queuing and publish-subscribe models. Two (or more) consumers from same group reading from same topic will never get same message (queuing model) but two (or more) consumers from different groups will (publish-subscribe model). This means that consumer group behaves like single logical consumer and specific consumers of the group (consumer nodes) are used for horizontal scaling of the logical consumer.

Since only one consumer from the group can read from particular partition, in case when group has more consumers then topic has partitions, all extra consumers will be in kind of hot stand by - they will periodically ask Kafka for messages, but they will get empty record collection. If one of active consumers crashes, one of extra consumer nodes will start to receive messages exactly from the position crashed node stopped processing them.

If there is single consumer node in a group reading from topic with two or more partitions, this consumer will be getting messages from all partitions.

By guaranteeing that only one consumer at the time can read from the partition, Kafka guarantees not just message delivery order but message processing order. Note that this guarantee applies for partition scope, not for entire topic (with multiple partitions), but this should be enough since we can almost always group messages to partitions by some business property which is relevant for processing order. This is main purpose of message keys. Messages with same keys will end up on same partition , so if we use business property relevant for processing order as message key, we got things covered.

Yesterday Kafka 0.10 is released as part of Confluent platform 3.0  but since consumer API, which this post is focused on, had it major change in version 0.9 and not changed in 0.10, so everything said here still applies.

Latest version of Kafka platform brings interesting things such as  Streams library and Control Center and version 0.9  brought Security and Kafka Connect and I hope I will grab some time to blog about some of them soon.

Author: Igor Buzatović