**Kafka Questions**

Apache Kafka Foundation Course - What is Apache Kafka

<https://www.learningjournal.guru/courses/kafka/kafka-foundation-training/what-is-kafka/>

**delete.topic.enable**

If you want to delete a topic, you can use topic management tool. But by default, deleting a topic is not allowed. You can't remove a topic because the default value for this parameter is false. That is reasonable protection for production environments. But in development or testing environment, you may want to delete topics. So, if you want Kafka to allow deleting a topic, you need to set this parameter to true.

**auto.create.topics.enable**

We have already discussed auto-create topic feature. If a producer starts sending messages to a non-existent topic, Kafka will create the topic automatically and accept the data. This behaviour is suitable for dev environments. But in a production environment, you may want to implement a more controlled approach. You can set this parameter to false, and Kafka will stop creating topics automatically. You can create topics manually using the topic management tool, and no one will be able to send data to a non-existent topic.

**default.replication.factor and num.partitions**

These two parameters are quite straightforward. The default values for both of them is one, and they are effective when you have auto create topics enabled. So, if Kafka is creating your topic automatically, the new topic will have only one partition and a single copy. If you want some other values, you can change the default settings accordingly.

**log.retention.ms and log.retention.bytes**

These two are critical and not obvious. So, whatever data you send to Kafka, it is not retained by Kafka forever. Kafka is not a database. You don't send data to Kafka for storage so that you can query it later. It is a message broker. It should deliver the data to the consumer and then clean it up. There is no reason to retain messages for longer than needed.  
Kafka gives you two options to configure the retention period. The default option is retention by time, and the default retention period is seven days. So, in this case, Kafka will clean up all the messages older than seven days. If you want to change the duration, you can specify your value for log.retention.ms configuration.  
Kafka gives you another option to define this retention period. You can specify it by size. That's where the second parameter log retention bytes is applicable. But this size applies to partition. So, if you set log.retention.bytes = 1 GB, Kafka will trigger a clean-up activity when the partition size reaches to 1 GB. Remember that it is not a topic size. It is partition size.

kafka Producers common Properties configurations interview questions :

Acknowledgements-

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface, application

Description automatically generated

Diagram

Description automatically generated

Text

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-7x/kafka-producer-workflow.jpg)Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated**max.in.flight.requests.per.connection : 5. It means producer don’t want to wait for acknowledgement for each request it he wants only wait on every Fifth request for ack.**

Apache Kafka Foundation Course - What is Apache Kafka

Welcome to Apache Kafka Tutorial at Learning journal. In this session, I will introduce you to Kafka. We will try to understand Kafka in less than 10 minutes. I am assuming that you have at least heard about Kafka and you already know that it is an Open Source project. Kafka was initially developed at LinkedIn and later open sourced in 2011. Since then it has evolved and established itself as a standard tool for building real-time data pipelines. Now it's securing its share in real-time streaming applications as well.  
The Kafka documentation says it is a distributed streaming platform. That's good for definition. But I want to know what it can do for me or what I can do using Kafka.

**Messaging System**

The official documentation says that Apache Kafka is similar to enterprise messaging system. I guess, you already understand a messaging system. In a typical messaging system, there are three components.

Producer or Publisher

Broker

Consumer.

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-5x/typical-messaging-system.jpg)Fig.1-Typical Messaging System

The producers are the client applications, and they send some messages.  
The Brokers receive those messages from publishers and store them.

The consumers read the message records from brokers.

**Kafka Use Case**

A messaging system looks very simple. Now let us look at the data integration problem in a large organization. I borrowed the below diagram from Jey Creps blog.

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-5x/data-integration-problem.jpg)Fig.1-Data Integration Problem

The above diagram shows the data integration requirement in a large enterprise.  
Does it look like a mess?  
There are many source systems and multiple destination systems. And you are given a task to create data pipelines to move data among those systems. For a growing company, the number of source and destination systems keep getting bigger and bigger. Finally, your data pipeline looks like a mess. I am sure that I don't need to explain that you can't manage and maintain that kind of data pipeline. Some part of your pipeline will keep breaking every day.  
However, if we can use a messaging system for solving that kind of integration problem, the solution may be neater, and cleaner as shown below.

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-5x/unified-messaging-system.jpg)Fig.1-Unified Messaging System

That's the idea discovered by the team at LinkedIn. Then they started evaluating existing messaging systems, but none of them meet their criteria to support the desired throughput and scale. Finally, they end up creating Kafka.

**What is Kafka?**

At the core, Kafka is a highly scalable and fault tolerant enterprise messaging system. Take a look at the [Apache Kafka diagram](https://kafka.apache.org/intro.html#_blank) from official documentation. I hope you understand the producer, consumer and the broker that the figure shows. At the top of the diagram, the Producer applications are sending messages to Kafka cluster. The Kafka cluster is nothing but a bunch of brokers running in a group of computers. They take message records from producers and store it in Kafka message log.  
At the bottom of the picture, there are consumer applications. They read messages from Kafka cluster, processes it and do whatever they want to do. They may want to send them to Hadoop, Cassandra, HBase or may be pushing it back again into Kafka for someone else to read these modified or transformed records.

**Kafka Streams**

**What is a stream?**  
Well, I will say continuous flow of data. or you can define it as a constant stream of messages.  
Kafka, as a messaging system is so powerful regarding throughput and scalability that it allows you to handle a continuous stream of messages. If you can just plug in some stream processing framework to Kafka, it could be your backbone infrastructure to create a real-time stream processing application. And that is what right side of the diagram is trying to explain. Those are some stream processing applications. They read a continuous stream of data from Kafka, process them and then either store them back in Kafka or send them directly to other systems. Kafka provides some stream processing APIs as well. So you can do a lot of things using Kafka stream processing APIs, or you can use other stream processing frameworks like Spark streaming or Storm.

**Kafka Connect**

The next thing is Kafka connector. These are the most compelling features. They are ready to use connectors to import data from databases into Kafka or export data from Kafka to databases. These are not just out of the box connectors but also a framework to build specialized connectors for any other application.

Summary

**Let us summarize all that we learned in this session.**

Kafka is a distributed streaming platform. You can use it as an enterprise messaging system. That doesn't mean just a traditional messaging system. You can use it to simplify complex data pipelines that are made up of a vast number of consumers and producers.

You can use it as a stream processing platform. There are two parts of stream processing. Stream and a Processing framework. Kafka gives you a stream, and you can plug in a processing framework.

Kafka also provides connectors to export and import bulk data from databases and other systems.

But implementing these things is not that simple. There is no plug and play component. You need to use APIs and write a bunch of code. You need to understand some configuration parameters and tune or customize Kafka behavior according to your requirement and use case.  
We will cover all these things in this training. So, keep watching.

**basic concepts associated with Kafka**

The objective of this article is to introduce you to the main terminologies and build a foundation to understand and grasp rest of the training.

Producer

Consumer

Broker

Cluster

Topic

Partitions

Offset

zookeeper

**Consumer Groups**

We will be using these terms extensively during our discussion of Apache Kafka. It is crucial that we both, myself and you have the same understanding of these concepts. So let me explain my understanding of these terms and some other related concepts associated with these keywords.

**Kafka Producer**

The first item is the producer. So, what is a producer?  
The producer is an application that sends data. Some people call it data, but we will call it a message or a message record. These messages can be anything ranging from a simple string to a complex object. Ultimately it is small to the medium-size piece of data. The message may have different meaning or schema for us. But for Kafka, it is a simple array of bytes.  
For example, if I want to send a file to Kafka, I will create a producer application and push each line of the file as a message. In this case, a message is one line of text. But for Kafka, it is just an array of bytes. Similarly, If I want to send all the records from a table, I will submit each row as a message, or if I want to send the result of a query. I will create a producer application, fire a query against my database, collect the result and start throwing each row as a message. So, while working with Kafka, if you want to send some data, you have to create a producer application. It is unlikely that you get a readymade producer that fits your purpose.

**Kafka Consumer**

The next thing is the consumer. The consumer is again an application that receives data. If producers are sending data, they must be sending it to someone. Right? The consumers are the recipients. But remember that the producers don't send data to a recipient address. They just send it to Kafka server. And anyone who is interested in that data can come forward and take it from Kafka server. So, any application that requests data from a Kafka server is a consumer, and they can ask for data send by any producer provided they have permissions to read it.  
So just continuing the file example, If I want to read the file sent by a producer, I will create a consumer application, then I will request Kafka for the data. The Kafka server will send me some messages. I think you remember that each message is a line of text in this example.  
So, the client application will receive some lines from Kafka server, it will process them and again request for some more messages. The client keeps demanding data, and Kafka server will keep giving message records as long as new messages are coming from the producer.

**Kafka Broker**

Now, let’s move on and try to understand a Broker. The broker is Kafka server. It is just a meaningful name given to Kafka server. And this title makes sense as well because all that Kafka does is act as a message broker between producer and consumer. The producer and consumer don not interact directly. They use Kafka server as an agent or a broker to exchange messages.

**Kafka Cluster**

Let's come to the next term. The cluster. This one is simple. If you have any background in distributed systems, you already know that a Cluster is a group of computers acting together for a common purpose. Since Kafka is a distributed system, so the cluster has the same meaning for Kafka. It is merely a group of computers, each executing one instance of Kafka broker.

**Kafka Topic**

Next item is the topic. We learned that producer sends data to Kafka broker. Then a consumer can ask for data from the Kafka broker. But the question is, Which data?  
Let's try to understand this by a simple conversation between Broker and the consumer.  
  
**Broker -** I am collecting data from multiple producers, which one do you want?  
**Consumer** - Give the data sent by producer ABC.  
**Broker -** Oh Man, producer ABC is pushing three different types of records. Which one do you want?  
**Consumer -** Well, send me the sales data.  
**Broker -** Ok, so you are looking for sales data. Two more producers are sending sales data.  
**Consumer -** Gosh, we need to have some identification mechanism.  
  
There comes the notion of the Kafka Topic. So, the topic is an arbitrary name given to a data set. We better say that it is a unique name for a data stream.  
For example, we create a topic called Global Orders, and every point of sales may have a producer. They send their order details as a message to the single Topic named Global Orders. And a subscriber interested in Orders can subscribe to the same Topic.

**Kafka Partitions**

By now, you learned that the broker would store data for a topic. This data can be enormous. It may be larger than the storage capacity of a single computer. In that case, the broker may have a challenge in storing that data. One of the obvious solutions is to break it into two or more parts and distribute it to multiple computers. Kafka is a distributed system that runs on a cluster of machines. So, it is self-evident that Kafka can break a topic into partitions and store one partition on one computer. And that's what the Partition means.  
You may be wondering that how Kafka will decide on the number of partitions. I mean, some topics may be large, but others may be relatively small. So how Kafka knows that it should create 100 partitions or just ten partitions could be enough?  
The answer is simple. Kafka doesn't take that decision. We, as a developer make that decision. When we create a topic, we make that decision, and Kafka broker will create that many partitions for our Topic. But remember that every Partition sits on a single machine. You can't break it again. So, do some estimation and simple math to calculate the number of partitions.

**Offsets**

Let's talk about offset. The offset is simple. It is a sequence number of a message in a partition. This number is assigned as the messages arrive in a partition. And these numbers, once assigned, they never change. They are immutable. This sequencing means that Kafka stores messages in the order of arrival within a partition. The first message gets an offset zero. The next message receives an offset one and so on. But remember that there is no global offset across partitions. Offsets are local to the partition. So, if you want to locate a message, you should know three things.  
Topic name, Partition number, and an offset number. If you have these three things, you can directly locate a message.

**Kafka Consumer Groups**

Now we are left with the last thing. The consumer groups. We already understand the Consumer. What is Consumer Group?  
It is a group of consumers. Several Consumers form a group to share the work. You can think of it like there is one large task and you want to divide it among multiple people, so you create a group, and members of the same group share the work. Let me give you an example.

A Kafka Example

Let's assume that we have a retail chain. In every store, there are few billing counters. You want to bring all the invoices from every billing counter to your data centre. Since you learned Kafka and you find Kafka as an excellent solution to transport data from billing locations to the data centre. You decided to implement it. The first thing you might want to do is to create a producer at every billing site. These Producers will send bills as a message to a Kafka Topic. The next thing you might want to do is to create a consumer. The Consumer will read data from Kafka Topic and write them into your data centre. It sounds like a perfect solution. Right? But there is a small problem. Think of the scale. You have hundreds of producers pushing data into a single topic. How will you handle that volume and velocity?

[Diagram

Description automatically generated](https://www.learningjournal.guru/_resources/img/jpg-7x/apache-kafka-example.jpg)Fig.1-Apache Kafka Example Use Case

You learned Kafka exceptionally well. So, you decided to create large Kafka cluster and partition your Topic. Correct? So, your Topic is partitioned and distributed across the Cluster. Now several brokers are sharing the workload to receive and store data. From the source side, you have many producer and several Brokers to share the workload. What about the destination side? You have a single unfortunate consumer.  
There comes the Consumer group. You create a Consumer group and start executing many Consumers and tell them to divide the work.  
So far so good. But how do we split the work? That's not a difficult question. I have 600 partitions. And I am starting 100 consumers. So why don't each of the consumer take six partitions? We will see, if they can't handle six partitions, we will start some more Consumers in the same group. We can go up to 600 Consumers, so each consumer will have just one partition to read.  
If you followed this example correctly, you understand that partitioning and consumer group is a tool for scalability. And notice that the maximum number of Consumers in a group is the total number of partitions you have on a topic. Kafka doesn't allow more than one Consumer to read from the same partition simultaneously. This restriction is necessary to avoid double reading of records.  
Great. I hope you learned core concepts of Kafka. Now you are familiar with the essential terminology that we will be using throughout the Kafka tutorials.

kafka interview questions and answers for experienced developer

**ChatGPT said:**

ChatGPT

Here’s a list of common Kafka interview questions tailored for an experienced developer, along with concise answers:

**General Questions**

1. **What is Apache Kafka?**
   * **Answer:** Kafka is a distributed streaming platform designed for high-throughput, fault-tolerant, and real-time data processing. It allows the publishing, subscribing to, storing, and processing of streams of records in a scalable manner.
2. **What are the key components of Kafka?**
   * **Answer:** The key components are:
     + **Producers:** Applications that publish messages to Kafka topics.
     + **Consumers:** Applications that subscribe to topics and process the messages.
     + **Brokers:** Kafka servers that store messages and serve client requests.
     + **Topics:** Categories to which records are published.
     + **Partitions:** Subdivisions of topics for parallel processing.
3. **What is a Kafka broker?**
   * **Answer:** A Kafka broker is a server that stores data and serves client requests. Each broker can handle hundreds of thousands of reads and writes per second, providing fault tolerance and scalability.
4. **What is the role of Zookeeper in Kafka?**
   * **Answer:** Zookeeper is used to manage the distributed brokers in Kafka. It helps in leader election, configuration management, and maintaining metadata about brokers, topics, and partitions.

**Producer and Consumer Questions**

1. **How does Kafka ensure message durability?**
   * **Answer:** Kafka ensures durability by writing messages to disk and replicating them across multiple brokers. The replication factor determines how many copies of a message are stored across brokers.
2. **What is the difference between at-most-once, at-least-once, and exactly-once delivery semantics in Kafka?**
   * **Answer:**
     + **At-most-once:** Messages may be lost but are never duplicated.
     + **At-least-once:** Messages are not lost but may be duplicated.
     + **Exactly-once:** Each message is delivered once and only once, achieved through idempotent producers and transactional consumers.
3. **What is a consumer group in Kafka?**
   * **Answer:** A consumer group is a group of consumers that work together to consume messages from topics. Each message is delivered to one consumer in the group, allowing for parallel processing of messages.

**Configuration and Performance Questions**

1. **What are some important producer configuration properties in Kafka?**
   * **Answer:** Important configurations include:
     + acks: Determines the number of acknowledgments required (0, 1, all).
     + retries: Number of retries for sending messages.
     + linger.ms: Time to wait before sending a batch of messages.
2. **How can you improve Kafka's performance?**
   * **Answer:** Performance can be improved by:
     + Increasing the number of partitions for parallelism.
     + Tuning batch sizes and linger time for producers.
     + Optimizing consumer fetch sizes.
     + Using appropriate compression types.

**Advanced Topics**

1. **What is the purpose of Kafka Streams?**
   * **Answer:** Kafka Streams is a client library for building real-time applications and microservices that process data in Kafka. It allows for stateful and stateless processing, windowing, and integration with other data sources.
2. **Can you explain the concept of "log compaction"?**
   * **Answer:** Log compaction is a Kafka feature that allows for the retention of the latest value for each key in a topic. It ensures that even if older messages are deleted, the latest state for each key is preserved.
3. **What is the difference between Kafka and traditional messaging systems?**
   * **Answer:** Kafka differs from traditional messaging systems in that it stores messages on disk for durability, allows for high throughput, supports distributed processing, and enables the reprocessing of historical data due to its log-based architecture.

**Troubleshooting and Monitoring**

1. **How do you monitor Kafka?**
   * **Answer:** Monitoring can be done using tools like Prometheus and Grafana, along with Kafka's JMX metrics. Key metrics to monitor include broker performance, consumer lag, throughput, and partition distribution.
2. **What steps would you take if a consumer is lagging behind?**
   * **Answer:**
     + Check consumer group metrics to understand the lag.
     + Increase the number of consumer instances or partitions.
     + Optimize the consumer's processing logic or configuration (e.g., increase fetch.size).
     + Review network and disk I/O performance.
3. **How do you handle schema evolution in Kafka?**
   * **Answer:** Schema evolution can be managed using a schema registry like Confluent Schema Registry, which supports versioning and allows for backward and forward compatibility. Avro or Protobuf can be used for serialization to handle schema changes effectively.

# **KafkaProducer and KafkaConsumer Example**

Here's a basic example of how to set up a **KafkaProducer** and **KafkaConsumer** in a Spring Boot application. This example will include the configuration for both producer and consumer, as well as the code to send and receive messages.

**1. Dependencies**

Make sure to add the following dependencies in your pom.xml if you’re using Maven:

xml

Copy code

<dependency>

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

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

</dependency>

<dependency>

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

<artifactId>spring-kafka</artifactId>

</dependency>

Or in build.gradle for Gradle:

gradle

Copy code

implementation 'org.springframework.boot:spring-boot-starter'

implementation 'org.springframework.kafka:spring-kafka'

**2. Kafka Configuration**

Create a configuration class to set up the Kafka producer and consumer properties.

java

Copy code

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

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

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

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

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import org.springframework.kafka.annotation.EnableKafka;

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

import java.util.HashMap;

import java.util.Map;

@EnableKafka

@Configuration

public class KafkaConfig {

private static final String BOOTSTRAP\_SERVERS = "localhost:9092";

// Producer Configuration

@Bean

public ProducerFactory<String, String> producerFactory() {

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

configProps.put(ProducerConfig.BOOTSTRAP\_SERVERS\_CONFIG, BOOTSTRAP\_SERVERS);

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

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

return new DefaultKafkaProducerFactory<>(configProps);

}

@Bean

public KafkaTemplate<String, String> kafkaTemplate() {

return new KafkaTemplate<>(producerFactory());

}

// Consumer Configuration

@Bean

public ConsumerFactory<String, String> consumerFactory() {

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

configProps.put(ConsumerConfig.BOOTSTRAP\_SERVERS\_CONFIG, BOOTSTRAP\_SERVERS);

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;

}

}

**3. Kafka Producer**

Create a service to send messages to a Kafka topic.

java

Copy code

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

import org.springframework.kafka.core.KafkaTemplate;

import org.springframework.stereotype.Service;

@Service

public class KafkaProducerService {

private static final String TOPIC\_NAME = "example-topic";

@Autowired

private KafkaTemplate<String, String> kafkaTemplate;

public void sendMessage(String message) {

kafkaTemplate.send(TOPIC\_NAME, message);

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

}

}

**4. Kafka Consumer**

Create a consumer service that listens to messages on the specified Kafka topic.

java

Copy code

import org.springframework.kafka.annotation.KafkaListener;

import org.springframework.stereotype.Service;

@Service

public class KafkaConsumerService {

@KafkaListener(topics = "example-topic", groupId = "group\_id")

public void consumeMessage(String message) {

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

}

}

**5. REST Controller to Test Kafka Producer**

To trigger the producer, create a simple REST controller that calls the KafkaProducerService.

java

Copy code

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

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

@RestController

@RequestMapping("/api/kafka")

public class KafkaController {

@Autowired

private KafkaProducerService kafkaProducerService;

@PostMapping("/publish")

public String publishMessage(@RequestParam("message") String message) {

kafkaProducerService.sendMessage(message);

return "Message published successfully";

}

}

**6. Run the Application**

1. Start your Kafka server on localhost:9092.
2. Run your Spring Boot application.
3. Use a tool like **Postman** or a browser to send a POST request to http://localhost:8080/api/kafka/publish?message=HelloKafka.

Example request:

bash

Copy code

POST /api/kafka/publish?message=HelloKafka

**Explanation of Components**

* **KafkaProducerService**: Sends messages to the specified Kafka topic using the KafkaTemplate.
* **KafkaConsumerService**: Listens to the specified Kafka topic and consumes messages, printing them to the console.
* **KafkaController**: Provides a REST endpoint to publish messages to Kafka, making it easy to test the producer functionality.

# **What is the role of zookeper in kafka**

In Apache Kafka, **ZooKeeper** serves as a centralized service that manages and coordinates Kafka brokers, ensuring smooth communication and fault tolerance within the Kafka cluster. However, starting with Kafka 2.8, Kafka has introduced a **KRaft mode** that enables it to operate without ZooKeeper, making ZooKeeper optional in newer Kafka setups. But in traditional Kafka architecture (prior to KRaft mode), ZooKeeper plays a crucial role in several ways:

**Key Roles of ZooKeeper in Kafka**

1. **Broker Management and Coordination**:
   * ZooKeeper keeps track of Kafka brokers and their statuses within the Kafka cluster. When a new broker starts or a broker goes offline, ZooKeeper updates its metadata about available brokers, helping the Kafka Controller (one of the brokers) to rebalance partitions and manage resources accordingly.
2. **Leader Election**:
   * In Kafka, each topic partition needs a leader broker to handle reads and writes. ZooKeeper manages the election process for partition leaders, ensuring that only one broker is the leader for a given partition at any time. If a leader broker goes down, ZooKeeper elects a new leader, promoting high availability and fault tolerance.
3. **Managing Metadata**:
   * Kafka relies on ZooKeeper to store metadata about topics, partitions, and configurations. This includes information about which brokers are assigned to which partitions, partition replicas, and the configuration of each broker.
4. **Producer and Consumer Coordination**:
   * ZooKeeper helps coordinate producers and consumers. For instance, it manages information about which consumer groups are consuming which partitions, and helps with consumer rebalancing whenever consumers join or leave a consumer group.
5. **Access Control and Quotas**:
   * Kafka uses ZooKeeper to store access control lists (ACLs) and quotas for various users, restricting access to topics and enforcing rate limits on API usage. This is part of the overall security and resource management framework of Kafka.
6. **Controller Election**:
   * ZooKeeper helps elect a single broker as the Kafka Controller, which is responsible for managing partition leadership and cluster metadata. If the controller broker goes down, ZooKeeper elects another broker as the controller to maintain cluster health and functionality.

**With KRaft Mode (Kafka Raft Metadata Mode)**

Since Kafka 2.8, Kafka has been evolving to become independent of ZooKeeper, using **KRaft** mode (Kafka Raft) to manage its metadata natively. In KRaft mode:

* Kafka stores metadata within the Kafka cluster itself, eliminating the need for ZooKeeper.
* Raft consensus algorithms replace ZooKeeper's role in leader election, metadata management, and broker coordination.