**Spring Boot - Integration with Kafka**

**Apache Kafka** is a distributed messaging system designed for high-throughput and low-latency message delivery. It is widely used in real-time data pipelines, streaming analytics, and other applications requiring reliable and scalable data processing. Kafka’s publish-subscribe model allows producers to send messages to topics, which subscribers then consume.

Integrating Kafka with Spring Boot enables developers to build robust and scalable applications capable of handling large volumes of data efficiently. Spring Boot provides the spring-kafka library, which simplifies the integration process by offering a set of APIs and configuration options that align with Spring's programming model.

**Kafka's Role and Architecture**

Kafka is a distributed streaming platform that functions as a message broker, facilitating the exchange of data between producers and consumers in real-time. It works based on the **publish-subscribe model** and is used to build real-time data pipelines and streaming applications. The main components in Kafka are:

* **Producer**: A client that sends messages to a Kafka topic. The producer publishes data to the Kafka cluster.
* **Consumer**: A client that reads messages from Kafka topics. Consumers subscribe to topics and process the data.
* **Topics**: Logical channels where producers send messages, and from which consumers read. Each topic can have multiple partitions, allowing Kafka to parallelize processing across consumers.
* **Broker**: A server that stores and serves messages. Kafka is designed to be distributed, so multiple brokers can be set up for scalability and fault tolerance.
* **Zookeeper**: Coordinates Kafka brokers, keeping track of topics, partitions, and message offsets.

**Configuring Kafka in Spring Boot**

Spring Boot provides a convenient way to configure Kafka using the **spring-kafka** project, part of the Spring ecosystem. Integration involves setting up Kafka-related configurations, including defining properties for producers, consumers, and the Kafka server itself.

**Kafka Configuration Properties**

Configuration properties are specified in the application.properties or application.yml file of the Spring Boot application. Key properties include:

* **Bootstrap Servers**: Specifies Kafka server addresses for both producers and consumers to send/receive messages.
* **Key and Value Serializers/Deserializers**: Kafka requires messages to be serialized into bytes. Producers serialize keys and values, and consumers deserialize them upon receiving.
* **Group ID**: Used by consumers to identify themselves as part of a group. Each consumer group reads from the same topic independently.

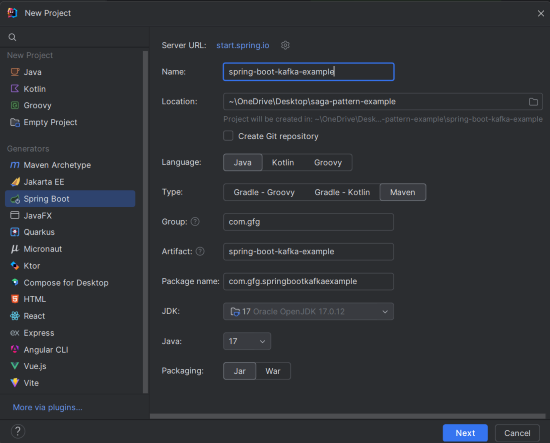
**Step-by-step Implementation to Integrate Spring Boot with Kafka**

**Step 1: Create a new Spring Boot project**

Create a new Spring Boot project using IntelliJ IDEA. Choose the following options:

* Name: spring-boot-kafka-example
* Language: Java
* Type: Maven
* Packaging: Jar

Click on the **Next**button.

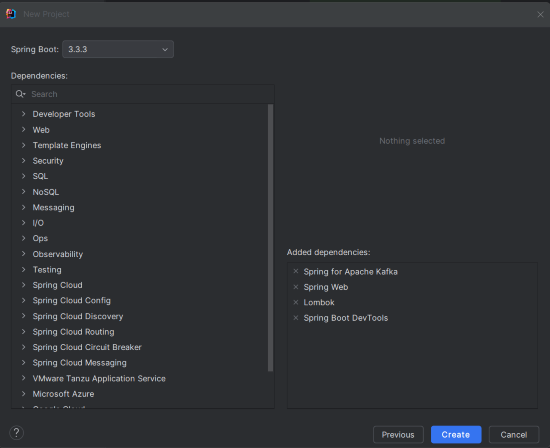


**Step 2: Add Dependencies**

Add the following dependencies into the Spring Boot project.

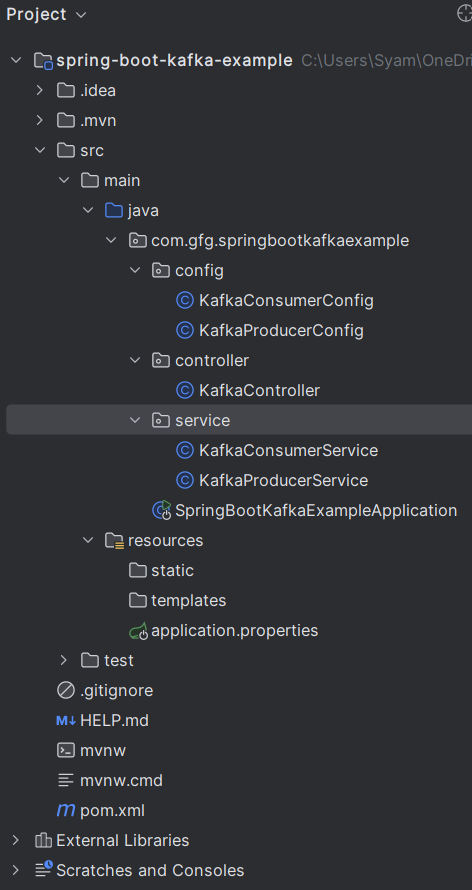
* Spring for Apache Kafka
* Spring Web
* Lombok
* Spring Boot DevTools

Click on the **Create**button.



**Project Structure**

Once the project is created, the file structure will look like the below image.



**Step 3: Configure Application Properties**

Open the application.properties file and add the following Kafka configuration.

spring.application.name=spring-boot-kafka-example

spring.kafka.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.key-serializer=org.apache.kafka.common.serialization.StringSerializer

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

**Step 4: Kafka Consumer Configuration**

Create the KafkaConsumerConfig class and this configuration class sets up the Kafka consumer of the Spring Boot application.

**config/KafkaConsumerConfig.java**

**package** **com.gfg.springbootkafkaexample.config**;

**import** **org.apache.kafka.clients.consumer.ConsumerConfig**;

**import** **org.apache.kafka.common.serialization.StringDeserializer**;

**import** **org.springframework.context.annotation.Bean**;

**import** **org.springframework.context.annotation.Configuration**;

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

**import** **org.springframework.kafka.config.ConcurrentKafkaListenerContainerFactory**;

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

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

**import** **java.util.HashMap**;

**import** **java.util.Map**;

@EnableKafka

@Configuration

**public** **class** **KafkaConsumerConfig** {

@Bean

**public** ConsumerFactory<String, String> consumerFactory() {

Map<String, Object> configProps = **new** HashMap<>();

configProps.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

configProps.put(ConsumerConfig.GROUP\_ID\_CONFIG, "group\_id");

configProps.put(ConsumerConfig.KEY\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

configProps.put(ConsumerConfig.VALUE\_DESERIALIZER\_CLASS\_CONFIG, StringDeserializer.class);

**return** **new** DefaultKafkaConsumerFactory<>(configProps);

}

@Bean

**public** ConcurrentKafkaListenerContainerFactory<String, String> kafkaListenerContainerFactory() {

ConcurrentKafkaListenerContainerFactory<String, String> factory = **new** ConcurrentKafkaListenerContainerFactory<>();

factory.setConsumerFactory(consumerFactory());

**return** factory;

}

}

**Step 5: Kafka Producer Configuration**

Create the KafkaProducerConfig class and this configuration class sets up the Kafka producer of the Spring Boot application.

**config/KafkaProducerConfig.java**

**package** **com.gfg.springbootkafkaexample.config**;

**import** **org.apache.kafka.clients.producer.ProducerConfig**;

**import** **org.apache.kafka.common.serialization.StringSerializer**;

**import** **org.springframework.context.annotation.Bean**;

**import** **org.springframework.context.annotation.Configuration**;

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

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

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

**import** **org.springframework.kafka.support.serializer.JsonSerializer**;

**import** **java.util.HashMap**;

**import** **java.util.Map**;

@Configuration

**public** **class** **KafkaProducerConfig** {

@Bean

**public** ProducerFactory<String, String> producerFactory() {

Map<String, Object> configProps = **new** HashMap<>();

configProps.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, "localhost:9092");

configProps.put(ProducerConfig.KEY\_SERIALIZER\_CLASS\_CONFIG, StringSerializer.class);

configProps.put(ProducerConfig.VALUE\_SERIALIZER\_CLASS\_CONFIG, JsonSerializer.class);

**return** **new** DefaultKafkaProducerFactory<>(configProps);

}

@Bean

**public** KafkaTemplate<String, String> kafkaTemplate() {

**return** **new** KafkaTemplate<>(producerFactory());

}

}

**Step 6: Kafka Producer Service**

Create the KafkaProducerService class and this service class sends the messages to the Kafka topic of the Spring application.

**service/KafkaProducerService.java**

**package** **com.gfg.springbootkafkaexample.service**;

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

**import** **org.springframework.stereotype.Service**;

@Service

**public** **class** **KafkaProducerService** {

**private** **static** **final** String TOPIC = "my\_topic";

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

**public** KafkaProducerService(KafkaTemplate<String, String> kafkaTemplate) {

**this**.kafkaTemplate = kafkaTemplate;

}

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

kafkaTemplate.send(TOPIC, message);

System.out.println("Message sent: " + message);

}

}

**Step 7: Kafka Consumer Service**

Create the KafkaConsumerService class and this service class listens to the Kafka topic and process the incoming messages of the Spring application.

**service/KafkaConsumerService.java**

**package** **com.gfg.springbootkafkaexample.service**;

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

**import** **org.springframework.stereotype.Service**;

@Service

**public** **class** **KafkaConsumerService** {

@KafkaListener(topics = "my\_topic", groupId = "group\_id")

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

System.out.println("Message received: " + message);

}

}

**Step 8: REST Controller**

Create the KafkaController class and this controller exposes the REST endpoint to send the messages to the Kafka topic of the Spring Boot application.

**controller/KafkaController.java**

**package** **com.gfg.springbootkafkaexample.controller**;

**import** **com.gfg.springbootkafkaexample.service.KafkaProducerService**;

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

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

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

@RestController

**public** **class** **KafkaController** {

**private** **final** KafkaProducerService kafkaProducerService;

**public** KafkaController(KafkaProducerService kafkaProducerService) {

**this**.kafkaProducerService = kafkaProducerService;

}

@GetMapping("/send")

**public** String sendMessage(@RequestParam String message) {

kafkaProducerService.sendMessage(message);

**return** "Message sent successfully";

}

}

**Step 9: Main Class**

This is the entry point of the Spring application. There is no changes are required in the main class.

**package** **com.gfg.springbootkafkaexample**;

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

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

@SpringBootApplication

**public** **class** **SpringBootKafkaExampleApplication** {

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

SpringApplication.run(SpringBootKafkaExampleApplication.class, args);

}

}

**pom.xml**

<?xml version="1.0" encoding="UTF-8"?>

**<project** xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd"**>**

**<modelVersion>**4.0.0**</modelVersion>**

**<parent>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-parent**</artifactId>**

**<version>**3.3.3**</version>**

**<relativePath/>** *<!-- lookup parent from repository -->*

**</parent>**

**<groupId>**com.gfg**</groupId>**

**<artifactId>**spring-boot-kafka-example**</artifactId>**

**<version>**0.0.1-SNAPSHOT**</version>**

**<name>**spring-boot-kafka-example**</name>**

**<description>**spring-boot-kafka-example**</description>**

**<url/>**

**<licenses>**

**<license/>**

**</licenses>**

**<developers>**

**<developer/>**

**</developers>**

**<scm>**

**<connection/>**

**<developerConnection/>**

**<tag/>**

**<url/>**

**</scm>**

**<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>**

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-devtools**</artifactId>**

**<scope>**runtime**</scope>**

**<optional>**true**</optional>**

**</dependency>**

**<dependency>**

**<groupId>**org.projectlombok**</groupId>**

**<artifactId>**lombok**</artifactId>**

**<optional>**true**</optional>**

**</dependency>**

**<dependency>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-starter-test**</artifactId>**

**<scope>**test**</scope>**

**</dependency>**

**<dependency>**

**<groupId>**org.springframework.kafka**</groupId>**

**<artifactId>**spring-kafka-test**</artifactId>**

**<scope>**test**</scope>**

**</dependency>**

**</dependencies>**

**<build>**

**<plugins>**

**<plugin>**

**<groupId>**org.springframework.boot**</groupId>**

**<artifactId>**spring-boot-maven-plugin**</artifactId>**

**<configuration>**

**<excludes>**

**<exclude>**

**<groupId>**org.projectlombok**</groupId>**

**<artifactId>**lombok**</artifactId>**

**</exclude>**

**</excludes>**

**</configuration>**

**</plugin>**

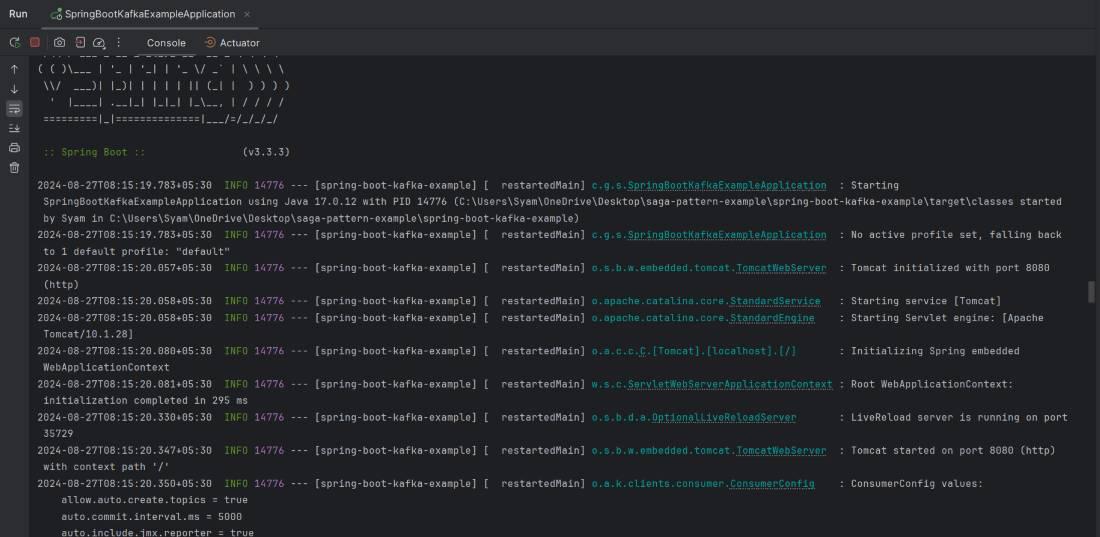
**</plugins>**

**</build>**

**</project>**

**Step 10: Run the Application**

Once completed the project, it will start and run at port 8080.



**Step 11: Start the Kafka and Zookeeper**

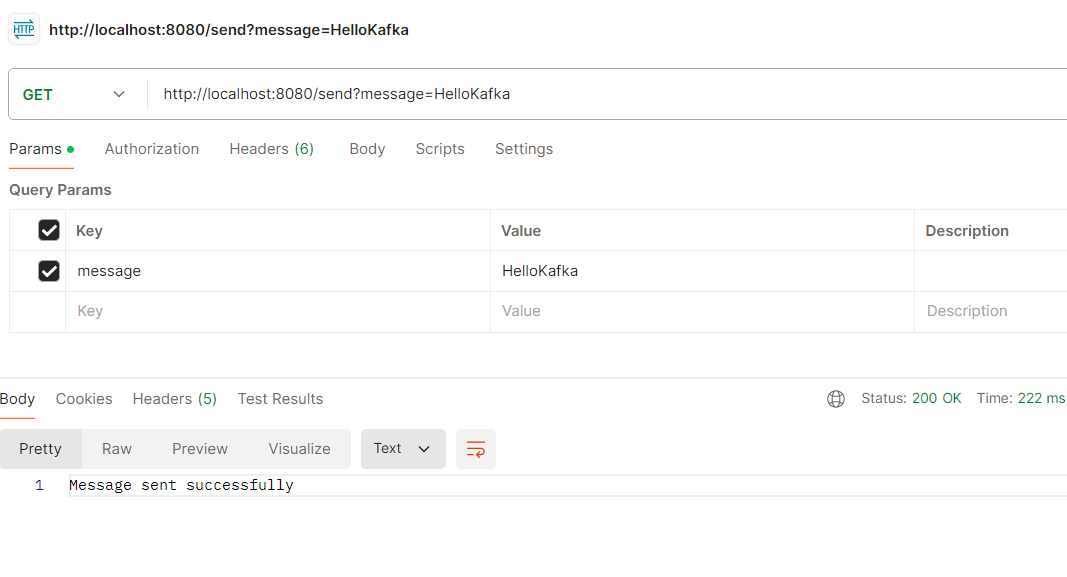
Make sure that the Kafka and zookeeper are running. We can refer this [article](https://www.geeksforgeeks.org/how-to-install-and-run-apache-kafka-on-windows/)to how to start and run the Kafka Server.

**Step 12: Test the Endpoint**

We can use the postman tool to send the GET request of the below endpoint.

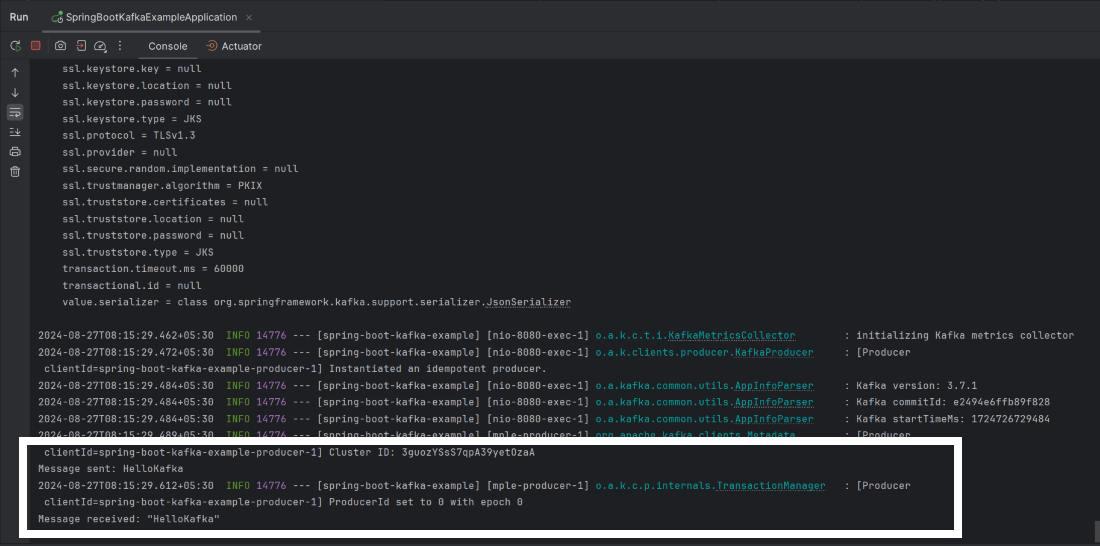
GET http://localhost:8080/send?message=HelloKafka

**Output:**



**Application Logs**

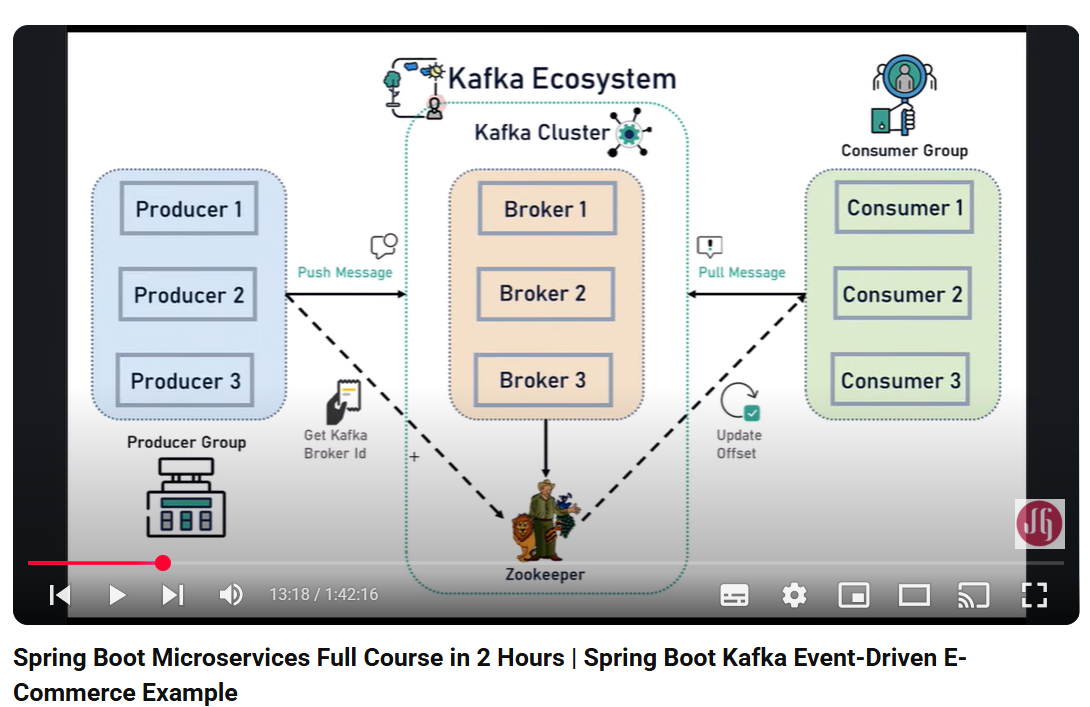
We can observe the console of the Spring Boot application to see the message being sent and received.



This example project demonstrates the basic implementation of the Kafka producer and consumer using the Spring Boot project. This project is structured to have separate the packages for the configuration, services, and controllers. This modular approach can be helpful in the maintaining and scaling the project.

**----------------------------------------------------**

What is Zookeeper ?



ZooKeeper is a centralized service used by Apache Kafka for managing and coordinating Kafka brokers, including maintaining configuration data, providing distributed synchronization, and performing leader elections. It acts as a central nervous system for the Kafka cluster, ensuring consistent and reliable operation.

Here's a breakdown of ZooKeeper's role in Kafka:

* **Broker Registration:** ZooKeeper keeps track of which brokers are part of the Kafka cluster.
* **Metadata Management:** ZooKeeper stores metadata about Kafka topics, partitions, and replicas.
* **Leader Election:** When a broker managing a partition fails, ZooKeeper facilitates the election of a new leader broker.
* **Configuration Management:** ZooKeeper stores configuration information for the Kafka cluster and topics.
* **Distributed Synchronization:** ZooKeeper provides synchronization services to ensure consistent state across the cluster.
* **Controller Management:** ZooKeeper elects and manages the Kafka controller, which is responsible for managing the cluster.