Apache Kafka Overview

1. What is Apache Kafka, and what are its main components?
2. Explain Kafka’s architecture briefly.
3. What is a Kafka broker?
4. What is a topic in Kafka?kindly try to answer all the given questions in details along with example in Java

1. What is Apache Kafka, and what are its main components?

Apache Kafka Overview

Apache Kafka is a distributed event-streaming platform designed for handling real-time data feeds. It provides a unified, high-throughput, low-latency platform to handle data in real-time. It is widely used for building real-time analytics pipelines and streaming applications.

Kafka is primarily used for:

- **Message Queueing**: Reliable message delivery between producers and consumers.
- **Event Streaming**: Processing and analyzing streams of data.
- **Integration**: As a distributed commit log for data exchange.

Main Components

1. **Producer**: Sends messages (records) to Kafka topics.
2. **Consumer**: Reads messages from Kafka topics.
3. **Topic**: A logical channel to which producers publish records and consumers subscribe.
4. **Broker**: Kafka server that stores and serves messages.
5. **Zookeeper/Controller**: Handles metadata and coordinates the Kafka cluster (modern versions use KRaft instead of ZooKeeper).
6. **Partitions**: Each topic is divided into partitions for parallelism and scalability.
7. **Offsets**: Unique identifiers for each message within a partition.

2. Explain Kafka’s architecture briefly

Kafka follows a **distributed publish-subscribe messaging system architecture**. Here’s how it works:

1. **Topics and Partitions**:
    - Topics are split into partitions for scalability.
    - Each partition is replicated across multiple brokers to ensure fault tolerance.
2. **Producers**:
    - Producers send data to a specific topic.
    - Producers can specify keys for partition routing.
3. **Brokers**:
    - Kafka brokers are responsible for storing and serving messages.
    - Each partition is assigned a leader, and all writes/read operations go through this leader.
4. **Consumers**:
    - Consumers subscribe to topics and read data in order from partitions.
    - Kafka uses consumer groups for parallel data consumption.
5. **Replication**:
    - Kafka ensures data reliability by replicating partitions across multiple brokers.
6. **Offset Management**:
    - Kafka tracks the position of consumers in the log using offsets.
    - Consumers can control offsets to re-read data or skip data.

---

3. What is a Kafka broker?

A Kafka broker is a server that:

- Stores data (topic partitions).
- Serves data to consumers and accepts data from producers.
- Handles replication of data for fault tolerance.
- Works in a cluster, where each broker has a unique ID.

In a Kafka cluster:

- Each broker can handle multiple topics and partitions.
- Load balancing and fault tolerance are managed through leader election and replication.

Example: If you have a cluster with 3 brokers and a topic with 6 partitions, Kafka will distribute these partitions among the brokers.

---

4. What is a topic in Kafka?

A **Kafka topic** is a category or feed name to which records are sent by producers. It's essentially a log for storing messages.

Key Features:

- Topics are divided into **partitions** for scalability.
- Data in a topic is persistent, and consumers can re-read data as needed.
- Data is identified within a partition by a unique **offset**.

Example in Java:

```java
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.clients.consumer.ConsumerRecords;
import org.apache.kafka.clients.consumer.ConsumerRecord;

import java.util.Properties;
import java.util.Collections;

public class KafkaExample {
    public static void main(String[] args) {
        String topic = "my-topic";

        // Producer
        Properties producerProps = new Properties();
        producerProps.put("bootstrap.servers", "localhost:9092");
        producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
        producer.send(new ProducerRecord<>(topic, "key1", "Hello Kafka!"));
        producer.close();

        // Consumer
        Properties consumerProps = new Properties();
        consumerProps.put("bootstrap.servers", "localhost:9092");
        consumerProps.put("group.id", "my-group");
        consumerProps.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
        consumerProps.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

        KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
        consumer.subscribe(Collections.singletonList(topic));

        ConsumerRecords<String, String> records = consumer.poll(1000);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("Offset: %d, Key: %s, Value: %s%n", record.offset(), record.key(), record.value());
        }
        consumer.close();
    }
}
```

Explanation of the Example

1. **Producer**:
   - Sends a record to the topic `my-topic`.
   - The key `key1` determines the partition.
2. **Consumer**:
   - Subscribes to `my-topic`.
   - Polls for records and processes them, printing the offset, key, and value.

This code demonstrates how Kafka's core components work together in a Java application.

5. What is the purpose of partitions in Kafka?
6. Explain the role of ZooKeeper in Kafka.
7. What are producers and consumers in Kafka?
8. Describe a Kafka cluster.kindly try to answer all the given questions in details along with example in Java

5. What is the purpose of partitions in Kafka?

Purpose of Partitions:

1. **Scalability**:
   - Partitions allow Kafka to parallelize data processing.
   - A topic can be divided into multiple partitions, distributed across brokers in the cluster.
2. **Fault Tolerance**:
   - Each partition is replicated across multiple brokers.
   - If a broker fails, another replica can take over.
3. **Data Ordering**:
   - Within a partition, Kafka guarantees data is read in the same order it was written.
4. **Load Balancing**:
   - By distributing partitions across brokers, Kafka ensures that the workload is balanced.

Example in Java:

```java
// Producer sending data to a specific partition
KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
ProducerRecord<String, String> record = new ProducerRecord<>("my-topic", 1, "key1", "Message for Partition 1");
producer.send(record);
```

In this example:

- The message is sent to partition `1` of the topic `my-topic`.

---

6. Explain the role of ZooKeeper in Kafka

ZooKeeper plays a critical role in Kafka's architecture (before KRaft was introduced):

Roles of ZooKeeper:

1. **Metadata Management**:
    - Stores metadata about Kafka brokers, topics, and partitions.
2. **Leader Election**:
    - Handles the leader election for topic partitions in the cluster.
3. **Configuration Management**:
    - Maintains cluster configuration for brokers and topics.
4. **Health Monitoring**:
    - Monitors the health of brokers and partitions.
5. **Fault Recovery**:
    - Helps recover from broker failures by reassigning partition leadership.

Transition from ZooKeeper to KRaft:

- Kafka now supports **KRaft (Kafka Raft)**, which eliminates the need for ZooKeeper, integrating metadata management directly within Kafka.

---

7. What are producers and consumers in Kafka?

Producers:

- Producers are applications that publish messages to Kafka topics.
- They decide which partition a message should be sent to, either by specifying a key or allowing Kafka to distribute messages.

Consumers:

- Consumers are applications that subscribe to Kafka topics to read messages.
- Kafka supports **consumer groups** for parallel consumption of partitions.

Example in Java:

```java
// Producer Example
KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("my-topic", "key1", "Hello from Producer"));
producer.close();

// Consumer Example
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("my-topic"));

ConsumerRecords<String, String> records = consumer.poll(1000);
for (ConsumerRecord<String, String> record : records) {
    System.out.printf("Consumed message: %s from partition: %d%n", record.value(), record.partition());
}
consumer.close();
```

---

8. Describe a Kafka Cluster

A **Kafka cluster** is a distributed system consisting of multiple brokers working together.

Key Components of a Kafka Cluster:

1. **Brokers**:
    - Kafka servers responsible for storing and serving data.
    - Each broker has a unique ID and can host multiple partitions.
2. **Topics and Partitions**:
    - Topics are divided into partitions.
    - Partitions are distributed across brokers for scalability.
3. **Replication**:
    - Each partition has replicas across brokers for fault tolerance.
4. **ZooKeeper/KRaft**:
    - Used for metadata management and leader election.

Cluster Features:

- **Fault Tolerance**: Redundancy is achieved through partition replication.
- **Scalability**: More brokers can be added to handle increased workloads.

Example in Java:

Simulating interaction with a Kafka cluster:

```java
// Example with multiple brokers
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "broker1:9092,broker2:9092,broker3:9092");
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("cluster-topic", "key1", "Message to Kafka Cluster"));
producer.close();
```

In this example:

- The producer connects to multiple brokers (`` `broker1` ``, `` `broker2` ``, `` `broker3` ``) in the Kafka cluster.
- Kafka handles partition and replica assignment internally.

By leveraging these features, Kafka ensures high availability, fault tolerance, and scalability in a distributed environment.

9. What are consumer groups?
10. How does Kafka ensure message ordering?
11. What is an offset in Kafka?
12. How does Kafka handle data retention?kindly try to answer all the given questions in details along with example in Java

9. What are Consumer Groups?

Consumer Groups in Kafka

A **consumer group** is a collection of consumer instances that work together to consume data from a Kafka topic. The members of a consumer group divide the partitions of a topic among themselves for parallel consumption.

Key Features:

1. **Parallel Processing**:
    - Each partition is consumed by one consumer within a group, ensuring that messages are not processed multiple times.
    - If a group has more consumers than partitions, some consumers will remain idle.
2. **Scalability**:
    - Adding more consumers to a group allows parallel processing of messages across partitions.
3. **Fault Tolerance**:
    - If a consumer in the group fails, its partitions are reassigned to other consumers in the group.

Example in Java:

```java
// Consumer Group Example
Properties consumerProps = new Properties();
consumerProps.put("bootstrap.servers", "localhost:9092");
consumerProps.put("group.id", "my-consumer-group");
consumerProps.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
consumerProps.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("my-topic"));

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("Consumer Group: %s, Partition: %d, Offset: %d, Value: %s%n",
                          "my-consumer-group", record.partition(), record.offset(), record.value());
    }
}
```

10. How does Kafka ensure message ordering?

Message Ordering in Kafka

1. **Within a Partition**:
    - Kafka ensures that messages are written and read in the same order within a partition.
    - A producer can use keys to ensure that related messages are sent to the same partition.
2. **Across Partitions**:
    - Kafka does not guarantee ordering across partitions.

Example in Java:

```java
// Producer ensuring message ordering within a partition using a key
KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);

// All messages with the same key go to the same partition
producer.send(new ProducerRecord<>("my-topic", "key1", "Message 1"));
producer.send(new ProducerRecord<>("my-topic", "key1", "Message 2"));
producer.close();
```

In this example:

- Messages with `` `key1` `` will be sent to the same partition and processed in order.

---

11. What is an Offset in Kafka?

Offset in Kafka

- The **offset** is a unique identifier assigned to each message within a partition.
- It represents the position of the message in the partition log.
- Consumers use offsets to track which messages have been read.

Key Features:

1. **Per Partition**:
    - Offsets are unique within a partition but not across partitions.
2. **Manual or Automatic Offset Management**:
    - Kafka allows consumers to manage offsets manually or automatically.

Example in Java:

```java
// Reading and printing offsets
ConsumerRecords<String, String> records = consumer.poll(1000);
for (ConsumerRecord<String, String> record : records) {
    System.out.printf("Partition: %d, Offset: %d, Key: %s, Value: %s%n",
                      record.partition(), record.offset(), record.key(), record.value());
}

// Commit the offset manually
consumer.commitSync();
```

---

12. How does Kafka handle data retention?

Data Retention in Kafka

- Kafka retains messages based on:
    1. **Retention Time**: Configured using the `log.retention.hours` setting.
    2. **Log Size**: Configured using `log.retention.bytes`.
- Messages are stored in the log even after being consumed and are deleted only after the retention period or size limit is exceeded.

Key Features:

1. **Immutable Log**:
    - Kafka stores messages as an immutable log.

   - Consumers can re-read messages by specifying the offset.
2. **Retention Policies**:
   - Time-based: Messages older than a certain time are deleted.
   - Size-based: Messages are deleted when the log size exceeds a threshold.
3. **Compaction**:
   - Retains only the latest value for each key, based on `` `log.cleanup.policy` ``.

Example in Java:

```java
// Producer sending data
KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("retention-topic", "key1", "Value 1"));
producer.send(new ProducerRecord<>("retention-topic", "key2", "Value 2"));
producer.close();

// Consumer reading data before retention
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("retention-topic"));

ConsumerRecords<String, String> records = consumer.poll(1000);
for (ConsumerRecord<String, String> record : records) {
    System.out.printf("Offset: %d, Key: %s, Value: %s%n", record.offset(), record.key(), record.value());
}
```

In this example:

- The producer sends messages to `` `retention-topic` ``.
- The consumer reads messages before they are removed by the retention policy.

These detailed explanations and Java examples demonstrate how Kafka manages key functionalities for message processing in a distributed, fault-tolerant, and scalable way.

13. Explain "at least once" and "exactly once" delivery in Kafka.
14. What is Kafka Streams?
15. How does Kafka ensure fault tolerance?
16. What is log compaction?kindly try to answer all the given questions in details along with example in Java

13. Explain "At Least Once" and "Exactly Once" Delivery in Kafka

At Least Once Delivery

- Kafka ensures that a message is delivered to a consumer at least once.
- This guarantees reliability but can lead to duplicate messages if the consumer processes the same message multiple times (e.g., due to retries).

Example:

- A consumer reads a message but fails to commit the offset. Kafka may redeliver the message.

Exactly Once Delivery

- Kafka ensures that a message is delivered exactly once to a consumer.
- This is achieved using Kafka’s **idempotent producers** and **transactions**, which prevent duplicates even if retries occur.

Key Features:

1. **Idempotent Producers**:
    - Ensure that the same message sent multiple times is written only once to the log.
2. **Transactions**:
    - Group multiple operations (e.g., producing to and consuming from topics) into a single atomic unit.

Example in Java:

```java
// Producer with Exactly Once Semantics
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("enable.idempotence", "true");
producerProps.put("transactional.id", "my-transaction-id");
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.initTransactions();

try {
    producer.beginTransaction();
    producer.send(new ProducerRecord<>("my-topic", "key1", "Message with Exactly Once"));
    producer.commitTransaction();
} catch (Exception e) {
    producer.abortTransaction();
}
```

---

14. What is Kafka Streams?

Kafka Streams

- Kafka Streams is a lightweight library for building stream-processing applications.
- It processes real-time data from Kafka topics and produces results back to Kafka topics.

Key Features:

1. **Distributed and Fault-Tolerant**:
    - Processes data in a distributed manner across multiple nodes.
2. **Stateful Processing**:
    - Supports operations like aggregations, joins, and windowing.
3. **Exactly Once Semantics**:
    - Ensures reliable processing of messages.

Example in Java:

```java
// Kafka Streams Example
Properties streamsProps = new Properties();
streamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG, "streams-app");
streamsProps.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
streamsProps.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
streamsProps.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> inputStream = builder.stream("input-topic");
KStream<String, String> outputStream = inputStream.mapValues(value -> value.toUpperCase());
outputStream.to("output-topic");

KafkaStreams streams = new KafkaStreams(builder.build(), streamsProps);
streams.start();
```

This example:

- Reads messages from `input-topic`.
- Converts message values to uppercase.
- Writes the results to `output-topic`.

---

15. How Does Kafka Ensure Fault Tolerance?

Key Features Ensuring Fault Tolerance:

1. **Replication**:
    - Each partition is replicated across multiple brokers.
    - One broker acts as the leader; others are followers.
2. **Leader Election**:
    - If a broker fails, a follower becomes the new leader.
3. **Acknowledgments**:
    - Producers can specify acknowledgment levels (`acks`):
        - `acks=0`: Fire-and-forget.
        - `acks=1`: Acknowledgment from the leader.

        - `` `acks=all` ``: Acknowledgment from all replicas.
4. **Durability**:
    - Data is written to disk and can survive broker crashes.

Example in Java:

```java
// Producer with Acknowledgments
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("acks", "all");
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("my-topic", "key1", "Fault Tolerant Message"));
producer.close();
```

---

16. What is Log Compaction?

Log Compaction in Kafka

- Kafka uses **log compaction** to retain the latest value for each unique key in a topic.
- It ensures that Kafka only deletes old messages for keys if there’s a newer message for the same key.

Use Case:

- Storing the latest state for a key (e.g., user preferences, account balances).

How It Works:

1. Compaction is triggered based on Kafka’s configuration.
2. Old records with no updates for the same key are retained temporarily but marked for deletion.

Key Configurations:

- `log.cleanup.policy=compact`: Enables log compaction.
- `log.retention.bytes`: Specifies retention size.

Example in Java:

```java
// Producer with Log Compaction
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);

// Sending messages with the same key
producer.send(new ProducerRecord<>("compact-topic", "key1", "Old Value"));
producer.send(new ProducerRecord<>("compact-topic", "key1", "Latest Value"));
producer.close();
```

In this example:

- Kafka retains only the latest message for `key1` in the `compact-topic` due to log compaction.

These answers and Java examples illustrate Kafka’s capabilities for ensuring delivery guarantees, fault tolerance, stream processing, and efficient data management.

16. What is log compaction?kindly try to answer all the given questions in details along with example in Java
17. Explain replication in Kafka.
18. What is an ISR (In-Sync Replica)?
19. How does leader election work in Kafka?
20. How do partitions help in Kafka scaling?kindly try to answer all the given questions in details along with example in Java

16. What is Log Compaction?

Log Compaction in Kafka

Log compaction ensures that Kafka retains only the most recent value for each key in a topic. It allows Kafka to retain a compacted log of data where only the latest record for each key is preserved.

Benefits:

1. **State Storage**: Ideal for use cases like maintaining the latest state (e.g., user preferences, sensor readings).
2. **Reduced Storage**: Eliminates older, unnecessary records for each key.
3. **Data Durability**: Guarantees that the latest updates are always available.

How It Works:

1. Kafka periodically compacts logs based on the configuration `` `log.cleanup.policy=compact` ``.
2. Messages with the same key are deduplicated, retaining only the latest version.

Example in Java:

```java
// Producer for Log Compaction
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);

// Sending multiple updates for the same key
producer.send(new ProducerRecord<>("compact-topic", "user1", "Initial Data"));
producer.send(new ProducerRecord<>("compact-topic", "user1", "Updated Data"));
producer.close();
```

In this example:

- Kafka retains only the message ``"user1 -> Updated Data"`` after compaction.

---

17. Explain Replication in Kafka

Replication in Kafka

Replication is a mechanism that ensures fault tolerance by duplicating partitions across multiple brokers in a Kafka cluster.

Key Features:

1. **Partition Leader**:
    - Each partition has one leader broker that handles all reads and writes.
2. **Replicas**:
    - Additional copies of the partition are stored on other brokers.
3. **Fault Tolerance**:
    - If a leader fails, one of the replicas becomes the new leader.

Example in Java:

Replication is handled automatically by Kafka, but you can configure replication in the topic creation step.

```java
// Creating a topic with replication factor
Properties adminProps = new Properties();
adminProps.put("bootstrap.servers", "localhost:9092");

AdminClient adminClient = AdminClient.create(adminProps);
NewTopic newTopic = new NewTopic("replicated-topic", 3, (short) 2); // 3 partitions, replication factor of 2
adminClient.createTopics(Collections.singleton(newTopic));
adminClient.close();
```

This example creates a topic with 3 partitions and a replication factor of 2.

---

18. What is an ISR (In-Sync Replica)?

ISR (In-Sync Replica)

The **ISR** is a list of replicas that are fully synchronized with the leader broker for a partition.

Key Points:

1. **Leader and ISR**:
    - The leader maintains the ISR list.
2. **Fault Tolerance**:
    - Only ISR members are eligible to become the new leader if the current leader fails.
3. **Synchronization**:
    - Replicas not caught up with the leader are removed from the ISR.

Example in Java:

ISR is handled internally by Kafka, but you can monitor ISR status using Kafka metrics.

---

19. How Does Leader Election Work in Kafka?

Leader Election in Kafka

Leader election determines which broker will act as the leader for a partition.

Process:

1. **Initial Assignment**:
    - During topic creation, Kafka assigns a leader broker for each partition.
2. **Failure Detection**:
    - If the leader fails, ZooKeeper or Kafka's internal Raft-based controller elects a new leader from the ISR.
3. **High Availability**:
    - Only replicas in the ISR can become leaders.

Example in Java:

You can monitor leader status using the Kafka Admin API.

```java
// Describe topic to see leader details
DescribeTopicsResult describeTopicsResult = adminClient.describeTopics(Collections.singletonList("replicated-topic"));
Map<String, TopicDescription> topicDescriptions = describeTopicsResult.all().get();
TopicDescription description = topicDescriptions.get("replicated-topic");
description.partitions().forEach(partition -> {
    System.out.printf("Partition: %d, Leader: %s%n", partition.partition(), partition.leader().id());
});
```

---

20. How Do Partitions Help in Kafka Scaling?

Role of Partitions in Scaling:

1. **Parallelism**:
   - Partitions allow multiple producers and consumers to work simultaneously, increasing throughput.
2. **Load Distribution**:
   - Partitions are distributed across brokers, ensuring no single broker is overwhelmed.
3. **Independent Consumption**:
   - Consumers in the same group can process partitions in parallel, improving processing speed.

Example:

- If a topic has 10 partitions and 3 consumers in a group, partitions will be divided among the consumers.

Example in Java:

```java
// Producer sending data to specific partitions
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
```

```
KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);

// Send data to specific partitions
producer.send(new ProducerRecord<>("scaling-topic", 0, "key1", "Message for Partition 0"));
producer.send(new ProducerRecord<>("scaling-topic", 1, "key2", "Message for Partition 1"));
producer.close();
```

This example:

- Demonstrates how producers can distribute messages across partitions, enabling parallel processing by consumers.

These detailed explanations with Java examples illustrate Kafka's capabilities in log management, fault tolerance, leader election, and scalability through partitions and replication.

21. What is Kafka Connect?
22. How does Kafka ensure message durability?
23. What’s Kafka’s role in stream processing?
24. How is Kafka different from traditional messaging systems?kindly try to answer all the given questions in details along with example in Java

21. What is Kafka Connect?

Kafka Connect

Kafka Connect is a tool included in Apache Kafka to **integrate Kafka with external systems** like databases, file systems, and other message queues. It provides a scalable, fault-tolerant, and distributed way to stream data between Kafka and other systems.

Key Features:

1. **Source Connectors**:
    - Bring data from external systems into Kafka topics (e.g., databases, logs).

2. **Sink Connectors**:
    - Write data from Kafka topics to external systems (e.g., NoSQL databases, file systems).
3. **Schema Registry**:
    - Ensures data consistency across systems using schemas.
4. **Scalability**:
    - Automatically distributes tasks across the Kafka Connect cluster.
5. **Configurable**:
    - Requires minimal coding; configurations are specified in JSON files.

Example:

Suppose you want to stream MySQL data into Kafka. You can use a **JDBC Source Connector**.

---

22. How Does Kafka Ensure Message Durability?

Message Durability in Kafka

1. **Replication**:
    - Kafka replicates each partition across multiple brokers.
    - If one broker fails, replicas ensure the data is not lost.
2. **Write to Disk**:
    - Messages are stored on disk (not memory), ensuring durability even after broker restarts.
3. **Acks (Acknowledgments)**:
    - Producers can set acknowledgment levels:
        - `` `acks=1` ``: Leader writes message to disk and sends acknowledgment.
        - `` `acks=all` ``: Leader waits for all replicas in the ISR to write the message before acknowledgment.
4. **Retention Policies**:
    - Kafka retains messages for a configured period (`` `log.retention.hours` ``) or until log size exceeds limits (`` `log.retention.bytes` ``).

Example in Java:

```java
// Producer ensuring durability
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("acks", "all"); // Ensures all replicas have written the message
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("durable-topic", "key1", "Durable Message"));
producer.close();
```

---

23. What's Kafka's Role in Stream Processing?

Kafka's Role in Stream Processing

Kafka serves as the backbone for building **real-time stream processing systems** by:

1. **Data Ingestion**:
    - Kafka acts as the primary data source for stream processing.
    - It captures and stores events from producers.
2. **Distributed Processing**:
    - Kafka Streams and other frameworks (e.g., Apache Flink, Spark Streaming) process data in real time.
3. **Scalability**:
    - Kafka's partitioning allows stream processing to scale horizontally.

Kafka Streams Library:

Kafka provides its own library, **Kafka Streams**, for processing data directly from Kafka topics and writing the results back to Kafka.

Example in Java:

```java
// Kafka Streams Example
Properties streamsProps = new Properties();
streamsProps.put(StreamsConfig.APPLICATION_ID_CONFIG, "stream-processing-app");
streamsProps.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
streamsProps.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
streamsProps.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> inputStream = builder.stream("input-topic");
KStream<String, String> outputStream = inputStream.mapValues(value -> "Processed: " + value);
outputStream.to("output-topic");

KafkaStreams streams = new KafkaStreams(builder.build(), streamsProps);
streams.start();
```

This example:

- Reads messages from `` `input-topic` ``, processes them, and writes results to `` `output-topic` ``.

---

24. How is Kafka Different from Traditional Messaging Systems?

Feature	Kafka	Traditional Messaging Systems
Storage	Messages are retained on disk even after being consumed.	Messages are deleted after being consumed.
Scalability	Horizontal scaling using partitions.	Limited by broker capacity.
Performance	High throughput, optimized for large-scale systems.	Typically lower throughput.
Durability	Messages are replicated and persisted.	Often relies on memory, with optional disk persistence.
Consumer Model	Consumers can replay messages using offsets.	Messages are usually consumed only once.
Message Delivery	Supports at-least-once, at-most-once, and exactly-once.	Delivery guarantees depend on the implementation.
Stream Processing	Kafka provides built-in support via Kafka Streams.	Limited or no native stream processing support.
Usage	Event streaming, real-time analytics, log aggregation.	Primarily point-to-point or publish/subscribe messaging.

Summary with Example in Java

Traditional Messaging Example:

- A message is sent from one producer to one consumer (e.g., RabbitMQ).

```java
// Kafka Example
KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("example-topic", "key1", "Event-Driven Message"));
producer.close();

// Consumer reading the message
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("example-topic"));
ConsumerRecords<String, String> records = consumer.poll(1000);
for (ConsumerRecord<String, String> record : records) {
    System.out.printf("Offset: %d, Key: %s, Value: %s%n", record.offset(), record.key(), record.value());
}
consumer.close();
```

This comparison highlights Kafka’s suitability for modern event-driven, scalable, and fault-tolerant architectures.

25. What is the replication factor?
26. How do leaders and followers work in Kafka replication?
27. How do consumers read data in Kafka?
28. What are producer acknowledgments?kindly try to answer all the given questions in details along with example in Java

25. What is the Replication Factor?

Replication Factor in Kafka

- The **replication factor** specifies the number of copies (replicas) of a partition in a Kafka cluster.
- A higher replication factor improves **fault tolerance** because data is stored on multiple brokers.

Key Points:

1. **Leader and Followers**:
    - One replica acts as the **leader**, while the others are **followers**.
    - The leader handles all read and write operations.
2. **Fault Tolerance**:
    - If a broker fails, one of the followers can take over as the leader.
3. **Configuration**:
    - Specified during topic creation.
    - Example: A replication factor of 3 means there are 3 copies of each partition.

Example in Java:

```java
// Creating a topic with a replication factor of 2
Properties adminProps = new Properties();
adminProps.put("bootstrap.servers", "localhost:9092");

AdminClient adminClient = AdminClient.create(adminProps);
NewTopic newTopic = new NewTopic("replicated-topic", 3, (short) 2); // 3 partitions, replication factor 2
adminClient.createTopics(Collections.singleton(newTopic));
adminClient.close();
```

---

26. How Do Leaders and Followers Work in Kafka Replication?

Leaders and Followers in Kafka

- **Leader**:
    - The broker responsible for handling all read and write operations for a partition.
- **Followers**:
    - Replicas of the partition that synchronize with the leader.
    - Followers do not handle client requests directly.

Replication Workflow:

1. **Write Operation**:
    - Producers send data to the leader.
    - The leader replicates the data to followers.
2. **Read Operation**:
    - Consumers read data from the leader.
3. **Fault Tolerance**:
    - If a leader fails, Kafka elects a new leader from the **In-Sync Replica (ISR)** list.

Example in Java:

You can view leader details for a topic:

```java
// Describe topic to check leaders and replicas
Properties adminProps = new Properties();
adminProps.put("bootstrap.servers", "localhost:9092");

AdminClient adminClient = AdminClient.create(adminProps);
DescribeTopicsResult describeTopicsResult = adminClient.describeTopics(Collections.singletonList("replicated-topic"));
Map<String, TopicDescription> topicDescriptions = describeTopicsResult.all().get();
topicDescriptions.values().forEach(topic -> {
    topic.partitions().forEach(partition -> {
        System.out.printf("Partition: %d, Leader: %s, Replicas: %s%n",
                          partition.partition(), partition.leader(), partition.replicas());
    });
});
adminClient.close();
```

---

27. How Do Consumers Read Data in Kafka?

Consumers in Kafka

- Consumers read messages from topics, and each topic is divided into partitions.
- Kafka allows parallel consumption using **consumer groups**.

Steps for Consumer Data Reading:

1. **Subscription**:
    - Consumers subscribe to one or more topics.
2. **Polling**:
    - Consumers poll messages from the assigned partitions.
3. **Offset Tracking**:
    - Kafka tracks the position (offset) of each consumer within a partition.

Example in Java:

java

```java
// Kafka Consumer Example
Properties consumerProps = new Properties();
consumerProps.put("bootstrap.servers", "localhost:9092");
consumerProps.put("group.id", "my-consumer-group");
consumerProps.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
consumerProps.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("my-topic"));

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("Partition: %d, Offset: %d, Key: %s, Value: %s%n",
                          record.partition(), record.offset(), record.key(), record.value());
    }
}
```

---

28. What Are Producer Acknowledgments?

Producer Acknowledgments (Acks)

Producer acknowledgments determine how many Kafka brokers must confirm the receipt of a message before the producer considers it successfully sent.

Acknowledgment Levels:

1. **`` `acks=0` ``**:

    - The producer does not wait for acknowledgment from the broker.
    - Fast but unreliable (no guarantee the message was received).
2. **`acks=1`**:
    - The producer waits for an acknowledgment from the leader broker only.
    - Balances reliability and speed.
3. **`acks=all` (or `acks=-1`)**:
    - The producer waits for acknowledgments from all replicas in the ISR.
    - Ensures the highest durability.

Example in Java:

```java
// Producer with Acknowledgments
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("acks", "all"); // Wait for all replicas in ISR to acknowledge
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("acks-topic", "key1", "Message with Acknowledgments"), (metadata, exception) -> {
    if (exception == null) {
        System.out.printf("Message sent to partition %d with offset %d%n", metadata.partition(), metadata.offset());
    } else {
        exception.printStackTrace();
    }
});
producer.close();
```

When to Use Each Level:

- Use `acks=0` for high-speed logging where message loss is acceptable.
- Use `acks=1` for general use cases where performance is critical, but durability is reasonable.
- Use `acks=all` for critical data where durability is paramount.

These answers and Java examples provide detailed insights into Kafka replication, consumer behavior, and producer acknowledgments, highlighting their significance in Kafka's reliability and scalability.

29. What is the transaction API in Kafka?
30. Explain Kafka’s "exactly once" semantics.
31. How does Kafka manage backpressure?
32. What is linger.ms in Kafka producers?kindly try to answer all the given questions in details along with example in Java

29. What is the Transaction API in Kafka?

Kafka Transaction API

The Kafka Transaction API allows applications to produce and consume messages with **atomicity**, **consistency**, **isolation**, and **durability (ACID)** guarantees. It ensures that multiple operations (e.g., producing to multiple partitions, consuming from multiple topics) are treated as a single atomic transaction.

Use Cases:

1. **Atomic Writes**:
    - Ensure that messages across multiple partitions are committed together.
2. **Read-Process-Write**:
    - Consume messages from input topics, process them, and produce output messages atomically.
3. **Exactly Once Semantics**:
    - Prevent duplicate or incomplete processing.

Example in Java:

```java
// Producer with Transactions
Properties producerProps = new Properties();
```

```
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("transactional.id", "transaction-id"); // Unique ID for transactional producer
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.initTransactions();

try {
    producer.beginTransaction();
    producer.send(new ProducerRecord<>("topic1", "key1", "Message for topic1"));
    producer.send(new ProducerRecord<>("topic2", "key2", "Message for topic2"));
    producer.commitTransaction(); // Both messages are committed together
} catch (Exception e) {
    producer.abortTransaction(); // Rollback on failure
}
```

---

30. Explain Kafka's "Exactly Once" Semantics

Exactly Once Semantics (EOS)

Kafka's **exactly-once semantics** ensure that messages are neither lost nor processed more than once, even in failure scenarios. This is achieved through:

1. **Idempotent Producers**:
    - Ensures that duplicate messages sent by the producer are not written to the Kafka log.
    - Enabled by setting `` `enable.idempotence=true` ``.
2. **Transactional API**:
    - Allows atomic production and consumption of messages across topics and partitions.
3. **Consumer Offset Management**:
    - Kafka stores offsets in a topic (`` `__consumer_offsets` ``) and commits them transactionally with the messages.

Example in Java:

```java
// Producer with Exactly Once Semantics
Properties producerProps = new Properties();
```

```
    producerProps.put("bootstrap.servers", "localhost:9092");
    producerProps.put("enable.idempotence", "true");
    producerProps.put("transactional.id", "transaction-id");
    producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
    producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

    KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
    producer.initTransactions();

    try {
        producer.beginTransaction();
        producer.send(new ProducerRecord<>("exactly-once-topic", "key1", "EOS Message"));
        producer.commitTransaction();
    } catch (Exception e) {
        producer.abortTransaction();
    }
```

---

31. How Does Kafka Manage Backpressure?

Backpressure in Kafka

Kafka manages backpressure by controlling the rate of data flow between producers, brokers, and consumers:

1. **Producer Side**:
    - Producers use **buffering** and block when the broker cannot keep up.
    - Configurations like `` `max.in.flight.requests.per.connection` `` and `` `batch.size` `` control this behavior.
2. **Broker Side**:
    - Kafka brokers store messages on disk, which prevents memory overflow.
3. **Consumer Side**:
    - Consumers poll messages at their own pace.
    - If consumers fall behind, they process messages in smaller batches until they catch up.

Example in Java:

Configuring producer to manage backpressure:

```java
java

  Properties producerProps = new Properties();
  producerProps.put("bootstrap.servers", "localhost:9092");
  producerProps.put("batch.size", 16384); // Batch size in bytes
  producerProps.put("linger.ms", 10); // Wait time before sending a batch
  producerProps.put("max.in.flight.requests.per.connection", 5); // Limit simultaneous requests
  producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
  producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

  KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
  producer.send(new ProducerRecord<>("backpressure-topic", "key1", "Message with Backpressure Control"));
  producer.close();
```

---

32. What is `linger.ms` in Kafka Producers?

`linger.ms` in Kafka

- `linger.ms` is a producer configuration that controls how long Kafka will wait before sending a batch of messages.
- A higher value increases batching, improving throughput but at the cost of increased latency.

How It Works:

1. Messages are accumulated in the producer's buffer.
2. The producer sends a batch:
   - When the batch size exceeds `batch.size`.
   - When the waiting time exceeds `linger.ms`.

Example in Java:

```java
java

  // Producer with Linger.ms Configuration
  Properties producerProps = new Properties();
  producerProps.put("bootstrap.servers", "localhost:9092");
  producerProps.put("linger.ms", 5); // Wait for 5 ms before sending messages
  producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
```

```
    producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

    KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
    producer.send(new ProducerRecord<>("linger-topic", "key1", "Message with Linger.ms"));
    producer.close();
```

Use Cases:

- **Low Latency**: Use a small `` `linger.ms` `` value for real-time systems.
- **High Throughput**: Use a larger value for batch processing.

These detailed explanations with Java examples demonstrate Kafka’s advanced features like transactions, exactly-once semantics, backpressure management, and producer optimizations for performance and reliability.

33. How does Kafka handle message compression?
34. What is batch.size in producers?
35. How does Kafka handle consumer offsets?
36. What is auto.offset.reset in Kafka consumers?kindly try to answer all the given questions in details along with example in Java

33. How Does Kafka Handle Message Compression?

Message Compression in Kafka

Kafka allows producers to compress messages before sending them to brokers. This reduces network usage and storage requirements.

Compression Options:

1. **None**: No compression (default).
2. **gzip**: High compression ratio but slower.
3. **snappy**: Faster but lower compression ratio.

4. **lz4**: A balance between compression ratio and speed.
5. **zstd**: High compression ratio and fast compression (introduced in later Kafka versions).

How It Works:

1. Producers compress the message batch.
2. Brokers store the compressed batch as-is.
3. Consumers decompress messages while consuming.

Example in Java:

```java
// Producer with Compression
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("compression.type", "gzip"); // Enable gzip compression
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("compressed-topic", "key1", "Compressed Message"));
producer.close();
```

---

34. What is `batch.size` in Producers?

`batch.size` in Kafka Producers

- `batch.size` specifies the maximum size (in bytes) of a batch of messages that the producer will send to a broker.
- Larger batch sizes increase throughput but may introduce latency if the batch takes longer to fill.

Key Points:

1. If a batch reaches the `batch.size` limit, it is sent immediately, regardless of `linger.ms`.
2. If `linger.ms` expires before the batch fills up, the partial batch is sent.

Example in Java:

java

```java
// Producer with Batch Size
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("batch.size", 32768); // Batch size in bytes (32 KB)
producerProps.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
producerProps.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer<String, String> producer = new KafkaProducer<>(producerProps);
producer.send(new ProducerRecord<>("batch-topic", "key1", "Message in Batch"));
producer.close();
```

Use Cases:

- Use a larger `` `batch.size` `` for high-throughput systems where latency is less critical.
- Use a smaller size for low-latency systems.

---

35. How Does Kafka Handle Consumer Offsets?

Consumer Offsets in Kafka

Offsets represent the position of a consumer in a partition. Kafka uses offsets to track which messages have been consumed.

Offset Management:

1. **Automatic Offset Commit**:
    - Kafka automatically commits offsets at regular intervals (controlled by `` `auto.commit.interval.ms` ``).
2. **Manual Offset Commit**:
    - The consumer explicitly commits offsets after processing messages.

Offset Storage:

1. Kafka stores offsets in an internal topic (`` `__consumer_offsets` ``).

2. Consumers can replay messages by resetting offsets.

Example in Java:

Automatic Offset Commit:

```java
Properties consumerProps = new Properties();
consumerProps.put("bootstrap.servers", "localhost:9092");
consumerProps.put("group.id", "offset-group");
consumerProps.put("enable.auto.commit", "true"); // Automatically commit offsets
consumerProps.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
consumerProps.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("offset-topic"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("Offset: %d, Value: %s%n", record.offset(), record.value());
    }
}
```

Manual Offset Commit:

```java
consumer.commitSync(); // Synchronously commit offsets after processing
```

---

36. What is `auto.offset.reset` in Kafka Consumers?

`auto.offset.reset` in Kafka

- Determines the consumer's behavior when no committed offset exists or the committed offset is invalid (e.g., when a partition is deleted or reset).

Options:

1. `earliest`:
    - Starts reading messages from the beginning of the partition.
2. `latest` (default):
    - Starts reading messages from the end of the partition.
3. `none`:
    - Throws an error if no offset is found.

Example in Java:

```java
Properties consumerProps = new Properties();
consumerProps.put("bootstrap.servers", "localhost:9092");
consumerProps.put("group.id", "reset-group");
consumerProps.put("auto.offset.reset", "earliest"); // Start from the beginning if no offset exists
consumerProps.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
consumerProps.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);
consumer.subscribe(Collections.singletonList("reset-topic"));
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(1000);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("Offset: %d, Value: %s%n", record.offset(), record.value());
    }
}
```

Use Cases:

- `earliest`: Useful for replaying data.
- `latest`: Suitable for real-time processing.
- `none`: Ensures consumers always start from a valid committed offset.

These detailed explanations with Java examples demonstrate Kafka's capabilities for compression, batching, offset management, and consumer behavior, providing insights into optimizing and controlling Kafka applications.

37. How do rebalances work in consumer groups?
38. What is the purpose of the Schema Registry?
39. How does Kafka handle topic deletion?
40. Explain commitSync() vs commitAsync() in Kafka consumers.kindly try to answer all the given questions in details along with example in Java

37. How Do Rebalances Work in Consumer Groups?

Rebalances in Consumer Groups

A rebalance occurs when the partitions of a topic are reassigned among the consumers in a consumer group. It ensures that all partitions are equally distributed among the consumers.

When Does Rebalancing Happen?

1. A new consumer joins the group.
2. A consumer leaves the group.
3. A partition is added to the topic.

Rebalancing Workflow:

1. Kafka's **Group Coordinator** reassigns partitions to consumers.
2. During a rebalance, consumers stop processing messages until the new assignments are finalized.

Minimizing Rebalance Impact:

- Use `` `max.poll.interval.ms` `` to adjust rebalance frequency.
- Enable **sticky assignment** to minimize partition movement during rebalancing.

Example in Java:

```java
Properties consumerProps = new Properties();
consumerProps.put("bootstrap.servers", "localhost:9092");
consumerProps.put("group.id", "rebalance-group");
consumerProps.put("enable.auto.commit", "false");
consumerProps.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");
consumerProps.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(consumerProps);

// Handling rebalances with a Rebalance Listener
consumer.subscribe(Collections.singletonList("rebalance-topic"), new ConsumerRebalanceListener() {
    @Override
    public void onPartitionsRevoked(Collection<TopicPartition> partitions) {
        System.out.println("Partitions revoked: " + partitions);
    }

    @Override
    public void onPartitionsAssigned(Collection<TopicPartition> partitions) {
        System.out.println("Partitions assigned: " + partitions);
    }
});
```

---

38. What Is the Purpose of the Schema Registry?

Schema Registry in Kafka

The Schema Registry manages schemas for Kafka topics, ensuring data consistency between producers and consumers. It is part of the **Confluent Platform**.

Key Features:

1. **Schema Evolution**:
    - Supports schema compatibility (backward, forward, or full).
    - Allows changes to schemas without breaking consumers or producers.
2. **Data Validation**:
    - Producers validate messages against the schema before sending them.
    - Consumers deserialize messages based on the schema.
3. **Centralized Repository**:

- Stores schemas in a centralized location.

Example Use Case:

- Producers use Avro, JSON, or Protobuf to serialize data. The Schema Registry ensures consumers understand the serialized data format.

Example in Java:

```java
Properties producerProps = new Properties();
producerProps.put("bootstrap.servers", "localhost:9092");
producerProps.put("key.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
producerProps.put("value.serializer", "io.confluent.kafka.serializers.KafkaAvroSerializer");
producerProps.put("schema.registry.url", "http://localhost:8081");

KafkaProducer<String, GenericRecord> producer = new KafkaProducer<>(producerProps);

// Example schema
String schemaString = "{\"type\":\"record\",\"name\":\"User\",\"fields\":[{\"name\":\"name\",\"type\":\"string\"},{\"name\":\"age\",\"type
\":\"int\"}]}";
Schema schema = new Schema.Parser().parse(schemaString);

GenericRecord record = new GenericData.Record(schema);
record.put("name", "John");
record.put("age", 30);

producer.send(new ProducerRecord<>("schema-topic", "key1", record));
producer.close();
```

---

39. How Does Kafka Handle Topic Deletion?

Topic Deletion in Kafka

Kafka allows topics to be deleted, but the process involves several configurations and steps.

Key Configurations:

1. **Enable Deletion**:

    - Set `delete.topic.enable=true` in the broker configuration.
2. **Replication**:
    - All replicas of a partition must be removed for the topic to be fully deleted.

Topic Deletion Steps:

1. The topic is marked for deletion.
2. Kafka removes all associated partitions and their data.
3. Kafka deletes the topic metadata.

Example in Java:

```java
Properties adminProps = new Properties();
adminProps.put("bootstrap.servers", "localhost:9092");

AdminClient adminClient = AdminClient.create(adminProps);
adminClient.deleteTopics(Collections.singletonList("topic-to-delete")).all().whenComplete((result, error) -> {
    if (error == null) {
        System.out.println("Topic deleted successfully.");
    } else {
        error.printStackTrace();
    }
});
adminClient.close();
```

---

40. Explain `commitSync()` vs `commitAsync()` in Kafka Consumers

`commitSync()`:

1. **Synchronous Offset Commit**:
    - Blocks until Kafka acknowledges the offset commit.
2. **Reliability**:
    - Guarantees that offsets are committed before proceeding.
3. **Performance**:

  - Slower due to blocking.

`commitAsync()`:

1. **Asynchronous Offset Commit**:
   - Sends the offset commit request and does not wait for acknowledgment.
2. **Reliability**:
   - Offset commits may fail silently.
3. **Performance**:
   - Faster but less reliable.

Example in Java:

`commitSync()`:

```java
consumer.poll(1000).forEach(record -> {
    System.out.printf("Offset: %d, Value: %s%n", record.offset(), record.value());
});
consumer.commitSync(); // Block until the offset is committed
```

`commitAsync()`:

```java
consumer.poll(1000).forEach(record -> {
    System.out.printf("Offset: %d, Value: %s%n", record.offset(), record.value());
});
consumer.commitAsync((offsets, exception) -> {
    if (exception == null) {
        System.out.println("Offsets committed: " + offsets);
    } else {
        exception.printStackTrace();
    }
});
```

When to Use Each:

- Use `` `commitSync()` `` when reliability is critical (e.g., exactly-once processing).
- Use `` `commitAsync()` `` for faster processing where occasional offset commit failures are acceptable.

These detailed explanations with Java examples illustrate Kafka’s features for handling rebalances, schema management, topic deletion, and offset commits, showcasing how to manage these scenarios effectively.