Day 1 – Lab2:

1. Introduction

In this lab we'll use Java and Maven to create a typical Java based producer and consumer of Kafka messages. You have two options for running Maven and Java. Either you install them natively or you can run the tools through a docker container.

Two separate labs

This lab consists of two parts. In the first part, we'll create a producer that can create messages in Kafka. In the second part, we'll consume these messages.

Both the `consumer` and `producer` are written in Java.

2. Developing Kafka Applications - Producer API

In this lab, you will create a Kafka Producer using the Java API. The next lab will be the creation of the Kafka Consumer so that you can see an end-to-end example using the API.

Objectives

1. Create topics on the Kafka server for the producer to send messages
2. Understand what the producer Java code is doing
3. Build & package the producer
4. Run the producer

Prerequisites

Like the previous lab, Docker will be used to start a Kafka and Zookeeper server. We will also use a Maven Docker image to compile & package the Java code and an OpenJDK image to run it.

You should have a text editor available with Java syntax highlighting for clarity. You will need a basic understanding of Java programming to follow the lab although coding will not be required. The Kafka Producer example will be explained and then you will compile and execute it against the Kafka server.

Instructions

All the directory references in this lab is relative to where you expended the lab files and `labs/02-Publish-And-Subscribe`

1. Open a terminal in this lesson's root directory.

2. Start the Kafka and Zookeeper containers using Docker Compose:

3. `$ docker-compose up`

4. Open an additional terminal window in the lesson directory.

5. Open `producer/src/main/java/app/Producer.java` in your text editor. This class is fairly simple Java application but contains all the functionality necessary to operate as a Kafka Producer. The application has two main responsibilities:
    - Initialize and configure a KafkaProducer
    - Send messages to topics with the `KafkaProducer` object

    To create our producer object, we must create an instance of `org.apache.kafka.clients.producer.KafkaProducer` which requires a set of properties for initialization. While it is possible to add the properties directly in Java code, a more likely scenario is that the configuration would be externalized in a properties file.
    Open `resources/producer.properties` and you can see that the configuration is minimal.

    - `acks` is the number of acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent. A setting of `all` is the strongest guarantee available.
    - Setting `retries` to a value greater than 0 will cause the client to resend any record whose send fails with a potentially transient error.
    - `bootstrap.servers` is our required list of host/port pairs to connect to Kafka. In this case, we only have one server. The Docker Compose file exposes the Kafka port so that it can be accessed through `localhost`.
    - `key.serializer` is the serializer class for key that implements the `Serializer` interface.
    - `value.serializer` is the serializer class for value that implements the `Serializer` interface.
    - `boostrap.servers.docker` is only used if we detect that the code is running inside a Docker container.

    There are many configuration options available for Kafka producers that should be explored for a production environment.

6. Once you have a `KafkaProducer` instance, you can post messages to a topic using the `ProducerRecord`. A `ProducerRecord` is a key/value pair that consists of a topic name to

which the record is being sent, an optional partition number, an optional key, and required value.
In our lab, we are going to send a bunch of messages in a loop. Each iteration of the loop will consist of sending a message to the `user-events` topic with a key/value pair. Every so often, we also send a randomized message to the `global-events` topic. The message sent to `global-events` does not have a key specified which normally means that Kafka will assign a partition in a round-robin fashion but our topic only contains 1 partition.
After the loop is complete, it is important to call `producer.close()` to end the producer lifecycle. This method blocks the process until all the messages are sent to the server. This is called in the `finally` block to guarantee that it is called. It is also possible to use a Kafka producer in a try-with-resources statement.

7. Now we are ready to compile the lab. In a terminal, change to the lab's `producer` directory and run the following command:

If you are on a Mac (or in linux), this should work:

```
$ docker run -it --rm -v "$(cd "$PWD/../.."; pwd)":/course-root -w "/course-root/$(basename
$(cd "$PWD/.."; pwd))/$(basename "$PWD")" -v "$HOME/.m2/repository":/root/.m2/repository
maven:3-jdk-11 ./mvnw clean package
```

On a windows machine, you have to replace the `$PWD` with the current directory and the `$HOME` with a directory where you have the `.m2` folder.

7. With the producer now built, run it with the following command:

```
8. $ docker run --network 02-publish-and-subscribe_default --rm -it -v "$PWD:/pwd" -w /pwd
   openjdk:11 java -jar target/pubsub-producer-*.jar
9. SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
10. SLF4J: Defaulting to no-operation (NOP) logger implementation
11. SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
12. ...
```

13. *Don't* stop the Kafka and Zookeeper servers because they will be used in the next lab focusing on the Consumer API.

Conclusion

We have now successfully sent a series of messages to Kafka using the Producer API. In the next lab, we will write a consumer program to process the messages from Kafka.

Congratulations, this lab is complete!

3.Developing Kafka Applications - Consumer API

In this lab, you will create a Kafka Consumer using the Java API. This is the continuation of the previous lab in which a `KafkaProducer` was created to send messages to two topics: `user-events` and `global-events`.

Objectives

1. Understand what the consumer Java code is doing
2. Compile and run the consumer program
3. Observe the interaction between producer and consumer programs

Prerequisites

Like the previous lab, Docker will be used to start a Kafka and Zookeeper server. We will also use a Maven Docker image to compile & package the Java code and an OpenJDK image to run it. You should have already completed the previous `KafkaProducer` lab so that there are messages ready in the Kafka server for the Consumer to process.

Instructions

1. Open `consumer/src/main/java/com/example/Consumer.java` in your favorite text editor. Like the producer we saw in the previous lab, this is a fairly simple Java class but can be expanded upon in a real application. For example, after processing the incoming records from Kafka, you would probably want to do something interesting like store them in HBase for later analysis. This application has two main responsibilities:
    - Initialize and configure a `KafkaConsumer`
    - Poll for new records in an infinite loop

    The first thing to notice is that a `KafkaConsumer` requires a set of properties upon creation just like a `KafkaProducer`. You can add these properties directly to code but a better solution is to externalize them in a properties file.
2. Open `resources/consumer.properties` and you see that the required configuration is minimal like the producer.
    - `bootstrap.servers` is our required list of host/port pairs to connect to Kafka. In this case, we only have one server.
    - `key.deserializer` is the deserializer class for key that implements the `Deserializer` interface.
    - `value.deserializer` is the deserializer class for value that implements the `Deserializer` interface.

- `group.id` is a string that uniquely identifies the group of consumer processes to which this consumer belongs. In our case, we are just using the value of `test` for an example.
- `auto.offset.reset` determines what to do when there is no initial offset in Zookeeper or Kafka from which to read records. The first time that a consumer is run will be the first time that the Kafka broker has seen the consumer group that the consumer is using. The default behavior is to position newly created consumer groups at the end of existing data which means that the producer data that we ran previously would not be read. By setting this to `earliest`, we are telling the consumer to reset the offset to the smallest offset.
- `boostrap.servers.docker` is only used if we detect that the code is running inside a Docker container.

Like the producer, there are many configuration options available for Kafka consumer that should be explored for a production environment.

3. Open `consumer/src/main/java/com/example/Consumer.java` again. A consumer can subscribe to one ore more topics. In this lab, you can see that the consumer will listen to messages from two topics.

4. Once the consumer has subscribed to the topics, the consumer then polls for new messages in an infinite loop.

   For each iteration of the loop, the consumer will fetch records for the topics. On each poll, the consumer will use the last consumed offset as the starting offset and fetch sequentially. The `poll` method takes a timeout in milliseconds to spend waiting if data is not available in the buffer.
   The returned object of the `poll` method is a `ConsumerRecords` that implements `Iterable` containing all the new records. From there our example lab just uses a `switch` statement to process each type of topic. In a real application, you would do something more interesting here than output the results to `stdout`.
5. Now we are ready to compile and run the lab. In a terminal, change to the `lab` directory and run the following command:
   ```
   $ docker run -it --rm -v "$(cd "$PWD/../.."; pwd)":/course-root -w "/course-root/$(basename $(cd "$PWD/.."; pwd))/$(basename "$PWD")" -v "$HOME/.m2/repository":/root/.m2/repository maven:3-jdk-11 ./mvnw clean package
   ```

On a windows machine, you have to replace the `$PWD` with the current directory and the `$HOME` with a directory where you have the `.m2` folder.

6. With the consumer now built, run it with the following command:

```
$ docker run --network 02-publish-and-subscribe_default --rm -it -v "$PWD:/pwd" -w /pwd openjdk:11 java -jar target/pubsub-consumer-*.jar
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
...
```

After the consumer has processed all the messages, start the producer again in another terminal window and you will see the consumer output the messages almost immediately. The consumer will run indefinitely until you press `ctrl-c` in the terminal window.

12. [OPTIONAL] Play with the performance. Before we shut down Kafka, you may want to spend some time playing with the performance of the programs. We have created an optional lab for this which you can run here before going to step 8 and shutting down Kafka. Here is a link to the lab.

13. Finally, change back into the `docker/` directory in order to shut down the Kafka and Zookeeper servers.

```
14. $ docker-compose down
```

Conclusion

We have now seen in action a basic producer that sends messages to the Kafka broker and then a consumer to process them. The examples we've shown here can be incorporated into a larger, more sophisticated application.

Congratulations, this lab is complete!