NOVATEC

Apache Kafka / Confluent Platform

Exercises

Authors: Oliver Berger, Christopher Weiß, Uwe Eisele, Nadja Hagen

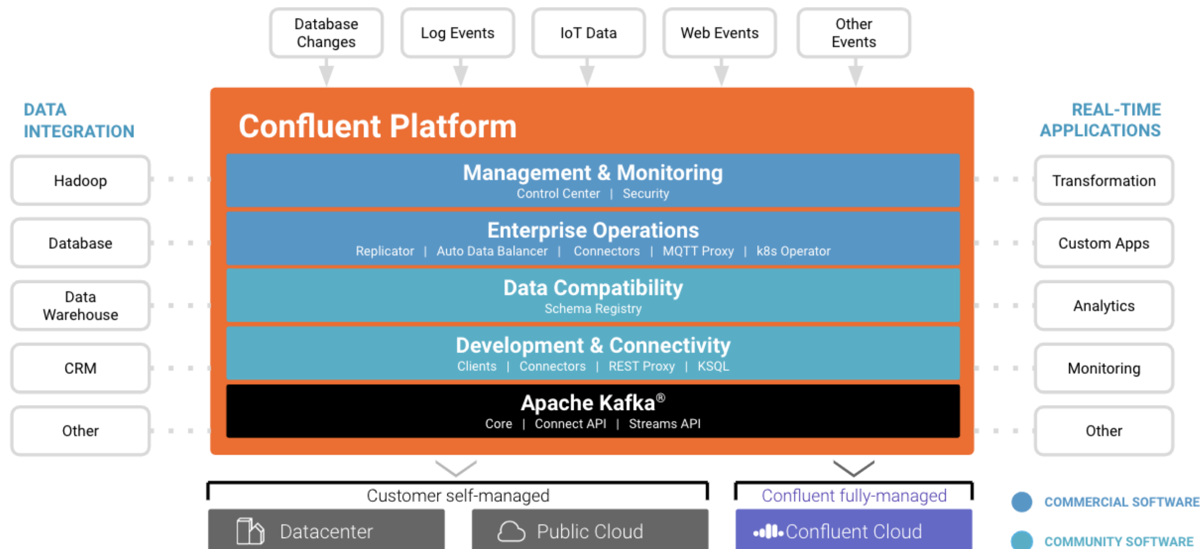
Date Created: 04.12.2020

Content

1 Installation ... **3**
1.1 Confluent Platform All-In-One ... 3
2 Kafka via Console ... **4**
2.1 Tasks ... 4
2.2 Writing Records into Kafka ... 4
2.3 Reading Records from Kafka ... 5
2.4 Topics with Multiple Partitions ... 5
3 Kafka via Java API ... **6**
3.1 Tasks ... 6
3.2 Writing Records into Kafka ... 7
3.3 Reading Records from Kafka ... 8
3.4 Additional Tasks (Advanced) ... 9

1 Installation

For the exercise, we use the *Confluent Platform*. Confluent Platform is built on Apache Kafka and extends it with many useful features and tools.



Quelle: https://docs.confluent.io/current/platform.html

1.1 Confluent Platform All-In-One

The installation is done using Docker Compose[1,2]. First, we download the necessary Docker Compose file. To do this, we use the following commands:

```
$ git clone https://github.com/confluentinc/cp-all-in-one
$ cd cp-all-in-one
$ git checkout 6.0.1-post
$ cd cp-all-in-one-community
```

In this directory you should now find the file *docker-compose.yml*. Now, we just need to download the necessary Docker images and start the Confluent Platform. This is easily done with a single command:

```
$ docker-compose up -d zookeeper broker
```

Several Docker images will be downloaded, so do not be surprised if it takes a few minutes for everything to download. When everything has gone through successfully, you should see the following output:

[1] https://docs.docker.com/compose/install/
[2] According to Confluent, you need 8GB Docker Memory Allocation for this. If your computer does not have the necessary amount of memory, you can also install the normal Apache Kafka either via Docker Compose (http://wurstmeister.github.io/kafka-docker/) or do a local installation (steps 1 + 2 from: https://kafka.apache.org/quickstart).

```
Starting zookeeper ... done
Creating broker    ... done
```

Zookeeper and Kafka Broker are now up and running.

2 Kafka via Console

To access the included Kafka command line tools, we need to 'dial in' to the Docker container of the Kafka Broker via shell with[3]:

```
$ docker exec -it broker bash
```

Now we have access to useful tools with which we can, among other things, write records into Kafka via console commands and read them out again.

2.1 Tasks

Try to solve the following tasks. In the next paragraphs, you will find hints and a short documentation which will guide you through the tasks if you need some help.

- Create a topic with 3 partitions.
- Write numbered records without keys into the topic (e.g., value1, value2, value3, etc.) using the *kafka-console-producer*.
- Read the topic with the *kafka-console-consumer*. What do you notice?
- Find out how to use *kafka-console-consumer* to read from a specific partition and check to which partitions your records were added (hint: just type '*kafka-console-consumer*' and press *Enter* to see the list of parameters).
- Enter key-value pairs in the topic and check if the entries with the same keys really end up in the same partition.

2.2 Writing Records into Kafka

With the *kafka-console-producer* we can add data to a specific topic. For this we have to pass two parameters:

- **bootstrap.server:** List of brokers in the Kafka cluster (it is sufficient to specify a single broker). The default port for Kafka brokers is 9092.
- **topic:** The name of the topic to describe.

```
# kafka-console-producer --bootstrap-server localhost:9092 --topic words
>
```

[3] This step is not necessary when installing Kafka manually. However, you must be in the Kafka folder for the following examples to work.

After executing the command, you will see the > symbol. Now you can type in entries and send them with the *Enter* key. We will see a warning message first, because we did not create the topic *words* before. Fortunately, Kafka is quite tolerant by default and creates the topic for us automatically (with one partition).

The created entries are only created with a value. If we want to add a key to the value, we can specify a separator. Which separator we want to use, we have to tell the *kafka-console-producer* by parameter (first end the previous process with *ctrl-c*):

```
# kafka-console-producer --bootstrap-server localhost:9092 --topic words --property parse.
key=true --property key.separator=:
>
```

Now we can specify a key-value pair such as *'1:Hello'*.

2.3 Reading Records from Kafka

If we want to read out the records again, we can start the *kafka-console-consumer*. Again, we need to pass a few properties:

- **bootstrap.server***:* List of brokers in the Kafka cluster (it is sufficient to specify a single broker). The default port of the Kafka broker is *9092*.
- **topic**: The name of the topic we want to describe.
- **from-beginning**: If we do not pass this parameter, only new entries will be output that were entered after the start of the *kafka-console-consumer*.

```
# kafka-console-consumer --bootstrap-server localhost:9092 --topic words --from-beginning
Hello
World
!
key1:value1
42:the answer
```

2.4 Topics with Multiple Partitions

Since our topic *words* was created automatically by Kafka, it has only one partition. This is functional, but not usual. Therefore, we want to continue a little more realistically and create a topic with 3 partitions.

Via the console we can execute the following command:

```
# kafka-topics --bootstrap-server localhost:9092 --create --topic numbers --partitions 3
```

We can now fill the topic with new data using *kafka-console-producer*.

3 Kafka via Java API

The *kafka-console-consumer* application is particularly useful to get to know Kafka quickly and easily. Normally, however, it is our own applications that wants to communicate with Kafka (reading or writing). For Java and other languages in the JVM environment (Kotlin, Groovy, Scala, etc.), there is a client library that we can easily integrate via Maven Dependency.

```
<dependencies>
   …
   <dependency>
      <groupId>org.apache.kafka</groupId>
      <artifactId>kafka-clients</artifactId>
      <version>2.6.0</version>
   </dependency>
   …
</dependencies>
```

This library contains both the *Producer*- and *Consumer-APIs* with which we can access Kafka directly in source code.

3.1 Tasks

- Clone the github repository for this exercise: https://github.com/nadjahagen/frauas_kafka_exercises
  It contains a skeleton for the following tasks. Alternatively, you can create a Java/Maven project (*KafkaConsumerDemo*) in an IDE of your choice.
- Use the template of *KafkaProducerDemo*. Generate data and write it to the topic *words* in Kafka.
- Write code into the class *KafkaConsumerDemo* to read records from the existing Kafka Topic *words* and output them to the console with *System.out.println()*.
- Run both applications simultaneously and observe the output in the console.
- Customise your application to display all records starting with offset 0 (hint: we learned about the *--from-beginning* parameter in *kafka-console-consumer*, find the right property key/value in *ConsumerConfig* and set it). If you already started the application, you won't see any effect. What is the reason for this? How can you achieve to always read from the beginning?
- How could we adapt the above example to continuously display data instead of constantly restarting the application? What happens if you increase the argument *timeout* of the *poll()* method?

3.2 Writing Records into Kafka

Let us rebuild the example from 2.2. in Java by also adding a string to the topic *words*.

```
public class KafkaProducerDemo {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
                                "localhost:9092");
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
                              "org.apache.kafka.common.serialization.StringSerializer");
        properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
                                "org.apache.kafka.common.serialization.StringSerializer");

        ProducerRecord<String, String> record = new ProducerRecord<>("words", "Hello World!");

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

        producer.send(record);

        producer.close();

    }
}
```

The configuration is done via properties. To avoid having to remember all the property keys, the library offers an Enum called *ProducerConfig*, in which all the relevant keys can be found.

We set the three mandatory properties:

- **BOOTSTRAP_SERVERS_CONFIG:** Address of our Kafka Broker
- **KEY_SERIALIZER_CLASS_CONFIG**: Serialiser matching the key
- **VALUE_SERIALIZER_CLASS_CONFIG**: Serializer matching the Value

BOOTSTRAP_SERVERS_CONFIG we already know from the *kafka-console-producer*, there the property was called '--bootstrap-servers'. The two serializers are new and have to be specified with a fully qualified class name. Since Kafka stores everything internally as a byte array, we have to tell the producer how to serialise the value we pass into a byte array.

For the simple data types, Kafka already offers ready-made serialisers, e.g:

- `StringSerializer`
- `LongSerializer`
- `DoubleSerializer`
- `Etc.`

For more complex data types, such as JSON, there are many libraries that can be additionally integrated into Maven.

The key-value pairs that we want to send to Kafka must be passed to the *ProducerRecord* class. This class has many different constructors, of which we choose the simplest variant (Topic-Name, Value).

We need the *KafkaProducer* to communicate with Kafka. With the *send()* method we can transmit the record.

3.3 Reading Records from Kafka

Reading out records is similar. We create the necessary properties and pass them to the constructor of the class *KafkaConsumer* (analogous to KafkaProducer).

Now we can tell the consumer which topics it should read out. We do this with the *subscribe()* method.

With the *poll()* method we can now read the records from the specified Kafka Topics for a defined period (in this example: 2000ms). The *poll()* method returns a *ConsumerRecords* object with the corresponding results. It should be noted, however, that only records that arrived in the Kafka Topic during the time period of the *poll()* call are considered.

To make this example work, one important Consumer property is missing. Can you remember which property it is?

```
public class KafkaConsumerDemo {

    public static void main(String[] args) {
        Properties properties = new Properties();
        properties.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                                    "localhost:9092");
        properties.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                                 "org.apache.kafka.common.serialization.StringDeserializer");
        properties.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                                    "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(properties);

        consumer.subscribe(Arrays.asList("words"));

        try {
            ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(2000));
              System.out.println("Records read: " + records.count());
              for (ConsumerRecord<String, String> record : records) {
                  System.out.println("offset: " + record.offset() + ", key: " + record.key() + ", value: " +
record.value());
             }
        } finally {
            consumer.close();
        }

    }
}
```

How the target architecture will look like:

3.4 Additional Tasks (Advanced)

- It is important to properly quit a program and to make sure that the consumer is closed. How could you ensure this when your application *KafkaConsumerDemo* is running using a while-loop? Have a look on how to use *ShutDownHooks* in combination with a *CountDownLatch*.
- Extend *KafkaProducerDemo* to take arguments from the command line which will then be written into a topic in Kafka.