LEARNING apache-kafka

#apache-kafka

Table of Contents

About

Chapter 1: Getting started with apache-kafka

Remarks

Kafka is a high throughput publish-subscribe messaging system implemented as distributed, partitioned, replicated commit log service.

Taken from official Kafka site

> **Fast**
>
> A single Kafka broker can handle hundreds of megabytes of reads and writes per second from thousands of clients.
>
> **Scalable**
>
> Kafka is designed to allow a single cluster to serve as the central data backbone for a large organization. It can be elastically and transparently expanded without downtime. Data streams are partitioned and spread over a cluster of machines to allow data streams larger than the capability of any single machine and to allow clusters of co-ordinated consumers
>
> **Durable**
>
> Messages are persisted on disk and replicated within the cluster to prevent data loss. Each broker can handle terabytes of messages without performance impact.
>
> **Distributed by Design**
>
> Kafka has a modern cluster-centric design that offers strong durability and fault-tolerance guarantees.

Examples

Installation or Setup

Step 1. Install Java 7 or 8

Step 2. Download Apache Kafka at: http://kafka.apache.org/downloads.html

For example, we will try download Apache Kafka 0.10.0.0

Step 3. Extract the compressed file.

On Linux:

```
tar -xzf kafka_2.11-0.10.0.0.tgz
```

On Window: Right click --> Extract here

Step 4. Start Zookeeper

```
cd kafka_2.11-0.10.0.0
```

Linux:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Windows:

```
bin/windows/zookeeper-server-start.bat config/zookeeper.properties
```

Step 5. Start Kafka server

Linux:

```
bin/kafka-server-start.sh config/server.properties
```

Windows:

```
bin/windows/kafka-server-start.bat config/server.properties
```

Introduction

Apache Kafka™ is a distributed streaming platform.

Which means

1-It lets you publish and subscribe to streams of records. In this respect it is similar to a message queue or enterprise messaging system.

2-It lets you store streams of records in a fault-tolerant way.

3-It lets you process streams of records as they occur.

It gets used for two broad classes of application:

1-Building real-time streaming data pipelines that reliably get data between systems or applications

2-Building real-time streaming applications that transform or react to the streams of data

> Kafka console scripts are different for Unix-based and Windows platforms. In the examples, you might need to add the extension according to your platform. Linux: scripts located in `bin/` with `.sh` extension. Windows: scripts located in `bin\windows\` and

with `.bat` extension.

Installation

Step 1: Download the code and untar it:

```
tar -xzf kafka_2.11-0.10.1.0.tgz
cd kafka_2.11-0.10.1.0
```

Step 2: start the server.

> to be able to delete topics later, open `server.properties` and set `delete.topic.enable` to true.

Kafka relies heavily on zookeeper, so you need to start it first. If you don't have it installed, you can use the convenience script packaged with kafka to get a quick-and-dirty single-node ZooKeeper instance.

```
zookeeper-server-start config/zookeeper.properties
kafka-server-start config/server.properties
```

Step 3: ensure everything is running

You should now have zookeeper listening to `localhost:2181` and a single kafka broker on `localhost:6667`.

Create a topic

We only have one broker, so we create a topic with no replication factor and just one partition:

```
kafka-topics --zookeeper localhost:2181 \
    --create \
    --replication-factor 1 \
    --partitions 1 \
    --topic test-topic
```

Check your topic:

```
kafka-topics --zookeeper localhost:2181 --list
test-topic

kafka-topics --zookeeper localhost:2181 --describe --topic test-topic
Topic:test-topic    PartitionCount:1    ReplicationFactor:1 Configs:
Topic: test-topic   Partition: 0    Leader: 0    Replicas: 0 Isr: 0
```

send and receive messages

Launch a consumer:

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test-topic
```

On another terminal, launch a producer and send some messages. By default, the tool send each line as a separate message to the broker, without special encoding. Write some lines and exit with CTRL+D or CTRL+C:

```
kafka-console-producer --broker-list localhost:9092 --topic test-topic
a message
another message
^D
```

The messages should appear in the consumer therminal.

Stop kafka

```
kafka-server-stop
```

start a multi-broker cluster

The above examples use only one broker. To setup a real cluster, we just need to start more than one kafka server. They will automatically coordinate themselves.

Step 1: to avoid collision, we create a `server.properties` file for each broker and change the `id`, `port` and `logfile` configuration properties.

Copy:

```
cp config/server.properties config/server-1.properties
cp config/server.properties config/server-2.properties
```

Edit properties for each file, for example:

```
vim config/server-1.properties
broker.id=1
listeners=PLAINTEXT://:9093
log.dirs=/usr/local/var/lib/kafka-logs-1

vim config/server-2.properties
broker.id=2
listeners=PLAINTEXT://:9094
log.dirs=/usr/local/var/lib/kafka-logs-2
```

Step 2: start the three brokers:

```
    kafka-server-start config/server.properties &
    kafka-server-start config/server-1.properties &
    kafka-server-start config/server-2.properties &
```

Create a replicated topic

```
kafka-topics --zookeeper localhost:2181 --create --replication-factor 3 --partitions 1 --topic
replicated-topic

kafka-topics --zookeeper localhost:2181 --describe --topic replicated-topic
Topic:replicated-topic  PartitionCount:1    ReplicationFactor:3 Configs:
Topic: replicated-topic Partition: 0    Leader: 1   Replicas: 1,2,0 Isr: 1,2,0
```

This time, there are more information:

- "leader" is the node responsible for all reads and writes for the given partition. Each node will be the leader for a randomly selected portion of the partitions.
- "replicas" is the list of nodes that replicate the log for this partition regardless of whether they are the leader or even if they are currently alive.
- "isr" is the set of "in-sync" replicas. This is the subset of the replicas list that is currently alive and caught-up to the leader.

Note that the previously created topic is left unchanged.

test fault tolerance

Publish some message to the new topic:

```
kafka-console-producer --broker-list localhost:9092 --topic replicated-topic
hello 1
hello 2
^C
```

Kill the leader (1 in our example). On Linux:

```
ps aux | grep server-1.properties
kill -9 <PID>
```

On Windows:

```
wmic process get processid,caption,commandline | find "java.exe" | find "server-1.properties"
taskkill /pid <PID> /f
```

See what happened:

```
kafka-topics --zookeeper localhost:2181  --describe --topic replicated-topic
Topic:replicated-topic  PartitionCount:1    ReplicationFactor:3 Configs:
Topic: replicated-topic Partition: 0    Leader: 2   Replicas: 1,2,0 Isr: 2,0
```

The leadership has switched to broker 2 and "1" in not in-sync anymore. But the messages are still there (use the consumer to check out by yourself).

Clean-up

Delete the two topics using:

```
kafka-topics --zookeeper localhost:2181 --delete --topic test-topic
kafka-topics --zookeeper localhost:2181 --delete --topic replicated-topic
```

Read Getting started with apache-kafka online: https://riptutorial.com/apache-kafka/topic/1986/getting-started-with-apache-kafka

Chapter 2: Consumer Groups and Offset Management

Parameters

Parameter	Description
group.id	The name of the Consumer Group.
enable.auto.commit	Automatically commit offsets; *default: true*.
auto.commit.interval.ms	The minimum delay in milliseconds between to commits (requires `enable.auto.commit=true`); *default: 5000*.
auto.offset.reset	What to do when there is no valid committed offset found; *default: latest*.(+)
(+) Possible Values	**Description**
earliest	Automatically reset the offset to the earliest offset.
latest	Automatically reset the offset to the latest offset.
none	Throw exception to the consumer if no previous offset is found for the consumer's group.
anything else	Throw exception to the consumer.

Examples

What is a Consumer Group

As of Kafka 0.9, the new high level KafkaConsumer client is availalbe. It exploits a new built-in Kafka protocol that allows to combine multiple consumers in a so-called *Consumer Group*. A Consumer Group can be describes as a single logical consumer that subscribes to a set of topics. The partions over all topics are assigend to the physical consumers within the group, such that each patition is assigned to exaclty one consumer (a single consumer can get multiple partitons assigned). The indiviual consumers belonging to the same group can run on different hosts in a distributed manner.

Consumer Groups are identified via their `group.id`. To make a specific client instance member of a Consumer Group, it is sufficient to assign the groups `group.id` to this client, via the client's configuration:

```
Properties props = new Properties();
props.put("group.id", "groupName");
// ...some more properties required
new KafkaConsumer<K, V>(config);
```

Thus, all consumers that connect to the same Kafka cluster and use the same `group.id` form a Consumer Group. Consumers can leave a group at any time and new consumers can join a group at any time. For both cases, a so-called *rebalance* is triggered and partitions get reassigned with the Consumer Group to ensure that each partition is processed by exaclty one consumer within the group.

Pay attention, that even a single `KafkaConsumer` forms a Consumer Group with itself as single member.

Consumer Offset Management and Fault-Tolerance

KafkaConsumers request messages from a Kafka broker via a call to `poll()` and their progress is tracked via *offsets*. Each message within each partition of each topic, has a so-called offset assigned—its logical sequence number within the partition. A `KafkaConsumer` tracks its current offset for each partition that is assigned to it. Pay attention, that the Kafka brokers are not aware of the current offsets of the consumers. Thus, on `poll()` the consumer needs to send its current offsets to the broker, such that the broker can return the corresponding messages, i.e,. messages with a larger consecutive offset. For example, let us assume we have a single partition topic and a single consumer with current offset 5. On `poll()` the consumer sends if offset to the broker and the broker return messages for offsets 6,7,8,...

Because consumers track their offsets themselves, this information could get lost if a consumer fails. Thus, offsets must be stored reliably, such that on restart, a consumer can pick up its old offset and resumer where it left of. In Kafka, there is built-in support for this via *offset commits*. The new `KafkaConsumer` can commit its current offset to Kafka and Kafka stores those offsets in a special topic called `__consumer_offsets`. Storing the offsets within a Kafka topic is not just fault-tolerant, but allows to reassign partitions to other consumers during a rebalance, too. Because all consumers of a Consumer Group can access all committed offsets of all partitions, on rebalance, a consumer that gets a new partition assigned just reads the committed offset of this partition from the `__consumer_offsets` topic and resumes where the old consumer left of.

How to Commit Offsets

KafkaConsumers can commit offsets automatically in the background (configuration parameter `enable.auto.commit = true`) what is the default setting. Those auto commits are done within `poll()` (which is typically called in a loop). How frequently offsets should be committed, can be configured via `auto.commit.interval.ms`. Because, auto commits are embedded in `poll()` and `poll()` is called by the user code, this parameter defines a lower bound for the inter-commit-interval.

As an alternative to auto commit, offsets can also be managed manually. For this, auto commit should be disabled (`enable.auto.commit = false`). For manual committing `KafkaConsumers` offers two methods, namely commitSync() and commitAsync(). As the name indicates, `commitSync()` is a blocking call, that does return after offsets got committed successfully, while `commitAsync()` returns immediately. If you want to know if a commit was successful or not, you can provide a call back handler (`OffsetCommitCallback`) a method parameter. Pay attention, that in both commit calls, the consumer commits the offsets of the latest `poll()` call. For example. let us assume a single partition topic with a single consumer and the last call to `poll()` return messages with offsets 4,5,6. On commit, offset 6 will be committed because this is the latest offset tracked by the consumer client. At the same time, both `commitSync()` and `commitAsync()` allow for more control what offset you want to commit: if you use the corresponding overloads that allow you to specify a `Map<TopicPartition, OffsetAndMetadata>` the consumer will commit only the specified offsets (ie, the map can contain any subset of assigned partitions, and the specified offset can have any value).

Semantics of committed offsets

A committed offset indicates, that all messages up to this offset got already processed. Thus, as offsets are consecutive numbers, committing offset `X` implicitly commits all offsets smaller than `X`. Therefore, it is not necessary to commit each offset individually, and committing multiple offsets at once, happens but just committing the largest offset.

Pay attention, that by design it is also possible to commit a smaller offset than the last committed offset. This can be done, if messages should be read a second time.

Processing guarantees

Using auto commit provides at-least-once processing semantics. The underlying assumption is, that `poll()` is only called after all previously delivered messages got processed successfully. This ensures, that no message get lost because a commit happens *after* processing. If a consumer fails before a commit, all messages after the last commit are received from Kafka and processed again. However, this retry might result in duplicates, as some message from the last `poll()` call might have been processed but the failure happened right before the auto commit call.

If at-most-once processing semantics are required, auto commit must be disabled and a manual `commitSync()` directly after `poll()` should be done. Afterward, messages get processed. This ensure, that messages are committed *before* there are processed and thus never read a second time. Of course, some message might get lost in case of failure.

How can I Read Topic From its Beginning

There are multiple strategies to read a topic from its beginning. To explain those, we first need to understand what happens at consumer startup. On startup of a consumer, the following happens:

1. join the configured consumer group, which triggers a rebalance and assigns partitions to the consumer
2. look for committed offsets (for all partitions that got assigned to the consumer)
3. for all partitions with valid offset, resume from this offset
4. for all partitions with not valid offset, set start offset according to `auto.offset.reset` configuration parameter

Start a new Consumer Group

If you want to process a topic from its beginning, you can simple start a new consumer group (i.e., choose an unused `group.id`) and set `auto.offset.reset = earliest`. Because there are no committed offsets for a new group, auto offset reset will trigger and the topic will be consumed from its beginning. Pay attention, that on consumer restart, if you use the same `group.id` again, it will not read the topic from beginning again, but resume where it left of. Thus, for this strategy, you will need to assign a new `group.id` each time you want to read a topic from its beginning.

Reuse the same Group ID

To avoid setting a new `group.id` each time you want to read a topic from its beginning, you can disable auto commit (via `enable.auto.commit = false`) before starting the consumer for the very first time (using an unused `group.id` and setting `auto.offset.reset = earliest`). Additionally, you should not commit any offsets manually. Because offsets are never committed using this strategy, on restart, the consumer will read the topic from its beginning again.

However, this strategy has two disadvantages:

1. it is not fault-tolerant
2. group rebalance does not work as intended

(1) Because offsets are never committed, a failing and a stopped consumer are handled the same way on restart. For both cases, the topic will be consumed from its beginning. (2) Because offset are never committed, on rebalance newly assigned partitions will be consumer from the very beginning.

Therefore, this strategy only works for consumer groups with a single consumer and should only be used for development purpose.

Reuse the same Group ID and Commit

If you want to be fault-tolerant and/or use multiple consumers in your Consumer Group, committing offsets is mandatory. Thus, if you want to read a topic from its beginning, you need to manipulate committed offsets at consumer startup. For this, `KafkaConsumer` provides three methods `seek()`, `seekToBeginning()`, and `seekToEnd()`. While `seek()` can be used to set an arbitrary offset, the second and third method can be use to seek to the beginning or end of a partition, respectively. Thus, on failure and on consumer restart seeking would be omitted and the consumer can resume where it left of. For consumer-stop-and-restart-from-beginning, `seekToBeginning()` would be called explicitly before you enter your `poll()` loop. Note, that `seekXXX()` can only be used after a consumer joined a group -- thus, it's required to do a "dummy-poll" before using `seekXXX()`. The overall code would be something like this:

```
if (consumer-stop-and-restart-from-beginning) {
    consumer.poll(0); // dummy poll() to join consumer group
    consumer.seekToBeginning(...);
}
```

```
// now you can start your poll() loop
while (isRunning) {
    for (ConsumerRecord record : consumer.poll(0)) {
        // process a record
    }
}
```

Read Consumer Groups and Offset Management online: https://riptutorial.com/apache-kafka/topic/5449/consumer-groups-and-offset-management

Chapter 3: Custom Serializer/Deserializer

Introduction

Kafka stores and transports byte arrays in its queue. The (de)serializers are responsible for translating between the byte array provided by Kafka and POJOs.

Syntax

- public void configure(Map<String, ?> config, boolean isKey);
- public T deserialize(String topic, byte[] bytes);
- public byte[] serialize(String topic, T obj);

Parameters

parameters	details
config	the configuration properties (`Properties`) passed to the `Producer` or `Consumer` upon creation, as a map. It contains regular kafka configs, but can also be augmented with user-defined configuration. It is the best way to pass arguments to the (de)serializer.
isKey	custom (de)serializers can be used for keys and/or values. This parameter tells you which of the two this instance will deal with.
topic	the topic of the current message. This lets you define custom logic based on the source/destination topic.
bytes	The raw message to deserialize
obj	The message to serialize. Its actual class depends on your serializer.

Remarks

Before version 0.9.0.0 the Kafka Java API used `Encoders` and `Decoders`. They have been replaced by `Serializer` and `Deserializer` in the new API.

Examples

Gson (de)serializer

This example uses the gson library to map java objects to json strings. The (de)serializers are generic, but they don't always need to be !

Serializer

Code

```
public class GsonSerializer<T> implements Serializer<T> {

    private Gson gson = new GsonBuilder().create();

    @Override
    public void configure(Map<String, ?> config, boolean isKey) {
        // this is called right after construction
        // use it for initialisation
    }

    @Override
    public byte[] serialize(String s, T t) {
        return gson.toJson(t).getBytes();
    }

    @Override
    public void close() {
        // this is called right before destruction
    }
}
```

Usage

Serializers are defined through the required `key.serializer` and `value.serializer` producer properties.

Assume we have a POJO class named `SensorValue` and that we want to produce messages without any key (keys set to `null`):

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
// ... other producer properties ...
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", GsonSerializer.class.getName());

Producer<String, SensorValue> producer = new KafkaProducer<>(properties);
// ... produce messages ...
producer.close();
```

(`key.serializer` is a required configuration. Since we don't specify message keys, we keep the `StringSerializer` shipped with kafka, which is able to handle `null`).

---

deserializer

Code

```
public class GsonDeserializer<T> implements Deserializer<T> {

    public static final String CONFIG_VALUE_CLASS = "value.deserializer.class";
    public static final String CONFIG_KEY_CLASS = "key.deserializer.class";
    private Class<T> cls;

    private Gson gson = new GsonBuilder().create();


    @Override
    public void configure(Map<String, ?> config, boolean isKey) {
        String configKey = isKey ? CONFIG_KEY_CLASS : CONFIG_VALUE_CLASS;
        String clsName = String.valueOf(config.get(configKey));

        try {
            cls = (Class<T>) Class.forName(clsName);
        } catch (ClassNotFoundException e) {
            System.err.printf("Failed to configure GsonDeserializer. " +
                    "Did you forget to specify the '%s' property ?%n",
                    configKey);
        }
    }


    @Override
    public T deserialize(String topic, byte[] bytes) {
        return (T) gson.fromJson(new String(bytes), cls);
    }


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

Usage

Deserializers are defined through the required `key.deserializer` and `value.deserializer` consumer properties.

Assume we have a POJO class named `SensorValue` and that we want to produce messages without any key (keys set to `null`):

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
// ... other consumer properties ...
props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer", GsonDeserializer.class.getName());
props.put(GsonDeserializer.CONFIG_VALUE_CLASS, SensorValue.class.getName());

try (KafkaConsumer<String, SensorValue> consumer = new KafkaConsumer<>(props)) {
    // ... consume messages ...
}
```

Here, we add a custom property to the consumer configuration, namely `CONFIG_VALUE_CLASS`. The `GsonDeserializer` will use it in the `configure()` method to determine what POJO class it should handle (all properties added to `props` will be passed to the `configure` method in the form of a map).

Read Custom Serializer/Deserializer online: https://riptutorial.com/apache-kafka/topic/8992/custom-serializer-deserializer

Chapter 4: kafka console tools

Introduction

Kafka offers command-line tools to manage topics, consumer groups, to consume and publish messages and so forth.

Important: Kafka console scripts are different for Unix-based and Windows platforms. In the examples, you might need to add the extension according to your platform.

Linux: scripts located in `bin/` with `.sh` extension.

Windows: scripts located in `bin\windows\` and with `.bat` extension.

Examples

kafka-topics

This tool let you list, create, alter and describe topics.

List topics:

```
kafka-topics  --zookeeper localhost:2181 --list
```

Create a topic:

```
kafka-topics  --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --
topic test
```

creates a topic with one partition and no replication.

Describe a topic:

```
kafka-topics  --zookeeper localhost:2181 --describe --topic test
```

Alter a topic:

```
# change configuration
kafka-topics  --zookeeper localhost:2181 --alter --topic test --config
max.message.bytes=128000
# add a partition
kafka-topics  --zookeeper localhost:2181 --alter --topic test --partitions 2
```

(Beware: Kafka does not support reducing the number of partitions of a topic) (see this list of configuration properties)

kafka-console-producer

This tool lets you produce messages from the command-line.

Send simple string messages to a topic:

```
kafka-console-producer --broker-list localhost:9092 --topic test
here is a message
here is another message
^D
```

(each new line is a new message, type ctrl+D or ctrl+C to stop)

Send messages with keys:

```
kafka-console-producer --broker-list localhost:9092 --topic test-topic \
        --property parse.key=true \
        --property key.separator=,
key 1, message 1
key 2, message 2
null, message 3
^D
```

Send messages from a file:

```
kafka-console-producer --broker-list localhost:9092 --topic test_topic < file.log
```

kafka-console-consumer

This tool let's you consume messages from a topic.

> to use the old consumer implementation, replace `--bootstrap-server` with `--zookeeper`.

Display simple messages:

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic test
```

Consume old messages:

In order to see older messages, you can use the `--from-beginning` option.

Display key-value messages:

```
kafka-console-consumer  --bootstrap-server localhost:9092 --topic test-topic \
    --property print.key=true \
    --property key.separator=,
```

kafka-simple-consumer-shell

This consumer is a low-level tool which allows you to consume messages from specific partitions, offsets and replicas.

Useful parameters:

- `parition`: the specific partition to consume from (default to all)
- `offset`: the beginning offset. Use `-2` to consume messages from the beginning, `-1` to consume from the end.
- `max-messages`: number of messages to print
- `replica`: the replica, default to the broker-leader (-1)

Exemple:

```
kafka-simple-consumer-shell  \
    --broker-list localhost:9092 \
    --partition 1 \
    --offset 4 \
    --max-messages 3 \
    --topic test-topic
```

displays 3 messages from partition 1 beginning at offset 4 from topic test-topic.

kafka-consumer-groups

This tool allows you to list, describe, or delete consumer groups. Have a look at this article for more information about consumer groups.

> if you still use the old consumer implementation, replace `--bootstrap-server` with `--zookeeper`.

List consumer groups:

```
kafka-consumer-groups  --bootstrap-server localhost:9092 --list
octopus
```

Describe a consumer-group:

```
kafka-consumer-groups  --bootstrap-server localhost:9092 --describe --group octopus
GROUP          TOPIC           PARTITION  CURRENT-OFFSET  LOG-END-OFFSET  LAG        OWNER
octopus        test-topic      0          15              15              0          octopus-
1/127.0.0.1
octopus        test-topic      1          14              15              1          octopus-
2_/127.0.0.1
```

Remarks: in the output above,

- `current-offset` is the last committed offset of the consumer instance,
- `log-end-offset` is the highest offset of the partition (hence, summing this column gives you the total number of messages for the topic)
- `lag` is the difference between the current consumer offset and the highest offset, hence how far behind the consumer is,
- `owner` is the `client.id` of the consumer (if not specified, a default one is displayed).

Delete a consumer-group:

> deletion is only available when the group metadata is stored in zookeeper (old consumer api). With the new consumer API, the broker handles everything including metadata deletion: the group is deleted automatically when the last committed offset for the group expires.

```
kafka-consumer-groups --bootstrap-server localhost:9092 --delete --group octopus
```

Read kafka console tools online: https://riptutorial.com/apache-kafka/topic/8990/kafka-console-tools

Chapter 5: Producer/Consumer in Java

Introduction

This topic shows how to produce and consume records in Java.

Examples

SimpleConsumer (Kafka >= 0.9.0)

The 0.9 release of Kafka introduced a complete redesign of the kafka consumer. If you are interested in the old `SimpleConsumer` (0.8.X), have a look at this page. If your Kafka installation is newer than 0.8.X, the following codes should work out of the box.

Configuration and initialization

> Kafka 0.9 no longer supports Java 6 or Scala 2.9. If you are still on Java 6, consider upgrading to a supported version.

First, create a maven project and add the following dependency in your pom:

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.9.0.1</version>
    </dependency>
</dependencies>
```

Note : don't forget to update the version field for the latest releases (now > 0.10).

The consumer is initialised using a `Properties` object. There are lots of properties allowing you to fine-tune the consumer behaviour. Below is the minimal configuration needed:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("group.id", "consumer-tutorial");
props.put("key.deserializer", StringDeserializer.class.getName());
props.put("value.deserializer", StringDeserializer.class.getName());
```

The `bootstrap-servers` is an initial list of brokers for the consumer to be able discover the rest of the cluster. This doesn’t need to be all the servers in the cluster: the client will determine the full set of alive brokers from the brokers in this list.

The `deserializer` tells the consumer how to interpret/deserialize the message keys and values. Here, we use the built-in `StringDeserializer`.

Finally, the `group.id` corresponds to the consumer group of this client. Remember: all consumers of a consumer group will split messages between them (kafka acting like a message queue), while consumers from different consumer groups will get the same messages (kafka acting like a publish-subscribe system).

Other useful properties are:

- `auto.offset.reset`: controls what to do if the offset stored in Zookeeper is either missing or out-of-range. Possible values are `latest` and `earliest`. Anything else will throw an exception;

- `enable.auto.commit`: if `true` (default), the consumer offset is periodically (see `auto.commit.interval.ms`) saved in the background. Setting it to `false` and using `auto.offset.reset=earliest` - is to determine where should the consumer start from in case no committed offset information is found. `earliest` means from the start of the assigned topic partition. `latest` means from the highest number of available committed offset for the partition. However, Kafka consumer will always resume from the last committed offset as long as a valid offset record is found (i.e. ignoring `auto.offset.reset`. The best example is when a brand new consumer group subscribes to a topic. This is when it uses `auto.offset.reset` to determine whether to start from the beginning (earliest) or the end (latest) of the topic.

- `session.timeout.ms`: a session timeout ensures that the lock will be released if the consumer crashes or if a network partition isolates the consumer from the coordinator. Indeed:

  > When part of a consumer group, each consumer is assigned a subset of the partitions from topics it has subscribed to. This is basically a group lock on those partitions. As long as the lock is held, no other members in the group will be able to read from them. When your consumer is healthy, this is exactly what you want. It’s the only way that you can avoid duplicate consumption. But if the consumer dies due to a machine or application failure, you need that lock to be released so that the partitions can be assigned to a healthy member. source

The full list of properties is available here http://kafka.apache.org/090/documentation.html#newconsumerconfigs.

Consumer creation and topic subscription

Once we have the properties, creating a consumer is easy:

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>( props );
consumer.subscribe( Collections.singletonList( "topic-example" ) );
```

After you have subscribed, the consumer can coordinate with the rest of the group to get its partition assignment. This is all handled automatically when you begin consuming data.

Basic poll

The consumer needs to be able to fetch data in parallel, potentially from many partitions for many topics likely spread across many brokers. Fortunately, this is all handled automatically when you begin consuming data. To do that, all you need to do is call `poll` in a loop and the consumer handles the rest.

`poll` returns a (possibly empty) set of messages from the partitions that were assigned.

```
while( true ){
    ConsumerRecords<String, String> records = consumer.poll( 100 );
    if( !records.isEmpty() ){
        StreamSupport.stream( records.spliterator(), false ).forEach( System.out::println );
    }
}
```

The code

Basic example

This is the most basic code you can use to fetch messages from a kafka topic.

```
public class ConsumerExample09{

    public static void main( String[] args ){

        Properties props = new Properties();
        props.put( "bootstrap.servers", "localhost:9092" );
        props.put( "key.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer" );
        props.put( "value.deserializer",
"org.apache.kafka.common.serialization.StringDeserializer" );
        props.put( "auto.offset.reset", "earliest" );
        props.put( "enable.auto.commit", "false" );
        props.put( "group.id", "octopus" );

        try( KafkaConsumer<String, String> consumer = new KafkaConsumer<>( props ) ){
            consumer.subscribe( Collections.singletonList( "test-topic" ) );

            while( true ){
                // poll with a 100 ms timeout
                ConsumerRecords<String, String> records = consumer.poll( 100 );
                if( records.isEmpty() ) continue;
                StreamSupport.stream( records.spliterator(), false ).forEach(
System.out::println );
            }
        }
    }
}
```

Runnable example

> The consumer is designed to be run in its own thread. It is not safe for multithreaded use without external synchronization and it is probably not a good idea to try.

Below is a simple Runnable task which initializes the consumer, subscribes to a list of topics, and executes the poll loop indefinitely until shutdown externally.

```
public class ConsumerLoop implements Runnable{
    private final KafkaConsumer<String, String> consumer;
    private final List<String> topics;
    private final int id;

    public ConsumerLoop( int id, String groupId, List<String> topics ){
        this.id = id;
        this.topics = topics;
        Properties props = new Properties();
        props.put( "bootstrap.servers", "localhost:9092");
        props.put( "group.id", groupId );
        props.put( "auto.offset.reset", "earliest" );
        props.put( "key.deserializer", StringDeserializer.class.getName() );
        props.put( "value.deserializer", StringDeserializer.class.getName() );
        this.consumer = new KafkaConsumer<>( props );
    }

    @Override
    public void run(){
        try{
            consumer.subscribe( topics );

            while( true ){
                ConsumerRecords<String, String> records = consumer.poll( Long.MAX_VALUE );
                StreamSupport.stream( records.spliterator(), false ).forEach(
System.out::println );
            }
        }catch( WakeupException e ){
            // ignore for shutdown
        }finally{
            consumer.close();
        }
    }


    public void shutdown(){
        consumer.wakeup();
    }
}
```

Note that we use a timeout of `Long.MAX_VALUE` during poll, so it will wait indefinitely for a new message. To properly close the consumer, it is important to call its `shutdown()` method before ending the application.

A driver could use it like this:

```
public static void main( String[] args ){

    int numConsumers = 3;
    String groupId = "octopus";
    List<String> topics = Arrays.asList( "test-topic" );

    ExecutorService executor = Executors.newFixedThreadPool( numConsumers );
    final List<ConsumerLoop> consumers = new ArrayList<>();

    for( int i = 0; i < numConsumers; i++ ){
        ConsumerLoop consumer = new ConsumerLoop( i, groupId, topics );
        consumers.add( consumer );
        executor.submit( consumer );
    }

    Runtime.getRuntime().addShutdownHook( new Thread(){
        @Override
        public void run(){
            for( ConsumerLoop consumer : consumers ){
                consumer.shutdown();
            }
            executor.shutdown();
            try{
                executor.awaitTermination( 5000, TimeUnit.MILLISECONDS );
            }catch( InterruptedException e ){
                e.printStackTrace();
            }
        }
    } );
}
```

SimpleProducer (kafka >= 0.9)

Configuration and initialization

First, create a maven project and add the following dependency in your pom:

```
<dependencies>
    <dependency>
        <groupId>org.apache.kafka</groupId>
        <artifactId>kafka-clients</artifactId>
        <version>0.9.0.1</version>
    </dependency>
</dependencies>
```

The producer is initialized using a `Properties` object. There are lots of properties allowing you to fine-tune the producer behavior. Below is the minimal configuration needed:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("client.id", "simple-producer-XX");
```

The `bootstrap-servers` is an initial list of one or more brokers for the producer to be able discover the rest of the cluster. The `serializer` properties tell Kafka how the message key and value should be encoded. Here, we will send string messages. Although not required, setting a `client.id` since is always recommended: this allows you to easily correlate requests on the broker with the client instance which made it.

Other interesting properties are:

```
props.put("acks", "all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
```

You can control the *durability of messages* written to Kafka through the `acks` setting. The default value of "1" requires an explicit acknowledgement from the partition leader that the write succeeded. The strongest guarantee that Kafka provides is with `acks=all`, which guarantees that not only did the partition leader accept the write, but it was successfully replicated to all of the in-sync replicas. You can also use a value of "0" to maximize throughput, but you will have no guarantee that the message was successfully written to the broker's log since the broker does not even send a response in this case.

`retries` (default to >0) determines if the producer try to resend message after a failure. Note that with retries > 0, message reordering may occur since the retry may occur after a following write succeeded.

Kafka producers attempt to collect sent messages into batches to improve throughput. With the Java client, you can use `batch.size` to control the maximum size in bytes of each message batch. To give more time for batches to fill, you can use `linger.ms` to have the producer delay sending. Finally, compression can be enabled with the `compression.type` setting.

Use `buffer.memory` to limit the total memory that is available to the Java client for collecting unsent messages. When this limit is hit, the producer will block on additional sends for as long as `max.block.ms` before raising an exception. Additionally, to avoid keeping records queued indefinitely, you can set a timeout using `request.timeout.ms`.

The complete list of properties is available here. I suggest to read this article from Confluent for more details.

Sending messages

The `send()` method is asynchronous. When called it adds the record to a buffer of pending record sends and immediately returns. This allows the producer to batch together individual records for efficiency.

The result of send is a `RecordMetadata` specifying the partition the record was sent to and the offset it was assigned. Since the send call is asynchronous it returns a `Future` for the RecordMetadata that will be assigned to this record. To consult the metadata, you can either call `get()`, which will block until the request completes or use a callback.

```
// synchronous call with get()
RecordMetadata recordMetadata = producer.send( message ).get();
// callback with a lambda
producer.send( message, ( recordMetadata, error ) -> System.out.println(recordMetadata) );
```

The code

```
public class SimpleProducer{

    public static void main( String[] args ) throws ExecutionException, InterruptedException{
        Properties props = new Properties();

        props.put("bootstrap.servers", "localhost:9092");
        props.put("acks", "all");
        props.put("retries", 0);
        props.put("batch.size", 16384);
        props.put("linger.ms", 1);
        props.put("buffer.memory", 33554432);
        props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        props.put("value.serializer",
"org.apache.kafka.common.serialization.StringSerializer");
        props.put( "client.id", "octopus" );

        String topic = "test-topic";

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

        for( int i = 0; i < 10; i++ ){
            ProducerRecord<String, String> message = new ProducerRecord<>( topic, "this is
message " + i );
            producer.send( message );
            System.out.println("message sent.");
        }

        producer.close(); // don't forget this
    }
}
```

Read Producer/Consumer in Java online: https://riptutorial.com/apache-kafka/topic/8974/producer-consumer-in-java

Credits

S. No	Chapters	Contributors
1	Getting started with apache-kafka	Ali786, Community, Derlin, Laurel, Mandeep Lohan, Matthias J. Sax, Mincong Huang, NangSaigon, Vivek
2	Consumer Groups and Offset Management	Matthias J. Sax, Sönke Liebau
3	Custom Serializer/Deserializer	Derlin, G McNicol
4	kafka console tools	Derlin
5	Producer/Consumer in Java	Derlin, ha9u63ar