

Now-a-days, many modern systems require data to be processed for the targeted purpose as soon as it becomes available. For example, as a developer, if we talk about a Logging or Monitoring system, we often require the data as soon as a problem occurs. In a nutshell, there is a high requirement of faster & robust data delivery. Here, Apache Kafka can be a good option to solve our purpose. In a single line, Apache Kafka acts as a mediator to transmit data between applications that generate data and applications that consume data. Needless to say, we are going to discuss about ‘How to work with Apache Kafka in Spring Boot?’ here in this topic.

If you want to know the steps how to make Kafka in Action in your Spring Boot Project, You need to stay here in this blog for a couple of minutes. You will get step by step information in this article itself. Let’s start discussing our topic ‘How to work with Apache Kafka in Spring Boot?’ and the related concepts.

**What are the limitations of using JMS (Java Message Service)?**

1) It is a Java language dependent. Thus, both the participants: Producer and Consumer must be Java Applications only.

2) It works on Protocol (TCP). It doesn’t support other protocol.

3) If the message is very big /large in size, then MOM (only one Message Broker Software) behaves very slow.

4) In case of multiple Producer and Consumers, it doesn’t support scaling. Hence, we can’t create multiple MOM instances.

5) Moreover, there is a possibility of data loss, if MOM is down or MOM is not responding. Because of a single instance, if the producer sent message to MOM, and MOM is down. Then, the consumer will not receive messages.

♦ \* JMS is the best for the smaller scale applications such as in case of less number of producers and less number of consumers.

**What is Apache Kafka?**

Apache Kafka is an open-source software platform developed by Apache Software Foundation written in Java and Scala. The project aims to offer a unified, high-throughput, low-latency platform for handling real-time data feeds. Kafka can connect to external systems (for data import/export) via Kafka Connect and provides Kafka Streams, a Java stream processing library.

Apache Kafka is a distributed data streaming platform that can publish, subscribe to, store, and process streams of records in real time. It is designed to handle data streams from multiple sources and deliver them to multiple consumers. In short, it moves massive amounts of data, not just from point A to B, but from points A to Z and anywhere else you need, all at the same time.

[Apache Kafka](https://kafka.apache.org/intro) is an alternative to a traditional enterprise messaging system. It started out as an internal system developed by Linkedin to handle 1.4 trillion messages per day, but now it’s an open source data streaming solution for a variety of enterprise needs.

**Kafka APIs**

Kafka has the following five core APIs for Java and Scala:

1) The [Admin API](https://kafka.apache.org/documentation.html#adminapi) to manage and inspect topics, brokers, and other Kafka objects.

2) The [Producer API](https://kafka.apache.org/documentation.html#producerapi) to publish (write) a stream of events to one or more Kafka topics.

3) The [Consumer API](https://kafka.apache.org/documentation.html#consumerapi) to subscribe to (read) one or more topics and to process the stream of events produced to them.

4) The [Kafka Streams API](https://kafka.apache.org/documentation/streams) to implement stream processing applications and microservices. It provides higher-level functions to process event streams, including transformations, stateful operations like aggregations and joins, windowing, processing based on event-time, and more. Input is read from one or more topics in order to generate output to one or more topics, effectively transforming the input streams to output streams.

5) The [Kafka Connect API](https://kafka.apache.org/documentation.html#connect) to build and run reusable data import/export connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka. For example, a connector to a relational database like PostgreSQL might capture every change to a set of tables. However, in practice, you typically don’t need to implement your own connectors because the Kafka community already provides hundreds of ready-to-use connectors.

**Why Kafka is used?**

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) Additionally, Kafka supports integration with non-java technologies even via REST calls.

4) Moreover, it is protocol independent as we can write code using TCP, FTP, HTTP, etc.

5) Equally important, 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 (like [Netflix Eureka in Microservices](https://javatechonline.com/how-to-register-discover-microservices-using-netflix-eureka/)).

**What are the benefits of using Kafka over other techniques?**

1) Kafka offers fast message delivery. Moreover, a single Kafka broker can serve thousands of clients by handling megabytes of reads and writes per seconds.

2) Messages in Kafka are persistent as they are replicated within the cluster in order to prevent any data loss.

3) In Kafka, there is a provision for data partitioning and streamlining over a cluster of machines to handle larger data.

4) Additionally, Kafka offers fault tolerance mechanism and durability.

5) Moreover, Kafka can process a very large size message. The maximum size of the message that Kafka server can receive is 1000000 bytes.

**What are the various components in Kafka?**

Sometimes, the whole 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:

**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.

**Topic**

A Kafka topic represents a channel through which data is streamed. Furthermore, Producers publish messages to topics, and consumers read messages from the topic they subscribe to. Moreover, Topics organize and structure messages. Its like particular types of messages will be published to particular topics. They are defined by unique names within a Kafka cluster. However, there is no limit on the number of topics that can be created.

**Brokers**

In fact, **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.

**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?**

Like Kafka, **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.

**What is the role of Zookeeper in Kafka?**

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.

**How to install Kafka Software in your System?**

**Step#1 : How to download Kafka?**

Here, we have used Kafka 2.6.0 and executed our example successfully. Even, you can try the latest available and stable version.

1) Visit below link using Browser to download Kafka  
<https://www.apache.org/dyn/closer.cgi?path=/kafka/2.6.0/kafka_2.12-2.6.0.tgz>

2) Click on **HTTP** **LINK** (Looks like: https://mirrors.estointernet.in/apache/kafka/2.6.0/kafka\_2.12-2.6.0.tgz)

..it will be downloaded as .tgz format…

**Step#2 : How to Install Kafka?**

1) Extract to a folder (Used Software: 7zip)

> Right click on kafka\_2.12-2.6.0.tgz > 7Zip > Extract Here

…extracted to kafka\_2.12-2.6.0.tar

2) Again extract

..now we can see folder ‘kafka\_2.12-2.6.0’.

3) Copy this folder to a drive of your system like ‘C:/’ or ‘D:/’ drive.

That’s it for the installation part.

**Apache Kafka Spring Boot Example**

Since our final target is to learn ‘How to work with Apache Kafka in Spring Boot?’, it’s time to go through it step by step. We already have installed Kafka software as described in the previous section. Now, let’s go ahead and find the solution for ‘How to work with Apache Kafka in Spring Boot?’

**Step#1: Create a new Spring Boot Starter Project using STS**

While creating Starter Project select ‘Spring Web’, ‘Spring for Apache Kafka’, and ‘Sprong Boot DevTools’ as starter project dependencies. Even If you don’t know how to create a Spring Boot Starter Project, Kindly visit [Internal Link](https://javatechonline.com/saving-data-into-database-using-spring-boot-data-jpa-step-by-step-tutorial/#Step_1_Creating_Starter_Project_using_STS).

**Step#2: Apply @EnableKafka at your main class**

In order to get features of Apache Kafka with Spring Boot, we need to apply @EnableKafka at the main class as below.

**package com.dev.spring.kafka;**

**import org.springframework.boot.SpringApplication;**

**import org.springframework.boot.autoconfigure.SpringBootApplication;**

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

**@SpringBootApplication**

**@EnableKafka**

**public class SpringBoot2ApacheKafkaTestApplication {**

**public static void main(String[] args) {**

**SpringApplication.run(SpringBoot2ApacheKafkaTestApplication.class, args);**

**}**

**}**

**Step#3: Create a custom MessageRepository class**

Here, we will create a custom MessageRepository class. Furthermore, we will create a List and add each incoming message in that List. Moreover, we will create two methods; one to add a message, and another to retrieve all messages. For example , below code demonstrates the concept.

**package com.dev.spring.kafka.message.repository;**

**import java.util.ArrayList;**

**import java.util.List;**

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

**@Component**

**public class MessageRepository {**

**private List<String> list = new ArrayList<>();**

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

**list.add(message);**

**}**

**public String getAllMessages() {**

**return list.toString();**

**}**

**}**

**Step#4: Create a MessageProducer class**

In order to produce the messages and send to the Topic, we will create a MessageProducer class as below.

**package com.dev.spring.kafka.sender;**

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

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

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

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

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

**@Component**

**public class MessageProducer {**

**private Logger log =LoggerFactory.*getLogger*(MessageProducer.class);**

**@Autowired**

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

**@Value("${myapp.kafka.topic}")**

**private String topic;**

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

**log.info("MESSAGE SENT FROM PRODUCER END -> " + message);**

**kafkaTemplate.send(topic, message);**

**}**

**}**

**Step#5: Create a MessageConsumer class**

Now, this is the turn to create a MessageConsumer class which consumes the messages sent by the Producer via Topic.

**package com.dev.spring.kafka.consumer;**

**import org.slf4j.Logger;**

**import org.slf4j.LoggerFactory;**

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

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

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

**import com.dev.spring.kafka.message.repository.MessageRepository;**

**@Component**

**public class MessageConsumer {**

**private Logger log = LoggerFactory.*getLogger*(MessageConsumer.class);**

**@Autowired**

**private MessageRepository messageRepo;**

**@KafkaListener(topics = "${myapp.kafka.topic}", groupId = "xyz")**

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

**log.info("MESSAGE RECEIVED AT CONSUMER END -> " + message);**

**messageRepo.addMessage(message);**

**}**

**}**

**Step#6: Create a RestController as KafkaRestController class**

Now, we will create a RestController which will take messages from Web Browser as an input to the Producer. In other words, we will send messages to Kafka to process them via a Rest call.

**package com.dev.spring.kafka.controller;**

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

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

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

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

**import com.dev.spring.kafka.message.repository.MessageRepository;**

**import com.dev.spring.kafka.sender.MessageProducer;**

**@RestController**

**public class KafkaRestController {**

**@Autowired**

**private MessageProducer producer;**

**@Autowired**

**private MessageRepository messageRepo;**

**//Send message to kafka**

**@GetMapping("/send")**

**public String sendMsg(**

**@RequestParam("msg") String message) {**

**producer.sendMessage(message);**

**return "" +"'+message +'" + " sent successfully!";**

**}**

**//Read all messages**

**@GetMapping("/getAll")**

**public String getAllMessages() {**

**return messageRepo.getAllMessages() ;**

**}**

**}**

**Step#7: Create an application.yml file**

In the end, We will create an application.yml file as below. Please mind to remove the application.properties file from ‘src/main/resources’ folder and include this file. If you keep both the files, application.properties will be effective by default.

application.yml

server:

port: 9090

spring:

kafka:

producer:

bootstrap-servers: localhost:9092

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

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

consumer:

bootstrap-servers: localhost:9092

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

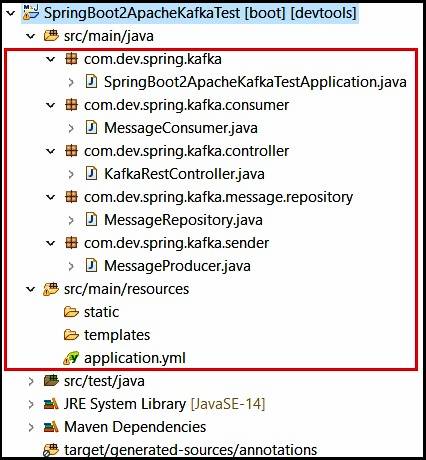
value-deserializer: org.apache.kafka.common.serialization.StringDeserializer

myapp:

kafka:

topic: myKafkaTest

Your project structure would look like something below screen.



**How to test the Application?**

In order to test the application, follow below steps.

1) Start Zookeeper

**cmd>cd C:\kafka\_2.12-2.6.0**

**cmd> .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties**

2) Start Kafka setup

**cmd> cd C:\kafka\_2.12-2.6.0**

**cmd> .\bin\windows\kafka-server-start.bat .\config\server.properties**

3) Create a Topic

**cmd>.\bin\windows\kafka-topics.bat --create --zookeeper localhost:2181 --replication-factor 1 --partitions 1 --topic myKafkaTest**

4) Run Spring Boot Application

5) After starting application Enter below URLs on the Browser and test the results

**♦ http:// localhost:9090/send?msg=I like**

**♥ http:// localhost:9090/send?msg=to work on**

**♦ http:// localhost:9090/send?msg=Kafka**

**♥ http:// localhost:9090/send?msg=with Spring Boot**

**http://localhost:9090/getAll**

6) Also check your output in the console. You will see output something like below screen.

