Kafka & Kafka Stream with Spring Boot

- Apache Kafka = **Event Streaming Platform**
- To build near **real-time integration** across multiple systems
- **Kafka in microservices architecture**

Messaging Systems

- They provide a communication mechanism between applications, sometimes also referred to as **A2A**
- They are widely used in enterprise applications

Example Use Case:

Let's suppose we have these three different systems within the organization:

- HR System
- Marketing System
- AD System

The HR System should handle the following events:

- Employees joining the company
- Employees leaving the company

The Marketing System should do the following:

- Allow access to employees joining the company
- Revoke access to employees leaving the company

The AD System:
Should store Employees credentials

Without a messaging system we can think of a solution like the one described below:

- The HR System pushes data to AD via a scheduled process which runs every day
- The Marketing System pulls data from HR System via a scheduled process which runs every hour

- **There is a delay when using a scheduled mechanism**
- Using a messaging system we can handle this kind of situation in an almost real-time fashion

Using Kafka the solution would be the one described below:

- As the HR System receives data, it sends that to Kafka in real-time
- The Marketing System and the AD System read the data from Kafka and process it according to their requirements

Sending and receiving data happens in near real-time fashion (**ms**).

When multiple applications (M source and N target applications) need to exchange data, without using a messaging system:

- Many **integration points** (M*N)
- Different implementations with different **data exchange protocols** (database link, API, file) and different data formats (binary, csv, json, xml)
- Source and Target applications need to maintain reliable connection to avoid data loss

Solution using Messaging Systems:

- We can decouple the applications
- The Source System sends data to the Messaging System
- The Target System takes data from the Messaging System
- **We can define a standard to send and receive data** (json)
- Messaging Systems are designed in such a manner that they will have **high throughput for data communication**

Kafka is not only a Messaging System but also a **Streaming Platform.**

Kafka

Apache Kafka is a **distributed** **Streaming Platform**.

A Streaming Platform has **three key capabilities**:

- Publish and subscribe to streams of records
- Store streams of records in a **fault-tolerant durable way**
- Process streams of records **as they occur**

Publisher = Producer
Subscriber = Consumer = Listener

- Kafka is more than a Messaging System
- **Each record sent to Kafka is stored on disk**
- Kafka has great support for making the **records highly available**
- We can create multiple Kafka nodes (cluster) and replicate the records across those nodes.
- When there is a network failure and a node is unavailable, the available nodes will take over and the message will be stored.
- **Publishers and Consumers can still work as usual without knowing that a specific node is down**. The available nodes will serve publishers and consumers.
- The records can stay in Kafka as long as we need and we only need to make sure that the disk storage is enough. This is the reason why we say that records are stored in a durable way
- **Kafka uses topics for storing messages**
- Each message that comes to a Kafka topic is stored in the order it arrives
- If a publisher publishes the messages M1, M2, M3 the subscriber will process the messages in the same order
- As a **Streaming platform**, Kafka has **very high performance** and can work with high volume of data, **when configured properly**

Use cases:

- **Storing application logs**
- **Storing application data like coordinates from GPS trackers every 10s (IoT data)**
- **Decoupling system dependencies and process records asynchronously**
- **Streaming messages in the big data ecosystem because of its high throughput** (**withdrawal and deposit transactions can be sent to Kafka to determine if a transaction is a fraud or not**)

Kafka Distributions

- Apache Kafka
- Confluent (www.confluent.io)

Confluent has additional tools for monitoring, managing Kafka clusters.

Configuration

Under the Kafka home directory create a new directory named “data” and under “data” create the two subdirectories:

- zookeeper
- kafka

These directories will contain Zookeeper and Kafka data.

In Kafka home directory open the config directory and edit the dataDir value in zookeeper.properties file.
From Kafka in directory run the following command:

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

The above command will start Zookeeper **on port 2181**

In Kafka home directory open the config directory and edit the **log.dirs** value in server.properties file.

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

The above command will start the broker on port **9092**

Kafka Basic Concepts

Kafka can be thought of as a “**warehouse**” for messages

The “producer” application, also known as “publisher”, sends the messages to the warehouse instead of sending them directly to the “receiver” or “consumer” application.
The message is stored “somewhere” in the warehouse based on the publisher instructions.
The receiver subscribes to specific storage “areas”. Whenever a message comes to these areas the consumer will receive the message and process it.

Storage areas are called “**topics**”.

- In Kafka each topic can have one or more partitions
- Each partition will store the incoming messages in the same order as they arrive
- **We can let Kafka handle partition assignment for us or we can instruct Kafka on which partition to use to save the message**
- **Kafka maintains an index, called offset, which is a pointer to the last record sent to a consumer** in the most recent poll
- **The offset is maintained for each partition**
- You can have as many topics as you need as long as the storage capacity is enough.
- Each message will be stored on the hard drive up to a certain period of time, known as the **retention period**
- You can set the value of the retention period as you need
- Each topic is identified by the name which was given on creation
- **Each message that comes to Kafka is immutable**, i.e. **once is written it cannot be changed**

Partition and Offset

- Each topic can have one or more partitions for accepting and storing messages
- **It would be nice to process the messages in parallel using multiple consumers**
- **Partitions is the way to achieve parallelism**

- **Each partition in Kafka can have at most one consumer**
- For example if we have one topic with two partitions we can have at most two consumers running in parallel
- Messages are stored as they come
- The order is guaranteed at partition level
- **The order is not guaranteed across partitions**
- **Messages will be assigned to a random partition by Kafka unless we provide the key**
- **Each message sent to a partition gets an incremental id, i.e. an offset, starting from 0**
- **The offset is per partition**
- We must define the number of partitions when we create the topic
- We can add partitions later, after the topic has been created
- We cannot delete an existing partition
- Deleting a partition means deleting the data stored in the partition (data loss)

Kafka Producer

- The Producer is the application sending messages to Kafka
- As developers we should specify which topic to write the message to and the message
- Kafka will automatically select the partition, using a Round Robin approach, but we can override this behavior
- We can directly specify the partition we want the message to be sent to
- We can specify a **key** in the record and **Kafka will hash the key and select the topic based on the hash value**. **So all the messages with the same key will be sent to the same partition if no new partitions have been added meantime**

Consumer and Consumer Group

- The Consumer is the application reading messages from Kafka
- The Consumer application reads messages from topic partitions in the order they come
- **Each partition can have at most one consumer for each consumer group**
- **One Kafka Consumer can read from multiple partitions**

Single Consumer for all the partitions

Let’s suppose we have one topic with three partitions
We have a single consumer and it will process the messages from all the partitions

Consumers number < Partitions number

- Let’s suppose we have a topic with three partitions
- We have two consumers to speed-up the process
- One consumer will read from one partition and the other consumer will read from the other two partitions
- Which consumer will read from one partition, which consumer will read from two partitions, which consumer will read from which partition depends on Kafka configuration

Consumers number = Partitions number

- Let’s suppose we have a topic with three partitions
- We have three consumers to speed-up the process
- **There will be one-to-one mapping between consumers and partitions**
- Which consumer will read from which partition depends on the Kafka configuration

Consumers number > Partition number

- Let’s suppose we have a topic with three partitions
- **We have four consumers but one of them will be idle**
- Kafka allows at most one consumer per partition
- **The idle consumer will come in to play if one of the other consumers go down**

Kafka Consumer Group

- **Consumers can be grouped based on functionalities**
- We can use partitions and consumer groups to consume messages in parallel
- **If we have a topic with one partition we can have two consumers belonging to different consumer groups consuming in parallel from the partition (each message is processed twice)**
- **Consumer groups are independent (the process of each consumer group will not interfere with each other)**
- **If we have a topic with three partitions and two functionalities (two consumer groups) each consumer will consume from three partitions**
- **If one of the two functionalities is slow** we can add two consumers to the consumer group associated with that functionality and so there will be one consumer consuming from each partition in one of the two consumer groups

Consumer Offset

- **It is an index representing the message position in the partition**
- **Kafka saves the consumer offset**
- **It is a checkpoint that indicates last read**
- If the consumer goes down and then it goes up again it will remember the last message read and continue from there

Delivery Semantics

- **The consumer chooses when to save (commit) the offset**
- **"At most once" delivery semantics** happens when the offset is committed immediately after the read
- If something goes wrong during processing the message will not be re-processed
- **"At least once"** delivery semantics happens when the offset is committed after the message is processed
- **If something goes wrong during processing the offset will not be committed so the message can be re-processed**
- **We must design an idempotent consumer so that reprocessing will not have impact on the system for example creating duplicated transactions**
- **"Exactly once" delivery semantics is hard to implement**

Zookeeper

- **Kafka is designed for HA**
- Kafka cluster: group of Kafka brokers
- **Zookeeper is responsible to manage brokers within the cluster**
- Zookeeper is responsible for **balancing the cluster** when a new broker joins the cluster or a broker goes down
- Zookeeper is responsible **for synchronizing the data among brokers**
- Zookeeper server must be up and running for Kafka to start

Kafka Commands

From Kafka bin directory we can execute the following commands to create a topic, list topics, describe a topic and delete a topic:

- **./kafka-topics.sh** --**bootstrap-server** localhost:9092 --**create** --**topic** my_topic_name --**partitions** 1 --**replication-factor** 1
- **./kafka-topics.sh** --**bootstrap-server** localhost:9092 --**list**
- **./kafka-topics.sh** --**bootstrap-server** localhost:9092 --**describe** --**topic** my_topic_name
- **./kafka-topics.sh** --**bootstrap-server** localhost:9092 --**delete** --**topic** my_topic_name

We can create a Springboot with Kafka project on https://start.spring.io/ adding the **spring-kafka** dependency.
This dependency provides a KafkaTemplate<K,V> to send messages and a @KafkaListener annotation to consume messages.

For Spring Boot >= 2.2 we need to add the following configuration in application.properties:

```
spring.kafka.consumer.auto-commit=true
```

Consumer Offset

- auto.offset.reset=latest (default value)
- auto.offset.reset=earliest

Using "latest" the consumer will only consume the messages that are sent after the consumer started. Using "earliest" the consumer will consume all the messages, from beginning.

In application.properties set the consumer offset like below:

```
spring.kafka.consumer.auto-offset-reset=earliest
```

Producing message with Key

The same key goes always to the same partition as long as we do not change the partitions number.

Let's create a topic with three partitions by running the following command:

./kafka-topics.sh --**bootstrap-server** localhost:9092 --**create** --**topic** my_multi_partitions_topic_name --**partitions** 3 --**replication-factor** 1

Let's run the describe command to get details about the newly created topic:

./kafka-topics.sh --**bootstrap-server** localhost:9092 --**describe** --**topic** my_multi_partitions_topic_name

Let's create three console consumers each consuming from a different partition of the topic from the beginning:

- **./kafka-console-consumer.sh** --**bootstrap-server** localhost:9092 --**topic** my_multi_partitions_topic_name --**offset** earliest --**partition** 0
- **./kafka-console-consumer.sh** --**bootstrap-server** localhost:9092 --**topic** my_multi_partitions_topic_name --**offset** earliest --**partition** 1
- **./kafka-console-consumer.sh** --**bootstrap-server** localhost:9092 --**topic** my_multi_partitions_topic_name --**offset** earliest --**partition** 2

Multiple consumers for each topic

When we publish data faster than we consume **then the consumer process is a bottleneck**. One solution is to use **multiple concurrent consumers per topic**. **The number of consumers per topic depends on the number of partitions of the topic**. The ideal situation is to have a number of consumers equals to the number of topic partitions. This will allow to parallelize and speed-up message processing. When the number of consumers is greater than the number of partitions then some consumers will be idle which is not very efficient. **Adding consumers to process messages in a topic is easy as Spring will manage consumers and we do not need any multi-thread programming**. **By default the annotation @KafkaListener configures one single consumer**. **We can verify this inspecting the logs**. **The logs show in fact that the container has created a single consumer and there is only one thread**. If we want to increase the number of consumers we can use the **concurrency parameter** in the annotation. For example with concurrency = 2 the container will spin-up two consumers and there will be two separate threads. In this case if 3 is the number of topic partitions, one consumer will consume from two partitions and the other will consume from a single partition. With concurrency = 3 each consumer will consume from a single partition. If we use concurrency = 4 then one consumer will be idle.
We can add one partition to the topic so that the number of consumers will be equal to the number of the topic partitions:
./kafka-topics.sh --**bootstrap-server** localhost:9092 --**alter** --**topic** my_multi_partitions_topic_name --**partitions** 4

This way we can decrease processing time.

Producing JSON message

Kafka can accept any string. This flexibility comes with its disadvantages. Indeed if two producers produce each with their own proprietary format then the consumers need to bee aware of the two formats. JSON is the standard for data interchange. Every programming language has a library to create and parse a JSON string. The consumer can focus on the business logic rather than focusing on the parsing logic. In Java we usually use Jackson or GSON to work with JSON. In Spring we use Jackson (jackson-core, jackson-databind,, jackson-annotations).

We will see how to:
- consume a JSON message
- consume a message with different Consumer Groups

When the same data needs to be used to implement different functionalities we can consume the data using different consumer groups. For example we may have two distinct functionalities like:

- consume data to update dashboard
- consume data to send a notification

In this case two consumer groups will read the same message to implement two different functionalities.
It is possible to get a description of a consumer group from the command line. The LAG is one of the outputs of this command. The LAG is the delta between the offset of the last produced message and the last consumer committed's message.

Rebalancing

Given I modify a topic adding one partition while Kafka producer and consumer applications are running Kafka automatically rebalances after a while. The **default time required for rebalancing** is 5 minutes for consumer and producer (300000ms). When you add a partition to a topic initially with a single partition, after a while the producer will start producing on two different partitions and consumers in a consumer group will reorganize accordingly. This can be inspected in logs.

Kafka Configuration

It is possible to override the default rebalancing behavior by adding a configuration class.

Kafka Configuration Reference

https://kafka.apache.org/documentation/#producerconfigs
https://kafka.apache.org/documentation/#consumerconfigs

metadata.max.age.ms = The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.

Message Filter

Spring Kafka provides the functionality to filter incoming messages on a topic. **If a message matches the filter criteria specified by the consumer it will be processed**, **otherwise it will be discarded**. The message will remain on the Kafka topic. Filtering can be applied at the Listener level. We can have two listeners for the same topic. One of them listens to all messages whereas the other listens to messages that match a specified criteria.

Kafka Listener Error handler

Exceptions can occur on the producer side or on the consumer side. **Handling exception on the producer side is straightforward**. **We use a try-catch block and if any error occurs we do not publish to Kafka**. We simply handle the exception in the catch block. Handling consumer exceptions is slightly more complex. **Spring default behavior** is to **log the exception and continue with the next message**. We can implement our own handler to have our own exception handling logic. For example we can send our logs to Elastic search. Elastic search is a search engine product that can be used for many things including log aggregation **and visualization**.

Let's consider the following use case:

- An item order is published to the topic
- when the item quantity is LE 0 an exception is thrown on the consumer side

We can implement our own error handler by creating a Spring bean implementing the **ConsumerAwareListenerErrorHandler** interface. Then we can use this error handler for our consumer method specifying it in the @KafkaListener annotation errorHandler.

Global Error Handler

We can have an error handler which applies to all Kafka consumers. In case we need to send logs to Elastic search for the failed records it would be better to handle it in a global error handler.

Let's consider the following use case:

- A random number is published to the topic
- when the number is LT < 0 an exception is thrown on the consumer side

The global error handler implements the **ConsumerAwareListenerErrorHandler** interface. We need to register the global error handler in the "**kafkaListenerContainerFactory**" bean. To this end in the configuration class we need to define such bean and set its handler field to the global error handler. The bean also needs to read the existing Kafka configuration .
In case a consumer has a Kafka Listener error handler we will not see the global exception handler coming into play for it **unless we re-throw the exception in the handler**.

Retrying Consumer

Why Retry Mechanism?

In case of errors we can apply different approaches:

- Use Spring Kafka default behavior , i.e. log the failed messages
- Build a custom Error handler, i.e. not just logging but sending a notification
- Add a re-try mechanism

A re-try mechanism is a good solution in case there is a service which needs to be invoked as part of the processing and this service may be not available. In this case it makes sense to have a re-try logic where we re-try to invoke the service **n** times every **m** seconds. If after n times the exception still happens the error handler will come into play.

How to implement a re-try mechanism?
We need to define a new container factory bean specific for the re-try. For this container factory we need to set the re-try template.

As part of the implementation we need to create an object of type **RetryTemplate**. For this object we need to set the re-try and back-off policies. The re-try policy will be of type SimpleRetryPolicy and there we specify the number of retries. The back-off policy will be of type FixedBackoffPolicy and there we specify the interval between re-tries. The new container factory will be used in the @KafkaListener annotation.

Spring Retry enables us to automatically re-invoke a failed operation. **This is useful where the errors may be temporary, like when a service is momentary down due to network problem**.
Spring Kafka provides integration with Spring Retry for re-trying a failed consumer operation. You can look at the documentation for the RetryPolicy and BackoffPolicy implementations.

Dead Letter Topic

If message processing keeps failing it might be due to:

- A non-technical issue (wrong data)
- A permanent technical issue (service endpoint has changed)

Spring provides the capability to send failed messages to another topic (dead letter topic) in case the processing of the message keeps failing. A dead letter topic is a combination of a re-try mechanism with a producer publishing to another topic.
The message that goes to the dead letter topic is called **dead letter record**.
To handle dead letter publishing Spring provides class **DeadLetterPublishingRecoverer**. By default the dead letter record will be sent to a topic named:

<original_topic_name>.DLT.

The record will be sent to the same partition number. As a consequence dead letter topics must have at least the same partition count as the original topic.
Sometimes the default behavior is not what we want. For example it is not a good naming strategy mixing dot with underscore in the topic names.

Let’s consider the following use case for a dead letter topic:

- We publish some data
- if a specific condition applies to the data we throw and exception on the consumer side
- we re-try processing the record for 4 times
- after 4 failed re-try attempts the message is finally published to a dead letter topic
- another consumer listens to failed records sent to the dead letter topic for processing them

For example the consumer could send a notification containing information about the failed record.

RabbitMQ versus Kafka

Both RabbitMQ and Kafka are excellent open-source messaging system solutions.
Let’s see some of the high level differences between the two of them.

Message Retention:

In Kafka the message is stored on disk for a configurable amount of time, i.e. according to a **retention policy**. This means that messages can be re-processed by the same consumer or by other consumers. In RabbitMQ the message will be deleted upon consumer acknowledgment of successful processing. **So in RabbitMQ the message will be deleted when a single consumer consumes the message successfully**. So if we wanted to re-process the message the publisher should re-publish it again.

Message Routing:

Kafka has no routing mechanism for messages. **RabbitMQ has a routing mechanism based on exchanges** which route the message into one or more queues. **In Kafka it is the publisher to tell which topic and optionally partition the message should go**. RabbitMQ can route the message based on a routing key. In RabbitMQ there are different types of exchanges: Fanout, Topic, Direct. Each exchange type has different functionalities. The publisher can attach a routing key and RabbitMQ will route the message to one or more queues.

Multiple Consumers:

Kafka achieves multiple consumers by using partitioning. Each topic in Kafka can have multiple partitions. **Each partition has a single consumer per consumer group**. So message ordering in Kafka is guaranteed at partition level. **In RabbitMQ one queue can have multiple consumers. So message ordering is not guaranteed**. This may be useful if we have fast producers and slow consumers. However there is a drawback in RabbitMQ approach. Multiple consumers might cause the “competing consumers” pattern where messages in the queue might not be processed in order.

Consumer Push/Pull Model:

Kafka consumers pull the messages from Kafka topic partitions. So consumers are the active part in Kafka. The pull model guarantees message ordering as one partition can be accessed by only one consumer. RabbitMQ will push the message into each registered consumer. To avoid flooding the consumer, in case publisher works on faster rate than consumer can handle, consumer can set the pre-fetch limit which is basically the number of unprocessed messages that can be accepted from the consumer. This push model ensures that message workload is distributed fairly even, among consumers.

Whether to use Kafka or RabbitMQ depends on the use case:

- In some companies like e-commerce ones where it is necessary to handle fast transactions Kafka is the messaging system of choice
- Some other companies with lower **transaction rate** use RabbitMQ
- RabbitMQ is somewhat easier to manage than Kafka. Kafka has several things to be managed like **cluster replication**, producer and consumer config
- **Kafka is good for scalability and it is usually a choice on big data streaming**. The transactions from ATM, credit cards, and other channels **can be streamed to Kafka** and analyzed for fraud detection
- Developer-wise both Kafka and RabbitMQ provide support for commonly used programming languages. In Java we can use **Spring boot which provides ready-to-use functionalities to talk with Kafka or RabbitMQ.** But also other libraries are supported like **Python**, Go or C#
- The learning curve of RabbitMQ is somewhat easier
- It is advisable to work with a single messaging system within an organization to avoid to double the effort required to work and maintain a messaging system

Kafka in Microservice Architecture & Pattern

Example application