Lab 5 Event-Driven and Message Queue with Apache Kafka

Updated: 2024-11-15.

Introduction to the lab

Learning outcomes

- Deploy Apache Kafka in Docker.
- Create an application to produce and consume messages with Apache Kafka and Spring Boot.

References and suggested readings

- "Guide to Setting Up Apache Kafka Using Docker"

Submission

This is a **week lab**. You may submit as you go but be sure to complete and submitted by the deadline outlined in the class plan (see slides 10 and 11 from the first class).
Remember to create the "lab5" folder in your Git repository. Be sure to create and update the lab "notebook, e.g. in 🗁lab5/README.md . [The notebook is where you take notes of quick reference info, links and visuals, so you can study from this content later; it is an import part of your submission.]

5.1 Configure Apache Kafka with Docker

Apache Kafka is an open-source distributed event streaming platform thousands of companies use for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. This guide aims to help you use it as a message queue, produce and consume messages, and design messages and different queues. The documentation allow you to understand how it works and what are the use cases.

To manage Kafka, there are several tools that you can use. You can use that, if you would like to, Kafdrop, Kafka Tool, etc.

a. Deploy the Apache Kafka with a docker (use docker compose, here is an example). It already brings the Kafdrop, but you can use additional tool to manage Kafka. It allow you to configure a Kafka with single node.
b. It is possible to organize the messages in categories of messages, and topic is the mechanism used by Kafka. Each topic has a name unique across the entire Kafka Cluster. The message can be sent and read from specific topics, and you can have multiple subscribers for the same topics. Let's start by creating a topic:

```
$ docker exec lab05-kafka-1 kafka-topics --create --topic lab05 --partitions 1 --
replication-factor 1 --bootstrap-server kafka:9092
```

c. Open Kafdrop (http://localhost:9009) and verify if the topic was created.
d. With the Kafka topic in place, let's publish and consume some messages. First, start a consumer by running the following command:

```
$ docker exec lab05-kafka-1 kafka-console-consumer --topic lab05 --from-beginning
--bootstrap-server kafka:9092
```

e. Using the above command, it will be possible to consume all the messages sent over to this topic. Additionally, we used *--from-beginning* to consume all messages sent over the topic from the beginning. Let's also look at publishing the data to this Kafka topic, but be sure *you open other terminal*:

```
$ docker exec -it lab05-kafka-1 bash
$ kafka-console-producer --topic lab05 --broker-list kafka:9092
Type any message
```

By using the above command, we can generate and send messages to the lab05 topic.

f. What happens if you open multiple consumer terminals? And if you open multiple terminals with producers? What happens with old messages? Explain it.

5.2 Create a producer and consumer

a. Create a new Poetry project with Python to produce and consume any random message to Kafka. You can use the basic usage of Petry documentation. Files will be named producer_example.py and consumer_example.py. Read the documentation and usage (https://kafka-python.readthedocs.io/en/master/usage.html). If you have problems with Python >=3.12.X, see here a solution. Verify if you can produce and consume messages from the topic with the following name: "lab05_nMec" (e.g. lab05_99999)

b. The producer should produce the message with the following format:

```
message = {'nMec': '99999', 'generatedNumber': 1, 'type: 'fibonacci'}
```

It should produce all the messages of the sequence until the *nMec* number (<= *nMec*). Run it only once.

c. The consumer should read all the messages from the topic. What is the last message?

d. If you run the consumer multiple times, does it read all the messages? Why?

Note: read about consumer commit and retention time.

5.3 Create a consumer in Java integrated with Spring Boot

Apache Kafka can be used in multiple languages, and you can produce events in one application and consume them in a different application, independently of the language, as it is a distributed cluster. Follow the Spring for Apache Kafka tutorial to create your first Kafka Application with Spring Boot: https://docs.spring.io/spring-kafka/reference/quick-tour.html

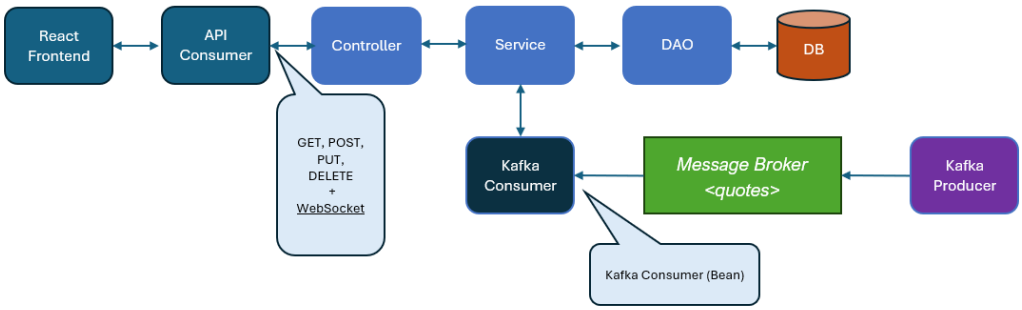
a. Create a new project in Spring Boot with the capability to listen to messages (section Sending a Message).
b. Use the *producer.py* (from 3.2) and verify if you receive the messages in the Spring Boot application (section Receiving a Message).
c. Add the capability to send a message and make sure that you can also receive it from *consumer.py* (from 3.2).
d. Implement a Serializer in Spring Boot to deserialize the message produced by producer.py. You can use the following example: https://howtodoinjava.com/kafka/spring-boot-jsonserializer-example/. Some helpful configurations for *application.properties*:

```
spring.kafka.consumer.value-deserializer:
org.springframework.kafka.support.serializer.JsonDeserializer
spring.kafka.consumer.properties.spring.json.trusted.packages=*
spring.kafka.consumer.properties[spring.json.value.default.type]=com.ies.dem
o_kafka_spring.entities.Message
spring.kafka.producer.properties.spring.json.add.type.headers=false
```

Note: in producer.py, make sure that the messages are sent with *json.dumps* method.

5.4 Wrapping-up and integrating concepts

Consider the last exercise from the previous lab (Exercise 4.4). Extend the exercise to support new quotes added by an external process and send them through message broker. It is also expected that the frontend will update with new quotes without requiring a refresh in the browser.



a) Create a Python script to generate new quotes for movies (you can rely on exercise 5.2) for every 5 to 10 seconds and introduce them in a Kafka topic.

b) Extend the SpringBoot to consume the quotes from Kafka topic, and introduce them into the database.

c) Extend the React Frontend to show the latest received 5 quotes.

d) Extend the React Frontend to update the quotes without require to refresh the browser. Consider to use Websocket technology using the SpringBoot Websocket guides.

e) Deploy the full application to Docker containers. Consider using docker compose (related example).