**Limitation of Java Message Services**

1. JMS are language dependent and both (producer and consumer) must be java application.
2. They support of TCP protocol but not other protocols
3. If message is very big/large in size then MOM behaves very slow.
4. In case of multiple producers and consumers, it won’t support scaling.
5. There might be a possibility of data loss, if MOM is down or if it’s not responding. Because of single instance, if producer is sending messages and MOM id down then consumer wont receive a message.

**Apache Kafka usage**

1. However, Kafka is implemented in the Java Language, but it supports integration with different technologies and concepts like Spark, Scala, Hadoop, BigData, etc.
2. Because of it’s cluster design, Kafka supports transfer of data between multiple complex systems.
3. Kafka supports integration with non-java technologies even via REST calls.
4. it is protocol independent as we can write code using TCP, FTP, HTTP, etc.
5. Kafka supports multiple message brokers. It means Horizontal scaling of the broker software is possible here.
6. Kafka takes the support of Zookeeper to handle load balancing.

**Apache Kafka**

* Open source software platform developed by Apache Foundation
* Written in java and scala
* Offers unified, high-throughput, low latency platform for handling real time data feeds.
* *It’s a distributed data streaming platform that can publish, subscribe to, store and process streams of records in real time.*
* *Design to handle* ***data streams from multiple sources and deliver them to multiple consumers.***

**Kafka Ecosystem**

Kafka system is also known as **Kafka Cluster** as It can consist of multiple elements / nodes/servers. And this is the reason Kafka is categorized as a distributed system. However, The four major components of Kafka are:

1. **Producer**: A Kafka producer acts as a data source that writes, optimizes, and publishes messages to one or more Kafka topics. Kafka producers also serialize, compress, and load balance data among brokers through partitioning.
2. **Topic**: It’s a channel through which a data is streamed. Furthermore, producer publish messages to topics and consumer reads messages from topic they are subscribed to.
3. **Brokers**: Brokers are the software components that run on a node.Many people in the industry define a Kafka broker as a server running in a Kafka cluster. In other words, a Kafka cluster consists of a number of brokers. Typically, multiple brokers form the Kafka cluster and achieve load balancing and reliable redundancy and failover. Brokers use Apache ZooKeeper for the management and coordination of the cluster. Each broker instance is capable of handling read and write quantities. Each broker has a unique ID and is responsible for partitions of one or more topic logs.
4. Consumer: Kafka Consumers read messages from the topics to which they subscribe. Consumers will belong to a consumer group. Each consumer within a particular consumer group will have responsibility for reading a subset of the partitions of each topic that it is subscribed to.

Data in the Kafka cluster is distributed amongst several brokers. There are several copies of the same data in the Kafka cluster. They are called **replicas**. This mechanism makes Kafka even more reliable, fault-tolerant, and stable. If an error occurs with one broker, the  another broker will start to perform the functions of the broken component. Hence, there are no chances of any information loss.

**What is Zookeeper and its role?**

* **ZooKeeper** is also an open source tool provided by the Apache Software Foundation.
* It provides a centralized service in distributed systems such as providing configuration information, synchronization, naming registry, and other group services over large clusters.
* **Kafka** uses **Zookeeper** in order to track the status of nodes in the Kafka cluster

**Role of Zookeeper**

While working with any distributed system, there should be a way to coordinate tasks. In our context, Kafka is a distributed system that uses ZooKeeper to co-ordinate its tasks. However, there are some other technologies like Elasticsearch and MongoDB who have their own built-in mechanisms for coordinating tasks.

1) When working with Apache **Kafka**, the primary role of **ZooKeeper** is to track the status of nodes in the **Kafka** cluster and also maintain a list of **Kafka** topics and messages.

2) In fact, **ZooKeeper** coordinates the brokers/cluster topology.

3)**ZooKeeper** acts as a consistent file system for configuration information. Moreover, it contains a list of all **Kafka** brokers with it. It notifies **Kafka**, if any broker goes down, or partition goes down or new broker is up or partition is up.

4) ZooKeeper also accesses how much data each client is allowed to read/write.

5) In addition, Kafka uses Zookeeper to store offsets of messages consumed for a specific topic and partition by a specific Consumer Group.

***Messages contained in the partitions are assigned a unique ID number that is called the offset. The role of the offset is to uniquely identify every message within the partition.***

**Spring Boot with Apache Kafka**

**Kafka Cluster**

Since kafka is distributed system, it acts as a cluster.

Kafka cluster consist of set of brokers. It contains minimum 3 brokers.

**Kafka Broker**

Broker is the kafka server. Its an alternate name given to server.Kafka asts as message broker between producer and consumer.

Producer and consumer dont interact directly. They use kafka server as agent or borker to exchange a message.

**Kafka Topic**

its like table in database or folder system present inside broker.

Topic is a mechanism which identifies which message a consumer wants to consume from the broker.

Topic is identified by a name. Broker can have multiple topic.

**Kafka partition**

Kafka topics are divided into number of partitions, which contains records in an unchangeable sequence.

Kafka brokers will store message for a topic. But the capacity of data can be enormous and it may not possible to store into a single computer. Therefore it will partitioned into multiple parts and distributed among multiple computers, since Kafka is distributed system.

**Offset**

Offset is sequence of ids given to message as the arrive at partition. Once the offset is assigned it will never be changed. The first message gets an offset zero. The next message receives an offset one and so on.

**Installing Apache Kafka and running based on some commands**

**STEP 1:** download and install kafka

[**https://www.apache.org/dyn/closer.cgi?path=/kafka/2.6.0/kafka\_2.12-2.6.0.tgz**](https://www.apache.org/dyn/closer.cgi?path=/kafka/2.6.0/kafka_2.12-2.6.0.tgz)

**STEP 2:** START THE KAFKA ENVIRONMENT

# **Start the ZooKeeper service** (open new terminal)

.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

# **Start the Kafka broker service** (open new terminal)

.\bin\windows\kafka-server-start.bat .\config\server.properties

**STEP 3:** CREATE A TOPIC TO STORE YOUR EVENTS (open new terminal)

.\bin\windows\kafka-topics.bat --create --topic topic\_demo --bootstrap-server localhost:9092

**STEP 4:** WRITE SOME EVENTS INTO THE TOPIC (in same or previous terminal)

.\bin\windows\kafka-console-producer.bat --topic topic\_demo --bootstrap-server localhost:9092

**STEP 5:** READ THE EVENTS

.\bin\windows\kafka-console-consumer.bat --topic topic\_demo --from-beginning --bootstrap-server localhost:9092 hello world topic demo

**Spring boot Application with Kafka**

1. **Configure Kafka Producer and Consumer in an application.properties File**

In the *application.properties* file, add Kafka broker address as well as Consumer and Producer related configuration.

Open the *application.properties* file and the following content to it:

spring.kafka.consumer.bootstrap-servers: localhost:9092

spring.kafka.consumer.group-id: group-id

spring.kafka.consumer.auto-offset-reset: earliest

spring.kafka.consumer.key-deserializer: org.apache.kafka.common.serialization.StringDeserializer

spring.kafka.consumer.value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

spring.kafka.producer.bootstrap-servers: localhost:9092

spring.kafka.producer.key-serializer: org.apache.kafka.common.serialization.StringSerializer

spring.kafka.producer.value-serializer: org.apache.kafka.common.serialization.StringSerializer

Let's understand the above spring boot provided Kafka properties:

*spring.kafka.consumer.group-id* - specifies a unique string that identifies the consumer group this consumer belongs to.

*spring.kafka.consumer.auto-offset-reset property* - specifies what to do when there is no initial offset in Kafka or if the current offset does not exist anymore on the server (e.g. because that data has been deleted):

* earliest: automatically reset the offset to the earliest offset
* latest: automatically reset the offset to the latest offset
* none: throw an exception to the consumer if no previous offset is found for the consumer’s group
* anything else: throw an exception to the consumer.

*spring.kafka.consumer.key-deserializer* - specifies the deserializer class for keys.

*spring.kafka.consumer.value-deserializer* - specifies the deserializer class for values.

*spring.kafka.producer.key-deserializer* - specifies the serializer class for keys.

*spring.kafka.producer.value-deserializer* - specifies the serializer class for values.

**2. Create Kafka Topic**

To create a topic on startup, add a bean of type *NewTopic*. If the topic already exists, the bean is ignored. We will use the topic name "javaguides" in this example.

Let's create a *KafkaTopicConfig* file and add the following content:

package net.javaguides.springbootkafka;

import org.apache.kafka.clients.admin.NewTopic;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.kafka.config.TopicBuilder;

@Configuration

public class KafkaTopicConfig {

@Bean

public NewTopic javaguidesTopic(){

return TopicBuilder.name("myTopic-events ")

.build();

}

}

**5. Create Kafka Producer**

Creating a producer will write our messages on the topic.

**KafkaTemplate**

Well, Spring boot provides an auto-configuration for Spring’s *KafkaTemplate* so you can autowire it directly in your own beans.

For example:

**package** com.nt;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.kafka.core.KafkaTemplate;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** MyKafkaProducer {

@Autowired

**private** KafkaTemplate<String, String> kafkaTemplate;

**public** **void** sendMessage(String message) {

kafkaTemplate.send("myTopic-events", message);

}

}

Create a *utils* package and within this package create *AppConstants* with the following content:

package net.javaguides.springbootkafka.utils;

public class AppConstants {

public static final String TOPIC\_NAME = "javaguides";

public static final String GROUP\_ID = "group\_id";

}

*KafKaProducer* class uses *KafkaTemplate* to send messages to the configured topic name.

**Create Kafka consumer**

* Kafka Consumer is  the service that will be responsible for reading messages and processing them according to the needs of your own business logic.
* Here, we told our method **consume (String message)** to subscribe to the user’s topic and just emit every message to the application log. In your real application, you can handle messages the way your business requires you to.

**package** com.nt;

**import** org.springframework.kafka.annotation.KafkaListener;

**import** org.springframework.stereotype.Component;

@Component

**public** **class** MyKafkaConsumer {

@KafkaListener(topics = "myTopic-events", groupId = "group-id")

**public** **void** consume(String message) {

System.***out***.println("Message received -> %s" + message);

}

}

**Create REST API to Send Message**

Create controller package, within controller package create *KafkaProducerController* with the following content to it:

**package** com.nt;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.http.ResponseEntity;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.RequestParam;

**import** org.springframework.web.bind.annotation.RestController;

@RestController

**public** **class** MyController {

@Autowired

**private** MyKafkaProducer kafkaProducer;

@GetMapping("/publish")

**public** ResponseEntity<String> publish(@RequestParam("message") String message) {

kafkaProducer.sendMessage(message);

**return** ResponseEntity.*ok*("Message sent to kafka topic");

}

}

**See Topic Messages via Command Line:**

**Go to location and run below command**

kafka-console-consumer.bat --topic **<topic name>**--from-beginning --bootstrap-server localhost:9092Location: D:\Ankur\_Wadatkar\Softwares\Kafka\bin\windows

Eg:

kafka-console-consumer.bat --topic dhanu\_topic --from-beginning --bootstrap-server localhost:9092

Make sure to change the topic name. In our case "javaguides" is the topic name.

**Demo**

Let's run the Spring boot application and have the demo. Make sure that Zookeeper and Kafka services should be up and running.

Open a browser and hit the below link to call a REST API:

[**http://localhost:8080/api/v1/kafka/publish?message=hello%20world**](http://localhost:8080/api/v1/kafka/publish?message=hello%20world)