Introduction to Kafka

Prerequisites

- Prior Knowledge or Working Experience with Spring Boot/Framework
- Knowledge about building Kafka Clients using Producer and Consumer API
- Knowledge about building RESTFUL APIs using Spring Boot
- Experience working with Spring Data JPA
- Java 11 or Higher is needed
- Intellij , Eclipse or any other IDE is needed

What is real time data streaming ?

- Real-time data streaming is the process by which big volumes of data are processed quickly such that a firm extracting the info from that data can react to changing conditions in real time.

- It is beneficial for various scenarios, such as real-time data analytics, system log monitoring, sentiment analysis on social media, and handling data in IoT applications.

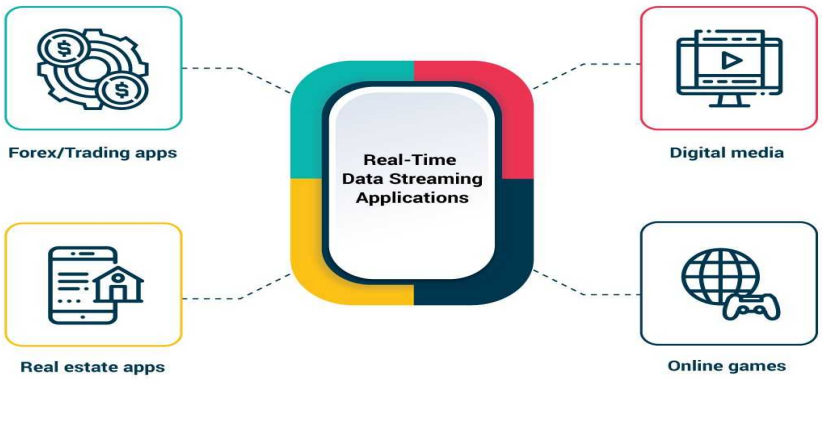


Figure 1

Why Kafka ?

Figure 2

- Kafka is a popular choice for handling real-time data streaming and processing in modern applications.
- Kafka is distributed and scalable, handling large data volumes across multiple servers or clusters.
- Kafka ensures high throughput and low latency, processing millions of messages per second.
- Kafka is fault-tolerant and highly available, with message replication ensuring data reliability and preventing loss during failures.
- Kafka provides message persistence through a distributed commit log, allowing for data replay and reprocessing.
- Kafka offers flexibility and integration with various systems and frameworks, providing client APIs for different programming languages and supporting connectors for seamless data integration.
- Kafka enables stream processing by integrating with different systems and frameworks through APIs and connectors.

- Kafka has a thriving ecosystem and active community, providing a wealth of resources and support.

- Kafka is trusted by industry leaders for handling high-volume data streams in critical applications.

What is a Messaging System?

A messaging system enables multiple applications or systems to communicate with one another by exchanging messages. Two types of patterns in messaging systems :

i) P2P (Point - to - Point) :

- P2P messaging sends messages from one sender to one receiver, which is useful for one-to-one communication. Messages are stored in a queue, which can only be eaten by one consumer at a time, and are removed as soon as they are read by consumers.

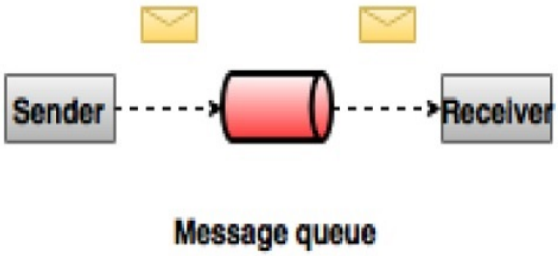


Figure 3

ii) Pub/Sub messaging :

- Pub/Sub messaging sends messages from one sender to multiple receivers, who receive a copy of the message when sent to a specific topic.

- Kafka consumers have the option to subscribe to one or more topics and read every message inside those topics.

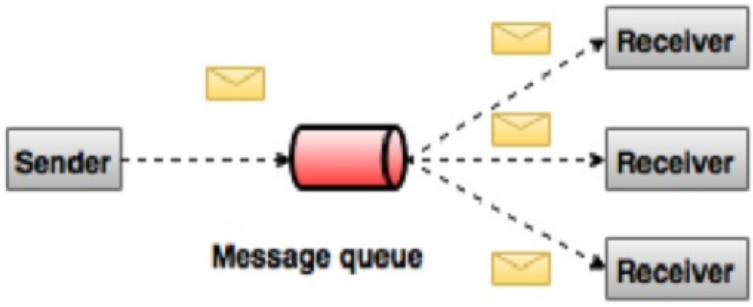


Figure 4

What is Kafka ?

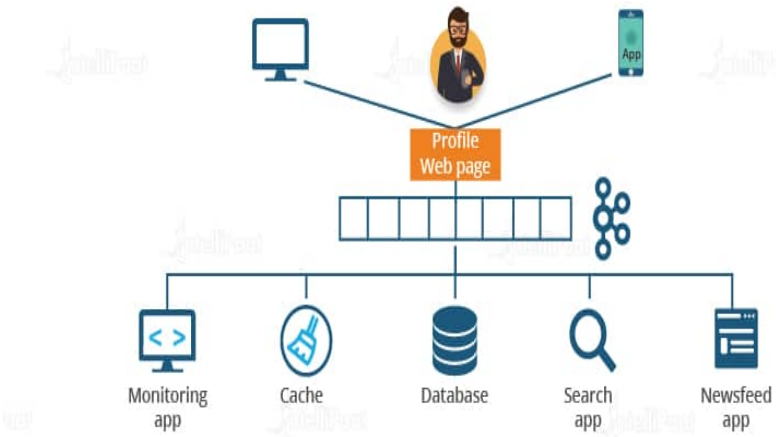


Figure 5

- Kafka is an open-source distributed event streaming platform.
  - It was initially developed by LinkedIn and is now maintained by the Apache Software Foundation.
  - Kafka is designed to handle high-volume, real-time data streams.

- It provides a scalable, fault-tolerant, and durable messaging system.
- Kafka uses a distributed commit log to store messages on disk, ensuring persistence and fault tolerance.
- It supports the publish-subscribe messaging model, allowing multiple consumers to receive messages from one or more producers.
- Kafka is highly scalable and can handle millions of messages per second.
- It has a rich ecosystem with various tools, libraries, and connectors for seamless integration with other systems.

Where Kafka is used?

- **Banking** : Financial institutions use Apache Kafka for real-time regulatory compliance, cybersecurity, and fraud detection, as well as stock market trading applications.

- **Retail** : Companies use Apache Kafka to create omni-channel experiences, manage deliveries, inventory, and recommend products, and monitor user traffic.

- **Healthcare** : Data streaming applications such as IoT devices and HIPAA-compliant record-keeping systems are essential for medical staff to respond to system warnings.

- Real-time data streaming, event-driven architectures, and enabling flexible and scalable communication between distributed system components.

Companies Using Kafka

Figure 6

What is a Distributed System ?

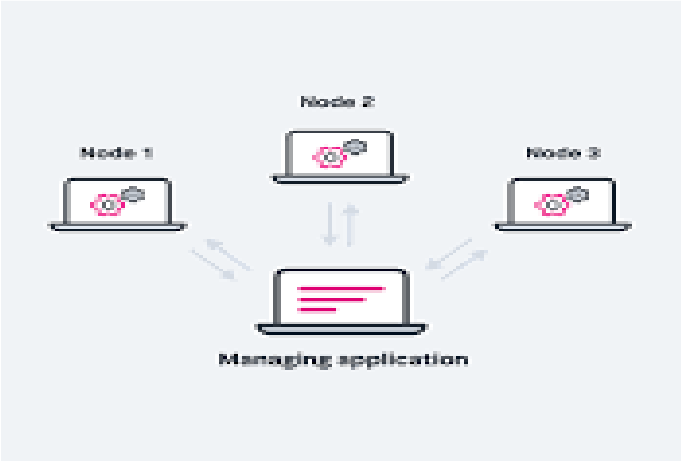


Figure 7

A distributed system is a network of computers or nodes that collaborate to solve complex tasks beyond the capacity of a single machine.

Characteristics of distributed systems:

- **Decentralization**:In a distributed system, nodes operate autonomously without a central authority or control.

- **Communication and Message Passing**:Nodes in a distributed system communicate through message passing via different channels.

- **Scalability**: Distributed systems can scale horizontally to handle larger workloads and improve performance by adding more nodes.

- **Fault Tolerance**: Distributed systems are resilient to failures through techniques like replication and fault detection, ensuring uninterrupted operation.

- **Consistency and Coordination**:Distributed systems ensure consistency and coordination through consensus algorithms and coordination protocols.

- **Load Balancing**: Distributed systems use load balancing techniques for efficient resource utilization.

- **Distributed Computing Paradigms**: Distributed computing encompasses different paradigms for specific use cases, including client-server, peer-to-peer, distributed databases, and file systems.

- **Performance and Latency**: Distributed systems prioritize low latency and high performance by minimizing network communication overhead and data transfer times.

UseCase Of Distributed System

- Cloud computing
- Big data processing
- Internet-scale services
- Artificial intelligence

Kafka Architecture

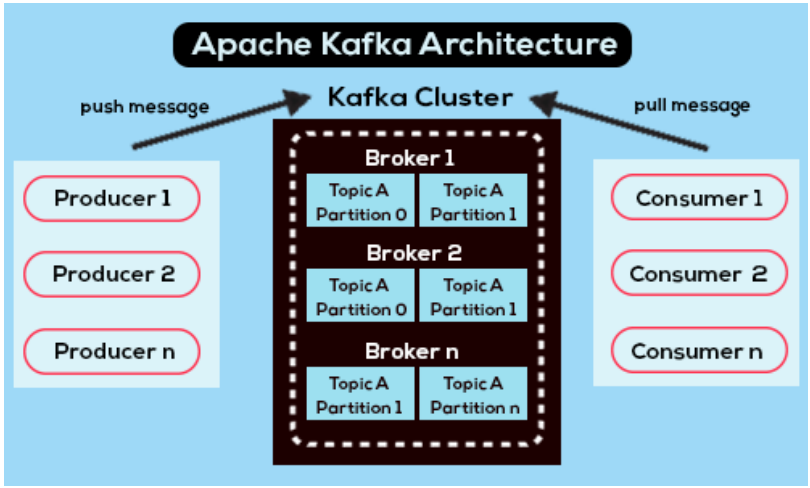


Figure 8

Kafka is built on a distributed publish-subscribe messaging system architecture. Its architecture consists of several key components and concepts working together to enable efficient and reliable data streaming.

Kafka Terminologies

1. **Topics**:In Kafka, topics categorize records and enable publish-subscribe messaging. They can be partitioned for scalability and parallel processing.

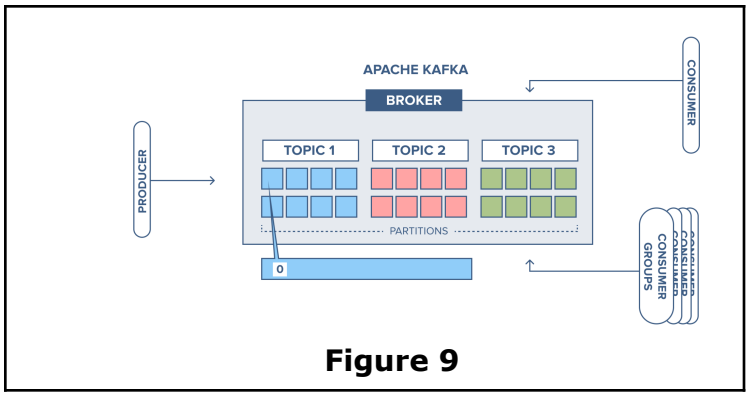


Figure 9

2. **Producers**: Producers publish records to Kafka topics, ensuring high throughput and distributed operation.

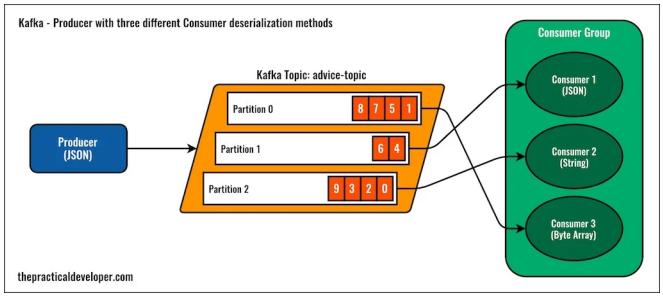


Figure 10

3. **Consumers**: Consumers consume Kafka topics' records, subscribe to them, and form consumer groups for load balancing and parallel processing.

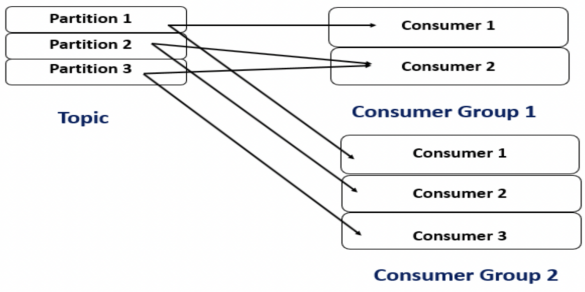


Figure 11

4. **Brokers:** Kafka cluster comprises brokers, who store and serve records from producers, allowing horizontal scaling by adding more brokers.

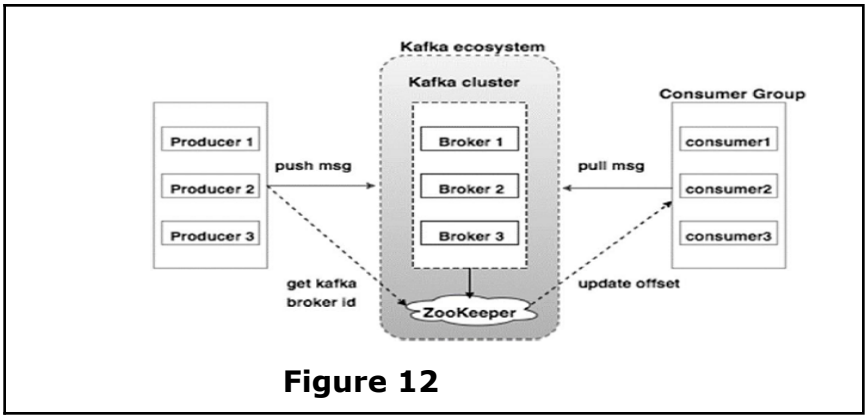


Figure 12

5. **Partitions**: Kafka partitions enable parallelism and scalability by distributing data across multiple brokers.

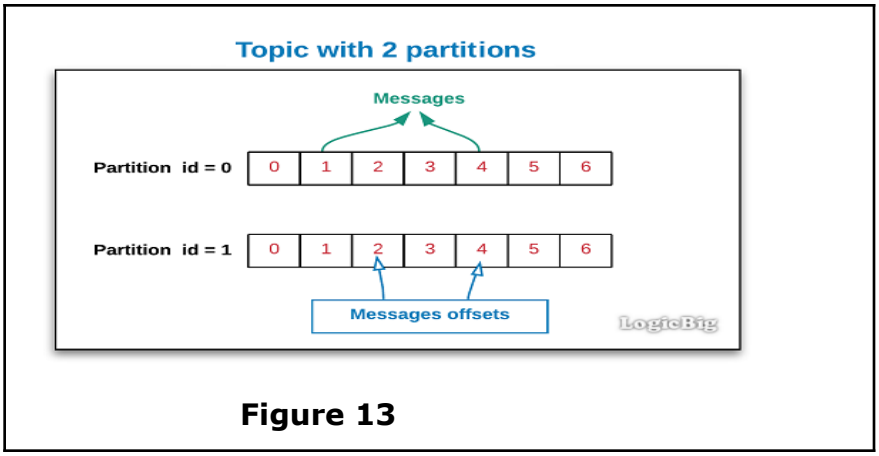


Figure 13

6. **Replication**: Kafka offers data replication for fault tolerance and reliability, ensuring durability and availability across multiple brokers.

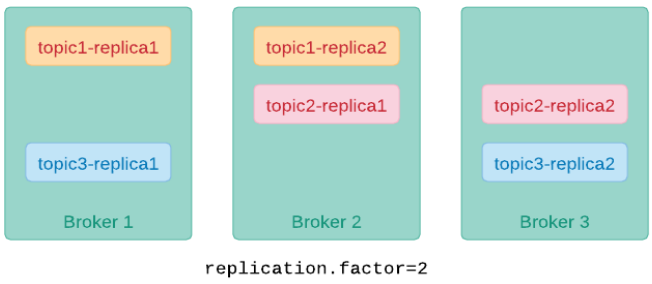


Figure 14

7. **ZooKeeper**:Kafka uses Apache ZooKeeper for cluster coordination, managing metadata, broker status, and leader election.

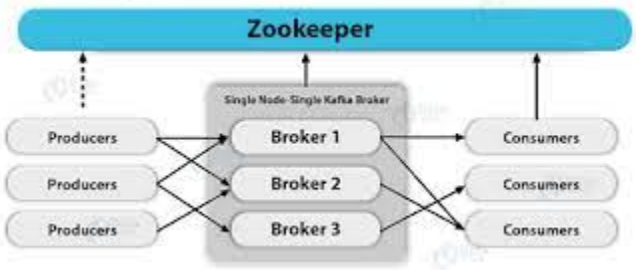


Figure 15

8. **Offsets**: Kafka uses offsets to track consumer progress and resume reading from where they left off.

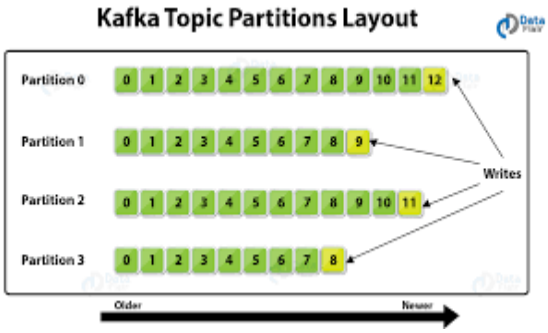


Figure 16

9. **Records:**Kafka stores publish-subscribe messages as records with key/value pairs and metadata, including timestamps.

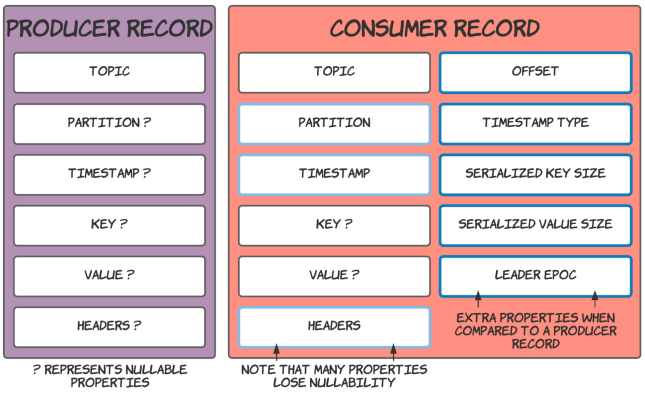


Figure 17

Kafka architecture enables high-volume, fault-tolerant, scalable data streaming with durability, low latency, and seamless integration.

Key components of Kafka (Keywords)

1.Broker

- Kafka is a distributed system that implements the basic features of an ideal publish-subscribe system.
- Each host in the Kafka cluster runs a server called a broker that stores messages sent to the topics and serves consumer requests.

Brokers in a Publish-Subscribe System

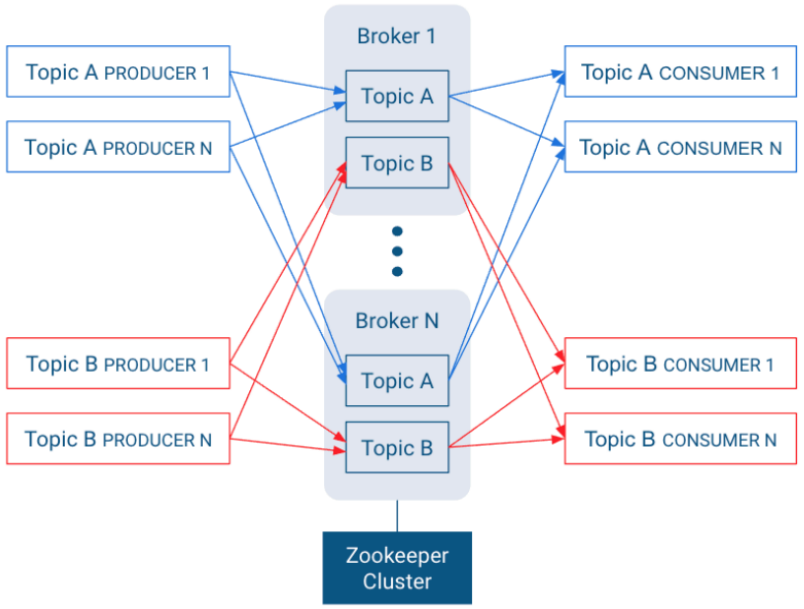


Figure 18

- Kafka is designed to run on multiple hosts, with one broker per host.
  - If a host goes offline, Kafka does its best to ensure that the other hosts continue running.

    - This solves part of the "**No Downtime**" and "**Unlimited Scaling**" goals of the ideal publish-subscribe system.

- Kafka brokers all talk to Zookeeper for distributed coordination.

- Topics are replicated across brokers.
    - Replication is an important part of "**No Downtime**", "**Unlimited Scaling**," and "**Message Retention**" goals.

- There is one broker(controller)which is responsible for coordinating the cluster. That broker is called the controller.

2.Topics

- In the publish-subscribe system, messages from one publisher(producers) will have to find their way to the subscriber(consumers).
    - To achieve this, Kafka introduces the concept of topics, which allow for easy matching between producers and consumers.

- A topic is a queue of messages that share similar characteristics.
    - For example, a topic might consist of instant messages from social media or navigation information for users on a web site.
    - Topics are written by one or more producers and read by one or more consumers.

- Each topic is identified by its name. This name is part of a global namespace of that Kafka cluster.
    - Producer or Consumer connects to the publish-subscribe system to read from or write to a specific topic.

3. Partitions

- Topics in Kafka consist of partitions(Like A table consisting of records /rows).

- Topics in Kafka are divided into a configurable number of parts, which are known as partitions.

- Partition is where the message lives inside the topic.

- Each message in the partition is ordered incrementally with an ID which is known as offset ID

Records in a Topic are Stored in Partitions, Partitions are Replicated across Brokers

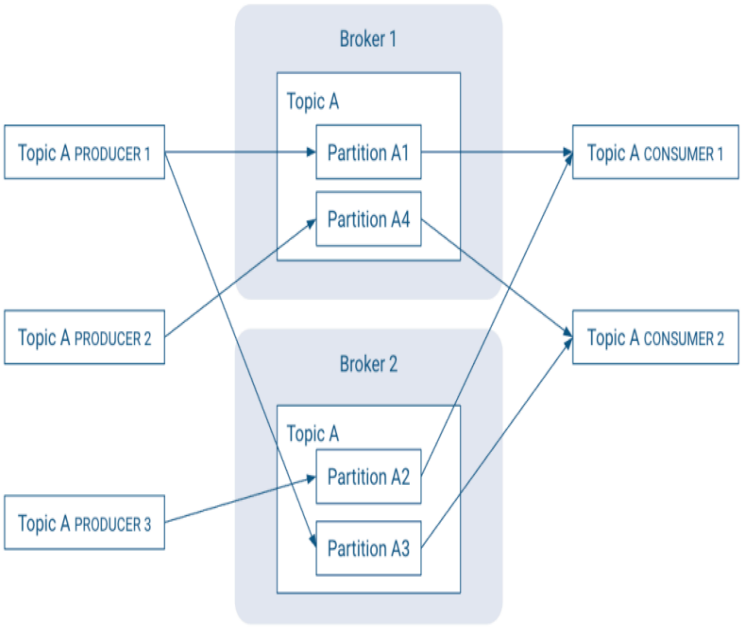


Figure 19

- Partitions are the key to keeping good record throughput. Choosing the correct number of partitions and partition replications for a topic.

- Spreads leader partitions evenly on brokers throughout the cluster

- Makes partitions within the same topic are roughly the same size

- Balances the load on brokers.

4. Replica

- Topic replica refers to a copy of a topic partition that is stored on a different broker within a Kafka cluster.

- Replicas are used to provide fault tolerance and data redundancy.

Key points about topic replicas:

- **Replication Factor:**
  - Replication factor determines the number of replicas for each partition of the topic.

  - The replication factor defines how many copies of the data will be maintained across the cluster.

- **Leader and Followers:**
  - Each partition of a topic has one replica designated as the leader and the remaining replicas as followers.

  - The leader replica handles all read and write requests for that partition, while the followers passively replicate data from the leader.

- **High Availability:**
  - Topic replicas provide high availability.

  - If the leader replica fails, one of the followers automatically takes over as the new leader, ensuring that the topic remains accessible and the data remains available.

- **Data Replication:**
  - Kafka uses a mechanism called the "replication protocol" to replicate data between replicas.

- The leader replica receives writes and then replicates the data to its followers using a combination of TCP-based protocol and file-based replication.

- **Fault Tolerance:**
  - Topic replicas ensure fault tolerance by distributing data across multiple brokers.

  - If a broker or replica fails, the data remains available on other replicas, allowing for seamless recovery and continuity of data processing.

- **Durability:**
  - Kafka writes messages to the leader replica, and once the data is successfully replicated to the followers, it is considered durable.

  - This replication mechanism ensures that data is not lost even if a broker fails.

By having multiple replicas of a topic partition, Kafka provides reliability, scalability, and fault tolerance, making it suitable for handling large-scale data streams and mission-critical applications.

Replication Factor

- The replication factor in Apache Kafka refers to the number of copies (replicas) of each topic partition that are maintained across the Kafka cluster. It determines the level of data redundancy and fault tolerance for a topic.

Key features about the replication factor:

- **Configuring Replication Factor:**
  - Creation of a topic in Kafka, you specify the replication factor as a parameter.

    - It can be set based on your desired level of fault tolerance and performance requirements.

- **Minimum Replication Factor:**
    - The replication factor must be greater than or equal to 1. It ensures that at least one replica of each partition is available in the cluster.

- **Replica Placement:**
    - Kafka distributes replicas across different brokers in the cluster to ensure fault tolerance and high availability. The replicas for a partition are typically distributed across multiple brokers, preferably on different machines, racks, or data centers.

- **Leader and Follower Replicas:**
    - Each partition of a topic has one leader replica and multiple follower replicas.

    - The leader replica handles all read and write requests for the partition, while the followers replicate the data from the leader.

    - The leader and follower roles can change dynamically in response to failures or rebalancing.

- **Fault Tolerance**:
    - The replication factor ensures that even if a broker or replica fails, the data remains accessible and durable.

    - If a leader replica fails, one of the followers is automatically elected as the new leader, maintaining the availability of the topic.

- **Performance Impact:**

- Increasing the replication factor introduces additional network overhead and disk I/O due to replication and synchronization of data.

- Replication factor is a trade-off between fault tolerance and performance.

- **Recommended Replication Factor:**
  - Replication factor recommended to set minimum of 3, ensuring that each partition has three replicas.

  - This provides fault tolerance for up to two replica failures.

  - By configuring an appropriate replication factor, you can ensure data durability, high availability, and fault tolerance in your Kafka cluster.

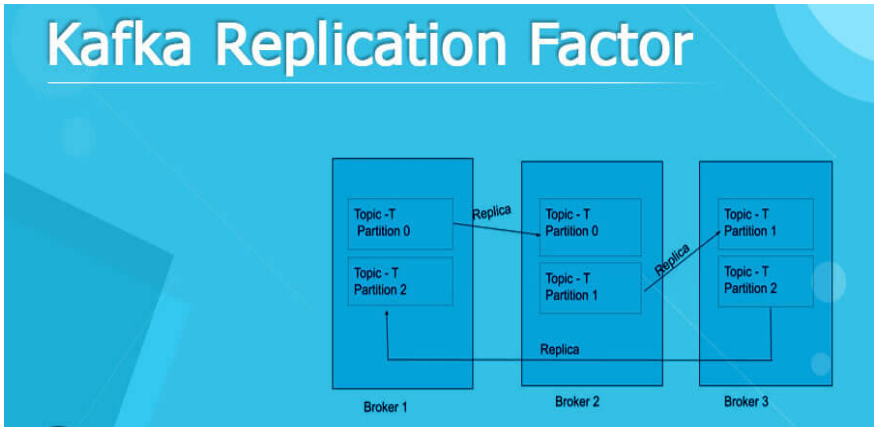


Figure 20

What if both Replica and data is present ?

- Kafka provides one broker's partition as a **leader**, and the rest of them become its **followers**.
  - The followers(brokers) will be allowed to synchronize the data.

- But, in the presence of a leader, none of the followers is allowed to serve the client's request.
- These replicas are known as ISR(in-sync-replica). So, Apache Kafka offers multiple ISR(in-sync-replica) for the data.
- Only the leader is allowed to serve the client's request. The leader handles all the read and writes operations of data for the partitions.

- The leader and its followers are determined by the zookeeper

Kafka Cluster

A Kafka cluster consists of multiple Kafka brokers (servers).

- Brokers work together to provide a distributed and fault-tolerant environment.

- Cluster architecture enables scalability, reliability, and high-performance data handling.

- Data is organized into topics, which are further divided into partitions.

- Each partition has one leader broker and multiple follower brokers.

- Data replication ensures fault tolerance and data availability.

- Apache ZooKeeper is used for maintaining cluster metadata and coordination.

- Kafka clusters can scale horizontally by adding more brokers.

- High availability is achieved through distributed partitioning and leader election.

- Data durability is ensured by storing data on disk and configurable retention policies.

- A Kafka cluster can be expanded without downtime.

- These clusters are used to manage the persistence and replication of message data.

- Zookeeper manages the cluster /Brokers/Leader/Follower
-
- It registers and load balance the brokers.

Producers

Producers are the components responsible for publishing data to Kafka topics. They are client applications or processes that generate and send messages to Kafka brokers.

Key features about Kafka producers:

- Producers publish data to Kafka topics, which are logical categories or feeds to which messages are written.

- Producers can send messages to one or more topics.

- Producers can choose to send messages to specific partitions within a topic or rely on the default partitioning mechanism.

- Producers can control the ordering of messages within a partition by setting the message key.

- Producers are designed to be highly scalable and can handle large message volumes and high message throughput.

- Producers can be developed using various programming languages using Kafka client libraries.

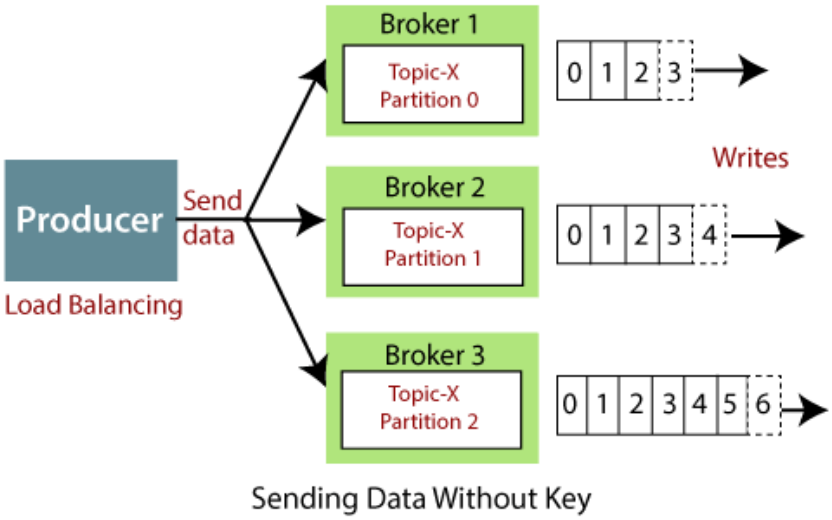


Figure 21

Consumers

Consumers are the components responsible for subscribing to Kafka topics and processing the published messages.

- Consumers read data from Kafka topics and consume it based on their own pace and processing capabilities.

Key features about Kafka consumers:

- Consumers subscribe to one or more Kafka topics and receive messages published to those topics.

- Consumers can be part of a consumer group, where multiple consumers work together to consume messages from a topic. Each consumer in a group processes a subset of the partitions within the topic.

- Consumers read messages from partitions in an ordered manner, ensuring the sequential processing of messages within each partition.

- Consumers can control their position in the topic by committing their current offset, which represents the position of the last processed message.

- Consumers can choose to manually manage the offset or rely on Kafka's built-in offset management system.

- Consumers can be implemented in various programming languages using Kafka client libraries.

- Consumers can process messages in real-time or in batches, depending on their processing requirements.

- Consumers can handle failures and recover their position in the topic by leveraging Kafka's offset management and consumer group coordination mechanisms.

- Consumers can scale horizontally by adding more consumer instances to a consumer group, allowing for higher message processing throughput.

- Consumers can also perform parallel processing by increasing the number of threads within a consumer instance.

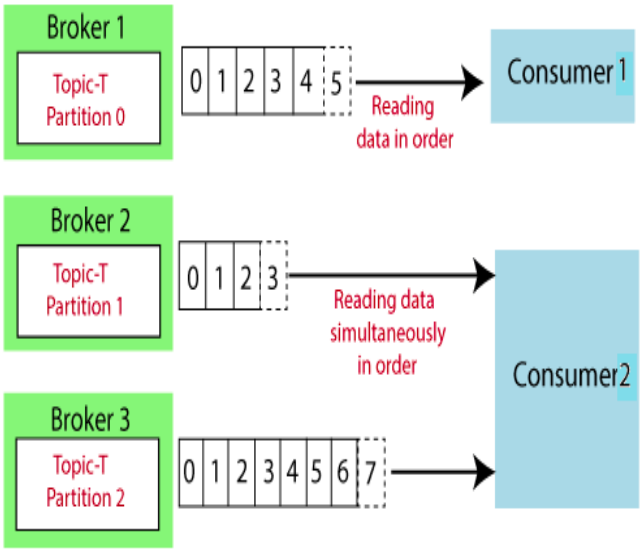
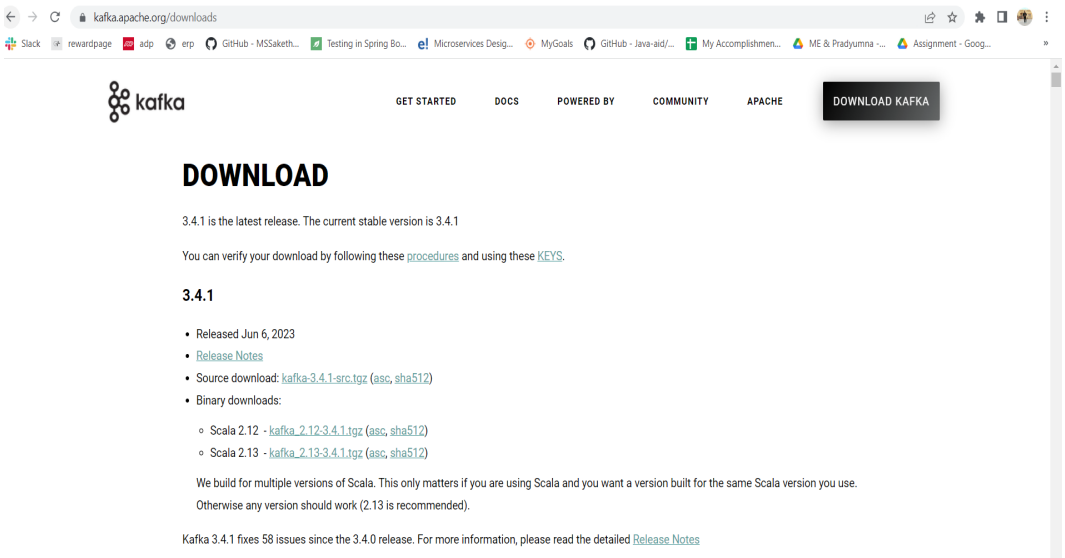


Figure 22

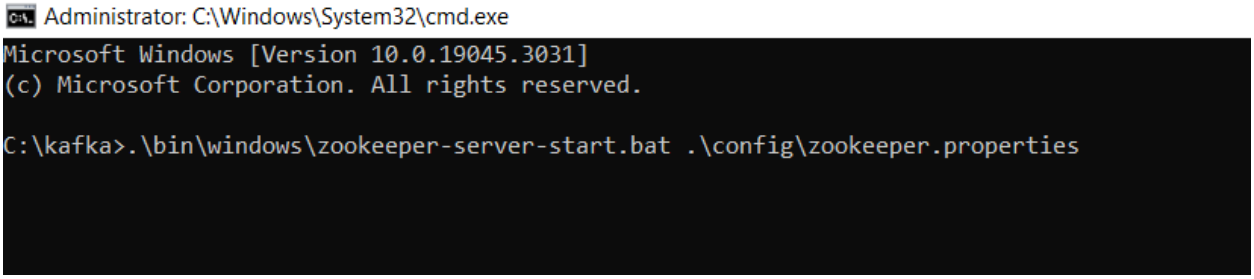
Installation of Zookeeper and Apache Kafka

➢ Download the setup file from the official Kafka.

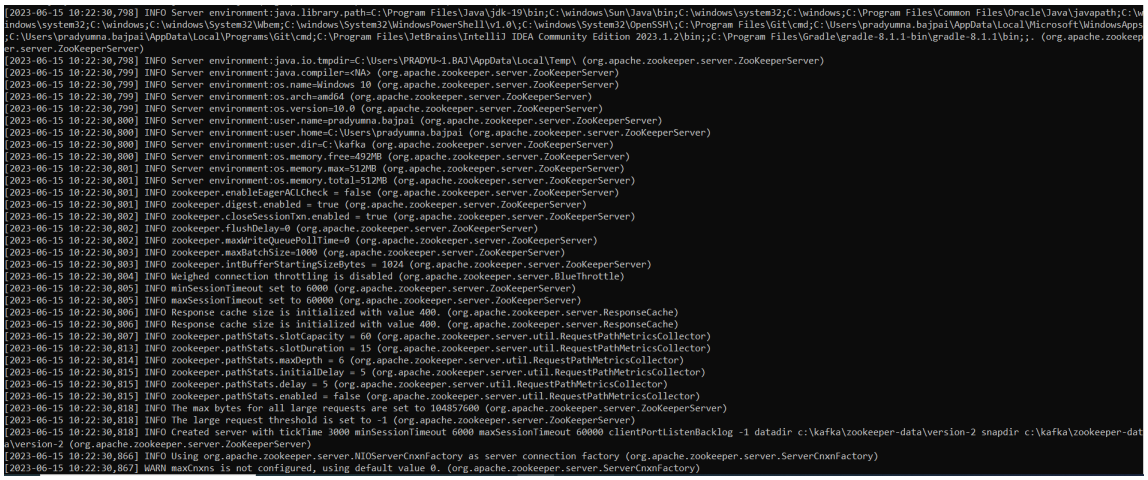


➢ Extract the Zip file downloaded

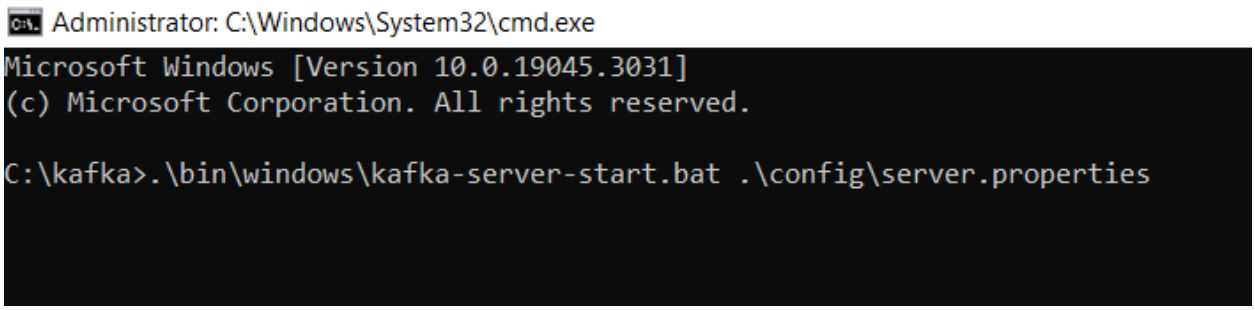
- ➢ Copy the folder to C drive or D drive
- ➢ Open command prompt inside Kafka Folder and run
  - For Windows:- .\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties
  - For Ubuntu:- bin/zookeeper-server-start.sh config/zookeeper.properties



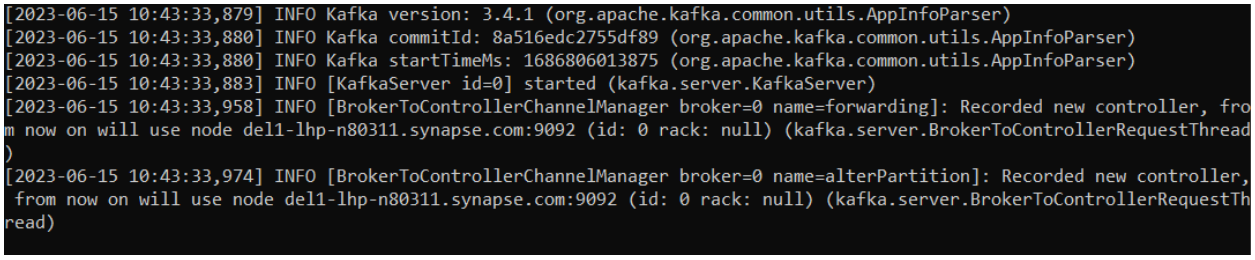
Output:-



- ❖ Steps to Start Kafka
  - ➢ **Open command prompt inside Kafka Folder and run**
    - **For Windows:- .\bin\windows\kafka-server-start.bat .\config\server.properties**
    - **For Ubuntu:- bin/kafka-server-start.sh config/server.properties**



Output:-



- By default the broker server will using 9092 port

Demo 1:Kafka Consumer and Producer

➢ Create a topic. Open a new command prompt window for this step.

C:\kafka\bin\windows>kafka-topics.bat--create --bootstrap-server localhost:9092 --topic test

➢ Open two new command prompt windows for the producer and consumer.

➢ In the producer window, you can enter messages that will be sent to the topic.

C:\kafka\bin\windows>kafka-console-producer.bat --broker-list localhost:9092 --topic test

➢ In the consumer window, you can receive and display the messages from the topic.

C:\kafka\bin\windows>kafka-console-consumer.bat --topic test --bootstrap-server localhost:9092 --from-beginning

Demo 2:Multiple Zookeeper and Kafka Demo

- **To run multiple Zookeeper at same time:-**
  - Ensure that you have downloaded and extracted the ZooKeeper binary distribution on your Windows machine.
  - **Configure ZooKeeper:**
    - Create multiple ZooKeeper configuration files, such as zookeeper 1.properties, zookeeper 2.properties, etc., by copying the default zookeeper.properties file.
    - Modify the dataDir property in each configuration file to specify different data directories for each ZooKeeper instance.
  - Modify the clientPort property to specify unique client ports for each instance.
  - Optionally, modify other properties like tickTime, initLimit, syncLimit, etc., if desired.
    - **tickTime=2000**
    - **initLimit=5**
    - **syncLimit=2**
    - **Start ZooKeeper:**
  - Open multiple command prompt or terminal windows.
  - In each window, navigate to the Kafka installation directory.
  - Run the following command for each ZooKeeper instance, replacing zookeeper 1.properties, zookeeper 2.properties, etc., with the respective configuration file names:
    - **bin\windows\zookeeper-server-start.bat config\zookeeper1.properties**
  - **Configure Kafka:**
    - Create multiple Kafka server properties files, such as server1.properties, server2.properties, etc., by copying the default server.properties file.
    - Modify the following properties in each configuration file:
    - broker.id: Set a unique ID for each Kafka broker.

        - listeners: Specify unique listener configurations (host:port) for each instance.
        - **server.1=localhost:2888:3888**
        - **server.2=localhost:2889:3889**
        - **server.3=localhost:2890:3890**
    - log.dirs: Set different data directories for each instance.
    - zookeeper.connect: Specify the connection string with multiple ZooKeeper addresses.
    - **Start Kafka:**
    - In separate command prompt or terminal windows, navigate to the Kafka installation directory.
    - Run the following command for each Kafka instance, replacing server1.properties, server2.properties, etc., with the respective configuration file names:
        - **bin\windows\kafka-server-start.bat config\server1.properties**
    - **Create a topic:**
    - In a new command prompt or terminal window, navigate to the Kafka installation directory.
    - Run the following command to create a topic, replacing <topic-name>, <replication-factor>, and <num-partitions> with your desired values:
        - **bin\windows\kafka-topics.bat --create --topic <topic-name> --replication-factor <replication-factor> --partitions <num-partitions> --bootstrap-server localhost:9092**
    - **Produce and consume messages:**
        - In separate command prompt or terminal windows, navigate to the Kafka installation directory.
        - To produce messages, run the following command, replacing <topic-name> with the topic you created:

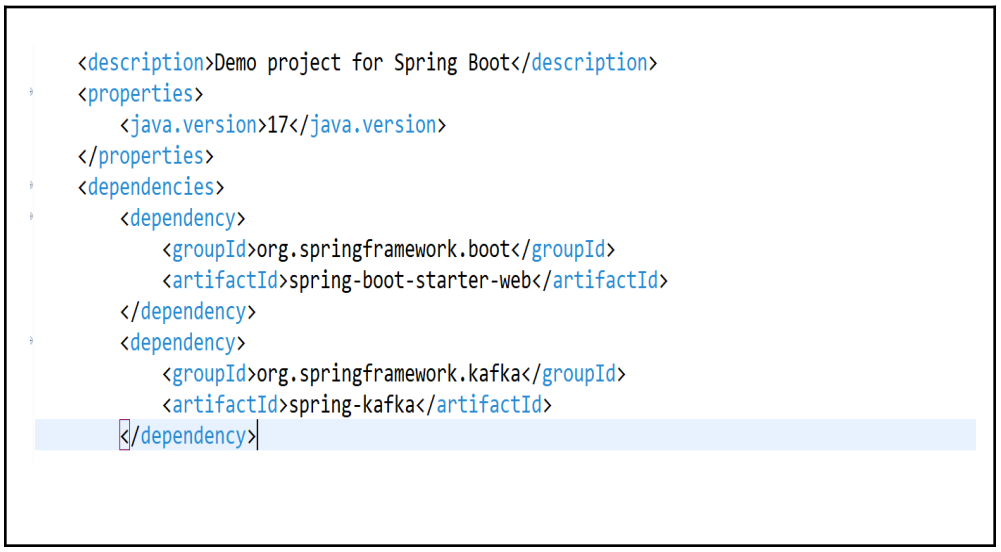
        - **bin\windows\kafka-console-producer.bat --topic <topic-name> --bootstrap-server localhost:9092**

        - To consume messages, run the following command, replacing <topic-name> with the topic you created:

- **bin\windows\kafka-console-consumer.bat --topic <topic-name> --bootstrap-server localhost:9092**

Demo 3:Apache Kafka with SpringBoot

- Set up Apache Kafka:
- Download and install Apache Kafka on your machine.
- Start a Kafka broker by running the following command in a terminal or command prompt:
  - **bin\windows\kafka-server-start.bat config\server.properties**
- **Create a Spring Boot project:**
  - Create a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io).
  - Include the required dependencies: Spring for Apache Kafka and Spring Web.



```
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>17</java.version>
</properties>
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.kafka</groupId>
        <artifactId>spring-kafka</artifactId>
    </dependency>
```

- **Configure Kafka properties:**
- Open the application.properties file in your Spring Boot project.
- Add the following properties to configure the connection to Kafka:
  - **spring.kafka.bootstrap-servers=localhost:9092**
  - **spring.kafka.consumer.group-id=my-group-id**

- Modify the bootstrap servers and consumer group ID as per your Kafka setup.
- Create POJO/Java class named as Message whose object is shared between KafkaProducer and KafkaConsumer

```java
package com.glo.app.kafkaMessageDemo.model;

public class Message {
    private String content;
   public Message() {
    }
    public Message(String content) {
        this.content = content;
    }

    public String getContent() {
        return content;
    }
    public void setContent(String content) {
        this.content = content;
    }
    @Override
    public String toString() {
        return "Message{" +
                "content='" + content + '\'' +
                '}';
    }
}
```

- **Create a Kafka producer:**
  - Create a Java class to define a Kafka producer named **MessageProducerService**.
  - Use the @EnableKafka annotation on your application class or a configuration class.
  - Autowire the KafkaTemplate to send messages:

    @Autowired
    private KafkaTemplate<String, String> kafkaTemplate

  - Implement a method to send messages:

public void sendMessage(String topic, String message) {
kafkaTemplate.send(topic, message);
}

- **Create a Java Class named MessageProducerService which will act as KafkaProducer**

```java
import org.springframework.beans.factory.annotation.Autowired;

@Service
public class MessageProducerService {
    private static final String TOPIC = "demo_topic";

    @Autowired
    private final KafkaTemplate<String, Object> kafkaTemplate;

    @Autowired
    public MessageProducerService(KafkaTemplate<String, Object> kafkaTemplate) {
        this.kafkaTemplate = kafkaTemplate;
    }

    public void sendMessage(Message message) {
        kafkaTemplate.send(TOPIC, message);
    }
}
```

- **Create a Kafka consumer:**
  - Create a Java class to define a Kafka consumer named MessageConsumerService.
  - Use the **@KafkaListener** annotation on a method to specify the topic to listen to and handle received messages:
  - **@KafkaListener(topics = "demo_topic")**
    public void receiveMessage(String message) {
    // Process the received message
    System.out.println("Received message: " + message);

}

```
package com.glo.app.kafkaMessageDemo.service;

import org.slf4j.LoggerFactory;

@Service
public class MessageConsumerService {

    private static final org.slf4j.Logger LOGGER=LoggerFactory.getLogger(MessageConsumerService.class);

    @KafkaListener(topics = "demo_topic", groupId = "demo_group")
    public void consumeMessage(Message message) {
        LOGGER.info(String.format("Message Received -> %s", message.toString()));

        System.out.println("-----Received message:------ " + message);
        // Process the received message as per your requirement
 }

}

```

- **Create a Controller named(MessageController) and expose a Rest Endpoint as /publish which is used by KafkaProducer to publish the Message**

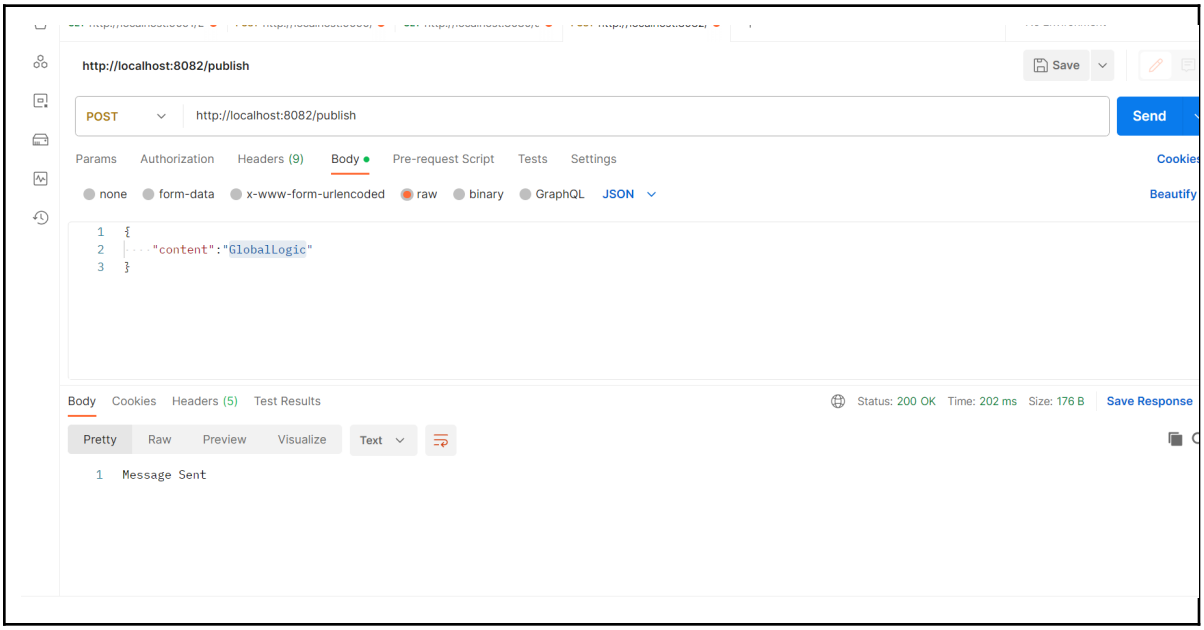
```
import org.springframework.beans.factory.annotation.Autowired;

@RestController
public class MessageController {

    @Autowired
    private  MessageProducerService producerService = null;


    @Autowired
    public MessageController(MessageProducerService producerService) {
        super();
        this.producerService = producerService;
    }
      @PostMapping("/publish")
        public String publishMessage(@RequestBody Message message) {
            producerService.sendMessage(message);
            return "Message Sent";
        }
    public MessageController() {
        super();
        // TODO Auto-generated constructor stub
    }
```

- Build and run the Spring Boot application.
- Run the application, and the Kafka consumer will start listening to the specified topic.
- Use the Kafka producer to send messages to the topic, and the consumer will receive and process them as given below


http://localhost:8082/publish
Save
POST
http://localhost:8082/publish
Send
Params
Authorization
Headers (9)
Body
Pre-request Script
Tests
Settings
Cookies
none
form-data
x-www-form-urlencoded
raw
binary
GraphQL
JSON
Beautify
{
"content":"GlobalLogic"
}
Body
Cookies
Headers (5)
Test Results
Status: 200 OK
Time: 202 ms
Size: 176 B
Save Response
Pretty
Raw
Preview
Visualize
Text
Message Sent

Summary

- Kafka is an open-source distributed event streaming platform.
- Kafka is a popular choice for handling real-time data streaming and processing in modern applications.
- Kafka is built on a distributed publish-subscribe messaging system architecture
- Each host in the Kafka cluster runs a server called a broker that stores messages sent to the topics and serves consumer requests.
- Topics categorize records and enable publish-subscribe messaging.
- Each host in the Kafka cluster runs a server called a broker that stores messages sent to the topics and serves consumer requests.
- Consumers consume Kafka topics' records, subscribe to them, and form consumer groups for load balancing and parallel processing.
- Kafka partitions enable parallelism and scalability by distributing data across multiple brokers.
- Kafka offers data replication for fault tolerance and reliability, ensuring durability and availability across multiple brokers.
- Apache Kafka with Spring Boot provides a powerful combination for building distributed messaging systems.
- Integrating Kafka with Spring Boot simplifies the development process by providing convenient abstractions and configurations.
- Spring Boot's integration with Kafka makes it easy to handle different message patterns, scale the application, and leverage advanced Kafka features.