Week3 - Kafka Homework #1

Overview

This homework will help you master the essential aspects and operation of Apache Kafka and understand of Kafka with regard to administration, operation and application development.

Submission

You need to provide exercises commands output history and source code

Grading

This Homework is graded 10 scores from overall 100 for the course.

Penalty

For every day after the deadline maximum score for the homework is reduced by 10%.

Assignments

Exercise 1:

Install Confluent Platform community components on local environment using Step 1 from:
https://docs.confluent.io/current/quickstart/cos-quickstart.html
Use CLI tools to get info on kafka and zookeeper and send/receive text messages

Exercise 2:

Install 3 node kafka cluster using
https://github.com/confluentinc/cp-docker-images/blob/master/examples/kafka-cluster/docker-compose.yml and 1-4 steps of instructions from
https://docs.confluent.io/current/installation/docker/installation/automatic-data-balancing.html#docker-client-setting-up-a-three-node-kafka-cluster

Exercise 3:

This exercise is to give you time to explore the very depths of a Kafka cluster, i.e. Zookeeper. Zookeeper is used as a runtime configuration database by Kafka brokers
Procedure:

1. Use *zookeeper-shell* to connect to the Zookeeper instance
   ./bin/zookeeper-shell.sh :2181

2. Use **help** to learn about available commands
3. Use **ls** and **get** commands to explore the Kafka znodes
   ls /brokers/ids
   get /brokers/ids/0
   get /controller

NOTE: zookeeper-shell comes with no support for command history and so consider the official Apache Zookeeper's **zkCli** for a better command-line experience (and command history).

Exercise 4:

In this exercise you will use **kafka-topics** shell script to manage a topic on a multi-broker Kafka cluster.

1. Use a 3-broker Kafka cluster from Ex. 2
2. Review the available options of **kafka-topics** shell script
   Use *--help* option to print usage information
3. Create a topic *t1* with 3 partitions and replication factor of 2
   Use *--create* command-line option
   ```
   ./bin/kafka-topics.sh --zookeeper :2181 --create --topic t1 --partitions 3 --replication-factor 2
   ```
4. List all available topics
   Use *--list* command-line option
5. Review topic configuration of t1 topic
   Use *--describe* command-line option
   ```
   ./bin/kafka-topics.sh --zookeeper :2181 --describe --topic t1
   ```
6. Focus on Leader, Replicas, ISR

Exercise 5:

In this exercise you will create a consumer group and observe what happens when the number of members changes.

1. Create a topic t2 with 3 partitions
2. Start a new consumer c1 in a consumer group CG1
3. Start a new producer and send messages

   At this point you should have 3 partitions and 1 consumer. Observe what and how messages are consumed. Simply send messages so you can identify what message used what partition.

4. Start a new consumer c2 in the CG1 consumer group
5. Observe what messages are consumed by the consumers
6. Start a new consumer c3 in the CG1 consumer group
7. Observe what messages are consumed by the consumers

8. Shut down any of the running consumers
9. Observe which consumer takes over the partition
10. Shut down the partition leader
11. Observe what happens with the clients and whether they can communicate with the cluster
12. Use *kafka-topics* to check out the leader

Exercise 6:

Developing Kafka Producer

1. Read the javadoc of KafkaProducer to know how to use the Producer API (to send messages to Kafka)
2. Create a new Scala/sbt project in IntelliJ IDEA
    i. Leave the defaults for the versions of sbt and Scala
    ii. They usually are the latest versions
3. Define dependency for Kafka Clients API library
    i. Use mvnrepository to know the proper entry for `kafka-clients` dependency
4. Write the code of a Kafka producer
    i. Name of the object: KafkaProducerApp
    ii. Start with an empty `Properties` object and fill out the missing properties per exceptions at runtime
    iii. Use ProducerConfig constants (not string values for properties)
    iv. Don't forget to `close` the producer (so the messages are actually sent out to the broker)
5. Run the producer
    i. Use `kafka-console-consumer` to receive the messages
6. Fix the slf4j logging errors
    i. Define dependencies for `slf4j-api` and `slf4j-log4j12` in `build.sbt`
    ii. Create `src/main/resources/log4j.properties` as the configuration file
    iii. Use Apache Kafka's `config/log4j.properties` as the sample (4 first non-comment lines)
7. Review the available `send` methods in KafkaProducer
8. Read the javadoc of KafkaProducer.send(ProducerRecord<K,V> record, Callback callback)
    i. Explore Callback interface
9. Answer the following questions with regards to the Callback interface:
    i. What does Callback give when you send a message to an existing topic?
    ii. What happens when sending a message to an unexisting topic?
    iii. Mind the automatic topic creation feature

Exercise 7:

Write a new Kafka application `PartitionerDemo` (using Kafka Producer API) as follows:

1. Write a custom `Partitioner`
    i. Implement Partitioner interface
    ii. Review Cluster (which is one of the input arguments of `Partitioner.partition` method)
    iii. Decide what to do when requested for a partition ID in `partition` method
2. Write a KafkaProducer
    i. Register the custom `Partitioner` using ProducerConfig.PARTITIONER_CLASS_CONFIG property
    ii. Use a `Callback` input object (to `Producer.send`) and display the partition ID
3. Create a topic with 2 partitions (on a single Kafka broker)
    i. Hint: Use as many `kafka-console-consumer` as many partitions are in use
    ii. Use `kafka-console-consumer` with `--property print.key=true` to print keys
4. Execute the application and the following tests. Observe the behaviour.
    i. Define the partition key in `Producer.send`
    ii. Don't specify the partition key
5. Use `kafka-topics --alter` to increase the number of partitions and observe how `Cluster` reflects the change
    i. `./bin/kafka-topics.sh --zookeeper :2181 --alter --topic PartitionerDemo-input --partitions 3`

Exercise 8:

Write a new Kafka application `CustomClassForKeyDemo` (using Kafka Producer API) as follows:

1. Write a custom class with two properties, e.g. `instID` and `empID`
    i. `case class InstEmp`
2. Implement a custom `Serializer` for the class
    i. Read Serializer
    ii. `class InstEmpSerializer extends Serializer`
3. Write a Kafka producer that uses the class for keys and the serializer
    i. Use `ProducerConfig.KEY_SERIALIZER` property to register the serializer
    ii. Use `InstEmp` for the keys of `ProducerRecord` objects (and any type for values)
4. Use `kafka-console-consumer` to print out the records

Exercise 9:

Developing Kafka Producer

1. Read the javadoc of KafkaConsumer to know how to use the Consumer API (to consume messages from Kafka)
2. Create a new Scala/sbt project in IntelliJ IDEA
3. Define dependency for Kafka Clients API library
    i. Use mvnrepository to know the proper entry for `kafka-clients` dependency
4. Write a Kafka consumer
    i. Name of the object: KafkaConsumerApp
    ii. Start with an empty `Properties` object and fill out the missing properties per exceptions at runtime
    iii. Use ConsumerConfig constants (not string values for properties)
    iv. Don't forget to `close` the consumer (so the messages are actually acknowledged to the broker)
5. Run the Kafka consumer
    i. Use `kafka-console-producer` to produce messages
6. Fix the slf4j logging errors
    i. Define dependencies for `slf4j-api` and `slf4j-log4j12` in `build.sbt`
    ii. Create `src/main/resources/log4j.properties` as the configuration file
    iii. Use Apache Kafka's `config/log4j.properties` as the sample (4 first non-comment lines)

Exercise 10:

Write a new Kafka application `WordCountPerLineApp` (using Kafka Producer and Consumer APIs) that does the following:

1. Consumes records from a topic, e.g. `input`
2. Counts words (in the value of a record)
3. Produces records with the unique words and their occurrences (counts)
    i. A record `key -> hello hello world` gives a record with the following value `hello -> 2, world -> 1` (and the same key as in the input record)
4. Produces as many records as there are unique words in the input record with their occurences (counts)
    i. A record `key -> hello hello world` gives two records in the output, i.e. `(hello, 2)` and `(world, 1)` as `(key, value)`